Session 4

Views

Functions

Stored Procedures

sourcemind

Views

- Views are pseudo-tables. That is, they are not real tables; nevertheless appear as ordinary tables to SELECT. A view can represent a subset of a real table, selecting certain columns or certain rows from an ordinary table. A view can even represent joined tables.
- A view can contain all rows of a table or selected rows from one or more tables. A view can be created from one or many tables, which depends on the written PostgreSQL query to create a view.

Create a view

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

This will create a view containing the columns that are in the film table at the time of view creation. Though * was used to create the view, columns added later to the table will not be part of the view.

Views usage

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data such that a user can only see limited data instead of complete table.
- Summarize data from various tables, which can be used to generate reports

Example

Consider, the COMPANY table is having the following records

```
 id | name  | age | address    | salary
----+-------+-----+------------+--------
  1 | Paul  |  32 | California |  20000
  2 | Allen |  25 | Texas      |  15000
  3 | Teddy |  23 | Norway     |  20000
  4 | Mark  |  25 | Rich-Mond  |  65000
  5 | David |  27 | Texas      |  85000
  6 | Kim   |  22 | South-Hall |  45000
  7 | James |  24 | Houston    |  10000
```

```
CREATE VIEW COMPANY_VIEW AS
SELECT ID, NAME, AGE
FROM  COMPANY;
```

```
SELECT * FROM COMPANY_VIEW;
```

```
 id | name  | age
----+-------+-----
  1 | Paul  |  32
  2 | Allen |  25
  3 | Teddy |  23
  4 | Mark  |  25
  5 | David |  27
  6 | Kim   |  22
  7 | James |  24
(7 rows)
```

DROP VIEW

```
DROP VIEW view_name;

DROP VIEW COMPANY_VIEW;
```

Procedural Languages

PostgreSQL allows user-defined functions to be written in other languages besides SQL and C. These other languages are generically called procedural languages (Pls)

There are currently four procedural languages available in the standard PostgreSQL distribution:

PL/pgSQL , PL/Tcl , PL/Perl, and PL/Python.

We will work with PL/pgSQL

Goals of PL/pgSQL

The design goals of PL/pgSQL were to create a loadable procedural language that

- 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,
- can be defined to be trusted by the server,
- is easy to use.

Functions created with PL/pgSQL can be used anywhere that built-in functions could be used.

Advantages

Every statement in SQL must be executed individually by the database server.

Performance issue

With PL/pgSQL you can group a block of computation and a series of queries inside the database server, thus having the power of a procedural language and the ease of use of SQL, but with considerable savings of client/server communication overhead.

- 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

Syntax

The basic syntax to create a function is as follows

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS $variable_name$
   DECLARE
      declaration;
      [...]
   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.
- **RETURNS** clause specifies that data type you are going to return from the function.
- **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. Here, we use this option for PostgreSQL.

Example

```
 id | name  | age | address    | salary
----+-------+-----+------------+--------
  1 | Paul  |  32 | California |  20000
  2 | Allen |  25 | Texas      |  15000
  3 | Teddy |  23 | Norway     |  20000
  4 | Mark  |  25 | Rich-Mond  |  65000
  5 | David |  27 | Texas      |  85000
  6 | Kim   |  22 | South-Hall |  45000
  7 | James |  24 | Houston    |  10000
(7 rows)
```

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

```
select totalRecords();
```

```
 totalrecords
--------------
            7
(1 row)
```

sourcemind

Structure of PL/pgSQL

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'function body text'
LANGUAGE plpgsql;
```

It is often helpful to use dollar quoting to write the function body, rather than the normal single quote syntax

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

Structure of PL/pgSQL

PL/pgSQL is a block-structured language. The complete text of a function body must be a block. A block is defined as:

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

A **label** is only needed if you want to identify the block for use in an EXIT statement, or to qualify the names of the variables declared in the block. If a label is given after END, it must match the label at the block's beginning.

- Each declaration and each statement within a block is terminated by a semicolon.
- A block that appears within another block must have a semicolon after END, as shown above; however the final END that concludes a function body does not require a semicolon.

Structure of PL/pgSQL

- A double dash (**--**) starts a comment that extends to the end of the line.
- A /* starts a block comment that extends to the matching occurrence of */.

- Any statement in the statement section of a block can be a subblock. Subblocks can be used for logical grouping or to localize variables to a small group of statements. Variables declared in a subblock mask any similarly-named variables of outer blocks for the duration of the subblock; but you can access the outer variables anyway if you qualify their names with their block's label.

Example

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints 50
    END;
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50
    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

Declarations

The general syntax of a variable declaration is:

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

- The **DEFAULT** clause, if given, specifies the initial value assigned to the variable when the block is entered. If the DEFAULT clause is not given then the variable is initialized to the SQL null value.
- The **CONSTANT** option prevents the variable from being assigned to after initialization, so that its value will remain constant for the duration of the block.
- If **NOT NULL** is specified, an assignment of a null value results in a run-time error. All variables declared as NOT NULL must have a nonnull default value specified.
- Equal (=) can be used instead of PL/SQL-compliant :=.

Examples

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
transaction_time CONSTANT timestamp with time zone := now();
```

Once declared, a variable's value can be used in later initialization expressions

```
DECLARE
  x integer := 1;
  y integer := x + 1;
```

Declaring Function Parameters

- Parameters passed to functions are named with the identifiers **$1, $2**, etc
- Optionally, aliases can be declared for **$n** parameter names for increased readability

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

```
name ALIAS FOR $n;
```

```
CREATE FUNCTION sales_tax(real) RETURNS real
AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Procedures

A procedure is a database object similar to a function.

- Procedures are defined with the **CREATE PROCEDURE** command, not **CREATE FUNCTION**.

- Procedures **do not return a function value**; hence **CREATE PROCEDURE** lacks a **RETURNS** clause. However, procedures can instead return data to their callers via output parameters.

Procedures

```
CREATE FUNCTION clean_emp() RETURNS void AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;
```

```
CREATE PROCEDURE clean_emp() AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;
```