Stored Procedures

Stored Procedures

- A stored procedure is a set of SQL statements with an assigned name, which are stored in a relational database management system as a group, so it can be reused and shared by multiple programs.

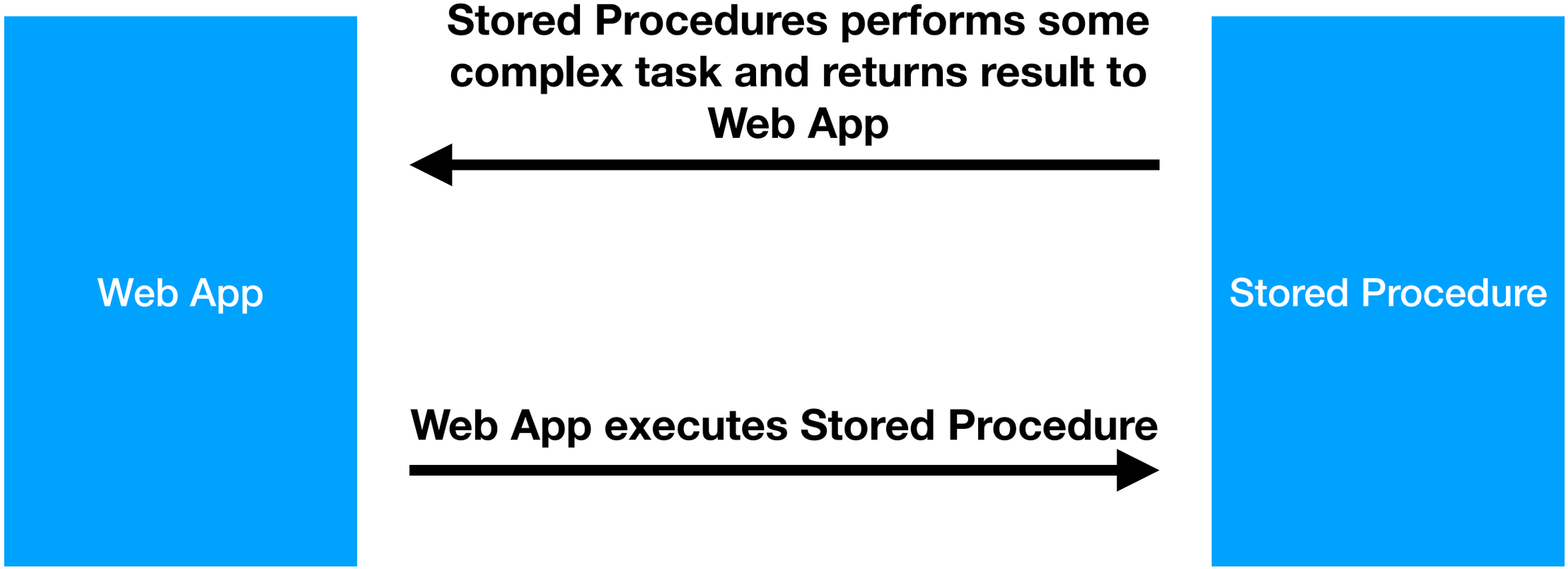
- Stored procedures can access or modify data in a database, but it is not tied to a specific database or object, which offers a number of advantages.

Stored Procedures

- The stored procedures define functions for creating triggers or custom aggregate functions.

- In addition, stored procedures also add many procedural features e.g., control structures and complex calculation. These allow you to develop custom functions much easier and more effective.

Stored Procedures



Advantage 1

- Reduce the number of round trips between application and database servers.
- All SQL statements are wrapped inside a function stored in the PostgreSQL database server so the application only has to issue a function call to get the result back instead of sending multiple SQL statements and wait for the result between each call.

Advantage 2

- Increase application performance because the user-defined functions are pre-compiled and stored in the PostgreSQL database server.

Advantage 2

- Be able to reuse in many applications. Once you develop a function, you can reuse it in any applications.

Disadvantages

- Slow in software development because it requires specialized skills that many developers do not possess.

- Make it difficult to manage versions and hard to debug.

- May not be portable to other database management systems e.g., MySQL or Microsoft SQL Server.

Stored Procedures

PostgreSQL divides the procedural languages into two main groups:

- Safe languages can be used by any users. SQL and PL/pgSQL are safe languages.

- Sand-boxed languages are only used by superusers because sand-boxed languages provide the capability to bypass security and allow access external sources. C is an example of a sandboxed language or unsafe language.

Stored Procedures

- By default, PostgreSQL supports three procedural languages: SQL, PL/pgSQL, and C.

- You can also load other procedural languages e.g., Perl, Python, JS, Ruby, Java and TCL into PostgreSQL using extensions.

Creating functions

- To create a new user-defined function in PostgreSQL, you use the **`CREATE FUNCTION`** statement as follows:

```
CREATE FUNCTION function_name(p1 type, p2 type)
 RETURNS type AS
BEGIN
 -- logic
END;
LANGUAGE language_name;
```

Creating functions

Let's examine the **`CREATE FUNCTION`** statement in more detail.

- First, specify the name of the function followed by the **`CREATE FUNCTION`** clause.

- Then, put a comma-separated list of parameters inside the parentheses following the function name.

Creating functions

- Next, specify the return type of the function after the **`RETURNS`** keyword.

- After that, place the code inside the **`BEGIN`** and **`END`** block. The function always ends with a semicolon (;) followed by the **`END`** keyword.

- Finally, indicate the procedural language of the function e.g., plpgsql in case PL/pgSQL is used.

Example

```
CREATE FUNCTION inc(val integer) RETURNS integer AS $$
BEGIN
RETURN val + 1;
END; $$
LANGUAGE PLPGSQL;
```

Example

```
SELECT inc(20);
```

	inc
1	21

```
SELECT inc(inc(20));
```

	inc
1	22

PL/pgSQL IN parameters

```
CREATE OR REPLACE FUNCTION get_sum(
 a NUMERIC,
 b NUMERIC)
RETURNS NUMERIC AS $$
BEGIN
 RETURN a + b;
END; $$

LANGUAGE plpgsql;
```

PL/pgSQL IN parameters

- The **`get_sum()`** function accepts two parameters: a, and b and returns a numeric.

- The data types of two parameters are **`NUMERIC`**.

- By default, the parameter's type of any parameter in PostgreSQL is **`IN`** parameter. You can pass the **`IN`** parameters to the function but you cannot get them back as a part of result.

PL/pgSQL IN parameters

```
SELECT get_sum(10,20);
```

	get_sum
1	30

PL/pgSQL OUT parameters

- The **`OUT`** parameters are defined as part of the function arguments list and are returned back as a part of the result.

- PostgreSQL supported the **`OUT`** parameters since version 8.1

PL/pgSQL OUT parameters

- To define **OUT** parameters, you use the **OUT** keyword as demonstrated in the following example:

```
CREATE OR REPLACE FUNCTION hi_lo(
 a NUMERIC,
 b NUMERIC,
 c NUMERIC,
        OUT hi NUMERIC,
 OUT lo NUMERIC)
AS $$
BEGIN
 hi := GREATEST(a,b,c);
 lo := LEAST(a,b,c);
END; $$

LANGUAGE plpgsql;
```

PL/pgSQL OUT parameters

The hi_lo function accepts 5 parameters:

- Three **`IN`** parameters: a, b, c.
- Two **`OUT`** parameters: hi (high) and lo (low).

PL/pgSQL OUT parameters

- Inside the function, we get the greatest and least numbers of three **`IN`** parameters using **`GREATEST`** and **`LEAST`** built-in functions.

- Because we use the **`OUT`** parameters, we don’t need to have a **`RETURN`** statement.

- The **`OUT`** parameters are useful in a function that needs to return multiple values without defining a custom type.

PL/pgSQL OUT parameters

```
SELECT hi_lo(10,20,30);
```

	hi_lo
1	(30,10)

PL/pgSQL OUT parameters

```
SELECT * FROM hi_lo(10,20,30);
```

	hi	lo
1	30	10

PL/pgSQL INOUT parameters

- The **`INOUT`** parameter is the combination **`IN`** and **`OUT`** parameters.

- It means that the caller can pass the value to the function.

- The function then changes the argument and passes the value back as a part of the result.

PL/pgSQL INOUT parameters

The following example shows you the square function that accepts a number and returns the square of that number.

```
CREATE OR REPLACE FUNCTION square(
 INOUT a NUMERIC)
AS $$
BEGIN
 a := a * a;
END; $$
LANGUAGE plpgsql;
```

PL/pgSQL INOUT parameters

```
SELECT square(4);
```

	square
1	16

PL/pgSQL VARIADIC parameters

- A PostgreSQL function can accept a variable number of arguments with one condition that all arguments have the same data type.
- The arguments are passed to the function as an array.
- See the following example:

PL/pgSQL VARIADIC parameters

```
CREATE OR REPLACE FUNCTION sum_avg(
 VARIADIC list NUMERIC[],
 OUT total NUMERIC,
         OUT average NUMERIC)
AS $$
BEGIN
    SELECT INTO total SUM(list[i])
    FROM generate_subscripts(list, 1) g(i);

    SELECT INTO average AVG(list[i])
    FROM generate_subscripts(list, 1) g(i);

END; $$
LANGUAGE plpgsql;
```

PL/pgSQL VARIADIC parameters

```
SELECT * FROM sum_avg(10,20,30);
```

	total	average
1	60	20

Function Overloading

- PostgreSQL allows more than one function to have the same name, so long as the arguments are different.

- If more than one function has the same name, we say those functions are overloaded.

- When a function is called, PostgreSQL determines the exact function is being called based on the input arguments.

Function Overloading

```
CREATE OR REPLACE FUNCTION get_rental_duration(p_customer_id
INTEGER)
 RETURNS INTEGER AS $$

DECLARE
 rental_duration INTEGER;
BEGIN
 -- get the rate based on film_id
 SELECT INTO rental_duration SUM( EXTRACT( DAY FROM return_date -
rental_date))
    FROM rental
 WHERE customer_id=p_customer_id;

 RETURN rental_duration;
END; $$
LANGUAGE plpgsql;
```

Function Overloading

- The **`get_rental_function`** function accepts **`p_customer_id`** as the argument.

- It returns the sum of duration (in days) that a specific customer rented DVDs.

- For example, we can get the rental duration of the customer with customer id **`232`**, we call the **`get_rental_duration`** function as follows:

Function Overloading

```sql
SELECT get_rental_duration(232);
```

	rental_duration
1	107

Function Overloading

- Suppose, we want to know the rental duration of a customer from a specific date up to now.

- We can add one more parameter **`p_from_date`** to the **`get_retal_duration()`** function or we can develop a new function with the same name but have two parameters as follows:

Function Overloading

```
CREATE OR REPLACE FUNCTION get_rental_duration(p_customer_id INTEGER,
p_from_date DATE)
 RETURNS INTEGER AS $$
DECLARE
 rental_duration integer;
BEGIN
 -- get the rental duration based on customer_id and rental date
 SELECT INTO rental_duration
             SUM( EXTRACT( DAY FROM return_date + '12:00:00' -
rental_date))
 FROM rental
 WHERE customer_id= p_customer_id AND
   rental_date >= p_from_date;

 RETURN rental_duration;
END; $$
LANGUAGE plpgsql;
```

Function Overloading

- This function has the same name as the first one except that it has two parameters.

- We say the **`get_rental_duration(integer)`** function is overloaded by the **`get_rental_duration(integer,date)`** function.

Function Overloading

```sql
SELECT get_rental_duration(232,'2005-07-01');
```

	rental_duration
1	85

Default values

```
CREATE OR REPLACE FUNCTION get_rental_duration(
 p_customer_id INTEGER,
 p_from_date DATE DEFAULT '2005-01-01'
 )
 RETURNS INTEGER AS $$
DECLARE
 rental_duration integer;
BEGIN
 -- get the rental duration based on customer_id and rental date
 SELECT INTO rental_duration
             SUM( EXTRACT( DAY FROM return_date + '12:00:00' -
rental_date))
 FROM rental
 WHERE customer_id= p_customer_id AND
   rental_date >= p_from_date;

 RETURN rental_duration;
END; $$
LANGUAGE plpgsql;
```

Function Overloading

```sql
SELECT get_rental_duration(232);
```

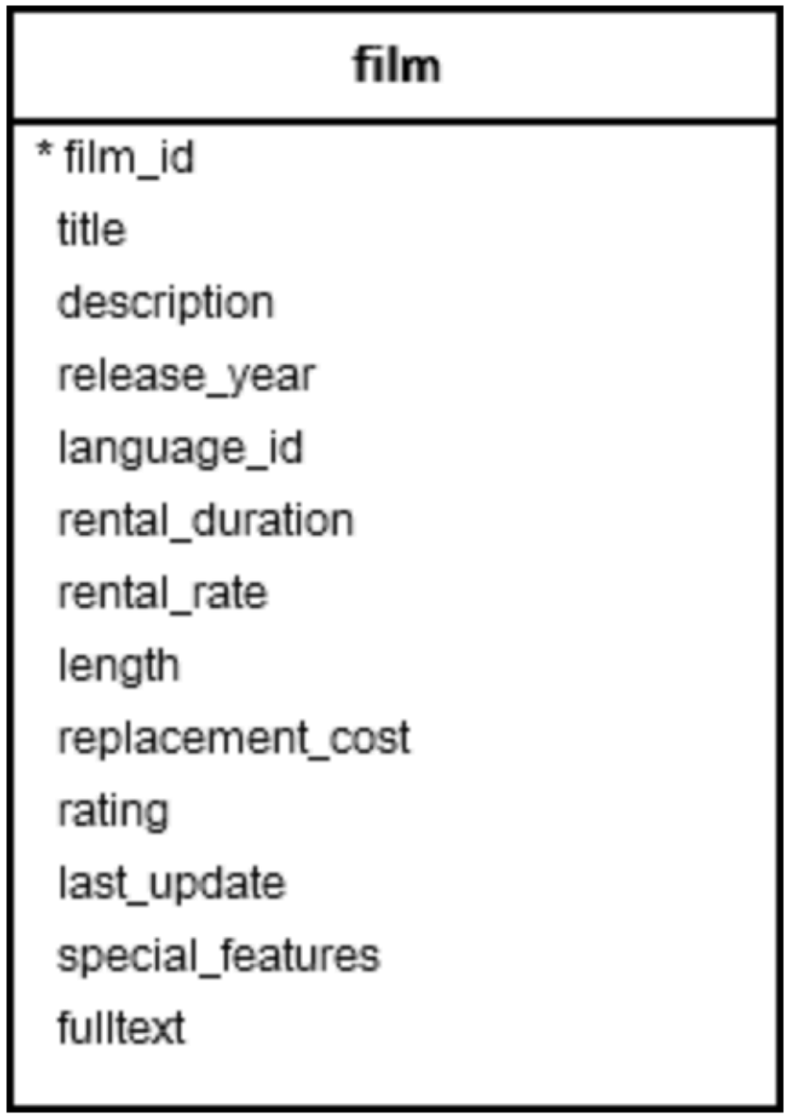
```
[Err] ERROR:  function get_rental_duration(integer) is not unique
LINE 1: SELECT get_rental_duration(232);
               ^
HINT:  Could not choose a best candidate function. You might need
to add explicit type casts.
```

Function Overloading

```
DROP FUNCTION get_rental_duration(INTEGER,DATE);
```

```
SELECT get_rental_duration(232);
```

Function That Returns A Table



Function That Returns A Table

```
CREATE OR REPLACE FUNCTION get_film (p_pattern VARCHAR)
 RETURNS TABLE (
 film_title VARCHAR,
 film_release_year INT
)
AS $$
BEGIN
 RETURN QUERY SELECT
 title,
 cast( release_year as integer)
 FROM
 film
 WHERE
 title ILIKE p_pattern ;
END; $$

LANGUAGE 'plpgsql';
```

Function That Returns A Table

- This `get_film(varchar)` function accepts one parameter p_pattern which is a pattern that you want to match with the film title.

- To return a table from the function, you use `RETURNS TABLE` syntax and specify the columns of the table. Each column is separated by a comma (,).

Function That Returns A Table

- In the function, we return a query that is a result of a **`SELECT`** statement.

- Notice that the columns in the **`SELECT`** statement must match with the columns of the table that we want to return.

- Because the data type of **`release_year`** of the film table is not integer, we have to convert it into integer using type cast.

Function That Returns A Table

```
SELECT * FROM get_film ('Al%');
```

film_title	film_release_year
Alabama Devil	2006
Aladdin Calendar	2006
Alamo Videotape	2006
Alaska Phantom	2006
Ali Forever	2006
Alice Fantasia	2006
Alien Center	2006
Alley Evolution	2006
Alone Trip	2006
Alter Victory	2006

Function That Returns A Table

```
SELECT get_film ('Al%');
```

get_film
("Alabama Devil",2006)
("Aladdin Calendar",2006)
("Alamo Videotape",2006)
("Alaska Phantom",2006)
("Ali Forever",2006)
("Alice Fantasia",2006)
("Alien Center",2006)
("Alley Evolution",2006)
("Alone Trip",2006)
("Alter Victory",2006)

Block Structure

A PL/pgSQL function is organized into blocks. The following illustrates the syntax of a complete block in PL/pgSQL:

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

Block Structure

Let’s examine the block structure in more detail:

- Each block has two sections called declaration and body. The declaration section is optional while the body section is required. The block is ended with a semicolon (;) after the **`END`** keyword.

- A block may have optional labels at the beginning and at the end. The label at the beginning and at the end must be the same. The block label is used in case you want to use the block in **`EXIT`** statement or you want to qualify the names of variables declared in the block.

Block Structure

- The declaration section is where you declare all variables used within the body section. Each statement in the declaration section is terminated with a semicolon (;).

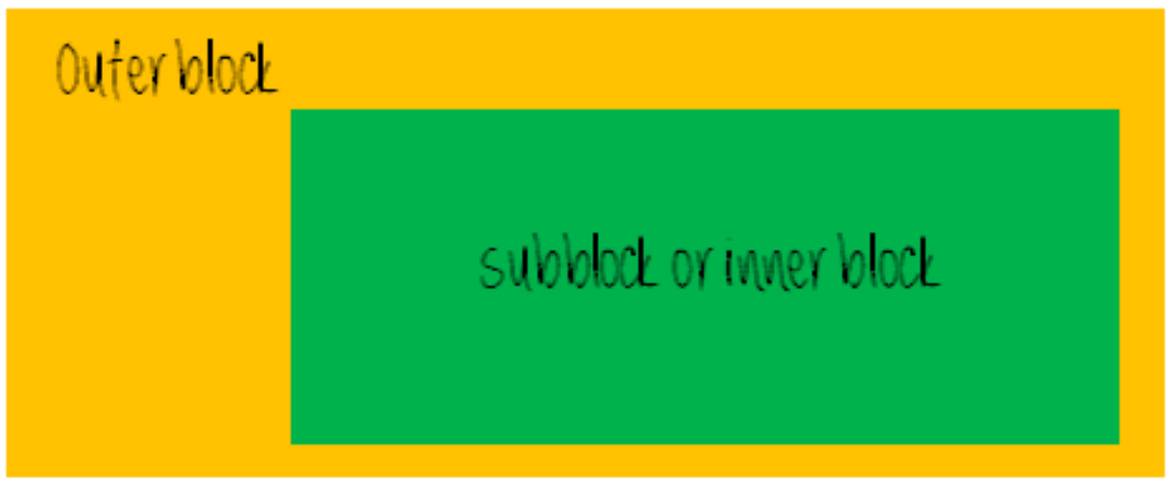
- The body section is where you put the logic of the block. It contains any valid statements. Each statement in the body section is also terminated with a semicolon (;).

Block Structure

```
DO $$
<<first_block>>
DECLARE
  counter integer := 0;
BEGIN
   counter := counter + 1;
   RAISE NOTICE 'The current value of counter is %', counter;
END first_block $$;
```

Subblock

- You can put a block inside the body of another block.
- This block nested inside another is called subblock.
- The block that contains the subblock is referred to as an outer block.



Subblock

- You often use subblocks for grouping statements so that a large block can be divided into smaller and more logical subblocks.

- The variables in the subblock can have the names as the ones in the outer block, even though it is not a good practice.

- When you define a variable within subblock with the same name as the one in the outer block, the variable in the outer block is hidden in the subblock.

- In case you want to access a variable in the outer block, you use block label to qualify its name;

Subblock

```
DO $$
<<outer_block>>
DECLARE
  counter integer := 0;
BEGIN
   counter := counter + 1;
   RAISE NOTICE 'The current value of counter is %', counter;

   DECLARE
       counter integer := 0;
   BEGIN
       counter := counter + 10;
       RAISE NOTICE 'The current value of counter in the subblock is %', counter;
       RAISE NOTICE 'The current value of counter in the outer block is %', outer_block.counter;
   END;

   RAISE NOTICE 'The current value of counter in the outer block is %', counter;

END outer_block $$;
```

Subblock

```
[2018-03-26 00:55:31] [00000] The current value of counter is 1
[2018-03-26 00:55:31] [00000] The current value of counter in the subblock is 10
[2018-03-26 00:55:31] [00000] The current value of counter in the outer block is 1
[2018-03-26 00:55:31] [00000] The current value of counter in the outer block is 1
[2018-03-26 00:55:31] completed in 27ms
```

Questions