Unit 8: Databases programming

- The PL/SQL language
- Constants and variables
- Error handling
- Control structures
- Stored procedures
- Cursors
- Triggers

Introduction

- Oracle → PL/SQL language
- Procedural programming language for database systems
- In Postgres → PL/pgSQL (similar)
- Purpose:
  - Create procedures and triggers
  - Extend standard SQL functionality with more complex operations

Basic types

- The basic data types available are the usual SQL ones (integer, numeric, varchar, ...)
- Likewise, they also take the regular, standard format for the values
- Text representation:
  - 'my string'
  - $$This is also a string$$
- The second syntax is recommended as it provides a more comfortable way to deal with special characters

Blocks

- PL/SQL code is organized in blocks
- Basic syntax:

-
  ```
  [ <<label>> ]
  [ declare
  <variable and constant declarations> ]
  begin
      <instruction>;
      [<instruction>; ...]
      ...
  end[label];
  ```

Blocks

Notes about the blocks:

- Optional declaration of variables and constants
- Optional label
- Body of the mandatory block → at least one statement delimited between *begin* and *end*
- Each statement ends with ;
- You can nest some blocks inside others, creating sub-blocks
- In a sub-block the identifiers of the upper block can be accessed (but as a rule of thumb it is <u>NOT</u> recommended)

Blocks

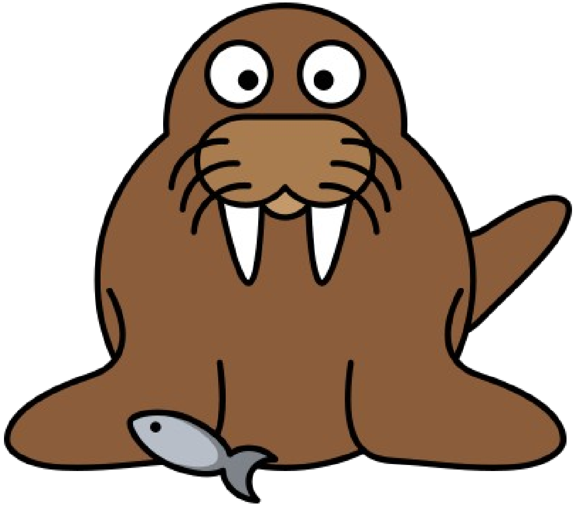
- Example:

```
do
$$ declare
  num_films integer := 0;
begin
   SELECT count(film_id)    -- total number of films
   INTO num_films
   FROM film;
   raise notice 'There are: % movies in the database', num_films;
end; $$
```

Constants and variables

- Declaration of a variable:
  - <variable_name> <data_type> [:= expression];
- Variables must be declared within the section of the corresponding block
- The rules that apply to variable names are usually similar to those of conventional programming

Constants and variables



- The := operator (also called walrus operator) is used to assign a value to a variable
- A special possibility: declaring a variable according to the same data type of a table column:
  - variable_name table_name.column_name%type;
  - Example: *film_title film.title%type;*

Constants and variables

- How to store data in variables coming from the database? → SELECT … INTO
- Example:

```
SELECT avg(age)
INTO average_age
FROM driver;
```

Constants and variables

- We can also save records in variables, or even lists
- Example:

```
declare
    selected_driver driver%rowtype;
```

- Alternatively, the following more generic syntax could be used:

```
declare
    selected_driver record;
```

Constants and variables

- Example:

-
  ```
  SELECT *
  INTO selected_driver
  FROM driver
  WHERE driver_id = '12345678A';
  ```

Constants and variables

- Then, all you have to do is access the record fields:

```
raise notice 'Full Name: % %'
selected_driver.firstname
selected_driver.lastname;
```

```
raise notice 'Age: %' selected_driver.age;
```

- ...and in the same way with the rest of the attributes

Constants and variables

- We also have means of declaring constants which, unlike variables, cannot be modified once initialized
- Example:

```
declare
    VAT constant numeric := 0.21;
    START_TIME constant time := now();
```

Handling errors

- If a message needs showing (no matter if it’s an error notification or just plain information) this statement must be used:

  raise level format;

- Where *level* can be one of the following:
  - debug
  - log
  - notice
  - info
  - warning
  - exception

Handling errors

- Examples:
  - raise info 'My information message %', now() ;
  - raise log 'My log message %', now();
  - raise debug 'My debug message %', now();
  - raise warning 'My warning message %', now();
  - raise notice 'My notice message %', now();

Handling errors

- Throwing an error:

```
raise exception 'Duplicate email: %', email
using hint = 'Please check your address';
```

- The generated output will be the following:

```
[Err] ERROR: Duplicate email: bbdd@correo.es
HINT: Please check your address
```

Assertions

- In order to debug feasible errors in a statement block, assertions are used
- An assertion is the verification that a specific condition is met
- If the assertion condition is not met when evaluated (in runtime), an error is displayed
- Syntax: assert <logical condition> [, <message>]

Assertions

- Example:

  ```
  assert num_films < 1000, 'The table already contains 1000 movie records';
  ```

- Returned output:

  ```
  MISTAKE: The table already contains 1000 movie records

  CONTEXT: PL/pgSQL function inline_code_block line 9 at ASSERT

  SQL state: P0004
  ```

Control structures

- Conditionals:
  - if – then
  - if – then – else
  - if – then – elsif
- Multiple statement conditional:
  - case – when

Control structures

- Simple conditional example:
-
  ```
  SELECT * FROM film
  INTO selected_film
  WHERE title = 'Academy Dinosaur';

  if not found then
      raise notice 'The movie could not be found';
  else
     raise notice 'Film Year: %', selected_film.release_year;
  end if;
  ```

Multiple conditional

- It is used to evaluate several conditions in a single control structure
- Syntax:

```
case <expression>
  when <expression1> [,<expression2>, ...] then
     <instructions>
[ ... ]
[ else
   <instructions> ]
end case;
```

Multiple conditional

- Example:

```
su $$
declare
    ratefilm.rental_rate%type;
    price_segment varchar;
begin
    SELECT rental_rate INTO rate
    FROM film
    WHERE title = 'Academy Dinosaur';
...
```

Multiple conditional

```
...
case rate
    when 0.99 then
        price_segment = 'Sale';
    when 2.99 then
        price_segment = 'Average';
    when 4.99 then
        price_segment = 'Premium';
    else
        price_segment = 'Uncategorized';
end case;
raise notice 'Price segment: %', price_segment;
```

Looping structures

- There are structures for iterating
- They are used to repeat sequences of instructions
- To get out of a loop, the *exit* keyword can be used
- It is also possible to jump to the next iteration by using *continue*

Looping structures

Looping statements:

- *loop* → termination is done via *exit* or *break*
- *while* → repeat a block of statements as long as a given condition is met
- *for* → typically used to go through a collection of values

Looping structures

- Example:

```
-- initially: i is 0, j is 1, counter is 0
loop
    exit when counter = n;
    counter := counter + 1;
    SELECT j, i + j INTO i, j;
end loop;
```

Looping structures

- Example:

```
while counter < n loop
    raise notice 'Counter value: %', counter;
    counter := counter + 1;
end loop;
```

Looping structures

- Example:

```
for counter in 1..10 loop
    raise notice 'Counter: %', counter;
end loop;
for counter in reverse 10..1 by 2 loop
    raise notice 'Counter: %', counter;
end loop;
```

Looping structures

- Example with a query result:

```
for film in SELECT title, length FROM film
loop
    raise notice 'The duration of the movie % is % minutes', f.title, f.length;
end loop;
```

Stored procedures

- With PL/SQL code, we can create functions
- Functions can have parameters as well as return a result
- Drawback: they cannot manipulate transactions (complex sequences of operations)

Stored procedures

- Example:

```
create function get_film_count(len_from int, len_to int)

returns int

language plpgsql

as

...
```

Stored procedures

```
...
$$ declare
    num_films integer;
begin
    SELECT count(id)
    INTO num_films
    FROM film
    WHERE length BETWEEN len_from AND len_to;
    return film_count;
end; $$;
```

Stored procedures

- To make up for the functions drawback as for the transactions, we can leverage of stored procedures
- The syntax is very similar to that of functions, but in this case it is not required to return a result
- The number of parameters can be zero as well

Stored procedures

- Example:

```
drop table if exist accounts;
create table accounts(
    id int generated by default as identity,
    namevarchar(100) not null,
    balancedec(15,2) not null,
    primary key(id)
);
insert into accounts(name,balance) values('Bob', 10000);
insert into accounts(name,balance) values('Alice', 10000);
```

Stored procedures

- Example:

```
create or replace procedure transfer(
    send integer,
    receiver integer,
    amount decimal
)
language plpgsql
as
...
```

Stored procedures

```
...
$$ start
    UPDATE accounts
    SET balance = balance - amount
    WHERE id = sender;

    UPDATE accounts
    SET balance = balance + amount
    WHERE id = receiver;

    commit;   -- this command confirms the changes made
end; $$
```

Stored procedures

- The operation in the above procedure cannot be performed with a function
- The changes will be transient as long as they are not confirmed by the *commit* statement
- To invoke the previous procedure, the *call* instruction must be used
- For example: *call transfer(1,2,1000);*
- In order to drop a procedure (like a table, view, index, ...) the statement: DROP PROCEDURE [IF EXISTS] is used

Cursors

- What are they? → a mechanism to “move” through the records that a query has returned as a result
- Statement:

```
declare
    cur_films cursor (year integer) for
    SELECT *
    FROM film
    WHERE release_year = year;
```

Cursors

- Operating mode → opening and use
- Example:

```
open cur_films(year := 2005);
fetch cur_films into row_film;
fetch last from row_film into title, release_year;
close cur_films;
```

Cursors

- Note that the *fetch* instruction is used to move through the records
- There are pointers to collect the data of the first record, the last, the previous, the next, ...
- It is also possible to make the cursor move forward or backwards
- Each record fetched with the cursor can be read, updated or deleted

Triggers

- They are arguably the most interesting resource from the PL/SQL language
- What are they? → blocks of code very similar to stored procedures
- Difference → they are not explicitly called by the user; instead, they are executed when a specific situation is detected in the database (they are "triggered")

Triggers

- Basic syntax:

```
CREATE FUNCTION trigger_function()
RETURNS TRIGGER
LANGUAGEPLPGSQL
AS
$$ BEGIN
-- sequence of instructions
END; $$
```

Triggers

- Basic syntax:

```
CREATE TRIGGER <trigger name>
{BEFORE | AFTER} { event }
ON <table>
[FOR [EACH] { ROW | STATEMENT }]
EXECUTE PROCEDURE trigger_function
```

Triggers

- Example:

```
CREATE TABLE employees(
    id INT PRIMARY KEY,
    first_name VARCHAR(40) NOT NULL,
    last_name VARCHAR(40) NOT NULL
);
```

Triggers

- Example:

```
CREATE TABLE employee_audits (
   id INT PRIMARY KEY,
   employee_id INT NOT NULL,
   last_name VARCHAR(40) NOT NULL,
   changed_on TIMESTAMP(6) NOT NULL
);
```

Triggers

- Example:

```
CREATE OR REPLACE FUNCTION log_last_name_changes()

RETURNS TRIGGER

LANGUAGE PLPGSQL

AS

…
```

Triggers

- Example:

```
...
$$ BEGIN
    IF NEW.last_name <> OLD.last_name THEN
        INSERT INTO employee_audits(employee_id, last_name, changed_on)
        VALUES(OLD.id, OLD.last_name, now());
    ENDIF;
    RETURN NEW;
END; $$
```

Triggers

- Example:

```
CREATE TRIGGER last_name_changes
BEFORE UPDATE ON employees
FOR EACH ROW
EXECUTE PROCEDURE log_last_name_changes();
```