Procedural Language

- In this section we will cover
  - About stored procedure
  - Functions / Stored procedures
  - Declaring variables
  - Handling statements
  - Control structures
  - Using cursors
  - Handling exceptions
  - Triggers
  - PLpython

About Stored Procedure

- SQL is a non-procedural language.
- Without the stored procedures, the application is forced to perform a number of SQL calls that will result in network exchanges between the application and the database.
- With stored procedures, the application can execute business processing in a single database call. Each SQL query of the PL/SQL code will be executed directly on the database without exchange with the application. The data processing is closer to the data itself.
- Complex processing are sometimes difficult to solve without a procedural language.

About stored procedure

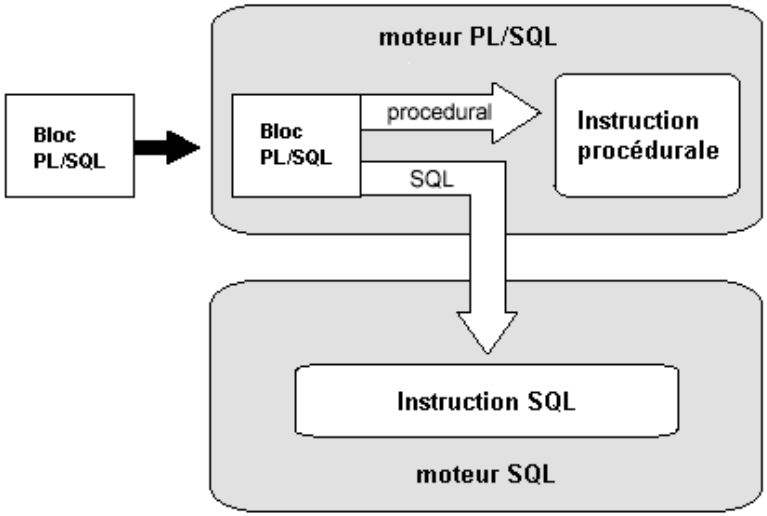
- As procedural languages are known by many, it is often easier to express one's needs in this way. Be careful: procedures apply to complex and frequent treatments.
- PL/PgSQL is simple to write, and offers a fast access to SQL.
- SQL indicates to the database what to do and procedural language indicates how to do it.
- A part of the business code may be deported to the database and is therefore shared on the database side.
- However, using a procedural language presents some limitations :
  - The portability is limited to the used RDBMS.
  - Difficult to manage versions and hard to debug.
  - Requires skills in stored procedure programming.

About Stored Procedure

- The most popular procedural language is PL/SQL :
  - Stand for Procedural Language extension to SQL
  - Developped by Oracle
- Each RDBMS implements its own procedural language :
  - Postgres :PL/pgSQL – heavily influenced by PL/SQL
  - Oracle : PL/SQL
  - Mysql : Msql
- Some RDBMS provide additional extensions
  - Postgres : PL/Python, PL/Perl, PL/R …
  - Oracle : PL/JAVA, PL/PRO C
  - Mysql : ?
- Even though the default procedural language provides a set of features. If you need to run some complex text processing or to handle files, some procedural languages such as PL/Perl, PL/C, PL/Python, PL/Java will be likely more convenient.

PL/SQL engine

- The PL/SQL engine resides in memory and processes the PL/SQL code instructions.
- When the PL/SQL engine encounters a SQL statement, a context switch is made to pass the SQL statement to the SQL engine.
- The PL/SQL engine waits for the SQL statement to complete and for the results to be returned before it continues to process subsequent statements in the PL/SQL block
- For any other language PL/Python, JAVA, … the mechanism is identical.



About Stored Procedure

- PL/SQL is supported only in certains database objects
- Functions
  - Functions are named objects that contains SQL and/or PL/SQL statements and returns a value of a specified data type
- Procedures
  - Procedures are named objects that contain SQL and/or PL/SQL statements
  - Theoretically, it's a function returning void
  - In practice, a procedure udpates data contrarely to function which computes data from a table.
  - For a long time, Postgres only implemented functions but since the last release stored procedure have been added.
- Triggers
  - are procedures fired when a certain events occurs.
- Anonymous blocks
  - Anonymous blocks are unnamed procedure
  - They are not stored in the database and consequently cannot be replayed without retyping the procedure.
  - They are passed to the PL/SQL engine for execution at run time

PL/SQL Block Structure

- DECLARE (optional)
  - Contains declarations of all variables
  - Constants, cursors, user-defined
- BEGIN (mandatory)
  - Implements the business logic
  - Must contain at least one instruction :
    - SQL
    - PL/SQL
- EXCEPTION (optional)
  - Specifies the actions to perform when errors occurs.
- END; (mandatory)

Anonymous block

```
DO $$                       -- DO (mandatory) indicates to Postgres
                            -- it's an anonymous block.
<< label >>                 -- you can define a label we will see the interest  later
DECLARE                     -- Optional section for declaring variables
BEGIN                       --  computing block (mandatory) :
                            --  has to contain at least one instruction
  Raise notice 'Hello';   -- Print Hello
END                        -- computing block end (mandatory)
$$
```

Procedures and Functions

- They are named PL/SQL program.
- They have the same block structure as anonymous blocks
- Procedures and functions are compiled and stored in the database in a compiled form.
- Any application can use them.
- They may have parameters.

Function

```
CREATE or replace FUNCTION
f_hello(v_myTxt text) RETURNS text
AS $BODY$
DECLARE
  v_hello text ='Hello';
BEGIN
  RETURN v_hello||' '||v_myTxt;
END
$BODY$
LANGUAGE plpgsql;

select f_hello('Guy');
```

https://www.postgresql.org/docs/11/sql-createfunction.html

Procedure

```
CREATE PROCEDURE
p_hello(v_myTxt varchar(50))
LANGUAGE plpgsql
AS $BODY$
DECLARE
   v_hello text ='Hello';
BEGIN
   Raise notice '% : % ', v_hello,
v_myTxt;
END
$BODY$;

call p_hello('Guy');
```

https://www.postgresql.org/docs/11/sql-createprocedure.html

Variables

- Variables can be used for:
  - Temporary storage of data
  - Manipulation of stored values
  - Reusability

- Variables are:
  - Declared and (optionally) initialized in the declarative section
  - Used and modified in the BEGIN / EXCEPTION section

Use of variables

- They can be defined in the definition block or in the prototype of a function or procedure with a block AS.

Anonymous block

```
DO $BODY$
DECLARE
  v_myTxt text ='Hello';
BEGIN
 Raise notice '%',v_myTxt;
END
$BODY$
LANGUAGE plpgsql;
```

Function

```
CREATE or replace FUNCTION
f_hello(v_myTxt text) RETURNS text
AS $BODY$
DECLARE
  v_hello text ='Hello';
BEGIN
  RETURN v_hello||' '||v_myTxt;
END
$BODY$
LANGUAGE plpgsql;

select f_hello('Guy');
```

Procedure

```
CREATE PROCEDURE
p_hello(v_myTxt varchar(50))
LANGUAGE plpgsql
AS $BODY$
DECLARE
   v_hello text ='Hello';
BEGIN
   Raise notice '% : % ', v_hello,
v_myTxt;
END
$BODY$;

call p_hello('Guy');
```

Types of variables

- Good practice : Define a Naming convention
  - Variable v_myvar
  - Constant variable c_myvar
  - Cursor cur_mycursor
  - Record rec_myrec
  - Type type_mytype
  - …
- PL/SQL variables:
  - Scalar : Scalar data types hold a single value.
  - Composite : Composite data types are a collection of any thing (user type, scalar, …)
  - Cursor : is a pointer on a table

Scalar variables

```
identifier [CONSTANT] datatype [NOT NULL] [:= | DEFAULT expr];
```

- Constant : Constrains the variable so that its value can not change
- NOT NULL : mandatory value

```
identifier table.column_name%TYPE;
```

```
identifier identifier%TYPE;
```

- Allow to reference a column or a variable type.
- This declaration is easy and avoid type error.

Scalar variables

- Scalar variables may be initialized from different ways.

- Manually
  - V_galaxy varchar(50) := ‘nebula’;
  - V_galaxy varchar(50) DEFAULT ‘nebula’;

- SQL functions
  - v_desc_size integer ;
  - v_desc_size:= LENGTH(‘nebula’);
  - v_desc_size:= to_number(‘100’)

- From a query
  - v_myVar varchar(50); ⇔ v_myVar otypedef.otype_descr%TYPE
  - SELECT otype_descr into strict v_myVar FROM otypedef where otype_bin = -100663296;

Composite variables : Records Type

- Composite variables
  - can hold multiple values (unlike scalar types) of anything.
  - Are user-defined and can be a subset of a row in a table
  - Are convenient for fetching a row of data from a table for processing

```
DO $$
DECLARE
rec_myrow RECORD;
BEGIN
   SELECT * 1 into strict rec_myrow FROM categories where category=1;
   Raise info Id category : % – Category Name : % ',rec_myrow.category,rec_myrow.categoryname;

END
$$
```

```
CREATE TYPE t_myType AS (category int, categoryname varchar(100));

DO $$
DECLARE
myrow_type t_myType;
BEGIN
   SELECT category, categoryname FROM categories where category=1 into strict myrow_type ;
   Raise info 'Id category : % – Category Name : % ',myrow_type.category,myrow_type.categoryname;

END
$$
```

Composite variables: ROWTYPE

- ROWTYPE : define the data type of a variable to the row structure of a database catalog object
  - The number and data types of the underlying database columns does not need be known.
  - Useful for handling data.
  - Simplifies maintenance, if the table structure changes your code does not need to be update.

```
DO $$
DECLARE
myrow_rowtype categories%ROWTYPE;
BEGIN
   SELECT * FROM categories where category=1 into strict myrow_rowtype ;
   Raise info 'Id category : % - Category Name : % ',myrow_rowtype.category,myrow_rowtype.categoryname;
   myrow_rowtype.categoryname = myrow_rowtype.categoryname || ' updated';
   --update categories SET category = myrow_rowtype.category, categoryname = myrow_rowtype.categoryname where
category=myrow_rowtype.category;
   update categories SET (category,categoryname)= ROW(myrow_rowtype.*) where category=myrow_rowtype.category;
   SELECT * FROM categories where category= 1 into strict myrow_rowtype ;
   Raise info 'Id category : % - Category Name : % ',myrow_rowtype.category,myrow_rowtype.categoryname;
END
$$
```

Variable scope

- A procedure can include many blocks which may include some other blocks and so on…

```
<< level1 >>
DECLARE …
BEGIN …

    << level2.1 >>
        DECLARE ...
        BEGIN ...
            << level3 >>
                DECLARE ...
                BEGIN ...

                EXCEPTION ...
                END level3;
        EXCEPTION ...
        END level2.1;

    << level2.2 >>
        DECLARE ...
        BEGIN ...

        EXCEPTION ...
        END level2.2;

EXCEPTION
END level1;
```

By default, variables of a parent block are visibles to child blocks.

Variables of a block always overload Parent block variables.
Use labels for referencing a variable from parent block.

Instruction set defined in a procedure are included in the same transation.

Manipulating Data

```
DO
$$
DECLARE
counter int;
BEGIN
select count(*) into counter from categories;
raise info 'The number of rows is %',counter;
END;
$$
LANGUAGE 'plpgsql'
```

```
DO
$$
DECLARE
counter int;
category categorie
categoryname cate
BEGIN
select count(*) into
raise info 'The num
SELECT category,categoryname into strict
category,categoryname  FROM categories where
category=1  ;
 Raise info 'Id category : % - Category Name : %
',category,categoryname;
END;
$$
LANGUAGE 'plpgsql'
```

```
ERREUR:  la référence à la colonne « category » est ambigu
LINE 1: SELECT category,categoryname  FROM categories where category...
               ^
DETAIL:  Cela pourrait faire référence à une variable PL/pgsql ou à la colonne d'une
table.
QUERY:  SELECT category,categoryname  FROM categories where category=1
CONTEXT:  fonction PL/pgsql inline_code_block, ligne 9 à instruction SQL
```

Good practice :
-> Define a Naming convention
-> declare variables in a labeled block

Manipulating Data

- Make changes to database tables by using DML commands:
  - SELECT
  - INSERT
  - UPDATE
  - DELETE
  - MERGE
- By default, if a procedure succeeds then the whole transaction is committed else it is rolled back.
- Executing Dynamic Commands
  - Use the EXECUTE command

- Executing Dynamic Commands
  - Oftentimes you want to generate dynamic commands inside your PL/pgSQL.
  - A procedure is compiled and afterwards executed. It is not possible to run dynamic SQL queries.
  - RDBMS allow to run dynamic SQL via the EXECUTE command.
  - Your dynamic query is always planned each time the statement is run.
  - You can run DDL, DCL , DML dynamically but take care with transactions. For example Oracle considers DDL as an atomic operation and commit all DML preceding the DDL.

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND
inserted <= $2' INTO c USING checked_user, checked_date;
```

```
EXECUTE 'SELECT count(*) FROM ' || quote_ident(tabname) || ' WHERE
inserted_by = $1 AND inserted <= $2' INTO c USING checked_user,
checked_date;
```

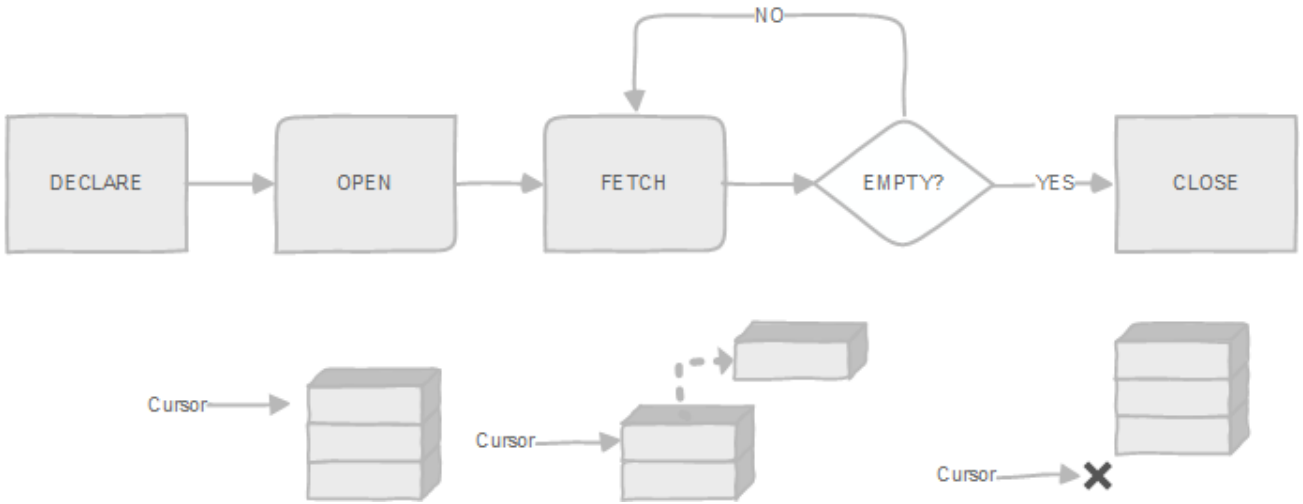
```
EXECUTE format('SELECT count(*) FROM %I ' 'WHERE inserted_by = $1 AND
inserted <= $2', tabname) INTO c USING checked_user, checked_date;
```

Cursors

- As a developper, you may want to retrieve multiple rows from a table and apply to each row a business processing.
- Cursor allows us to encapsulate a query result and process each individual row at a time.
- RDBMS allocates a private memory area for processing SQL statements. The SQL statement is parsed and processed in this area. You don’t have any control on this area.
- Each cursor has a set of attributes associated with it that allows the program to test the state of the cursor
  - ISOPEN: attribute is used to test whether or not a cursor is open.
  - FOUND : attribute is used to test whether or not a row is retrieved from the result set of the specified cursor after a FETCH on the cursor.
  - NOTFOUND : attribute is the logical opposite of %FOUND.
  - ROWCOUNT: attribute returns an integer showing the number of rows FETCHed so far from the specified cursor
- A cursor may be parameterized
  - CURSOR c1 (v_category NUMBER) IS SELECT * FROM categories WHERE category < v_category ;

Cursors

- Cursor operations:
  - Declaration : All access to cursors goes through cursor variables
  - Open : Before a cursor variable can be used to retrieve rows, it must be *opened*.
  - FETCH retrieves the next row from the cursor into a target, which might be a row variable, a record variable, or a comma-separated list of simple variables
  - UPDATE/DELETE : When a cursor is positioned on a table row, that row can be updated or deleted using the cursor to identify the row.
  - CLOSE release resources



Cursors with PostgreSQL

```
DO $$
DECLARE
   ref refcursor;
   row RECORD;
BEGIN
   OPEN ref FOR SELECT * FROM otypedef;
   LOOP
      FETCH ref INTO row;
      EXIT WHEN NOT FOUND;
      raise info 'row %', row.otype_descr;
   END LOOP;
   CLOSE ref;
END;
$$
```

```
DO $$
DECLARE
   cur_ref refcursor;
   rec_row RECORD;
BEGIN
   OPEN cur_ref FOR SELECT * FROM categories order by category;
   FETCH FIRST FROM cur_ref into rec_row;
   IF FOUND THEN
   raise info 'First row %', rec_row.category;
   END IF;
   FETCH cur_ref into rec_row;
   raise info 'Second row %', rec_row.category;
   MOVE NEXT FROM cur_ref;
   FETCH cur_ref INTO rec_row;
   raise info 'Fourth row %', rec_row.category;
   FETCH cur_ref INTO rec_row;
   raise info 'Fifth  row %', rec_row.category;
   MOVE FORWARD 2 FROM cur_ref;
   FETCH cur_ref INTO rec_row;
   raise info 'heighth row %', rec_row.category;

   MOVE LAST FROM cur_ref;
   raise info 'Cursor over result';
   IF NOT FOUND THEN
   raise info 'Last row %', rec_row.category;
   MOVE RELATIVE -1 from cur_ref;
   END IF;
   raise info 'Last row %', rec_row.category;
   CLOSE cur_ref;
END;
$$
;
```

Control structures : IF

```
IF condition THEN instructions
[ELSEIF condition THEN instructions]*
[ELSE instruction]
END IF;
```

```
DO $$
DECLARE
   cur_ref refcursor;
   rec_row RECORD;
   v_catname categories.categoryname%TYPE;
BEGIN
   OPEN cur_ref FOR SELECT * FROM categories where category=15 order by category;
   FETCH FIRST FROM cur_ref into rec_row;
   v_catname = rec_row.categoryname;
   IF NOT FOUND THEN
      raise info 'There is no row %', rec_row.category;
   ELSEIF v_catname = 'New' then
      raise info 'There is a great category  %', rec_row.category;
   ELSEIF v_catname = 'Sports' then
      raise info 'There is a great great category  %', rec_row.category;
   ELSE
      raise info 'There is a category  %', ec_row.category;
   END IF;
   CLOSE cur_ref;
END;
$$
LANGUAGE 'plpgsql';
```

Control structures : CASE

```
CASE variable
WHEN condition THEN instructions
ELSE instructions
END CASE
```

```
DO $$
DECLARE
    ref refcursor;
    row RECORD;
    catname categories.categoryname%TYPE;
BEGIN
    OPEN ref FOR SELECT * FROM categories where category=15 order
by category;
    FETCH FIRST FROM ref into row;
    catname = row.categoryname;
    CASE catname
    WHEN 'Sports' then
        raise info 'There is a great category  %', catname;
    WHEN'Games' then
        raise info 'There is a great great category  %', catname;
    ELSE
        raise info 'It''s not an important category  %', catname;
    END CASE;
    CLOSE ref;
END;
$$
LANGUAGE 'plpgsql';
```

Control structures : CASE

```
CASE variable
WHEN condition THEN instructions
ELSE instructions
END CASE
```

```
CASE
WHEN condition THEN instructions
ELSE instructions
END CASE
```

```
DO $$
DECLARE
   cur_ref refcursor;
   rec_row RECORD;
   v_catname categories.categoryname%TYPE;
BEGIN
   OPEN cur_ref FOR SELECT * FROM categories where category=15
order by category;
   FETCH FIRST FROM cur_ref into rec_row;
   v_catname = rec_row.categoryname;
   IF NOT FOUND THEN
      raise info 'There is no row %', rec_row.category;
   ELSEIF v_catname = 'New' then
      raise info 'There is a great category  %', rec_row.category;
   ELSEIF v_catname = 'Sports' then
      raise info 'There is a great great category  %', rec_row.category;
   ELSE
      raise info 'There is a category  %', rec_row.category;
   END IF;
   CLOSE cur_ref;
END;
$$
LANGUAGE 'plpgsql';
```

Control structures : LOOP

```
<<label>>]
LOOP
instructions
EXIT [<<label>>] WHEN condition;
CONTINUE WHEN condition;
instructions
END LOOP;
```

```
DO $$
DECLARE
   v_resultat int = 0;
BEGIN
<< myloop >>
LOOP
   raise info 'Resultat %', v_resultat;
   v_resultat := v_resultat + 1;
   EXIT  myloop  WHEN v_resultat > 10;
   CONTINUE WHEN v_resultat < 5;
   v_resultat := v_resultat + 1;
END LOOP;
END;
$$
LANGUAGE 'plpgsql';
```

Control structures : While

```
[ <<label>> ]
WHILE condition LOOP
   statements;
END LOOP;
```

```
 DO $$
DECLARE
   v_counter int = 0;
   v_nb int = 10;
BEGIN
<< exitloop >>
 WHILE v_counter <= v_nb
 LOOP
 raise info 'Resultat %', v_counter;
 v_counter := v_counter + 1 ;
 if v_counter > 9 then
    EXIT exitloop;
 end if;
 END LOOP ;
 raise info 'I go out';
END;
$$
LANGUAGE 'plpgsql';
```

Control structures : FOR

```
[ <<label>> ]
FOR loop_counter IN [ REVERSE ] from.. to [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

```
DO $$
DECLARE
v_counter int = 0;
BEGIN
    FOR v_counter IN REVERSE 5..1 BY 2  LOOP
      RAISE NOTICE 'Counter: %', v_counter;
    END LOOP;
END;
$$
```

Handling Exceptions

- An exception is an error in your code which is thrown at run time. The code does not work as expected. You expected the SELECT statement to retrieve only one row; however, it retrieved multiple rows. Such errors that occur at run time are called exceptions.

- An exception is a PL/SQL error that is raised during program execution.

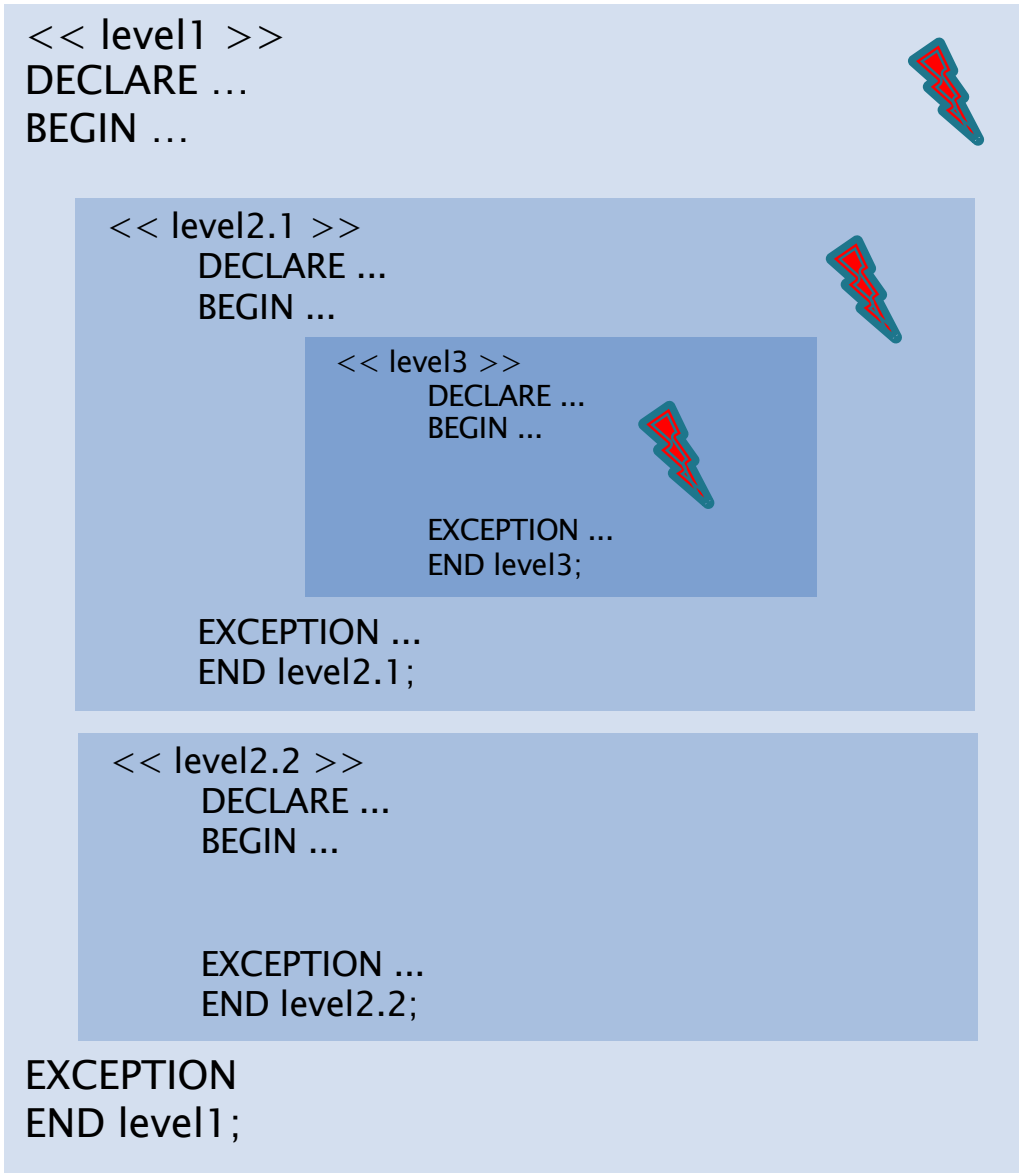
- An exception can be raised:
  - – Implicitly by the RDBMS
  - – Explicitly by the program/Developper

- Exception Types
  - Predefined error
  - User-defined error

- An exception can be handled:
  - – By trapping it with a handler
  - – By propagating it to the calling environment

```
DO $$
DECLARE
myrow_rowtype categories%ROWTYPE;
BEGIN
   SELECT * FROM categories into strict myrow_rowtype ;
   Raise info 'Id category : % ',myrow_rowtype.category;
EXCEPTION
    WHEN too_many_rows then
    Raise warning 'Catch an error %',SQLSTATE;
    Raise warning 'Description : %',SQLERRM;
    WHEN others then
    Raise warning 'Catch an other error %',SQLSTATE;
    Raise warning 'Description : %',SQLERRM;
END
$$
```

Handling Exceptions



All modifications run in the block are rolled back and
Variables keep their status.

Trapping Exception

- The EXCEPTION keyword starts the exception-handling section.

- Several exception handlers are allowed.
  - https://docs.postgresql.fr/current/errcodes-appendix.html
  - unique_violation, check_violation, no_data_found, too_many_rows …

- SQLSTATE : Returns the numeric value for the error code
- SQLERRM : Returns the message associated with the error number

- You can raise your own exceptions
  - raise exception ‘myexception’;

Triggers

- A trigger is a PL/SQL program as functions and stored procedures.
- A trigger is a procedure fired when a certain events occurs.
  - BEFORE insert / delete /update : it is fired BEFORE each affected row is changed.
  - AFTER insert / delete /update : it is fired AFTER each affected row is changed.
  - INSTEAD OF : RDBMS fires the trigger instead of executing the triggering event. Specific to complex view.
- It is associated to one and only one table/materialized view or view but you can create many triggers on the same object.
- A trigger may be fired in 2 ways :
  - FOR EACH ROW, it is fired for each updated row .
  - FOR EACH STATEMENT : it is fired only once for any given operation. Regardless of how many rows it modifies.
- When you use a trigger, special variables are created automatically in the top-level block :
  - 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
  - TG_OP : the operation/event name (UPDATE/DELETE/ INSERT) firing the trigger
  - https://www.postgresql.org/docs/11/plpgsql-trigger.html
- Watch out : If a trigger produces compilation errors, then it is still created, but it fails on execution. This means it effectively blocks all triggering DML statements until it is dropped or disabled.

Triggers

```
CREATE FUNCTION emp_stamp() RETURNS trigger
AS $emp_stamp$
BEGIN
    -- Check that empname and salary are given
    IF NEW.empname IS NULL
    THEN
        RAISE EXCEPTION 'empname cannot be null';
    END IF;
    IF NEW.salary < 0 THEN
        RAISE EXCEPTION '% cannot have a negative salary',NEW.empname;
    END IF;
    -- Remember who changed the payroll when
    NEW.last_date := current_timestamp;
    NEW.last_user := current_user;
    RETURN NEW;
END;
$emp_stamp$
LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_stamp
BEFORE INSERT OR UPDATE ON emp
FOR EACH ROW
EXECUTE FUNCTION emp_stamp();
```

PL/python

- PostgreSQL is one of RDBMS to provide the PL/python

```
CREATE FUNCTION py_func(argument-list)
                RETURNS return-type
AS $$
                 # PL/Python function
body $$
LANGUAGE plpythonu;
```

- Note that this procedure language is untrusted. It means it does not offer any way of restricting what users can do.

- An user can use a functions with an access to files and bypass PostgreSQL security or can change system configurations .

- Using this language is limited to super users.