PostgreSQL 9.3+

# 1. Stored Procedures and Functions in PostgreSQL

|  |  |
| --- | --- |
| **Stored Procedure** | **Function** |
| Use in an expression | [http://www.sqlines.com/_media/red_cross.png](http://www.sqlines.com/_detail/red_cross.png?id=postgresql:stored_procedures_functions) | [http://www.sqlines.com/_media/green_tick.gif](http://www.sqlines.com/_detail/green_tick.gif?id=postgresql:stored_procedures_functions) |
| Return a value | [http://www.sqlines.com/_media/red_cross.png](http://www.sqlines.com/_detail/red_cross.png?id=postgresql:stored_procedures_functions) | [http://www.sqlines.com/_media/green_tick.gif](http://www.sqlines.com/_detail/green_tick.gif?id=postgresql:stored_procedures_functions) |
| Return values as OUT parameters | [http://www.sqlines.com/_media/green_tick.gif](http://www.sqlines.com/_detail/green_tick.gif?id=postgresql:stored_procedures_functions) | [http://www.sqlines.com/_media/red_cross.png](http://www.sqlines.com/_detail/red_cross.png?id=postgresql:stored_procedures_functions) |
| Return a single result set | [http://www.sqlines.com/_media/green_tick.gif](http://www.sqlines.com/_detail/green_tick.gif?id=postgresql:stored_procedures_functions) | [http://www.sqlines.com/_media/green_tick.gif](http://www.sqlines.com/_detail/green_tick.gif?id=postgresql:stored_procedures_functions) (as a table function) |
| Return multiple result sets | [http://www.sqlines.com/_media/green_tick.gif](http://www.sqlines.com/_detail/green_tick.gif?id=postgresql:stored_procedures_functions) | [http://www.sqlines.com/_media/red_cross.png](http://www.sqlines.com/_detail/red_cross.png?id=postgresql:stored_procedures_functions) |

To return a result set from a PostgreSQL procedure, you have to specify *refcursor* return type, open and return a cursor:

CREATE OR REPLACE FUNCTION show\_cities() RETURNS refcursor AS $$

DECLARE

ref refcursor;

BEGIN

OPEN ref FOR SELECT city, state FROM cities;

RETURN ref;

END;

$$ LANGUAGE plpgsql;

To return multiple result sets, specify SETOF refcursor return type and use RETURN NEXT to return each cursor:

-- Procedure that returns multiple result sets (cursors)

CREATE OR REPLACE FUNCTION show\_cities\_multiple() RETURNS SETOF refcursor AS $$

DECLARE

ref1 refcursor; -- Declare cursor variables

ref2 refcursor;

BEGIN

OPEN ref1 FOR SELECT city, state FROM cities WHERE state = 'CA'; -- Open the first cursor

RETURN NEXT ref1; -- Return the cursor to the caller

OPEN ref2 FOR SELECT city, state FROM cities WHERE state = 'TX'; -- Open the second cursor

RETURN NEXT ref2; -- Return the cursor to the caller

END;

$$ LANGUAGE plpgsql;

2. Материализированные представления

<https://habrahabr.ru/post/195898/>

Материализированное представление — физический объект базы данных, содержащий в себе результаты некоторого запроса. Бесспорно, одно из самых ожидаемых новшеств. Посмотрим, каким образом работать с ним

**CREATE** **MATERIALIZED** **VIEW** mvw\_book **AS** **SELECT** book.**id**, author.first\_name || ' ' || author.last\_name **AS** author\_name, book.**name** **FROM** book **INNER** **JOIN** author **ON** author.**id** = book.author\_id;в **PostgreSQL**.

#### 3. Триггеры к событиям

**CREATE** **TABLE** article ( **id** SERIAL **NOT** NULL, **name** text **NOT** NULL, **CONSTRAINT** pk\_article\_id PRIMARY **KEY** ( **id** ), **CONSTRAINT** uk\_article\_name **UNIQUE** ( **name** ) );

**ALTER** **TABLE** article **ADD** **COLUMN** misc numeric;

**ALTER** **TABLE** article **ALTER** **COLUMN** misc **TYPE** text;

**ALTER** **TABLE** article **DROP** **COLUMN** misc; **DROP** **TABLE** article;

tg\_event = ddl\_command\_start, tg\_tag = **CREATE** **TABLE** tg\_event = ddl\_command\_end, tg\_tag = **CREATE** **TABLE** tg\_event = ddl\_command\_start, tg\_tag = **ALTER** **TABLE** tg\_event = ddl\_command\_end, tg\_tag = **ALTER** **TABLE** tg\_event = ddl\_command\_start, tg\_tag = **ALTER** **TABLE** tg\_event = ddl\_command\_end, tg\_tag = **ALTER** **TABLE** tg\_event = ddl\_command\_start, tg\_tag = **ALTER** **TABLE** tg\_event = ddl\_command\_end, tg\_tag = **ALTER** **TABLE** tg\_event = ddl\_command\_start, tg\_tag = **DROP** **TABLE** tg\_event = ddl\_command\_end, tg\_tag = **DROP** **TABLE**

To return multiple result sets, specify SETOF refcursor return type and use RETURN NEXT to return each cursor:

#### 4. Рекурсивные представления

Создадим таблицу и наполним ее тестовыми данными:

**CREATE** **TABLE** **directory**

(

**id** serial **NOT** NULL,

parent\_id integer,

**name** text **NOT** NULL,

**CONSTRAINT** pk\_directory\_id PRIMARY **KEY** ( **id** ),

**CONSTRAINT** fk\_directory\_parent\_id FOREIGN **KEY** ( parent\_id ) **REFERENCES** **directory** ( **id** ),

**CONSTRAINT** uk\_directory\_name **UNIQUE** ( parent\_id, **name** )

);

**INSERT** **INTO** **directory** ( parent\_id, **name** ) **VALUES** ( NULL, 'usr' ); *-- сгенерирован id = 1*

**INSERT** **INTO** **directory** ( parent\_id, **name** ) **VALUES** ( 1, 'lib' );

**INSERT** **INTO** **directory** ( parent\_id, **name** ) **VALUES** ( 1, 'include' );

**INSERT** **INTO** **directory** ( parent\_id, **name** ) **VALUES** ( NULL, 'var' ); *-- сгенерирован id = 4*

**INSERT** **INTO** **directory** ( parent\_id, **name** ) **VALUES** ( 4, 'opt' ); *-- сгенерирован id = 5*

**INSERT** **INTO** **directory** ( parent\_id, **name** ) **VALUES** ( 5, 'tmp' );

**INSERT** **INTO** **directory** ( parent\_id, **name** ) **VALUES** ( 4, 'log' ); *-- сгенерирован id = 7*

**INSERT** **INTO** **directory** ( parent\_id, **name** ) **VALUES** ( 7, 'samba' );

**INSERT** **INTO** **directory** ( parent\_id, **name** ) **VALUES** ( 7, 'news' );

**CREATE** **RECURSIVE** **VIEW** vw\_directory

( **id**, parent\_id, **name**, **path** ) **AS** **SELECT** **id**, parent\_id, **name**, '/' ||

**name** **FROM** **directory** **WHERE** parent\_id **IS** NULL **AND** **name** = 'var'

**UNION** ALL **SELECT** **d**.**id**, **d**.parent\_id, **d**.**name**, **t**.**path** || '/' || **d**.**name** **FROM** **directory** **d**

**INNER** **JOIN** vw\_directory **t** **ON** **d**.parent\_id = **t**.**id**;

**SELECT** \* **FROM** vw\_directory **ORDER** **BY** **path**;

**-- дерево**

**CREATE** **TABLE** geo ( **id** int **not** null primary **key**, parent\_id int **references** geo(**id**), **name** varchar(1000) );

**INSERT** **INTO** geo (**id**, parent\_id, **name**) **VALUES** (1, null, 'Планета Земля'), (2, 1, 'Континент Евразия'), (3, 1, 'Континент Северная Америка'), (4, 2, 'Европа'), (5, 4, 'Украина'), (6, 4, 'Германия'), (7, 5, 'Киев'), (8, 5, 'Львов'), (9, 6, 'Берлин');

WITH RECURSIVE r AS

( **SELECT** **id**, parent\_id, **name**, 1 **AS** **level** **FROM** geo **WHERE** **id** = 4

**UNION** ALL

**SELECT** geo.**id**, geo.parent\_id, geo.**name**, r.**level** + 1 **AS** **level** **FROM** geo **JOIN** r

**ON** geo.parent\_id = r.**id** )

**SELECT** \* **FROM** r; id | parent\_id | name | level

*----+-----------+-----------------+-------*

4 | 2 | Европа | 1

5 | 4 | Украина | 2

6 | 4 | Германия | 2

7 | 5 | Киев | 3

8 | 5 | Лювов | 3

9 | 6 | Берлин | 3

SELECT employee\_id, last\_name, manager\_id, LEVEL

FROM employees

CONNECT BY PRIOR employee\_id = manager\_id;

EMPLOYEE\_ID LAST\_NAME MANAGER\_ID LEVEL

----------- ------------------------- ---------- ----------

101 Kochhar 100 1

108 Greenberg 101 2

109 Faviet 108 3

110 Chen 108 3

111 Sciarra 108 3

112 Urman 108 3

113 Popp 108 3

SELECT last\_name, employee\_id, manager\_id, LEVEL

FROM employees

START WITH employee\_id = 100

CONNECT BY PRIOR employee\_id = manager\_id

ORDER SIBLINGS BY last\_name;

LAST\_NAME EMPLOYEE\_ID MANAGER\_ID LEVEL

------------------------- ----------- ---------- ----------

King 100 1

Cambrault 148 100 2

Bates 172 148 3

Bloom 169 148 3

Fox 170 148 3

Kumar 173 148 3

Ozer 168 148 3

Smith 171 148 3

De Haan 102 100 2

Hunold 103 102 3

Austin 105 103 4

Ernst 104 103 4

Lorentz 107 103 4

Pataballa 106 103 4

Errazuriz 147 100 2

Ande 166 147 3

Banda 167 147 3

#### 5. Латеральное присоединение

To return multiple result sets, specify SETOF refcursor return type and use RETURN NEXT to return each cursor:

Позволяет обращатся из подзапроса к сущностями из внешнего запроса. Пример использования (подсчет количества полей только для сущностей из схемы public):

**SELECT** **t**.table\_schema || '.' || **t**.table\_name,

q.columns\_count

**FROM** information\_schema.**tables** **t**,

LATERAL (

**SELECT** **sum** ( 1 ) **AS** columns\_count

**FROM** information\_schema.**columns** **c**

**WHERE** **t**.table\_schema **IN** ( 'public' ) **AND**

**t**.table\_schema || '.' || **t**.table\_name = **c**.table\_schema || '.' || **c**.table\_name

) q

**ORDER** **BY** 1;

#### 6. Изменяемые внешние таблицы

Новый модуль *postgres\_fdw*, позволяющий получить read/write доступ к данным, расположенным в другой БД. Ранее такая функциональность была в *dblink*, но в *postgres\_fdw* всё прозрачнее, стандартизированный синтаксис и можно получить лучшую производительность. Посмотрим каким способом можно использовать*postgres\_fdw*.  
  
Создадим новую БД *fdb* и в ней тестовую таблицу (она будет внешней по отношению к текущей БД):

**CREATE** **TABLE** city

(

country text **NOT** NULL,

**name** text **NOT** NULL,

**CONSTRAINT** uk\_city\_name **UNIQUE** ( country, **name** )

Вернемся в текущую БД и настроим внешний источник данных:

*-- создание расширения*

**CREATE** EXTENSION postgres\_fdw;

*-- добавление внешнего сервера*

**CREATE** **SERVER** fdb\_server FOREIGN **DATA** WRAPPER postgres\_fdw OPTIONS ( host 'localhost', dbname 'fdb' );

*-- отображения пользователя*

**CREATE** **USER** **MAPPING** **FOR** **PUBLIC** **SERVER** fdb\_server OPTIONS ( **password** 'pwd' );

*-- создание внешней таблицы*

**CREATE** FOREIGN **TABLE** fdb\_city ( country text, **name** text ) **SERVER** fdb\_server OPTIONS ( table\_name 'city' );

);

Теперь мы можем работать с внешней таблицей:

*-- добавляем запись*

**INSERT** **INTO** fdb\_city ( country, **name** ) **VALUES** ( 'USA', 'Las Vegas' );

*-- изменяем ее*

**UPDATE** fdb\_city **SET** **name** = 'New Vegas' **WHERE** **name** = 'New Vegas';

*-- смотрим, что получилось*

**SELECT** \* **FROM** fdb\_city;

#### 7. Функции и операторы для работы с типом JSON

Тип **JSON** появился в **PostgreSQL** 9.2, но функций было лишь две — array\_to\_json (конвертация массива в**JSON**) и row\_to\_json (конвертация записи в **JSON**). Теперь функций стало больше и можно вполне работать с этим типом:

**CREATE** **TYPE** t\_link **AS**

(

"from" text,

"to" text

);

**CREATE** **TABLE** param

(

**id** serial **NOT** NULL,

**name** text **NOT** NULL,

**value** **json** **NOT** NULL,

**CONSTRAINT** pk\_param\_id PRIMARY **KEY** ( **id** ),

**CONSTRAINT** uk\_param\_name **UNIQUE** ( **name** )

);

**INSERT** **INTO** param ( **name**, **value** ) **VALUES**

( 'connection', '{ "username" : "Administrator", "login" : "root", "databases" : [ "db0", "db1" ], "enable" : { "day" : 0, "night" : 1 } }'::**json** ),

( 'link', '{ "from" : "db0", "to" : "db1" }'::**json** );

*-- значение поля (запрос)*

**SELECT** **value** ->> 'username' **FROM** param **WHERE** **name** = 'connection';

*-- результат*

Administrator

*-- значение поля (по заданному пути) (запрос)*

**SELECT** **value** #>> '{databases,0}' **FROM** param **WHERE** **name** = 'connection';

*-- результат*

db0

*-- преобразование в SETOF ( key, value ) с типом text (запрос)*

**SELECT** json\_each\_text ( **value** ) **FROM** param;

*-- результат*

(username,Administrator)

(login,root)

(databases,"[ ""db0"", ""db1"" ]")

(enable,"{ ""day"" : 0, ""night"" : 1 }")

(from,db0)

(to,db1)

*-- значения ключей (запрос)*

**SELECT** json\_object\_keys ( **value** ) **FROM** param;

*-- результат*

username

login

databases

enable

from

to

*-- значение в виде записи (запрос)*

**SELECT** \* **FROM** json\_populate\_record ( null::t\_link, ( **SELECT** **value** **FROM** param **WHERE** **name** = 'link' ) );

*-- результат*

db0;db1

*-- значения массива (запрос)*

**SELECT** json\_array\_elements ( **value** -> 'databases' ) **FROM** param;

*-- результат*

"db0"

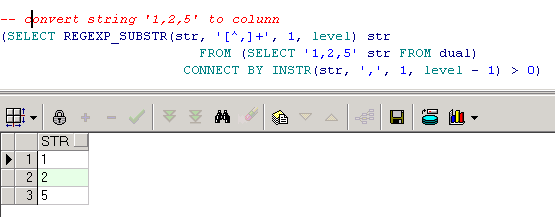
"db1"

#### 8. Преобразование строки типа ‘1,2,3,4,5’ в столбец

(SELECT REGEXP\_SUBSTR(str, '[^,]+', 1, level) str

FROM (SELECT '1,2,5' str FROM dual)

CONNECT BY INSTR(str, ',', 1, level - 1) > 0)



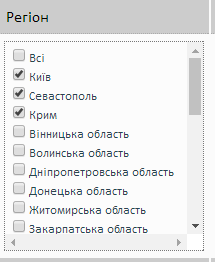
-- Example use in SQL in WHERE PART for filtering subset criteria (like list of cities)

and (p.id\_purchase\_type in

(SELECT REGEXP\_SUBSTR(str, '[^,]+', 1, level) str

FROM (SELECT p\_var\_pur\_type str FROM dual)

CONNECT BY INSTR(str, ',', 1, level - 1) > 0)



function F\_Q

(p\_app in varchar2,

p\_var\_region in varchar2 default null,

p\_goods\_code in varchar2 default '',

p\_date\_start in date default '01.01.1990',

p\_date\_end in date default sysdate,

p\_dis\_date\_start in date default '01.01.1990',

p\_dis\_date\_end in date sysdate)

return t\_purchase\_start\_ann\_table

pipelined is

-- use multi criteria projection

Select \* from table (F\_Q(1,’1,2,3,4’, ‘1,3,4,5’, …))

#### 8. ETL Example

Merge

MERGE INTO employees e

USING (SELECT \* FROM hr\_records WHERE start\_date > ADD\_MONTHS(SYSDATE, -1)) h

ON (e.id = h.emp\_id)

WHEN MATCHED THEN

UPDATE SET e.address = h.address

WHEN NOT MATCHED THEN

INSERT (id, address)

VALUES (h.emp\_id, h.address);

MERGE INTO dest\_tab a

USING source\_tab b

ON (a.object\_id = b.object\_id)

WHEN MATCHED THEN

UPDATE SET

owner = b.owner,

object\_name = b.object\_name,

object\_type = b.object\_type

WHEN NOT MATCHED THEN

INSERT (object\_id, owner, object\_name, object\_type)

VALUES (b.object\_id, b.owner, b.object\_name, b.object\_type);

**--- ETL**

-- очищення довідників

procedure p\_clear is

begin

for i in (select name\_dict

from s\_d

where is\_active = 1

order by num) loop

execute immediate ('delete from S\_' || i.name\_dict);

end loop;

execute immediate ('commit');

exception

when OTHERS then

p\_log\_message('p\_clear - ' || SQLERRM, 1);

return;

end;