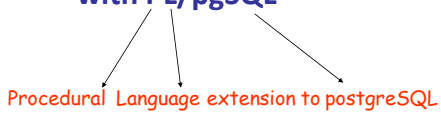
Programming in postgreSQL with PL/pgSQL



Why a Programming Language?

- Some calculations cannot be made within a query (examples?)
- Two options:
  - Write a program within the database to calculate the solution
  - Write a program that communicates with the database and calculates the solution
- Both options are useful, depending on the circumstances.
  - Option 1 reduces the communication need, and can be faster!

PL/pgSQL

- Specific for Postgres (similar languages available for other db systems)
- Allows using general programming tools with SQL, for example: loops, conditions, functions, etc.
  - This allows a lot more freedom than general SQL
- We write PL/pgSQL code in a regular file, for example firstPl.sql, and load it with \i in the psql console.
- Documentation available at: http://www.postgresql.org/docs/8.1/static/plpgsql.html#PLPGSQL-OVERVIEW

BASIC STRUCTURE OF A PL/PGSQL PROGRAM

PL/pgSQL Blocks

PL/pgSQL code is built of Blocks, with a unique structure:

```
DECLARE      (optional)
   /* All Variables Declared Here*/
BEGIN        (mandatory)
   /* Executable statements (what the block
      DOES!)*/
EXCEPTION    (optional)
   /* Exception handling*/
END;         (mandatory)
```


Creating a Function

```
CREATE OR REPLACE FUNCTION
   funcName(varName1 varType1,…)
RETURNS returnVarType AS '
DECLARE      (optional)
   /* All Variables Declared Here*/
BEGIN        (mandatory)
   /* Executable statements (what the block
      DOES!)*/
EXCEPTION    (optional)
   /* Exception handling*/
END;         (mandatory)
' language plpgsql
```

Example

```
Create or replace function
   myMultiplication(var1 integer, var2 integer)  returns
integer as ‘
BEGIN
    return var1*var2;
END;
’ language plpgsql
```


The Function Body String

- The body of the function is a string, from the standpoint of the db
- We can use quotes to create this string, or use dollar string encoding (will be used from now on in example)

```
Create or replace function
   myMultiplication(var1 integer, var2 integer)
   returns integer as $$
BEGIN
return var1*var2;
END;
$$ language plpgsql
```


The Return Value

- If the function returns a single parameter, you can use the return syntax below
- Must use a return statement to return the value

```
Create or replace function myMultiplication(var1
integer, var2 integer) returns integer as $$
BEGIN
return var1*var2;
END;
$$ language plpgsql
```

- Functions can also return multiple values (details omitted)

Calling Functions

first.sql:

```
Create or replace function
  addTax(price real) returns real as $$
begin
   Return  price*1.155;
end;
$$language plpgsql;
```

In the psql console write: \i first.sql

Then you can call the function using, e.g., :

```
Insert into pricesTable values(addTax(20));

Select (addTax(price)) from catalog;

Perform  addTax(20);
```


DECLARING VARIABLES

Defining Variables (1)

- All variables ***must be defined in the declare section***.
- The general syntax of a variable declaration is:

name [CONSTANT] *type* [NOT NULL]
[{DEFAULT | := } *expression*]

Examples:

```
user_id integer;
name CONSTANT integer := 10;
name CONSTANT integer DEFAULT 10;
url varchar NOT NULL := ‘http://www.abc.com’;
```

Declaring Variables (2): The %TYPE Attribute

- Examples

```
DECLARE
  sname          Sailors.sname%TYPE;
  fav_boat       VARCHAR(30);
  my_fav_boat    fav_boat%TYPE := 'Pinta';
```


Declaring Variables (3): The %ROWTYPE Attribute

- Declare a variable with the type of a ROW of a table.

```
reserves_record   Reserves%ROWTYPE;
```

- And how do we access the fields in reserves_record?

```
reserves_record.sid := 9;
Reserver_record.bid := 877;
```


Declaring Variables (4): Records

- A *record* is similar to row-type, but we don't have to predefine its structure

```
unknownRec record;
```


COMMON OPERATIONS WITHIN FUNCTION BODY

Some Common Operations

- In this part we discuss:
  - Using the result of a query within a function
  - Conditionals (if/then/else)
  - Loops
  - Exceptions

Select Into

- We will often wish to run a query, and take a query result, store it in a variable, and perform further calculations
- Storing the result in a variable is done using the *Select Into* command
- Note in the following slides what happens when applied to queries that return multiple rows

Select Into

```
Create or replace function
   sillyFunc(var1 integer) returns integer as $$
DECLARE
  s_var sailors%rowtype;
BEGIN
  select * into s_var from sailors;
  return s_var.age*var1;
END;
 $$language plpgsql
```

1. If select returns more than one result, the first row will be put into sp_var
2. If no rows were returned, nulls will be put in sp_var

Notice that unless 'Order by' was specified, the first row is not well defined

Select Into Strict

```
Create or replace function
  sillyFunc(var1 integer) returns integer as $$
DECLARE
  s_var sailors%rowtype;
BEGIN
  select * into strict s_var from sailors;
  return sp_var.age*var1;
END;
 $$language plpgsql
```

- In this case, if more or less than one row is returned, a run-time error will occur

Using Records in Select Into

```
DECLARE
v record;
BEGIN
  select * into v
  from Sailors S, Reserves R
  where S.sname='Sam' and S.sid = R.sid
END;
```


Checking if a Row was Returned By Select Into

```
Declare
      v record;
Begin
Select * into v from Sailors where age=4;
If not found then…
```


Conditioning

IF *boolean-expression*
THEN *statements*
END IF;

```
…
IF v_age > 22
THEN
   UPDATE employees
   SET salary = salary+1000
   WHERE eid = v_sid;
END IF;
…
```

Assume variables in blue were defined above the code fragment

More Conditioning

```
IF boolean-expression
    THEN statements
ELSIF boolean-expression
    THEN statements
ELSIF boolean-expression
     THEN statements
 …
 ELSE statements
 END IF ;
```

Example

```
CREATE or replace FUNCTION
    assessRate(rating real) RETURNS text AS $$
BEGIN
        if rating>9 then return 'great';
        elsif rating>7 then return 'good';
        elsif rating>5 then return 'keep on working';
        elsif rating>3 then return 'work harder!';
        else return 'you are hopeless';
        end if;
END;
$$ LANGUAGE plpgsql;
```

Select assessRate(6.7);

Another Example

- Write a function that when called by a user:
  - if user is already in table mylog, increment num_run.
  - Otherwise, insert user into table

mylog

who	num_run
Peter	3
John	4
Moshe	2


```
CREATE FUNCTION
   updateLogged() RETURNS void AS $$
DECLARE
   cnt integer;
BEGIN
   Select count(*) into cnt
   from mylog where who=user;
   If cnt>0 then
      update mylog
      set num_run = num_run + 1
      where who = user;
   else
      insert into mylog values(user, 1);
   end if;
end;
$$ LANGUAGE plpgsql;
```


Simple loop

LOOP
statements
END LOOP;

- Terminated by Exit or return
- Exit: only causes termination of the loop
- Can be specified with a condition: Exit when …

Examples

```
LOOP
-- some computations
IF count > 0 THEN EXIT;
END IF;
END LOOP;
```

```
LOOP
-- some computations
EXIT WHEN count > 0;
 END LOOP;
```


Continue

- The next iteration of the loop is begun

```
Create or replace function
myTest(var1 integer) returns integer as $$
DECLARE
   i integer;
BEGIN
   i:=1;
   loop
     exit when i>var1;
     i=i+1;
    continue when i<20;
    raise notice 'num is %',i;
  end loop;
  return i*var1;
END
 $$language plpgsql
```

What does this print for myTest(30)?

While loop

WHILE *expression*
LOOP
--*statements*
END LOOP ;

```
WHILE money_amount > 0 AND happiness < 9
LOOP
-- buy more
 END LOOP;
```


For loop

FOR *var* IN [REVERSE] *stRange* ..endRange
LOOP
statements
END LOOP;

```
FOR i IN 1..10 LOOP
    RAISE NOTICE 'i is %', i;
END LOOP;
```

The variable *var* is not declared in the declare section for this type of loop.

```
FOR i IN REVERSE 10..1 LOOP
-- some computations here
END LOOP;
```


Looping Through Query Results

FOR *target* IN *query*
LOOP
statements
END LOOP;

```
CREATE or replace FUNCTION
assessRates() RETURNS void AS $$
DECLARE
  i record;
BEGIN
  For i in select rating from ratings order by rating loop
     if i.rating>9 then raise notice 'great';
     elsif i.rating>7 then raise notice 'good';
     elsif i.rating>5 then raise notice 'keep on working';
     elsif i.rating>3 then raise notice 'work harder!';
     else raise notice 'you are hopeless';
     end if;
  end loop;
END; $$ LANGUAGE plpgsql;
```


Trapping exceptions

DECLARE
declarations
BEGIN
statements
EXCEPTION
WHEN *condition* [OR *condition* ...] THEN
  handler_statements
WHEN *condition* [OR *condition* ...] THEN
  handler_statements
...
END;

See http://www.postgresql.org/docs/8.1/static/errcodes-appendix.html for a list of all exceptions

Exception Example

```
Create or replace function
errors(val integer) returns real as $$
Declare
  val2 real;
BEGIN
  val2:=val/(val-1);
  return val2;
Exception
  when division_by_zero then
  raise notice 'caught a zero division';
  return 0;
End;
$$ LANGUAGE plpgsql;
```


Triggers

Triggers

- A trigger defines an action we want to take place whenever some event has occurred.
- When defining a trigger, you have to define:
  1. Triggering Event
  2. Trigger Timing
  3. Trigger Level

Triggering Event

- When defining a trigger, you must choose an event (or events) upon which you want the trigger to be *automatically* called
- Possible events:
  - Update of a specific table
  - Insert into a specific table
  - Delete from a specific table
- For example, if you define a trigger on inserting into table R, then whenever an insert is performed your trigger will be called by the database!

Trigger Timing

- Triggers run when a predefined event has occurred.
- The trigger can run before or after the event
- For example, if you define a trigger *before insert on R*, then after the user calls an insert command on R, but before it has been executed, your trigger will be called

Trigger Level

- Triggers run when a predefined event has occurred.
- The trigger level determines the number of times that the trigger will run.
- If the trigger level is statement, then the trigger will run once for the triggering event
- If the trigger level is row, then the trigger will run once for each row changed by the triggering event
- For example, a statement level trigger, defined upon delete will run once, for each delete statement. A row level trigger will run once for each row deleted by the delete statement

Defining Triggers

- There are two parts to defining a trigger:
  1. Writing a trigger function, i.e., a function with return type trigger
  2. Calling create trigger, defining triggering events, trigger timing and level, and using the trigger function

- We first explain #2, and then #1

Create Trigger: Timing

CREATE TRIGGER *name* { BEFORE | AFTER }
{ *event* [OR ...] } ON *table*
[FOR EACH ROW |STATEMENT]
EXECUTE PROCEDURE *funcname* (*arguments*)

```
CREATE TRIGGER emp_trig
BEFORE INSERT OR UPDATE ON employee
FOR EACH ROW
EXECUTE PROCEDURE emp_trig_func ();
```

Create Trigger: Triggering Event

CREATE TRIGGER *name* { BEFORE | AFTER }
{ *event* [OR ...] } ON *table*
[FOR EACH ROW |STATEMENT]
EXECUTE PROCEDURE *funcname* (*arguments*)

```
CREATE TRIGGER emp_trig
BEFORE INSERT OR UPDATE ON employee
FOR EACH ROW
EXECUTE PROCEDURE emp_trig_func ();
```


Create Trigger: Trigger Level

CREATE TRIGGER *name* { BEFORE | AFTER }
{ *event* [OR ...] } ON *table*
[FOR EACH ROW |STATEMENT]
EXECUTE PROCEDURE *funcname* (*arguments*)

```
CREATE TRIGGER emp_trig
BEFORE INSERT OR UPDATE ON employee
FOR EACH ROW
EXECUTE PROCEDURE emp_trig_func ();
```


Writing a Trigger Function

```
CREATE FUNCTION funcName() RETURNS trigger AS $$
```

- There are several variables automatically available for the trigger function:
  - **New:** Available for row level triggers, defined upon insert or update. Is a record containing the new values for the row
  - **Old:** Available for row level triggers, defined upon delete or update. Is a record containing the old values for the row
  - **TG_OP:** Name of the operation which caused the trigger
  - …

Example

```
CREATE FUNCTION toUpper() RETURNS trigger AS $$
BEGIN
   new.sname := UPPER(new.sname);
  return new;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER toUpperTrig
BEFORE INSERT or UPDATE  on Sailors
FOR EACH ROW execute procedure toUpper();
```


Important!
Row Level Triggers, BEFORE

- A return value of null signals to the trigger manager to skip the rest of the operation for this row
  - subsequent triggers are not fired for this row
  - the INSERT/UPDATE does not occur for this row.
- A return value that is non-null causes the operation to proceed with that row value.
  - Returning a row value different from the original value of NEW alters the row that will be inserted or updated (but has no direct effect in the DELETE case).

Important!
All Other Types of Triggers

- The return value of a BEFORE or AFTER statement-level trigger or an AFTER row-level trigger is always ignored;
  - it may as well be null.
- However, any of these types of triggers can still abort the entire operation by raising an error.

Another Example

```
CREATE TABLE emp (
   empname text,
   salary integer,
   last_date timestamp,
   last_user text );
```

49

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

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

50

Another Example: Backing Up Information

```
CREATE TABLE emp (
   empname text NOT NULL,
   salary integer );
```

```
CREATE TABLE emp_backup(
   operation char(1) NOT NULL,
   stamp timestamp NOT NULL,
   userid text NOT NULL,
   empname text NOT NULL,
   salary integer );
```

51

```
CREATE OR REPLACE FUNCTION process_emp_backup() RETURNS
    TRIGGER AS $$
BEGIN
IF (TG_OP = 'DELETE') THEN
  INSERT INTO emp_backup
  SELECT 'D', current_timestamp, current_user, OLD.*;
  RETURN null;
ELSIF (TG_OP = 'UPDATE') THEN
  INSERT INTO emp_backup
  SELECT 'U', current_timestamp, current_user, NEW.*;
  RETURN null;
ELSIF (TG_OP = 'INSERT') THEN
  INSERT INTO emp_backup
  SELECT 'I', current_timestamp, current_user, NEW.*;
  RETURN null;
END IF;
RETURN NULL;
END; $$ LANGUAGE plpgsql;
```

52

Example (cont)

```
CREATE TRIGGER emp_backup
AFTER INSERT OR UPDATE OR DELETE ON emp FOR EACH
ROW
EXECUTE PROCEDURE process_emp_backup ();
```

53

Statement Trigger Example

```
CREATE FUNCTION
shabbat_trig_func() RETURNS trigger AS $$
BEGIN
if (TO_CHAR(current_date,'DY')='SAT') then
 raise exception 'no work on shabbat!';
end if;
Return null;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER no_work_on_shabbat_trig
BEFORE INSERT or DELETE or UPDATE on sailors
for each statement
execute procedure shabbat_trig_func();
```

54