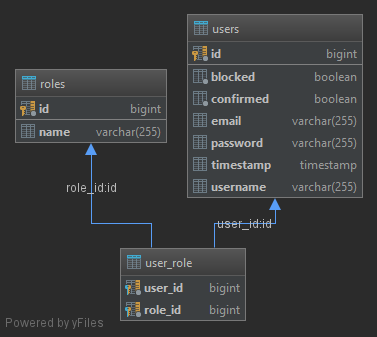
Яцкевич Фёдор, 522403

**Отчет по работе сайта на СУБД Postgresql**

Был произведен переход сайта с СУБД MySQL на PostgreSQl. В ходе перехода серьезных изменений совершено не было, т.к.:

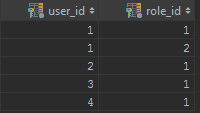
1. Обе СУБД имеют схожий синтаксис запросов
2. В данном приложении используется ORM Framework Hibernate (Spring Data JPA), позволяющий создавать универсальные запросы на языке HQL.
3. **Сложные группировки (GROUPING SETS, ROLLUP, CUBE)**

Рассмотрим таблицу user\_role



Для начала напишем запрос по получению из данной таблицы строки id роли и количество пользователей с данной ролью

Таблица содержит следующие данные:



Запрос:

**SELECT** role\_id, *COUNT*(\*)  
**FROM** user\_role  
**GROUP BY** role\_id;

Response:



Как видно суть оператора GROUP BY в том, что наборы строк с одинаковыми значениями преобразуются в одну строку, имеет смысл это только при использовании агрегатных функций примененных к этим группам.

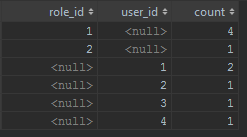
GROUPING SETS

Теперь изменим запрос так чтобы он выводил сначала список ролей с количеством пользователей имеющих данную роль, а затем список пользователей с количеством ролей у данного пользователя:

Запрос:

**SELECT** role\_id, user\_id, *COUNT*(\*)  
**FROM** user\_role  
**GROUP BY GROUPING SETS** (role\_id, user\_id);

Ответ:



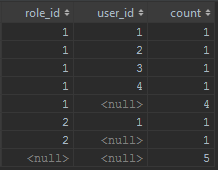
Как видно, для этого выл использован оператор GROUPING SETS в совокупности с GROUP BY.

Его использование похоже на последовательный вызов только оператора GROUP BY сначала с одним полем, а затем с другим.

ROLLUP

Выполним оператор ROLLUP:

**SELECT** role\_id, user\_id, *COUNT*(\*)  
**FROM** user\_role  
**GROUP BY ROLLUP** (role\_id, user\_id);



Как видно данный запрос равносилен запросу:

**SELECT** role\_id, user\_id, *COUNT*(\*)  
**FROM** user\_role  
**GROUP BY GROUPING SETS** ((role\_id, user\_id), (role\_id), ());

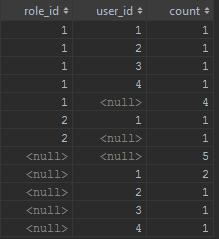
Данный оператор применим для обработки иерархических структур данных. Например подсчитать зарплату для подразделения, отдела, и т.д.

CUBE

Теперь выполним запрос с оператором CUBE:

**SELECT** role\_id, user\_id, *COUNT*(\*)  
**FROM** user\_role  
**GROUP BY CUBE** (role\_id, user\_id);

Получим ответ:



Данный запрос равносилен запросу:

**SELECT** role\_id, user\_id, *COUNT*(\*)  
**FROM** user\_role  
**GROUP BY GROUPING SETS** ((role\_id, user\_id), (role\_id), (user\_id), ());

Он создает группировки по заданному списку и всем его подмножествам.

1. **Оконные функции**

Оконная функция выполняет вычисления для набора строк, некоторым образом связанных с текущей строкой. Можно сравнить её с агрегатной функцией, но, в отличие от обычной агрегатной функции, при использовании оконной функции несколько строк не группируются в одну, а продолжают существовать отдельно. Внутри же, оконная функция, как и агрегатная, может обращаться не только к текущей строке результата запроса.

Синтаксис окна:

[ *имя\_существующего\_окна* ]

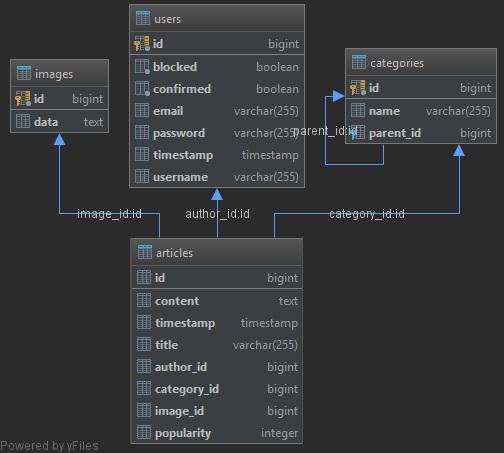
[ PARTITION BY *выражение* [, ...] ]

[ ORDER BY *выражение* [ ASC | DESC | USING *оператор* ] [ NULLS { FIRST | LAST } ] [, ...] ]

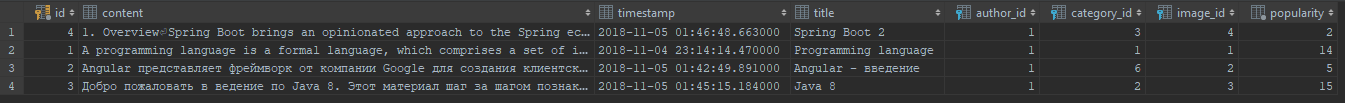
[ *определение\_рамки* ]

row\_number()

Для примера работы оконных функций сделаем запрос для получения основной информации о статьях, а также их номерах в рейтинге. Эта информация хранится в таблице articles:



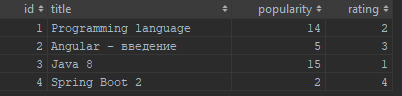
Данные из таблицы articles:



Запрос:

**SELECT** id, title, popularity, *row\_number*() **OVER** (**ORDER BY** popularity **DESC**) **AS** rating  
**FROM** articles  
**ORDER BY** id **ASC**;

Ответ:



В запросе после оператора OVER следует окно. В данном случае в качестве окна указан порядок обработки – по убыванию популярности.

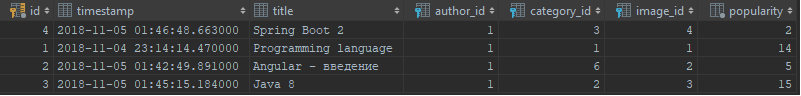
Здесь была использована функция row\_number(), которая получает номер текущей строки в её разделе, начиная с 1.

В конце запроса был добавлен ORDER BY, в противном случае результат бы был упорядочен как ORDER BY popularity DESC, как описано в окне.

Использование агрегатных функций в качестве оконных

Также агрегатные функции можно применять в качестве оконных. Для этого за названием функции должен следовать оператор OVER с определением окна. Продемонстрирую применение агрегатной функции в качестве оконной. Выведем из таблицы articles данные о статье, а также среднюю популярность для группы статей с категорией соответствующей текущей.

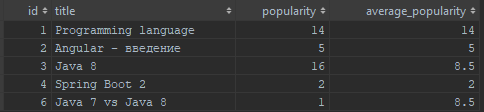
Исходные данные:



Запрос:

**SELECT** id, title, popularity, *avg*(popularity) **OVER** (**PARTITION BY** category\_id) **AS** average\_popularity  
**FROM** articles  
**ORDER BY** id **ASC**;

Ответ:



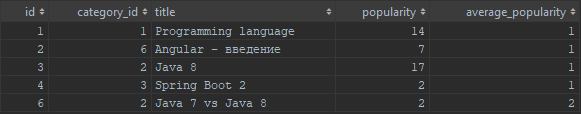
В данном случае был использован оператор PARTITION BY, который работает аналогично GROUP BY, но используется в определении окна. Преимущество оконной функции в данном случае состоит в том, что при использовании агрегатной функции можно вывести только category\_id и average\_popularity.

rank

функция rank выдаёт порядковый номер в разделе текущей строки для каждого уникального значения, по которому выполняет сортировку предложение ORDER BY.

Пример

**SELECT** id, category\_id, title, popularity, *rank*()  
 **OVER** (**PARTITION BY** category\_id **ORDER BY** popularity **DESC** ) **AS** average\_popularity  
**FROM** articles  
**ORDER BY** id **ASC**;



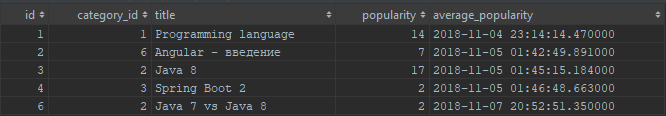
lead()

lead(*значение* anyelement [,*смещение* integer [,*по\_умолчанию* anyelement ]])

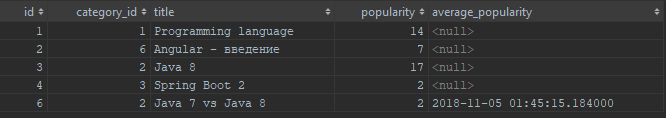
возвращает *значение* для строки, положение которой задаётся *смещением* от текущей строки к концу раздела; если такой строки нет, возвращается значение *по\_умолчанию* (оно должно иметь тот же тип, что и *значение*). Оба параметра *смещение* и *по\_умолчанию* вычисляются для текущей строки. Если они не указываются, то *смещение* считается равным 1, а *по\_умолчанию* — NULL

Для примера сделаем 3 запроса

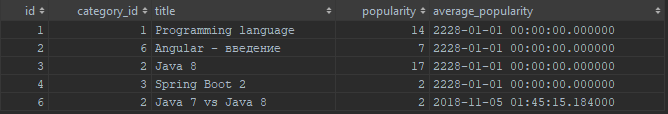
**SELECT** id, category\_id, title, popularity, *lead*(**timestamp**, 0)  
 **OVER** (**PARTITION BY** category\_id **ORDER BY timestamp DESC**) **AS** average\_popularity  
**FROM** articles  
**ORDER BY** id **ASC**;



**SELECT** id, category\_id, title, popularity, *lead*(**timestamp**)  
 **OVER** (**PARTITION BY** category\_id **ORDER BY timestamp DESC**) **AS** average\_popularity  
**FROM** articles  
**ORDER BY** id **ASC**;



**SELECT** id, category\_id, title, popularity, *lead*(**timestamp**, 1, '2228-01-01')  
 **OVER** (**PARTITION BY** category\_id **ORDER BY timestamp DESC**) **AS** average\_popularity  
**FROM** articles  
**ORDER BY** id **ASC**;



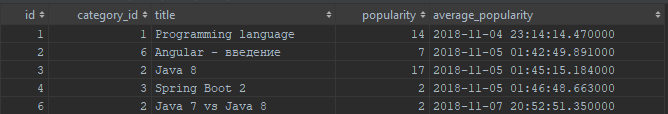
lag()

lag(*значение* anyelement [,*смещение* integer [,*по\_умолчанию*anyelement ]])

возвращает *значение* для строки, положение которой задаётся *смещением* от текущей строки к началу раздела; если такой строки нет, возвращается значение *по\_умолчанию* (оно должно иметь тот же тип, что и *значение*). Оба параметра *смещение* и *по\_умолчанию* вычисляются для текущей строки. Если они не указываются, то *смещение* считается равным 1, а *по\_умолчанию* — NULL

Ниже представлено 2 примера, которые возвращают одинаковый результат:

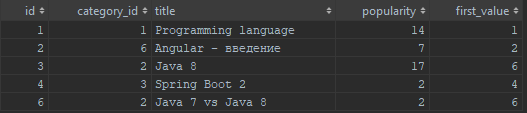
**SELECT** id, category\_id, title, popularity, *lead*(**timestamp**, 0)  
 **OVER** (**PARTITION BY** category\_id **ORDER BY timestamp ASC**) **AS** average\_popularity  
**FROM** articles  
**ORDER BY** id **ASC**;  
  
**SELECT** id, category\_id, title, popularity, *lag*(**timestamp**, 0)  
 **OVER** (**PARTITION BY** category\_id **ORDER BY timestamp DESC**) **AS** average\_popularity  
**FROM** articles  
**ORDER BY** id **ASC**;



first\_value()

возвращает *значение*, вычисленное для первой строки в рамке окна

**SELECT** id, category\_id, title, popularity, *first\_value*(id)  
 **OVER** (**PARTITION BY** category\_id **ORDER BY timestamp DESC**)  
**FROM** articles  
**ORDER BY** id **ASC**;



Есть ещё одно важное понятие, связанное с оконными функциями: для каждой строки существует набор строк в её разделе, называемый рамкой окна. По умолчанию, с указанием ORDER BY рамка состоит из всех строк от начала раздела до текущей строки и строк, равных текущей по значению выражения ORDER BY. Без ORDER BY рамка по умолчанию состоит из всех строк раздела.

В случае если необходимо применить оконную функцию к одному и тому же окну, стоит вынести определение окна, как в следующем примере

**SELECT** id, title, popularity, category\_id, *sum*(popularity) **OVER w AS** average\_popularity, *row\_number*() **OVER w  
FROM** articles  
**WINDOW w AS** (**PARTITION BY** category\_id **ORDER BY** popularity **DESC**)  
**ORDER BY** category\_id, id **ASC**;

1. **Иерархические запросы (CONNECT BY PRIOR, LEVEL SIBLINGS и другие псевдостолбцы)**

CONNECT BY

Оператор CONNECT BY был разработан Oracle еще до появления WITH RECURSIVE в спецификации SQL. В postgresql нет оператора CONNECT BY, но есть функция

connectby(text relname, text keyid\_fld, text parent\_keyid\_fld

[, text orderby\_fld ], text start\_with, int max\_depth

[, text branch\_delim ])

Для использования данной функции необходимо включить модуль tablefunc.

Для этого нужно ввести команду:

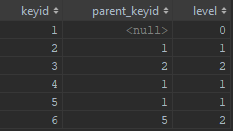
CREATE EXTENSION tablefunc;

Пример запроса к таблице categories при помощи данной функции:

Запрос:

**SELECT** \* **FROM** *connectby*('categories', 'id', 'parent\_id', '1', 10)  
 **AS** t(keyid **bigint**, parent\_keyid **bigint**, **level int**);

Результат:



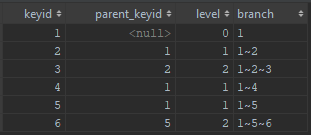
Теперь выведем подиерархию, где корень будет элемент с id = 2

**SELECT** id, **c**.parent\_id, **name**, **level  
FROM** categories **c**,  
 *connectby*('categories', 'id', 'parent\_id', '2', 10)  
 **AS** t(keyid **bigint**, parent\_keyid **bigint**, **level int**)  
**WHERE** t.keyid = **c**.id  
;



Также можно получить иерархию в сравнительно читабельном виде, используя аргумент 6й аргумент, характеризующий разделитель для вывода элемента относительно корня иерархии. Например:

**select** \* **from** *connectby*('categories', 'id', 'parent\_id', '1', 10, '~')  
 **AS** t(keyid **bigint**, parent\_keyid **bigint**, **level int**, branch **text**);



Получаем колонку branch

1. **Регулярные выражения (SIMILAR TO, POSIX, substring, regexp\_replace, regexp\_split\_to\_table)**

SIMILAR TO

Данный оператор, в отличие от оператора, LIKE может описывать более сложные регулярные выражения. Однако здесь используется SQL синтаксис для регулярных выражений, который несколько отличается от типичного синтаксиса регулярных выражений.

Для примера выведем список статей, в названии которых содержится одно или несколько слов Java.

Запрос:

**SELECT** id, title  
**FROM** articles  
**WHERE** title **SIMILAR TO** '(%Java%){1,}';

Ответ:



В данном случае вместо “(%Java%){1,}” можно было использовать “(%Java%)+”, но для наглядной демонстрации функциональности оператора SIMILAR TO, был использован именно первый вариант.

Также возможно использование оператора NOT SIMILAR TO

POSIX Regular expressions

В Postgresql есть 4 оператора, которые используются для оценки строки по регулярным выражениям: . Они означают соответственно «совпадает», «совпадает без учета регистра», «не совпадает», «не совпадает без учета регистра».

Для примера сделаем следующий запрос:

**SELECT** id, title  
**FROM** articles  
**WHERE** title ~\* '(.\*JAVA.\*){1,}';

В данном случае используется не чувствительный к регистру оператор, поэтому используя слово JAVA, мы получаем совпадение при встрече в строке слова Java. Как видно, здесь уже используется «классический» синтаксис (здесь он называется POSIX) для регулярных выражений. Здесь символ “.” Означает любой символ.

Результат запроса:



substring(string from regexp)

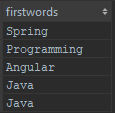
Функция substring предназначена для получения первой строки, соответствующей указанному регулярному выражению.

Для примера выведем первые слова в названии каждой статьи.

Запрос:

**SELECT** *substring*(title **from** '[a-zA-Z]\* ') **as** firstWords  
**FROM** articles

Ответ:



regexp\_replace('foobarbaz', 'b..', 'X')

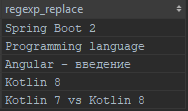
Данная функция позволяет заменить подстроки указанной строки, соответствующие регулярному выражению, на строку, указанную в качестве 3го аргумента.

К примеру выведем все названия статей, причем заменим слово Java без учета регистра на слово Kotlin.

Запрос:

**SELECT** *regexp\_replace*(title, '[j|J][a|A][v|V][a|A]', 'Kotlin', 'g')  
**FROM** articles;

Ответ:



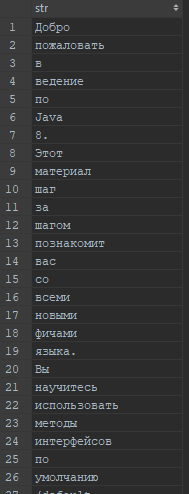
Отмечу, что в функцию в качестве 4го аргумента был передан флаг ‘g’, что позволило заменить не только первое вхождение регулярного выражения в строку, но также и все последующие.

regexp\_split\_to\_table()

Данная функция позволяет разбить строку на подстроки используя разделитель, указанный в качестве второго аргумента.

К примеру, сделаем следующий запрос:

**SELECT** str **FROM** *regexp\_split\_to\_table*((**SELECT content FROM** articles **where** "title" = 'Java 8'), '\s') **AS** str;



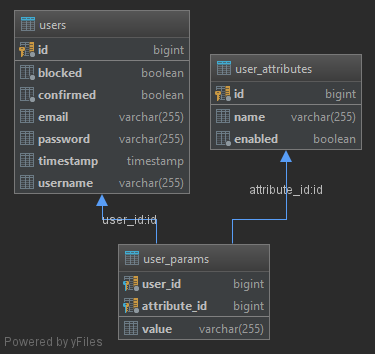
Как видно контент статьи “Java 8” был разбит на отдельные слова.

1. **crosstab(), CASE**

*crosstab()*

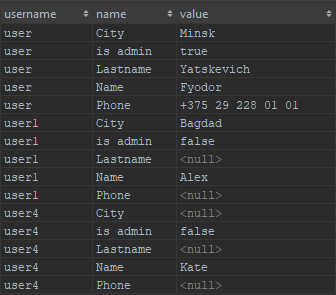
Модуль tablefunc также предоставляет функцию crosstab(), которая аналогична функции PIVOT в Oracle.

Для демонстрации функционала рассмотрим способы отображения данных из таблиц users, user\_attributes, user\_params. Сами таблицы связаны как приведено ниже



Выведем данные из них следующим запросом:

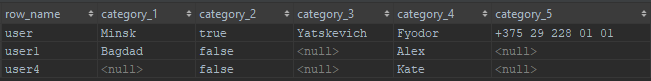
**select** u.username,  
 ua.name,  
 up.value  
**from** users u, user\_params up, user\_attributes ua  
**where** u.id = up.user\_id  
 **and** ua.id = up.attribute\_id  
**order by** u.username, ua.name, up.value;



Данная таблица состоит из 3 колонок. Её можно преобразовать таким образом, что в первой колонке все записи будут уникальными, записи из второй колонки попадут в заглавие (там где сейчас находится name и value), а данные из 3й колонки распределятся так, чтобы соответствовать вертикальному и горизонтальному заглавию таблицы.

Для этого и служит функция crosstab(). Преобразуем прежний запрос в новый:

**SELECT** \*  
**FROM** *crosstab*('  
 select  
 CAST(u.username as text),  
 CAST(ua.name as text),  
 CAST(up.value as text)  
 from  
 users u, user\_params up, user\_attributes ua  
 where  
 u.id = up.user\_id  
 and ua.id = up.attribute\_id  
 order by u.username, ua.name, up.value'  
 ) **AS** ct(row\_name **text**, category\_1 **text**, category\_2 **text**, category\_3 **text**, category\_4 **text**, category\_5 **text**);



Прежний запрос заносится в качестве строки в первый параметр функции, причем в добавок к этому тип полей преобразуется в text при помощи функции CAST. Возвращаемое функцией значение преобразуется в таблицу, с которой можно работать. Здесь указываются поля, которые должны попасть в таблицу: category\_1, category\_2, category\_3. Данная функция динамически определяет количество полей для таблицы.

Недостатком данной функции является то, что для корректной ее работы необходимо чтобы значения параметров обязательно должны быть, даже если они равны null (как на примере выше). Иначе функция работает некорректно.

*CASE*

CASE – это оператор ветвления, его синтаксис следующий

CASE WHEN *условие* THEN *результат*

[WHEN ...]

[ELSE *результат*]

END

Или простая форма:

CASE *выражение*

WHEN *значение* THEN *результат*

[WHEN ...]

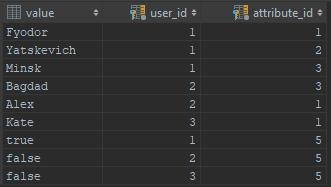
[ELSE *результат*]

END

Для демонстрации возможностей этого оператора создадим в таблице user\_attributes атрибут со значением “is admin” и «заполним его» для каждого пользователя в зависимости от того есть ли у него роль “ROLE\_ADMIN”.

Для заполнения воспользуемся следующим запросом:

**insert into** user\_params(user\_id, attribute\_id, **value**)  
**select** u.id,  
 (**select** id **from** user\_attributes **where name** = 'is admin'),  
 **CASE** ('ROLE\_ADMIN' **IN** (**SELECT name  
 FROM** roles r, user\_role ur  
 **WHERE** ur.user\_id = u.id  
 **AND** ur.role\_id = r.id))  
 **WHEN true  
 THEN** 'true'  
 **ELSE** 'false'  
 **END  
from** users u;



Запрос прошел успешно как видно по данным из таблицы user\_params выше.

* 1. **Рекурсивные запросы с WITH**

*WITH*

Оператор WITH позволяет сформировать подзапрос и присвоить ему имя перед началом выполнения SELECT запроса.

Синтаксис:

[ WITH [ RECURSIVE ] ***with\_query*** [, ...] ]

SELECT [ ALL | DISTINCT [ ON ( ***expression*** [, ...] ) ] ]

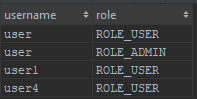
…

…

К примеру, можно создать временный VIEW.

Пример использования:

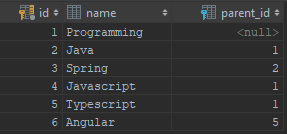
**WITH** tmp **as** (  
 **SELECT** u.id **as** user\_id, r.name **as role  
 FROM** users u, roles r, user\_role ur  
 **WHERE** ur.user\_id = u.id  
 **AND** ur.role\_id = r.id  
)  
**SELECT** u.username, tmp.role  
**FROM** users u, tmp  
**WHERE** u.id = tmp.user\_id;



*WITH RESURSIVE*

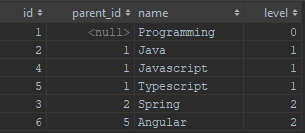
Оператор WITH RECURSIVE удобно использовать для построения иерархии.

Возьмем таблицу categories, содержащую иерархические данные:



И выведем данные из таблицы при помощи WITH RECURSIVE:

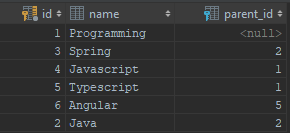
**WITH RECURSIVE** r **AS** (  
 **SELECT** id, parent\_id, **name**, 0 **as level  
 FROM** categories  
 **WHERE** parent\_id **IS NULL  
  
 UNION  
  
 SELECT c**.id, **c**.parent\_id, **c**.name, r.level + 1  
 **FROM** categories **c  
 JOIN** r **ON** (**c**.parent\_id = r.id)  
)  
**SELECT** \* **FROM** r;



Данный запрос работает следующим образом: SELECT до оператора UNION выполняется только при первом проходе. Все последующие проходы к результату прибавляются строки, полученные вторым SELECT’ом. Именно для этого и используется UNION (это стандартный оператор, который можно использовать во многих других ситуациях). UNION, в отличии от UNION ALL не добавляет повторяющиеся строки. Завершение проходов завершается, когда результат второго SELECT’a не содержит ни одной строки. Этого можно добиться, добавив условие WHERE, или другими способами.

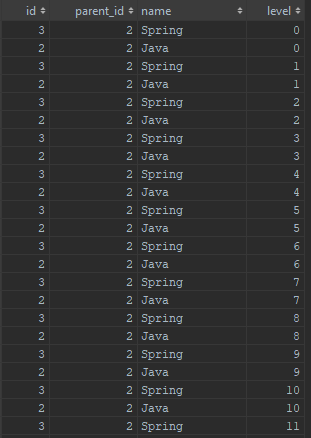
Теперь рассмотрим случай, когда в условии JOIN r ON для одной и той же записи поля id и parent\_id будут иметь одинаковые значения. Ожидается что произойдет зацикливание.

Исходные данные:



Запрос изменим, начнем с 2 ИД, чтобы войти в вечный цикл

**WITH RECURSIVE** r **AS** (  
 **SELECT** id, parent\_id, **name**, 0 **as level  
 FROM** categories  
 **WHERE** parent\_id = 2  
  
 **UNION  
  
 SELECT c**.id, **c**.parent\_id, **c**.name, r.level + 1  
 **FROM** categories **c  
 JOIN** r **ON** (**c**.parent\_id = r.id)  
 )  
**SELECT** \* **FROM** r;



В результате получаем таблицу, это не значит что вечный цикл прервался, просто результаты выводятся по ходу выполнения запроса. В доказательство можно выполнить запрос на получение count(\*):

**WITH RECURSIVE** r **AS** (  
 **SELECT** id, parent\_id, **name**, 0 **as level  
 FROM** categories  
 **WHERE** parent\_id = 2  
  
 **UNION  
  
 SELECT c**.id, **c**.parent\_id, **c**.name, r.level + 1  
 **FROM** categories **c  
 JOIN** r **ON** (**c**.parent\_id = r.id)  
 )  
**SELECT** *count*(\*) **FROM** r;

Теперь по запросу не возвращается никаких результатов, он просто выполняется.

1. **Использование MERGE, FLASHBACK**

В PostgreSQL отсутствует такой функционал.

1. **Внешние таблицы**

*Dblink*

Для получения доступа к данным на удаленной базе, в PostgreSQL есть функция dblink(). Ее синтаксис:

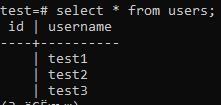
Для удобства лучше создать VIEW, в котором уже будет хранится информация о соединении с удаленной бд. Дальше этот VIEW можно будет использовать как обычную таблицу в БД, хоть она и находится в другой БД.

dblink(text connname, text sql [, bool fail\_on\_error]) returns setof record

dblink(text connstr, text sql [, bool fail\_on\_error]) returns setof record

dblink(text sql [, bool fail\_on\_error]) returns setof record

Для примера создадим локальную бд с именем ‘test’, а в ней таблицу users со следующими данными

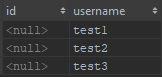


Что бы PostgreSQL распознал функцию dblink, необходимо подключить расширение:

**CREATE EXTENSION** dblink;

Сделаем запрос к базе test на получение таблицы users

**SELECT** proname **as** id, prosrc username **FROM** *dblink*('dbname=test user=postgres password=0000', 'select id, username from users')  
 **AS** t1(proname **name**, prosrc **text**);



Результат оборачиваем во VIEW и работает с ним как с обычной таблицей.

COPY

Данный оператор служит для импорта данных из файла в бд, или наоборот – для экспорта из бд в файл.

Синтаксис:

COPY ***table\_name*** [ ( ***column\_name*** [, ...] ) ]

FROM { '***filename***' | STDIN }

[ [ WITH ] ( ***option*** [, ...] ) ]

COPY { ***table\_name*** [ ( ***column\_name*** [, ...] ) ] | ( ***query*** ) }

TO { '***filename***' | STDOUT }

[ [ WITH ] ( ***option*** [, ...] ) ]

Пример экспорта:

**COPY** users **TO** 'D:/export.txt' **WITH DELIMITER** '|';

В файл выведется следующий текст:

1|f|f|yatskevichfyodor@gmail.com|$2a$11$vRiSjMQjKXuuCYw9OZVWeu/8/91Z1syPL7Jfv8RtSe3Gqvzilusnu|2018-11-04 23:11:49.775|user

2|f|f|email1@mail.com|$2a$11$svapxzztYPVUPEq2QPfdVO/Cwj7J0LFcaELGz3nLPvISSu21fVH72|2018-11-16 20:55:06.729|user1

3|f|f|email4@mail.com|$2a$11$E3YZFyxJWfHCGnAgzU4SheJVLvNaGaCBwob2gNUYXMVcfn27GgUkq|2018-11-16 20:56:57.604|user4

Т.е. данный файл содержит 3 строки.

Теперь эту же информацию импортируем в другую таблицу:

**create table** users\_new **as** (**select** \* **from** users **limit** 0);  
**COPY** users\_new **FROM** 'D:/import.txt' **WITH DELIMITER** '|';

В результате таблица users\_new будет содержать ту же информацию, что и таблица users.

Также в утилите psql есть команда \copy. Её действие идентично оператору COPY. Отличие состоит в том, что \copy – клиентская команда. Это значит что при подключении через psql к удаленному серверу при использовании \copy можно указывать путь к файлу на локальном компьютере. В то время как при использовании оператора COPY, можно указать только путь к файлу, доступному с сервера.

Существенным минусом данных команд является невозможность указать сложные разделители для строк и колонок. Для строк служат только такие разделители как \r, \n, \r\n. А для столбцов можно использовать только однобайтовый разделитель. В некоторых ситуациях (когда данные из файла содержат такие же символы, что и разделители) это вынуждает либо заносить все данные в один столбец, а потом обрабатывать их скриптом, чтобы распределить данные из этого столбца по другим столбцам; либо использовать другую специальную утилиту для импорта/экспорта.

Пример кода утилиты на языке Java

import java.io.FileInputStream;  
import java.io.PrintWriter;  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.PreparedStatement;  
import java.sql.Statement;  
import java.util.Scanner;  
  
public class HistoryImporter {  
  
 private Connection connection;  
  
 private void initConnection() throws Exception {  
 String[] dbProps = getDBprops();  
 try {  
 connection = DriverManager.*getConnection*(dbProps[0], dbProps[1], dbProps[2]);  
 } catch (Exception e) {  
 System.*err*.println(e.toString());  
 }  
 }  
  
 private String[] getDBprops() throws Exception {  
 FileInputStream inputStream = new FileInputStream("./dbProps.txt");  
 Scanner sc = new Scanner(inputStream);  
 String url;  
 String user;  
 String password;  
 try {  
 url = sc.nextLine();  
 user = sc.nextLine();  
 password = sc.nextLine();  
 } finally {  
 inputStream.close();  
 sc.close();  
 }  
  
 return new String[]{url, user, password};  
 }  
  
 public void importFile(String fileDir, String fileName) throws Exception {  
 try {  
 initConnection();  
  
 long counter = 0L;  
  
 createTable(fileName);  
  
 FileInputStream inputStream = null;  
 PrintWriter writer = null;  
 Scanner sc = null;  
 try {  
 inputStream = new FileInputStream(fileDir + fileName);  
 writer = new PrintWriter("Report\_" + fileName + ".txt", "UTF-8");  
 sc = new Scanner(inputStream, "UTF-8");  
 while (sc.hasNextLine()) {  
 String line = sc.nextLine();  
 String[] historyItemDataArray = line.split("\\$#c\\$#");  
 historyItemDataArray[0] = historyItemDataArray[0].replace("$#r$#", "\r");  
 historyItemDataArray[0] = historyItemDataArray[0].replace("$#n$#", "\n");  
  
 if (historyItemDataArray.length < 6) {  
 String message = "String below (number is " + (++counter) + ") consists of " + historyItemDataArray.length +" fields. Process it by hand." + "\n"  
 + "$START$" + line + "$END$";  
  
 writer.println(message);  
 System.*out*.println(message);  
 continue;  
 }  
  
 for (int i = 0; i < historyItemDataArray.length; i++) {  
 historyItemDataArray[i] = historyItemDataArray[i].replace("'", "''");  
 }  
 loadHistoryItemDataToDB(historyItemDataArray, fileName);  
  
 if (counter % 10000 == 0) {  
 System.*out*.println("Processing data... " + counter);  
 }  
 counter++;  
 }  
 System.*out*.println("\nHistory import finished!");  
 if (sc.ioException() != null) {  
 throw sc.ioException();  
 }  
 } finally {  
 if (inputStream != null) {  
 inputStream.close();  
 }  
 if (sc != null) {  
 sc.close();  
 }  
 if (writer != null) {  
 writer.close();  
 }  
 }  
 } finally {  
 if (connection != null) {  
 connection.close();  
 }  
 }  
 }  
  
 private void createTable(String fileName) throws Exception {  
 Statement statement = connection.createStatement();  
 String query = "" +  
 "create table if not exists \"import\_" + fileName + "\"(\n" +  
 " \"fieldname1\" text,\n" +  
 " \"fieldname2\" text,\n" +  
 " \"fieldname3\" timestamp,\n" +  
 " \"fieldname4\" text,\n" +  
 " \"fieldname5\" text,\n" +  
 " \"fieldname6\" text,\n" +  
 " \"fieldname7\" bigint,\n" +  
 " \"fieldname8\" text,\n" +  
 " \"fieldname9\" text,\n" +  
 " \"fieldname10\" bigint,\n" +  
 " \"fieldname11\" text\n" +  
 ");";  
 System.*out*.println("Creating table \"fedor\_import\_" + fileName + "\"...");  
 System.*out*.println("Query:\n" + query);  
 statement.executeUpdate(query);  
 System.*out*.println("Table has been created");  
 }  
  
 private void loadHistoryItemDataToDB(String[] historyItemDataArray, String fileName) throws Exception {  
 PreparedStatement pst = connection.prepareStatement("" +  
 "INSERT INTO \"\_import\_" + fileName + "\"(\"fieldname1\",\"fieldname2\",\"fieldname3\",\"fieldname4\",\"fieldname5\",\"fieldname6\")\n" +  
 "VALUES ('" + historyItemDataArray[0] + "', '" + historyItemDataArray[1] + "', '" + historyItemDataArray[2] + "', '" + historyItemDataArray[3] + "', '" + historyItemDataArray[4] + "', '" + historyItemDataArray[5] + "');");  
 pst.executeUpdate();  
 }  
}

1. **Администрирование**
2. **Триггеры**

В PL/pgSQL можно создавать триггерные процедуры, которые будут вызываться при изменениях данных или событиях в базе данных. Триггерная процедура создаётся командой CREATE FUNCTION, при этом у функции не должно быть аргументов, а типом возвращаемого значения должен быть trigger (для триггеров, срабатывающих при изменениях данных) или event\_trigger (для триггеров, срабатывающих при событиях в базе). Для триггеров автоматически определяются специальные локальные переменные с именами вида TG\_имя, описывающие условие, повлёкшее вызов триггера.

Синтаксис:

CREATE [ CONSTRAINT ] TRIGGER ***name*** { BEFORE | AFTER | INSTEAD OF } { ***event*** [ OR ... ] }

ON ***table***

[ FROM ***referenced\_table\_name*** ]

[ NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY DEFERRED } ]

[ FOR [ EACH ] { ROW | STATEMENT } ]

[ WHEN ( ***condition*** ) ]

EXECUTE PROCEDURE ***function\_name*** ( ***arguments*** )

where ***event*** can be one of:

INSERT

UPDATE [ OF ***column\_name*** [, ... ] ]

DELETE

TRUNCATE

Пример триггера по событию INSERT или UPDATE в таблицу users. Он вызывает функцию, которая выполняется перед выполнение вызываемой операции.

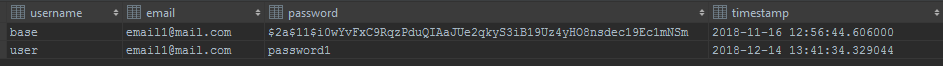
**DROP FUNCTION IF EXISTS** *emp\_stamp*();  
**CREATE FUNCTION** *emp\_stamp*() **RETURNS trigger AS**$$  
**BEGIN**  
 **IF NEW**.username <> 'user' **THEN  
 RAISE EXCEPTION** 'username should be "user"';  
 **END IF**;  
 **IF NEW**.email <> 'email1@mail.com' **THEN  
 RAISE EXCEPTION** 'email % is incorrect', **NEW**.email;  
 **END IF**;  
  
 **IF NEW**.password = 'password' **THEN  
 RAISE EXCEPTION** 'password is too obvious';  
 **END IF**;  
  
 **NEW**.timestamp := *current\_timestamp*;  
 **RETURN NEW**;  
**END**;  
$$ **LANGUAGE** plpgsql;  
  
**DROP TRIGGER IF EXISTS** my\_trigger **ON** users;  
**CREATE TRIGGER** my\_trigger **BEFORE INSERT OR UPDATE ON** users  
 **FOR EACH ROW EXECUTE PROCEDURE** *emp\_stamp*();

Данный триггер будет проверять значение, которое идет в таблицу users, и выдавать исключение (+ прерывать) операцию добавления данных, если username не равно ‘user’ или email не равен ‘email1@mail.com’ или password равен ‘password’.

Также триггер реализует автоматическое заполнение даты/времени добавления записи.

Выполним запросы для проверки:

**insert into** users(username, email, **password**) **values** ('user', 'email1@mail.com', 'password1');



Добавило юзера, прописало ему timestamp.

**insert into** users(username, email, **password**) **values** ('user1', 'email1@mail.com', 'password');



Получаем ошибку.

Когда функция на PL/pgSQL срабатывает как триггер, в блоке верхнего уровня автоматически создаются несколько специальных переменных, например переменная NEW, как в предыдущем примере.

Также триггеры можно использовать для ведения логов.

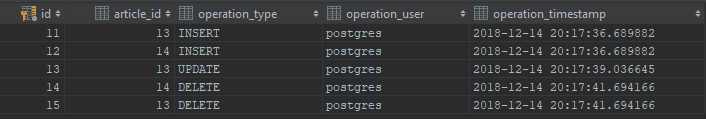
В примере ниже триггер отлавливает insert, update и delete запросы для таблицы articles и записывает их в таблицу article\_logs.

**drop table if exists** article\_logs;  
**create table** article\_logs (  
 id **bigserial primary key**,  
 article\_id **bigint**,  
 operation\_type **text**,  
 operation\_user **text**,  
 operation\_timestamp **timestamp**);  
  
**CREATE OR REPLACE FUNCTION** *article\_logging*() **RETURNS trigger AS**$$  
**BEGIN  
 if** TG\_OP = 'INSERT' **or** TG\_OP = 'UPDATE' **then  
 INSERT INTO** article\_logs(article\_id, operation\_type, operation\_user, operation\_timestamp)  
 **SELECT NEW**.id, TG\_OP, *current\_user*, *now*();  
 **elseif** TG\_OP = 'DELETE' **then  
 INSERT INTO** article\_logs(article\_id, operation\_type, operation\_user, operation\_timestamp)  
 **SELECT OLD**.id, TG\_OP, *current\_user*, *now*();  
 **end if**;  
 **RETURN NULL**;  
**END**;  
$$ **LANGUAGE** plpgsql;  
  
**DROP TRIGGER IF EXISTS** article\_logger **ON** articles;  
**CREATE TRIGGER** article\_logger **AFTER INSERT OR UPDATE OR DELETE ON** articles  
 **FOR EACH ROW EXECUTE PROCEDURE** *article\_logging*();

Для проверки выполним следующие запросы:

**insert into** articles(title) **values** ('test1'), ('test2');  
  
**update** articles **set** title = 'test3' **where** title = 'test1';  
  
**delete from** articles **where** title = 'test1' **or** title = 'test2' **or** title = 'test3';

В результате данные в таблице articles не изменятся, а данные в таблице article\_logs будут следующие:



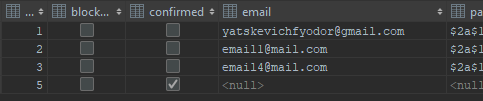
INSTEAD OF

В PostgreSQL ключевое слово INSEAD OF может использоваться только для представлений.

Пример использования INSTEAD OF:

Создадим представление подобное таблице users

**create view** test\_view **as select** \* **from** users;  
  
**insert into** test\_view(username, blocked, confirmed) **values** ('username1', **false**, **true**);  
  
**select** \* **from** test\_view;

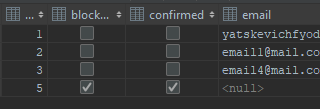


Создадим триггер, который при выполнении операции delete над test\_view, будет прерывать удаление и вместо этого устанавливать поле blocked = TRUE

**CREATE OR REPLACE FUNCTION** *user\_delete\_imitation*() **RETURNS trigger AS**$$  
**BEGIN  
 UPDATE** users **set** blocked = **TRUE where** id = **OLD**.id;  
 **RETURN OLD**;  
**END**;  
$$ **LANGUAGE** plpgsql;  
  
**DROP TRIGGER IF EXISTS** user\_delete\_imitator **ON** test\_view;  
**CREATE TRIGGER** user\_delete\_imitator **INSTEAD OF DELETE ON** test\_view  
 **FOR EACH ROW EXECUTE PROCEDURE** *user\_delete\_imitation*();

Проверим:

**delete from** test\_view **where** id = 5;



Как видно,триггер корректно отработал.

Триггер на шифрование пароля

Подключаем модуль pgcrypto

**create extension** pgcrypto;

Данный модуль обладает следующим функционалом:

- [**Стандартные функции хеширования**](https://postgrespro.ru/docs/postgrespro/9.5/pgcrypto#idp136797)

- [**Функции хеширования пароля**](https://postgrespro.ru/docs/postgrespro/9.5/pgcrypto#idp136833)

- [**Функции шифрования на базе PGP**](https://postgrespro.ru/docs/postgrespro/9.5/pgcrypto#idp137060)

- [**Низкоуровневые функции шифрования**](https://postgrespro.ru/docs/postgrespro/9.5/pgcrypto#idp137284)

- [**Функции получения случайных данных**](https://postgrespro.ru/docs/postgrespro/9.5/pgcrypto#idp137347)

Для простого примера, преобразуем пароль в шестнадцетеричный хэш md5.

digest() - Вычисляет двоичный хеш данных (*data*). Параметр *type* выбирает используемый алгоритм. Поддерживаются стандартные алгоритмы: md5, sha1, sha224, sha256, sha384 и sha512.

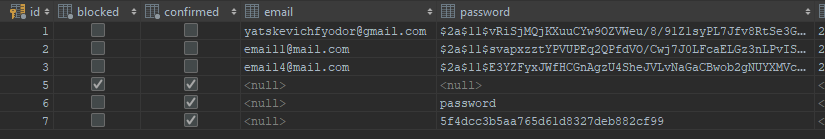
Если вы хотите получить дайджест в виде шестнадцатеричной строки, примените encode() к результату.

Напишем триггер:

**CREATE OR REPLACE FUNCTION** *password\_encryption*() **RETURNS trigger AS**$$  
**BEGIN  
 NEW**.password := *encode*(*digest*(**NEW**.password, 'md5'), 'hex');  
 **RETURN NEW**;  
**END**;  
$$ **LANGUAGE** plpgsql;  
  
**DROP TRIGGER IF EXISTS** password\_encrypter **ON** users;  
**CREATE TRIGGER** password\_encrypter **BEFORE INSERT OR UPDATE ON** users  
 **FOR EACH ROW EXECUTE PROCEDURE** *password\_encryption*();

Проверим как работает:

**insert into** users(username, **password**, confirmed, blocked) **values** ('encTest', 'password', **TRUE**, **FALSE**);



В таблицу была занесена запись с id = 7. Как видно, вместо ‘password’ был занесен хэш.