aws professional services

PL/pgSQL

March 2021

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)

https://www.postgresql.org/docs/current/static/errcodes-appendix.html

Error Code	Condition Name
Class 00 — Successful Completion	
00000	successful_completion
Class 01 — Warning	
01000	warning
0100C	dynamic_result_sets_returned
01008	implicit_zero_bit_padding
01003	null_value_eliminated_in_set_function
01007	privilege_not_granted
01006	privilege_not_revoked
01004	string_data_right_truncation
01P01	deprecated_feature
Class 02 — No Data (this is also a warning class per the SQL	
02000	no_data
02001	no_additional_dynamic_result_sets_returned
Class 03 — SQL Statement Not Yet Complete	
03000	sql_statement_not_yet_complete
Class 08 — Connection Exception	
08000	connection_exception
08003	connection_does_not_exist
08006	connection_failure
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconn
08007	transaction_resolution_unknown
08P01	protocol_violation
Class 09 — Triggered Action Exception	
09000	triggered_action_exception
Class 0A — Feature Not Supported	
0A000	feature_not_supported
Class 0B — Invalid Transaction Initiation	
0B000	invalid_transaction_initiation
Class 0F — Locator Exception	
0F000	locator_exception
0F001	invalid_locator_specification
Class 0L — Invalid Grantor	
0L000	invalid_grantor
0LP01	invalid_grant_operation
Class 0P — Invalid Role Specification	
0P000	invalid_role_specification
Class 0Z — Diagnostics Exception	
0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active
Class 20 — Case Not Found	
20000	case_not_found
Class 21 — Cardinality Violation	
21000	cardinality_violation
Class 22 — Data Exception	
22000	data_exception
2202E	array_subscript_error
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero
22005	error_in_assignment
2200B	escape_character_conflict
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
2202H	invalid_tablesample_argument
2202G	invalid_tablesample_repeat
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
2200H	sequence_generator_limit_exceeded
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid_text_representation
22P03	invalid_binary_representation
22P04	bad_copy_file_format
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid_xml_document
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
Class 23 — Integrity Constraint Violation	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Class 24 — Invalid Cursor State	
24000	invalid_cursor_state
Class 25 — Invalid Transaction State	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_trans
25004	inappropriate_isolation_level_for_branch_t
25005	no_active_sql_transaction_for_branch_trans
25006	read_only_sql_transaction
25007	schema_and_data_statement_mixing_not_suppo
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction
25P03	idle_in_transaction_session_timeout
Class 26 — Invalid SQL Statement Name	
26000	invalid_sql_statement_name
Class 27 — Triggered Data Change Violation	
27000	triggered_data_change_violation
Class 28 — Invalid Authorization Specification	
28000	invalid_authorization_specification
28P01	invalid_password
Class 2B — Dependent Privilege Descriptors Still Exist	
2B000	dependent_privilege_descriptors_still_exis
2BP01	dependent_objects_still_exist
Class 2D — Invalid Transaction Termination	
2D000	invalid_transaction_termination
Class 2F — SQL Routine Exception	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
Class 34 — Invalid Cursor Name	
34000	invalid_cursor_name
Class 38 — External Routine Exception	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Class 39 — External Routine Invocation Exception	
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
39P03	event_trigger_protocol_violated
Class 3B — Savepoint Exception	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
Class 3D — Invalid Catalog Name	
3D000	invalid_catalog_name
Class 3F — Invalid Schema Name	
3F000	invalid_schema_name
Class 40 — Transaction Rollback	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Class 42 — Syntax Error or Access Rule Violation	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation
42809	wrong_object_type
428C9	generated_always
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition
Class 44 — WITH CHECK OPTION Violation	
44000	with_check_option_violation
Class 53 — Insufficient Resources	
53000	insufficient_resources
53100	disk_full
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
Class 54 — Program Limit Exceeded	
54000	program_limit_exceeded
54001	statement_too_complex
54011	too_many_columns
54023	too_many_arguments
Class 55 — Object Not In Prerequisite State	
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
Class 57 — Operator Intervention	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
Class 57 — Operator Intervention	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped
Class 58 — System Error (errors external to PostgreSQL itself)	
58000	system_error
58030	io_error
58P01	undefined_file
58P02	duplicate_file
Class 72 — Snapshot Failure	
72000	snapshot_too_old
Class F0 — Configuration File Error	
F0000	config_file_error
F0001	lock_file_exists
Class HV — Foreign Data Wrapper Error (SQL/MED)	
HV000	fdw_error
HV005	fdw_column_name_not_found
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply
HV00N	fdw_unable_to_establish_connection
Class P0 — PL/pgSQL Error	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure
Class XX — Internal Error	
XX000	internal_error
XX001	data_corrupted
XX002	index_corrupted

aws professional services

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)
```


aws professional services

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 ?

aws professional services

Thank You !!!

aws professional services