Databases

PL/pgSQL

Bachelor in Informatics Engineering
Department of Informatics Engineering
University of Coimbra
2025/2026

Outline

- Procedures
- Functions
- Cursors

PL/pgSQL

- Loadable procedural language for the PostgreSQL database system
  - Similar to PL/SQL from Oracle
  - Concepts apply to most DBMS

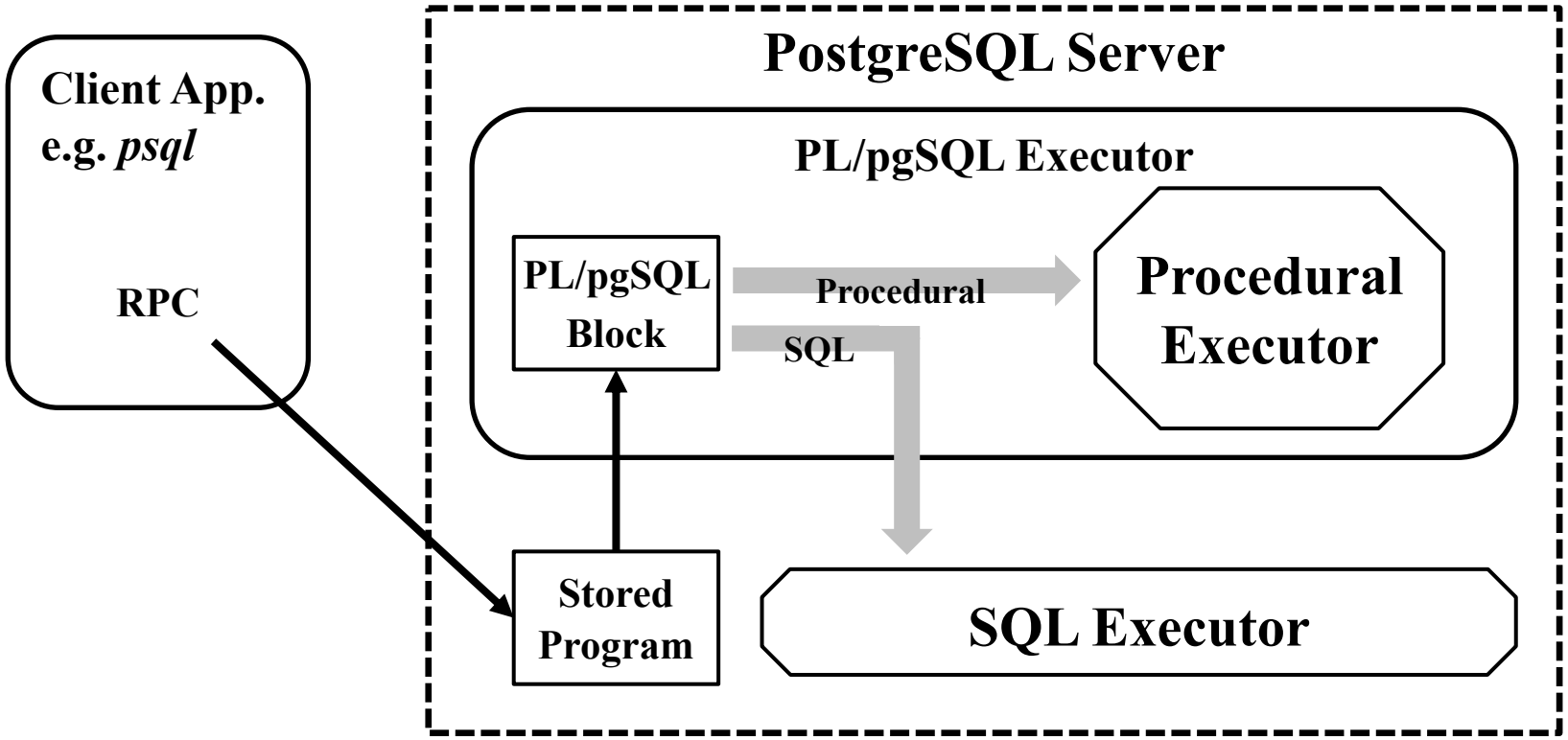
- The design goals of PL/pgSQL were to create a loadable language:
  - Can be used to create functions, procedures, and triggers
  - Adds control structures to the SQL language
  - Can perform complex computations
  - Inherits all user-defined types, functions, procedures, and operators from SQL
  - Can be defined to be trusted by the server
  - Is easy to use!

Advantages of Using PL/pgSQL

- SQL is the language PostgreSQL and most other relational databases use as query language
  - But every SQL statement must be executed individually by the database server
- Client must send each query, wait for it to be processed, receive the results, do some computation, then send further queries to the server
  - This incurs communication and network overhead if the client and the database server are on different machines

- PL/pgSQL can be used to group a block of computation and a series of queries inside the database server
  - Having the power of a procedural language and the ease of use of SQL

Procedures & Functions



call add_student('António Silva', 23212, ...);

Procedures

```
create [or replace] procedure procname ([arg1[, arg2,...]])
language plpgsql
as $$
declare
    -- declarations
begin
    -- actions
[exception
      -- exception handling]
end;
```

- Arguments:
  - [*argmode*] [*argname*] *argtype* [{ DEFAULT | = } *default_expr*]
- *argmode* can be:
  - IN: input arguments (default)
  - OUT: output arguments
  - INOUT: input/output arguments
  - VARIADIC: to accept a variable numbers of arguments (arrays only)

Procedures: Example

```
create or replace procedure proc1(a integer)
language plpgsql
as $$
<<proc1>>
begin
  insert into mytab values (a);
exception
  when others then
    raise exception 'error';
end;
$$;
```

```
call proc1(5);
```

```
drop procedure proc1;
```

Create and Execute & Functions

```
create [or replace] function funcname ([arg1[, arg2,...]])
returns datatype
language plpgsql
as $$
declare
    -- declarations
begin
    -- actions
    return value;
[exception
      -- exception handling]
end;
```

- Result is returned using the RETURN instruction

Functions: Example

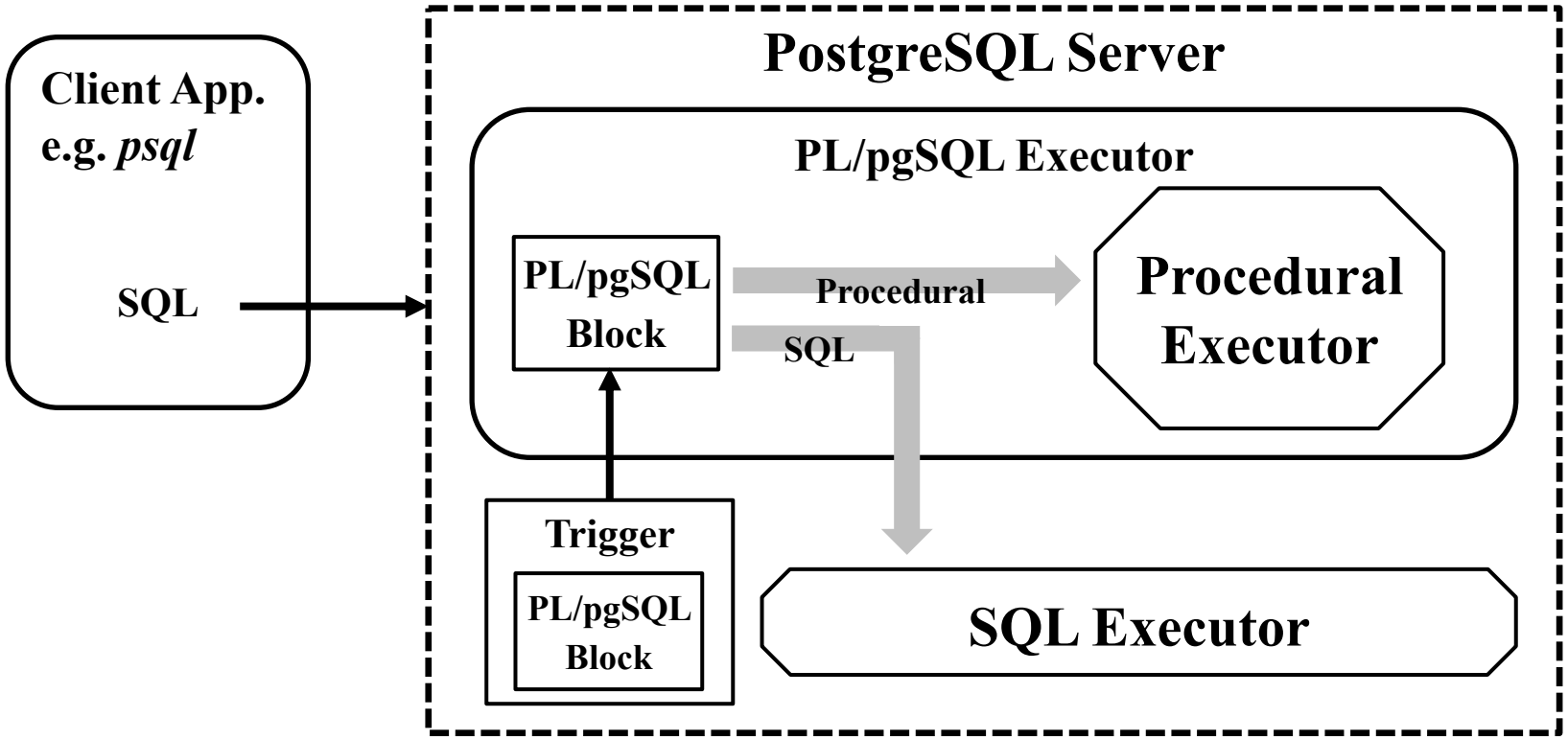
```
create or replace function func1() returns integer
language plpgsql
as $$
declare
  x integer;
begin
  select sum(col) into x from mytab;
  return(x);
exception
  when others then
    raise exception 'error';
end;
$$;
```

```
select func1();
```

```
var:=func1()
```

```
drop function func1();
```

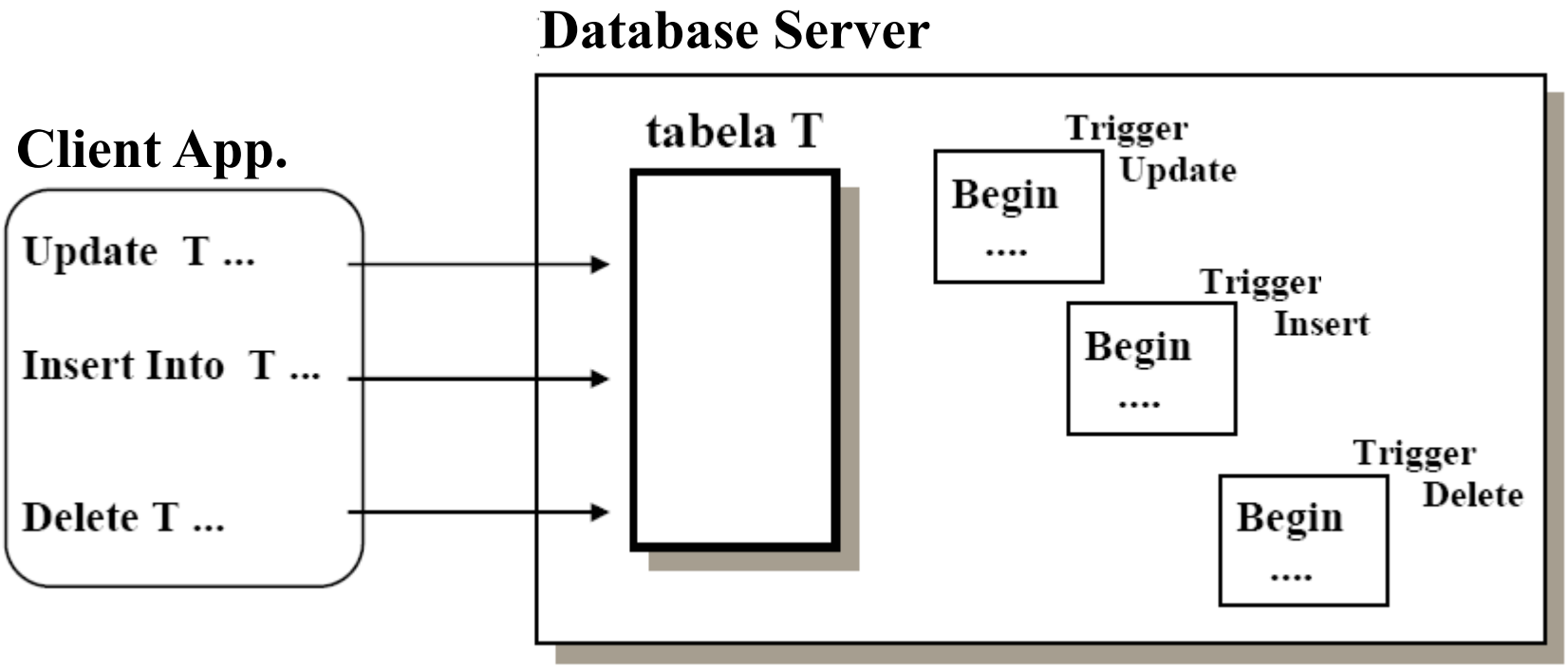
Triggers



insert into students values('António Silva', 23212, …);

Triggers: The Concept

- Program that runs automatically when some event occurs
  - e.g., insert, update and delete on table



Why Triggers? Examples…

- Implement advanced security mechanisms
- Implement complex integrity restrictions
- Allow the logging of all the operations performed
- Support auditing services (review of operations)
- Keep synchronous replicas
- Automatic computation of derived attributes
- Avoid invalid transactions
- Collect statistics
- …

Trigger Definitions

- Activation event
- Activation instant
- Trigger type
- Trigger restriction
- Trigger action

Creating a Trigger

- Instant: before, after, instead of
- Event: insert, update [of…],
  delete, truncate
- Type: for each row, for each statement
- Restriction: when
- Action: function to be executed when the trigger is activated

```
create trigger name
{ before | after | instead of } { event [ or ... ] }
on table
[ for [ each ] { row | statement } ]
[ when ( condition ) ]
execute procedure proc_name ( arguments );
```

Activation Instant

- BEFORE
  - The trigger is executed immediately before the execution of the command that activates the trigger
- AFTER
  - The trigger is executed immediately after the execution of the command that activates the trigger

- It is possible to have one trigger being executed before and another one after

Activation Event

- Event or SQL command SQL that activates the trigger to execute
- It can be an INSERT, UPDATE or DELETE in the table to which the trigger is associated
  - There are other event-based triggers (not seen in the course)
- Several events can be combined for the same trigger

```
create trigger t1
before update of sal on emp
...
```

```
create trigger t2
after insert on emp
...
```

```
create trigger t3
after insert or update or delete on emp
...
```

Trigger Type

- Row:
  - Defined using the clause FOR EACH ROW
  - Activated for each line affected by the command that activates the trigger
    - e.g., if the trigger is activated by an update that changes 20 lines of a table, then the trigger will be executed 20 times (or less, depending on the restriction)

- Statement:
  - Defined using the clause FOR EACH STATEMENT
  - Executed only once, independently of the number of rows affected by the command that activates the trigger

Row Triggers

- Row triggers have access to the old and new values of the line whose manipulation activated the trigger
  - **new**.*column* – new value (only for triggers activated by insert and update)
  - **old**.*column* – old value (only for triggers activated by update and delete)

- Row triggers fired BEFORE
  - can return NULL to skip the rest of the operation for that row (e.g., subsequent triggers are not fired; INSERT/UPDATE/DELETE does not occur)
  - for INSERT/UPDATE must return NEW to continue execution normally
  - for DELETE must return a non-NULL to continue normally
  - it is possible to alter the values directly in NEW to change what is stored

Instant/Type

- before each statement
  - Executed only once before the command that activates the trigger
- before each row
  - Executed before changing each line affected by the command that activates the trigger
- after each statement
  - Executed only once after the command that activates the trigger
- after each row
  - Executed after changing each line affected by the command that activates the trigger

Trigger Action

- Function that should be executed when the trigger is activated
  - Such function returns trigger
- Triggers can receive arguments (string literals) but do not have parameters specified
  - they are available at TG_NARG (number of arguments) and TG_ARGV[] (an array of arguments)
- The event that activated the trigger can be identified using the variable TG_OP

```
...
if (tg_op='DELETE') then
   insert into mytab values(old.nemp);
elsif (tg_op='INSERT') then
   insert into mytab values(new.nemp);
...
```

Example

```
create or replace function func_trig1() returns trigger
language plpgsql
as $$
begin
  insert into mytab values(new.ndep);
  return new;
end;
$$;
```

```
create trigger trig1
after insert on dep
for each row
execute procedure func_trig1();
```

```
insert into dep values(50,'Sales','Coimbra');
```

```
drop trigger trig1 on dep;
```

Declaring Variables

var_name [**constant**] *data_type* [**not null**] [:= *default_expr*]

```
declare
   v_sal numeric(7,2);
   v_account numeric not null = 0; (= works)
   married boolean := false; (:= is PLSQL compliant)
   annual_sal numeric(7,2) := salary * 14;
   v_last_name varchar(20);
   v_deptno dep.ndep%type;
begin
   ...
end;
```

Cursors

- We need a working area to temporarily store the output of SQL instructions, in order to be able to process that output
- PL/pgSQL construction that allows assigning a name to that working area and access the data stored

- Two *types* of cursors :
  - Implicit cursors
  - Explicit cursors

Declare Cursor

- Assigns a name to the cursor
- Define the SELECT to be execute
- It is possible to use arguments

declare
cursor_name **cursor** [(arguments)] **for**
select ...

```
declare
    c1 cursor for
       select nome, nemp
       from emp
       where sal > 2500;
    ...
```

Cursors and Records

- It is possible to declare variables of the type record and %rowtype

```
declare
   c1 cursor for
      select nome, nemp
      from emp
      where sal > 2500;
   v_cur record;
begin
   open c1;
   ...
   fetch c1
   into v_cur;
   ...
   insert into mytab
   values (v_cur.nemp);
   ...
```

```
declare
   c1 cursor for
      select *
      from emp
      where sal > 2500;
   v_cur emp%rowtype;
begin
   open c1;
   ...
   fetch c1
   into v_cur;
   ...
   insert into mytab
   values (v_cur.nemp);
   ...
```

FOR Loops and Cursors

- FOR loops automatically do:
  - Open cursor
  - FETCH one row in each interaction
  - Exit when all rows are processed
  - Close cursor

```
for var in cursor_name[(arguments)]
loop
    -- actions
    -- var is a record variable
end loop;
```

- The control variable is automatically declared

FOR Loops and Cursors: Example

```
do $$
declare
    c1 cursor for
        select * from emp
        where funcao = 'Vendedor' or sal > 2000
        for update;
        new_sal numeric;
begin
    for r in c1
    loop
        new_sal := (r.sal - 200) + r.sal*1.05;
        update emp set sal = new_sal
        where current of c1;
    end loop;
    commit;
end;
$$;
```

Logging

```
do $$
declare
        max_value INT;
begin
        SELECT MAX(nemp) INTO max_value FROM emp;
        RAISE NOTICE 'max emp: %' , max_value;
end;
$$;
```

Q&A



Databases

PL/pgSQL

Bachelor in Informatics Engineering
Department of Informatics Engineering
University of Coimbra
2025/2026