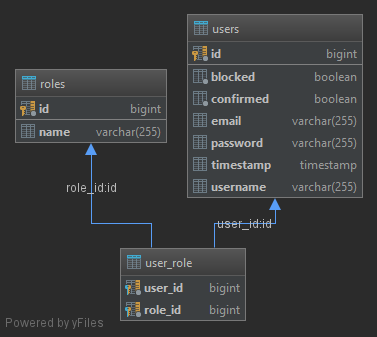
Яцкевич Фёдор, 522403

**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.:

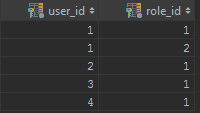
1. Обе СУБД имеют схожий синтаксис запросов
2. В данном приложении используется ORM Framework Hibernate (Spring Data JPA), позволяющий создавать универсальные запросы на языке HQL.
3. **Сложные группировки (GROUPING SETS, ROLLUP, CUBE)**

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.

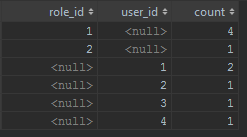
GROUPING SETS

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:



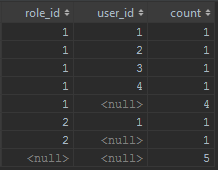
Как видно, для этого выл использован оператор GROUPING SETS в совокупности с GROUP BY.

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), ());

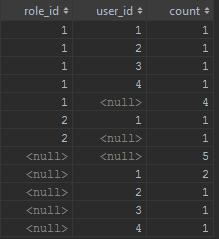
Данный оператор применим для обработки иерархических структур данных. Например подсчитать зарплату для подразделения, отдела, и т.д.

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:

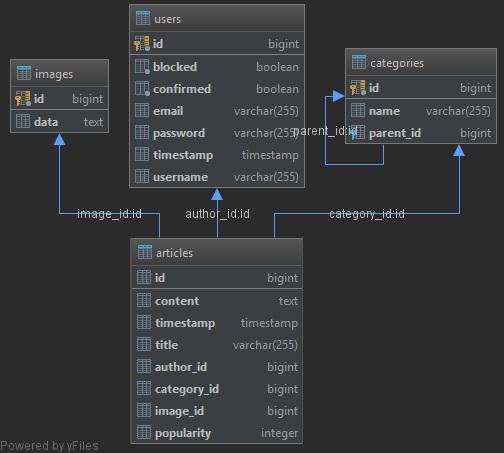
**SELECT** role\_id, user\_id, *COUNT*(\*)  
**FROM** user\_role  
**GROUP BY GROUPING SETS** ((role\_id, user\_id), (role\_id), (user\_id), ());

Он создает группировки по заданному списку и всем его подмножествам.

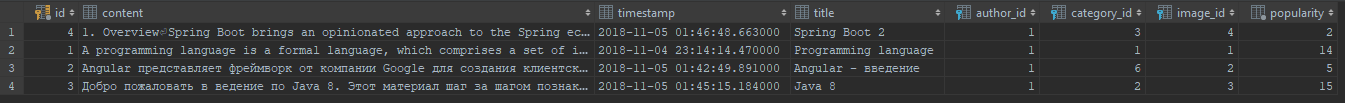
1. **Оконные функции**

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.

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:



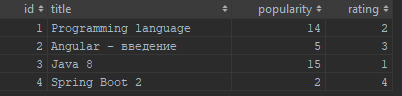
Данные из таблицы articles:



Запрос:

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

Ответ:



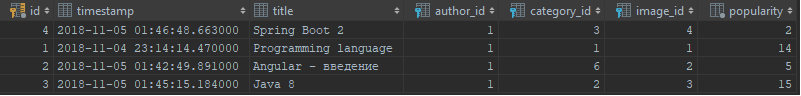
В запросе после оператора OVER следует окно. Окно – это некоторое выражение, описывающее набор строк, которые будет обрабатывать функция и порядок этой обработки. В данном случае в качестве окна указан порядок обработки – по убыванию популярности.

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, т.к. как описано в окне.

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.

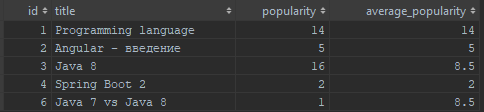
Исходные данные:



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.

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 GROUP 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:

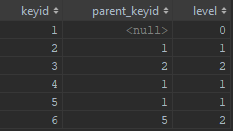
CREATE EXTENSION tablefunc;

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:



Судя по всему, используя данный запрос нельзя получить информацию из других столбцов таблицы, например, столбца name. Разве что используя вложенный запрос.