МГТУ ИМ. Н.Э. БАУМАНА

Базы Данных

Семинар 5

Оконные функции

Семинар начинаем с напоминания, что такое группировка и что можно получить в результате запроса с GROUP BY. (Неплохо также узнать, усвоили ли различия WHERE и HAVING).

Как напоминание. *Предложение **GROUP BY** используется для определения групп выходных строк, к которым могут применяться агрегатные функции (**COUNT**, **MIN**, **MAX**, **AVG** и **SUM**). Если это предложение отсутствует, и используются агрегатные функции, то все столбцы с именами, упомянутыми в **SELECT**, должны быть включены в агрегатные функции, и эти функции будут применяться ко всему набору строк, которые удовлетворяют предикату запроса. В противном случае все столбцы списка **SELECT**, не вошедшие в агрегатные функции, должны быть указаны в предложении **GROUP BY**. В результате чего все выходные строки запроса разбиваются на группы, характеризуемые одинаковыми комбинациями значений в этих столбцах. После чего к каждой группе будут применены агрегатные функции. Следует иметь в виду, что для **GROUP BY** все значения **NULL** трактуются как равные, то есть при группировке по полю, содержащему **NULL**-значения, все такие строки попадут в одну группу.*

Реляционная модель исходит из того факта, что строки в таблице не имеют порядка, являющегося прямым следствием теоретико-множественного подхода. Поэтому наивными выглядят вопросы новичков, спрашивающих: "А как мне получить последнюю добавленную в таблицу строку?" Ответом на вопрос будет "никак", если в таблице не предусмотрен столбец, содержащий дату вставки строки, или не используется последовательная нумерация строк, реализуемая во многих СУБД с помощью столбца с автоинкрементируемым значением. Тогда можно выбрать строку с максимальным значением даты или счетчика.

Теоретически каждая строка запроса обрабатывается независимо от других строк. Однако на практике часто требуется при обработке строки соотносить ее с предыдущими или последующими строками (например, для получения нарастающих итогов), выделять группы строк, обрабатываемые независимо от других и т.д. В ответ на потребности практики в ряде СУБД в языке SQL появились

соответствующие конструкции, в частности, функции ранжирования и оконные (аналитические) функции.

Оконные функции не изменяют выборку, а только добавляют некоторую дополнительную информацию о ней. Т.е. для простоты понимания можно считать, что СУБД сначала выполняет весь запрос (кроме сортировки и limit), а потом только просчитывает оконные выражения.
Синтаксис примерно такой:

```
функция OVER окно
```

Окно — это некоторое выражение, описывающее набор строк, которые будет обрабатывать функция и порядок этой обработки. Причем окно может быть просто задано пустыми скобками (), т.е. окном являются все строки результата запроса.
Например, в этом селекте к обычным полям id, header и score просто добавится нумерация строк.

```
SELECT id, section, header, score,
             row_number() OVER ()  AS num
FROM news;

 id | section |  header   | score | num
----+---------+-----------+-------+-----
  1 |       2 | Заголовок |    23 |   1
  2 |       1 | Заголовок |     6 |   2
  3 |       4 | Заголовок |    79 |   3
  4 |       3 | Заголовок |    36 |   4
  5 |       2 | Заголовок |    34 |   5
  6 |       2 | Заголовок |    95 |   6
  7 |       4 | Заголовок |    26 |   7
  8 |       3 | Заголовок |    36 |   8
```

В оконное выражение можно добавить ORDER BY, тогда можно изменить порядок обработки.

```
SELECT id, section, header, score,

          row_number() OVER (ORDER BY score DESC)   AS
rating
FROM news
ORDER BY id;

 id | section |  header   | score | rating
----+---------+-----------+-------+--------
  1 |       2 | Заголовок |    23 |      7
  2 |       1 | Заголовок |     6 |      8
  3 |       4 | Заголовок |    79 |      2
  4 |       3 | Заголовок |    36 |      4
  5 |       2 | Заголовок |    34 |      5
  6 |       2 | Заголовок |    95 |      1
  7 |       4 | Заголовок |    26 |      6
  8 |       3 | Заголовок |    36 |      3
```

Обратите внимание, что в конце всего запоса ORDER BY id, при этом рейтинг посчитан все равно верно. Т.е. посгрес просто отсортировал результат вместе с результатом работы оконной функции, один order ничуть не мешает другому.

В оконное выражение можно добавить слово PARTITION BY [expression],
например row_number() OVER (PARTITION BY section), тогда подсчет будет идти в каждой группе отдельно:

```
SELECT id, section, header, score,
                row_number() OVER (PARTITION BY section
ORDER BY score DESC)  AS

rating_in_section
FROM news
ORDER BY section, rating_in_section;
```

id	section	header	score	rating_in_section
2	1	Заголовок	6	1
6	2	Заголовок	95	1
5	2	Заголовок	34	2
1	2	Заголовок	23	3
4	3	Заголовок	36	1
8	3	Заголовок	36	2
3	4	Заголовок	79	1
7	4	Заголовок	26	2

Если не указывать партицию, то партицией является весь запрос.
В качестве функции можно использовать, так сказать, истинные оконные функции из мануала — это row_number(), rank(), lead() и т.д., а можно использовать функции-агрегаты, такие как: sum(), count() и т.д. Так вот, это важно, агрегатные функции работают слегка по-другому: если не задан ORDER BY в окне, идет подсчет по всей партиции один раз, и результат пишется во все строки (одинаков для всех строк партиции). Если же ORDER BY задан, то подсчет в каждой строке идет от начала партиции до этой строки.

```
SELECT transaction_id, change
FROM balance_change
ORDER BY transaction_id;

 transaction_id | change
----------------+--------
              1 |   1.00
              2 |  -2.00
              3 |  10.00
              4 |  -4.00
              5 |   5.50
```

и мы хотим узнать заодно, как менялся остаток на балансе при этом:

```
SELECT transaction_id, change,
                            sum(change) OVER (ORDER BY
transaction_id) as balance
FROM balance_change
ORDER BY transaction_id;

 transaction_id | change | balance
----------------+--------+---------
              1 |   1.00 |    1.00
              2 |  -2.00 |   -1.00
              3 |  10.00 |    9.00
              4 |  -4.00 |    5.00
              5 |   5.50 |   10.50
```

Т.е. для каждой строки идет подсчет в отдельном фрейме. В данном случае фрейм — это набор строк от начала до текущей строки (если было бы PARTITION BY, то от начала партиции).

Если же мы для агрегатной фунции sum не будем использовать ORDER BY в окне, тогда мы просто посчитаем общую сумму и покажем её во всех строках. Т.е. фреймом для каждой из строк будет весь набор строк
от начала до конца партиции.

```
SELECT transaction_id, change,
               sum(change) OVER () as result_balance
FROM balance_change
ORDER BY transaction_id;

 transaction_id | change | result_balance
----------------+--------+----------------
              1 |   1.00 |          10.50
              2 |  -2.00 |          10.50
              3 |  10.00 |          10.50
              4 |  -4.00 |          10.50
              5 |   5.50 |          10.50
```

Вот такая особенность агрегатных функций, если их использовать как оконные. На мой взгляд, это довольно-таки странный, интуитивно неочевидный момент SQL-стандарта.

Оконные функции можно использовать сразу по несколько штук, они друг другу ничуть не мешают, чтобы вы там в них не написали.

```
SELECT transaction_id, change,
       sum(change) OVER (ORDER BY transaction_id) as
balance,
    sum(change) OVER () as result_balance,
         round( 100.0 * sum(change) OVER (ORDER BY
transaction_id) / sum(change)

OVER (), 2) AS percent_of_result,
    count(*) OVER () as transactions_count
FROM balance_change
ORDER BY transaction_id;

transaction_id | change | balance | result_balance |
percent_of_result | transactions_count
----------------+--------+---------+----------------
+------------------+--------------------
               1 |   1.00 |    1.00 |          10.50 |
9.52 |                  5
               2 |  -2.00 |   -1.00 |          10.50 |
-9.52 |                  5
               3 |  10.00 |    9.00 |          10.50 |
85.71 |                  5
               4 |  -4.00 |    5.00 |          10.50 |
47.62 |                  5
               5 |   5.50 |   10.50 |          10.50 |
100.00 |                  5
```

Если у вас много одинаковых выражений после OVER, то можно дать им имя и вынести отдельно с ключевым словом WINDOW, чтобы избежать дублирования кода. Вот пример из мануала:

```
SELECT
    sum(salary) OVER w,
    avg(salary) OVER w
FROM empsalary
WINDOW w AS (PARTITION BY depname ORDER BY salary
DESC);
```

Здесь w после слова OVER идет без уже скобок.

Результат работы оконной функции невозможно отфильтровать в запросе с помощью WHERE, потому что оконные фунции выполняются после всей фильтрации и группировки, т.е. с тем, что получилось. Поэтому чтобы выбрать, например, топ 5 новостей в каждой группе, надо использовать подзапрос:

```
SELECT *
FROM (
    SELECT id, section, header, score,
                        row_number() OVER (PARTITION BY
section ORDER BY score DESC)  AS

rating_in_section
    FROM news
    ORDER BY section, rating_in_section
) counted_news
WHERE rating_in_section <= 5;
```

Еще пример для закрепления. Помимо row_number() есть несколько других функций. Например lag, которая ищет строку перед последней строкой фрейма. К примеру мы можем найти насколько очков новость отстает от предыдущей в рейтинге:

```
SELECT id, section, header, score,
               row_number() OVER w AS rating,
               lag(score) OVER w - score AS score_lag
FROM news
WINDOW w AS (ORDER BY score DESC)
ORDER BY score desc;

 id | section |  header   | score | rating | score_lag
----+---------+-----------+-------+--------+-----------
  6 |       2 | Заголовок |    95 |      1 |
  3 |       4 | Заголовок |    79 |      2 |        16
  8 |       3 | Заголовок |    36 |      3 |        43
  4 |       3 | Заголовок |    36 |      4 |         0
  5 |       2 | Заголовок |    34 |      5 |         2
  7 |       4 | Заголовок |    26 |      6 |         8
  1 |       2 | Заголовок |    23 |      7 |         3
  2 |       1 | Заголовок |     6 |      8 |        17
```

Прошу в коментариях накидать примеров, где особенно удобно применять оконные фунции. А также, какие с ними могут возникнуть проблемы, если таковые имеются.

Обобщенное табличное выражение

Common Table Expression (CTE) или **обобщенное табличное выражение (ОТВ)** - это временные результирующие наборы (т.е. результаты выполнения SQL запроса), которые не сохраняются в базе данных в виде объектов, но к ним можно обращаться.

Главной особенностью обобщенных табличных выражений является то, что с помощью них можно писать рекурсивные запросы. Случаи в которых может пригодиться использование ОТВ:

- Основной целью ОТВ является написание рекурсивных запросов, можно сказать для этого они, и были созданы;
- ОТВ можно использовать также и для замены представлений (VIEW), например, в тех случаях, когда нет необходимости сохранять в базе SQL запрос представления, т.е. его определение;

- Обобщенные табличные выражения повышают читаемость кода путем разделения запроса на логические блоки, и тем самым упрощают работу со сложными запросами;
- Также ОТВ предназначены и для многократных ссылок на результирующий набор из одной и той же SQL инструкции.

Обобщенное табличное выражение определяется с помощью конструкции WITH, и определить его можно как в обычных запросах, так и в функциях, хранимых процедурах, триггерах и представлениях.

Синтаксис:

```
WITH  [ ( column_name [ ,...n ] ) ]
 AS
  ( CTE_query_definition )
```

Где,

- common_table_expression_name – это псевдоним или можно сказать идентификатор обобщенного табличного выражения. Обращаться к ОТВ мы будем, как раз используя этот псевдоним;
- column_name – имя столбца, который будет определен в обобщенном табличном выражении. Использование повторяющихся имен нельзя, а также их количество должно совпадать с количеством столбцов возвращаемых запросом CTE_query_definition. Указывать имена столбцов необязательно, но только в том случае, если всем столбцам в запросе CTE_query_definition присвоены уникальные псевдонимы;
- CTE_query_definition - запрос SELECT, к результирующему набору которого, мы и будем обращаться через обобщенное табличное выражение, т.е. common_table_expression_name.

После обобщенного табличного выражения, т.е. сразу за ним должен идти одиночный запрос SELECT, INSERT, UPDATE, MERGE или DELETE.

Какие бывают обобщенные табличные выражения?

Они бывают простые и рекурсивные.

Простые ОТВ не включают ссылки на самого себя.

```
WITH TestCTE (UserID, Post, ManagerID) AS
(
      SELECT UserID, Post, ManagerID
      FROM TestTable
)
 SELECT * FROM TestCTE
```

В данном случае мы могли и не перечислять имена столбцов, так как они у нас уникальны. Можно было просто написать вот так:

```
WITH TestCTE AS
(     SELECT UserID, Post, ManagerID
      FROM TestTable
)
 SELECT * FROM TestCTE
```

Рекурсивные ОТВ используются для возвращения иерархических данных, например, классика жанра это отображение сотрудников в структуре организации (чуть ниже мы это рассмотрим).

Теперь допустим, что нам необходимо вывести иерархический список сотрудников, т.е. мы хотим видеть, на каком уровне работает тот или иной сотрудник. Для этого пишем рекурсивный запрос:

```
WITH TestCTE(UserID, Post, ManagerID, LevelUser)
 AS
 (
      -- Находим якорь рекурсии
      SELECT UserID, Post, ManagerID, 0 AS LevelUser
      FROM TestTable WHERE ManagerID IS NULL
      UNION ALL
      --Делаем объединение с TestCTE (хотя мы его еще не дописали)
      SELECT t1.UserID, t1.Post, t1.ManagerID, t2.LevelUser + 1
      FROM TestTable t1
      JOIN TestCTE t2 ON t1.ManagerID=t2.UserID
 )
 SELECT * FROM TestCTE ORDER BY LevelUser
```

В итоге, если мы захотим, мы можем легко получить список сотрудников определенного уровня, например, нам нужны только начальники отделов, для этого мы просто в указанный выше запрос добавим условие WHERE LevelUser = 1

При написании рекурсивного ОТВ нужно быть внимательным, так как неправильное его составление может привести к бесконечному циклу. Поэтому для этих целей есть опция MAXRECURSION, которая может ограничивать количество уровней рекурсии. Давайте представим, что мы не уверены, что написали рекурсивное обобщенное выражение правильно и для отладки применим после запроса инструкцию:

```
OPTION (MAXRECURSION 5)
```

Если запрос отработал корректно, значит проблем с рекурсией не возникло, запрос не был перван данной опцией.