PL/pgSQL

What to Expect from this Session

- What is PL/pgSQL?
- Why PL/pgSQL?
- PL/pgSQL:
  - Supported argument and result data types
  - Structure of PL/pgSQL
    - Syntax
    - Declaration
    - Basic Statements
    - Control Structures
  - Functions
  - Triggers
  - XML Support
  - Regex Support

Assumptions

General knowledge of DBMS and SQL language

What is PL/pgSQL ?

- (Procedural Language/PostgreSQL) is a loadable programming language supported by the PostgreSQL ORDBMS
- Fully featured programming language leveraging SQL but giving much more imperative control, as procedural calls
- Includes
  - the ability to use loops – for, while
  - conditionals - when, if–then-else
  - function (method) calls, including recursion
  - Error trapping, event handlers
- ANSI SQL compatible

Why PL/pgSQL ?

- Adds control structures to the SQL language.
- Can be used to create functions and trigger procedures.
- Can perform complex computations.
- Inherits all user-defined types, functions, and operators.
- Can be defined to be trusted by the server
- Extra round trips between client and server are eliminated
- Intermediate results that the client does not need do not have to be marshaled or transferred between server and client
- Multiple rounds of query parsing can be avoided

PL/pgSQL : Supported argument and result data types

- Functions written in PL/pgSQL can return or accept as arguments any scalar or array data type supported by PostgreSQL
- Also, accept or return any composite type (row type) specified by name.
- PL/pgSQL functions can be declared to accept a variable number of arguments by using the VARIADIC marker.
- PL/pgSQL functions can also be declared to accept and return the polymorphic types any element, any array.
- PL/pgSQL functions can also be declared to return a "set" (or table) of any data type that can be returned as a single instance.
- A PL/pgSQL function can be declared to return void if it has no useful return value.

PL/pgSQL : Structure

- block-structured language
- each statement within a block terminated by a semicolon
- A block that appears within another block must have a semicolon after END
- The final END that concludes a function body does not require a semicolon
- All keywords are case-insensitive and identifiers are implicitly converted to lower case unless double-quoted

PL/pgSQL : Syntax

```
[ <<label>> ]
    [ DECLARE
                declarations ]
    BEGIN
        statements…
    END [ label ];
```

PL/pgSQL : Declaration

Examples:

```
roll_no integer;
qty numeric(5);
description varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
qty integer DEFAULT 1;
roll_no CONSTANT integer := 10;
url varchar := 'http://example.com';
```

PL/pgSQL : Basic Statements

Anything not recognized as basic statement is considered an SQL command and sent to the database engine to execute.

Examples

```
variable:= expression;
    tax:= subtotal * 0.04;
    my_record.user_id := 30;
```

PL/pgSQL : Basic Statements

perform -- executes a call with no return result

- some SQL commands do not return rows, for example, INSERT without a RETURNING clause.

```
CREATE OR REPLACE FUNCTION test() RETURNS void AS $$
INSERT INTO mytable VALUES (30),(50)
$$ LANGUAGE sql;

PERFORM test();
```

PL/pgSQL : Basic Statements

into

- The result of a SQL command yielding a single row (possibly of multiple columns) can be assigned to a record variable, row-type variable, or list of scalar variables. This is done by writing the base SQL command and adding an INTO clause.

SELECT select_expressions INTO [STRICT] target FROM ... ;

Select col1, col2 into var1, var2 from table….

- where a target can be a record variable, a row variable, or a comma-separated list of simple variables and record/row fields.

PL/pgSQL : Basic Statements

execute -- Executing Dynamic Commands

- It can be useful to generate dynamic commands inside PL/pgSQL functions, that is, commands that will involve different tables or different data types each time they are executed. To handle this sort of problem, the EXECUTE statement is provided:

EXECUTE command-string [INTO [STRICT] target] [USING expression [, ...]];

- where command-string yields a text expression containing the command to be executed. The optional USING expressions supply values to be inserted into the command.

EXECUTE 'SELECT count(*) FROM employees WHERE manager_id<>0' into found_employee;

EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = '||checked_user||'AND inserted <= '||checked_date||' INTO c;

PL/pgSQL : Basic Statements

execute -- cont'd

Quoting Values In Dynamic Queries (execute statements)

When working with dynamic commands you will often have to handle escaping of single quotes.

```
EXECUTE 'UPDATE tbl SET ' || quote_ident(colname) || ' = ' || quote_literal(newvalue) || ' WHERE key
= ' || quote_literal(keyvalue);
```

quote_literal will always return null when called with a null argument, rendering the entire dynamic query string null. Avoid this problem by using the quote_nullable function, which works the same as quote_literal except that when called with a null argument it returns the string NULL.

```
EXECUTE 'UPDATE tbl SET ' || quote_ident(colname) || ' = ' || quote_nullable(newvalue) || ' WHERE
key = ' || quote_nullable(keyvalue);
```

PL/pgSQL : Basic Statements

execute -- cont'd

Dynamic SQL statements can also be safely constructed using the format function. For example:

```
EXECUTE format('UPDATE tbl SET %I = %L WHERE key = %L', colname, newvalue, keyvalue);
```

The format function can be used in conjunction with the USING clause:

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname) USING newvalue, keyvalue;
```

PL/pgSQL : Basic Statements

Obtaining the Result Status

There are several ways to determine the effect of a command. The first method is to use the GET DIAGNOSTICS command, which has the form:

GET DIAGNOSTICS variable = item [, ...];

GET DIAGNOSTICS integer_var = ROW_COUNT;

```
create function test_diag()
returns setof int
language plpgsql
as $$
declare
   n int;
begin
   return query select generate_series(1,5);
   get diagnostics n = row_count;
   return query select format ('%s', n)::int;
end $$;
```

PL/pgSQL : Basic Statements

DO

executes an anonymous block

```
DO $$DECLARE r record;
BEGIN
   FOR r IN SELECT table_schema, table_name FROM information_schema.tables
            WHERE table_type = 'VIEW' AND table_schema = 'public'
   LOOP
      EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' || quote_ident(r.table_name) || '
TO webuser';
   END LOOP;
END$$;
```

PL/pgSQL : Basic Statements

NULL;

Do nothing at all

```
BEGIN
                NULL;
END;

BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
                NULL; -- ignore the error
END;
```

PL/pgSQL : Basic Statements

RAISE

Use the RAISE statement to report messages and raise errors.

```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression [, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
RAISE ;
```

Allowed levels are DEBUG, LOG, INFO, NOTICE, WARNING, and EXCEPTION, with EXCEPTION being the default.

EXCEPTION raises an error

Other levels only generate messages of different priority levels.

Other priorities are reported to the client, written to the server log, or both

PL/pgSQL : Basic Statements

RAISE

Use the RAISE statement to report messages and raise errors.

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;   --the value of v_job_id will replace the % in the string

RAISE EXCEPTION 'Nonexistent ID --> %', user_id
                                        USING HINT = 'Please check your user ID';  --will abort the transaction with the given error message and hint

RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';                  --equivalent ways of setting the SQLSTATE

RAISE division_by_zero;
RAISE SQLSTATE '22012';  --condition name or SQLSTATE to be reported

RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;  --USING can be used to supply a custom error message, detail, or hint.
```

PL/pgSQL : Basic Statements

RAISE

Use the RAISE statement to report messages and raise errors.

```
do $$
begin
  raise info 'information message %', now() ;
  raise log 'log message %', now();
  raise debug 'debug message %', now();
  raise warning 'warning message %', now();
  raise notice 'notice message %', now();
end $$;
```

PL/pgSQL : Control Structures

IF

Syntax:

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

Here are three forms of IF statements:

IF … THEN … ENDIF

IF … THEN … ELSE … ENDIF

IF … THEN … ELSIF … THEN … ELSE … ENDIF

PL/pgSQL : Control Structures

CASE

Syntax:

```
CASE search-expression
      WHEN expression [, expression [ ... ]] THEN
                  statements
      [ WHEN expression [, expression [ ... ]] THEN
                  statements ... ]
      [ ELSE
                  statements ]
END CASE;
```

PL/pgSQL : Control Structures

CASE --cont'd

```
SELECT salary,
      CASE     WHEN department_id =90 THEN 'High Salary'
                        WHEN department_id =100 THEN '2nd grade salary'
      ELSE     'Low Salary'
  END
      AS salary_status
FROM employees
LIMIT 15;
```

Statements can include assignments
… THEN var := value

PL/pgSQL : Control Structures

Loops

With the LOOP, EXIT, CONTINUE, WHILE, FOR, and FOREACH statements, you can arrange for your PL/pgSQL function to repeat a series of commands.

Syntax: of a defined loop with label

```
[ <<label>> ]
LOOP
        statement;
        [...]
END LOOP;
```

PL/pgSQL : Control Structures

Loops

A label can help you to specify which loop to exit when you have nested loops.
Here is the syntax for an EXIT statement, within a LOOP:

```
[ <<label>> ]
LOOP
        statement;
        [...]
        EXIT [ label ] [ WHEN condition ];
END LOOP;
```

PL/pgSQL : Control Structures

While Loop

The WHILE loop repeats the block of statements until the specified condition becomes false. The specified condition is tested before each iteration of the statement block.

Here is t he syntax of the WHILE loop:

```
[ <<label>> ]
WHILE condition LOOP
        statement;
        [...]
END LOOP;
```

PL/pgSQL : Control Structures

For Loop

The FOR loop repeats a statement block over a specified range.

Here is the syntax of the FOR loop:
Syntax:

```
[ <<label>> ]
FOR identifier IN [ REVERSE ] expression1 .. expression2 LOOP
        statement;
        [...]
END LOOP;
```

PL/pgSQL : Control Structures

Looping Through Query Results

Using a variation of the FOR loop, you can iterate through and manipulate the results of a query.

[<<*label*>>]
FOR *target* IN *query* LOOP
        statements
END LOOP [*label*];

Where *target* is a record variable, row variable, or comma-separated list of scalar variables.

FOR mviews IN SELECT mview FROM cs_materialized_views ORDER BY sort_key LOOP
        statements
END LOOP

PL/pgSQL : Control Structures

FOREACH loop

Like a FOR loop, but iterates through the elements of an array value. The FOREACH statement to loop over an array is:

[<<*label*>>]
FOREACH *targets* IN ARRAY *expression* LOOP
    statements
END LOOP [*label*];

PL/pgSQL : Control Structures

FOREACH loop --cont'd

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
    DECLARE
            s int8 := 0;
            x int;
    BEGIN
        FOREACH x IN ARRAY $1 LOOP
            s := s + x;
    END LOOP;
    RETURN s;
END; $$ LANGUAGE plpgsql;
```

PL/pgSQL : Control Structures

ERROR Trapping (exception handling)

Any error occurring in a PL/pgSQL function aborts execution of the function, and surrounding transactions. You can trap and handle errors by using a BEGIN block with an EXCEPTION clause.

```
[ <<label>> ]
[ DECLARE
        declarations ]
BEGIN
        statements
EXCEPTION
        WHEN condition [ OR condition ... ] THEN
                handler_statements
        [ WHEN condition [ OR condition ... ] THEN
                handler_statements
                    ... ]
END;
```

PL/pgSQL : Control Structures

ERROR Trapping (exception handling)

Obtaining Information About an Error

Special variables

SQLSTATE contains the PostgreSQL error code that corresponds to the exception
SQLERRM contains the error message associated with the exception.

RAISE 'error code: % message: %', sqlstate, sqlerrm;

These variables are undefined outside exception handlers.

PL/pgSQL : Functions

allow operations within a single database function.

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS $[variable_name]$
                    DECLARE
                                        declarations;
                    BEGIN
                                < function_body >
                    RETURN { variable_name | value }
                    END;
$[variable_name]$.
LANGUAGE plpgsql.;
```

- **function-name** specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The function must contain a **return** statement.
- **RETURN** clause specifies that data type you are going to return from the function. The **return_datatype** can be a base, composite, or domain type, or can reference the type of a table column, or can be VOID.
- **function-body** contains the executable part.
- The AS keyword is used for creating a standalone function.
- **plpgsql** is the name of the language that the function is implemented in

PL/pgSQL : Functions

This function returns the total number of records in the COMPANY table

```
CREATE OR REPLACE FUNCTION totalRecords () RETURNS integer AS
$total$
        declare
                        total integer;
        BEGIN
                        SELECT count(*) into total FROM COMPANY;
        RETURN total;
        END;
$total$ LANGUAGE plpgsql;
```

When the above query is executed, the result would be −

```
testdb# CREATE FUNCTION
```

Call the function as follows:

```
testdb=# select totalRecords();
```

PL/pgSQL : Functions

```
CREATE OR REPLACE FUNCTION fnsomefunc(numtimes integer, msg text) RETURNS text AS
$$
        DECLARE
                strresult text;
        BEGIN
                        strresult := '';
                        IF numtimes > 0 THEN
                                FOR i IN 1 .. numtimes LOOP
                                        strresult := strresult || msg || E'\r\n';
                                END LOOP;
                        END IF;
                        RETURN strresult;
        END;
$$
LANGUAGE 'plpgsql' IMMUTABLE
SECURITY DEFINER
        COST 10;
```

- IMMUTABLE – output of the function can be expected to be the same if the inputs are the same. Other options are STABLE - will not change within a query given same inputs and VOLATILE - can be expected to change output even in the same query call
- SECURITY DEFINER - function runs in context (permissions) of the owner of the function.
- COST - set costs and estimated rows returned for a function. Defaults to 100 unless you change it.

PL/pgSQL : Functions

Named Parameters

```
CREATE OR REPLACE FUNCTION sum (i int, j int)
RETURNS int AS $$
        DECLARE
                sum int;
        BEGIN
                sum := i + j;
                RETURN sum;
        END;
$$ LANGUAGE plpgsql;

SELECT sum(41, 1);
sum
-----
42
(1 row)
```

PL/pgSQL : Functions

Parameter Alias

```
CREATE OR REPLACE FUNCTION sum (int, int)
RETURNS int AS $$
        DECLARE
                i ALIAS FOR $1;
                j ALIAS FOR $2;
                sum int;
        BEGIN
                sum := i + j;
                RETURN sum;
        END;
$$ LANGUAGE plpgsql;
```

```
SELECT sum(41, 1);
sum
-----
42
(1 row)
```

PL/pgSQL : Functions

Control Structures IF

```
CREATE OR REPLACE FUNCTION even (i int)
RETURNS boolean AS $$
        DECLARE
                tmp int;
        BEGIN
                tmp := i % 2;
                IF tmp = 0 THEN
                        RETURN true;
                ELSE
                         RETURN false;
                END IF;
        END;
$$ LANGUAGE plpgsql;

SELECT even(3), even(42);
even | even
------+------
 f          |         t
(1 row)
```

PL/pgSQL : Functions

Control Structures FOR LOOP

```
CREATE OR REPLACE FUNCTION factorial (i numeric)
RETURNS numeric AS $$
        DECLARE
                tmp numeric;
                result numeric;
        BEGIN
                result := 1;
                FOR tmp IN 1 .. i LOOP
                        result := result * tmp;
                END LOOP;
                RETURN result;
        END;
$$ LANGUAGE plpgsql;

SELECT factorial(42::numeric);
factorial
-----------------------------------------------------
1405006117752879898543142606244511569936384000000000
(1 row)
```

PL/pgSQL : Functions

Control Structures WHILE LOOP

```
CREATE OR REPLACE FUNCTION factorial (i numeric)
RETURNS numeric AS $$
        DECLARE
                tmp numeric;
                result numeric;
        BEGIN
                result := 1;
                 tmp := 1;
                WHILE tmp <= i LOOP
                                result := result * tmp;
                                tmp := tmp + 1;
                END LOOP;
                RETURN result;
        END;
$$ LANGUAGE plpgsql;

SELECT factorial(42::numeric);
factorial
-----------------------------------------------------
1405006117752879898543142606244511569936384000000000
(1 row)
```

PL/pgSQL : Functions

Control Structures RECURSIVE

```
CREATE OR REPLACE FUNCTION factorial (i numeric)
RETURNS numeric AS $$
        BEGIN
                IF i = 0 THEN
                        RETURN 1;
                ELSIF i = 1 THEN
                        RETURN 1;
                ELSE
                        RETURN i * factorial(i - 1);
                END IF;
        END;
$$ LANGUAGE plpgsql;

SELECT factorial(42::numeric);
factorial
-------------------------------------------------------
1405006117752879898543142606244511569936384000000000
(1 row)
```

PL/pgSQL : Functions

Control Structures PERFORM

```
CREATE OR REPLACE FUNCTION func_w_side_fx() RETURNS void AS
$$ INSERT INTO foo VALUES (41),(42) $$ LANGUAGE sql;

CREATE OR REPLACE FUNCTION dummy ()
RETURNS text AS $$
        BEGIN
                        PERFORM func_w_side_fx();
                        RETURN 'OK';
        END;
$$ LANGUAGE plpgsql;

SELECT dummy();

SELECT * FROM foo;
f1
----
41
42
(2 rows)
```

PL/pgSQL : Functions

Control Structures DYNAMIC SQL

```
CREATE OR REPLACE FUNCTION get_foo(i int)
RETURNS foo AS $$
        DECLARE
                rec RECORD;
        BEGIN
                        EXECUTE 'SELECT * FROM foo WHERE f1 = ' || i INTO rec;
                        RETURN rec;
        END;
$$ LANGUAGE plpgsql;

SELECT * FROM get_foo(42);
f1
----
42
(1 row)
```

PL/pgSQL : Functions

Control Structures CURSORS

```
CREATE OR REPLACE FUNCTION totalbalance()
RETURNS numeric AS $$
        DECLARE
                        tmp RECORD;
                        result numeric;
        BEGIN
                        result := 0.00;
                        FOR tmp IN SELECT * FROM foo LOOP
                                        result := result + tmp.f1;
                        END LOOP;
                        RETURN result;
        END;
$$ LANGUAGE plpgsql;

SELECT totalbalance();
totalbalance
--------------
83.00
(1 row)
```

PL/pgSQL : Functions

Control Structures CURSORS

```
CREATE OR REPLACE FUNCTION totalbalance(n1)
RETURNS numeric AS $$
        DECLARE
                        foo_fetch cursor (n1 numeric) for
                                select * from foo where f1=n1;
                        tmp RECORD;
                        result numeric;
                        v_n1 := n1;
        BEGIN
                        result := 0.00;
                        FOR tmp IN foo_fetch(v_n1) LOOP
                                        result := result + tmp.f1;
                        END LOOP;
                        RETURN result;
        END;
$$ LANGUAGE plpgsql;

SELECT totalbalance(83);
totalbalance
--------------
83.00
(1 row)
```

PL/pgSQL : Functions

Control Structures REFCURSORS

```
CREATE FUNCTION active_info(OUT p_queries refcursor, OUT p_locks refcursor)
AS $$
        BEGIN
                        OPEN p_queries FOR SELECT now()-query_start as runtime, pid, usename,
                        substring(query,1,50) as query
                        FROM pg_stat_activity
                        ORDER BY 1 DESC;

                        OPEN p_locks FOR SELECT l.mode, count(*) as k
                        FROM pg_locks l, pg_stat_activity a
                        WHERE a.pid = l.pid
                        AND a.usename = SESSION_USER
                        GROUP BY 1;
        END;
$$ LANGUAGE plpgsql;


SELECT active_info()
```

PL/pgSQL : Functions

Control Structures ERROR HANDLING

```
CREATE OR REPLACE FUNCTION safe_add(a integer, b integer)
RETURNS integer AS $$
        BEGIN
                RETURN a + b;
            EXCEPTION
                WHEN numeric_value_out_of_range THEN
                        -- do some important stuff
                        RETURN -1;
                WHEN OTHERS THEN
                        -- do some other important stuff
                        RETURN -1;
        END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL : Triggers

A trigger procedure is created with the CREATE FUNCTION command, declaring it as a function with no arguments and a return type of trigger.

```
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 IS NULL THEN
                                RAISE EXCEPTION '% cannot have null salary', NEW.empname;
                        END IF; -- Who works for us when they must pay for it?
                        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 PROCEDURE emp_stamp();
```

XML Support

- XML Type

```
CREATE TABLE test ( ..., data xml, ... );

INSERT INTO test(data) VALUES (XMLPARSE (DOCUMENT '<?xml
version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>'));

INSERT INTO test(data) VALUES (XMLPARSE (CONTENT
'abc<foo>bar</foo><bar>foo</bar>'));
```

XML Support

- XML Functions

```
SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');

SELECT xmlforest('abc' AS foo, 123 AS bar);

SELECT xmlcomment('hello');

SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

SELECT xml_is_well_formed('<>');


SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
            ARRAY[ARRAY['my', 'http://example.com']]);
```

Regex Functions

- Pattern matching

Operator	Description	Example
~	Matches regular expression, case sensitive	`'thomas' ~ '.*thomas.*'`
~*	Matches regular expression, case insensitive	`'thomas' ~* '.*Thomas.*'`
!~	Does not match regular expression, case sensitive	`'thomas' !~ '.*Thomas.*'`
!~*	Does not match regular expression, case insensitive	`'thomas' !~* '.*vadim.*'`

- SELECT regexp_match('foobarbequebaz', 'bar.*que');
- SELECT regexp_matches('foo', 'not there');
- SELECT foo FROM regexp_split_to_table('the quick brown fox jumps over the lazy dog', '\s+') AS foo;
- SELECT regexp_split_to_array('the quick brown fox jumps over the lazy dog', '\s+');

Questions ?

Thank You !!!