ЛЕКЦИЯ 6

Стандарт SQL

Массивы

- (будем рассматривать реализацию в PostgreSQL, т.к. общую документацию на все СУБД найти непросто)
- Объявление столбца как массив:

```
CREATE TABLE students (
    name text,
    scores int[],
    contacts varchar ARRAY -- or varchar[]
);
```

Ввод массива

- Константа массива (первичный ввод):

  '{ значение1, значение2, ... }‘

- Массивы могут быть многомерными

  '{{1,2,3},{4,5,6},{7,8,9}}'

- Пример:

```
INSERT INTO sal_emp
VALUES ('Carol',
'{20000, 25000, 25000, 25000}',
'{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

Работа с массивом

- Нумерация массива начинается с 1
- Например, есть таблица с сотрудниками и номерами телефонов

```
Demo=# SELECT * FROM Employees;
 id |    name     |              contact
----+-------------+-----------------------------------
  1 | Alice John  | {(408)-743-9045,(408)-567-7834}
  2 | Kate Joel   | {(408)-783-5731}
  3 | James Bush  | {(408)-745-8965,(408)-567-78234}
(3 rows)

Demo=#
```

- Тогда для вывода сотрудника и первого номера телефона нужно написать:

```
SELECT name, contact[1] FROM Employees;
```

```
Demo=# SELECT name, contact[1]
Demo-# FROM Employees;
    name     |    contact
-------------+----------------
 Alice John  | (408)-743-9045
 Kate Joel   | (408)-783-5731
 James Bush  | (408)-745-8965
(3 rows)


Demo=#
```

Изменение массива

- Заменить значения массива полностью:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}‘
WHERE name = 'Carol';
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
WHERE name = 'Carol';
```

- Изменить один элемент или срез

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
WHERE name = 'Bill';
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
WHERE name = 'Carol';
```

Изменение массива

- Значение можно сконструировать с помощью оператора конкатенации

```
SELECT ARRAY[1,2] || ARRAY[3,4];

SELECT my_array || ‘{1}’ FROM my_table;
```

- Оператор конкатенации позволяет вставить один элемент в начало или в конец одномерного массива
- Можно использовать функцию из postgresql

```
UPDATE table1

SET integer_array = array_append(integer_array, 5);
```

Поиск в массиве

- Можно искать по конкретному элементу массива

```
SELECT * FROM sal_emp
WHERE pay_by_quarter[1] = 10000 OR
      pay_by_quarter[2] = 10000;
```

- Можно искать по любому из (хотя бы одно):

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

- Поиск по всем элементам

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

unnest

- unnest – функция, позволяющая «развернуть» массив в набор строк

```
Demo=# SELECT * FROM Employees;
 id |    name    |                 contact
----+------------+-----------------------------------
  1 | Alice John | {(408)-743-9045,(408)-567-7834}
  2 | Kate Joel  | {(408)-783-5731}
  3 | James Bush | {(408)-745-8965,(408)-589-89347}
(3 rows)
```

```
Demo=# SELECT
Demo-#     name,
Demo-#     unnest(contact)
Demo-# FROM
Demo-#     Employees;
    name    |     unnest
------------+-----------------
 Alice John | (408)-743-9045
 Alice John | (408)-567-7834
 Kate Joel  | (408)-783-5731
 James Bush | (408)-745-8965
 James Bush | (408)-589-89347
(5 rows)

Demo=#
```

ROLLUP, CUBE и GROUPING SETS

- Операторы, расширяющие GROUP BY
- Часть OLAP (*online analytical processing*, интерактивная аналитическая обработка)- технология обработки данных, заключающаяся в подготовке суммарной (агрегированной) информации на основе больших массивов данных, структурированных по многомерному принципу
- Нужны для формирования отчетов, как со строкой общего итога, так и со строками промежуточных итогов
- Для начала вспомним, как работает GROUP BY

Дана таблица

- Владелец, тип сущности и название

```
student=> select * from objects;
 id | owner  |   type   |           name
----+--------+----------+--------------------------
  1 | user_1 | table    | region
  2 | user_1 | table    | job
  3 | user_1 | table    | link
  4 | user_1 | sequence | region_id_seq
  5 | user_1 | sequence | job_id_seq
  6 | user_1 | sequence | link_id_seq
  7 | user_1 | index    | pk_region
  8 | user_1 | index    | pk_link
  9 | user_1 | index    | ix_link_hash
 10 | user_1 | index    | pk_job
 11 | user_2 | table    | settings_template
 12 | user_2 | table    | error_pattern
 13 | user_2 | sequence | settings_template_id_seq
 14 | user_2 | index    | pk_error_pattern
 15 | user_2 | index    | pk_settings_template
 16 | user_3 | table    | test_table_1
(16 rows)
```

Запросы 1-3

```
student=> SELECT
student->     COUNT(*) total,                    -- сколько всего объектов
student->     COUNT(DISTINCT owner) owner_count,      -- скольким владельцам они принадлежат
student->     COUNT(DISTINCT type) type_count  -- к скольким разным типам они относятся
student-> FROM objects;
 total | owner_count | type_count
-------+-------------+------------
    16 |           3 |          3
(1 row)

student=> -- запрос 2: сколько объектов у каждого владельца
student=> SELECT owner, COUNT(*) ownerobj_count
student-> FROM objects
student-> GROUP BY owner;
 owner  | ownerobj_count
--------+----------------
 user_3 |              1
 user_2 |              5
 user_1 |             10
(3 rows)

student=> -- запрос 3: сколько объектов каждого типа
student=> SELECT type, COUNT(*) typeobj_count
student-> FROM objects
student-> GROUP BY type;
   type   | typeobj_count
----------+---------------
 sequence |             4
 index    |             6
 table    |             6
(3 rows)
```

Запросы 4-6

```
student=> -- запрос 4 -- сколько объектов каждого типа у каждого владельца
student=> SELECT owner, type, COUNT(*) obj_count
student-> FROM objects
student-> GROUP BY owner, type
student-> ORDER BY owner, type;
 owner  |   type    | obj_count
--------+-----------+-----------
 user_1 | index     |         4
 user_1 | sequence  |         3
 user_1 | table     |         3
 user_2 | index     |         2
 user_2 | sequence  |         1
 user_2 | table     |         2
 user_3 | table     |         1
(7 rows)
```

```
student=> -- запрос 5 -- сколько объектов каждого типа у каждого владельца
student=> SELECT type, owner, COUNT(*)
student-> FROM objects
student-> GROUP BY type, owner
student-> ORDER BY type, owner;
   type    | owner  | count
-----------+--------+-------
 index     | user_1 |     4
 index     | user_2 |     2
 sequence  | user_1 |     3
 sequence  | user_2 |     1
 table     | user_1 |     3
 table     | user_2 |     2
 table     | user_3 |     1
(7 rows)
```

```
student=> -- та же самая информация, но в другом порядке вывода
student=> -- можно записать иначе
student=> -- запрос 6
student=> SELECT owner, type, COUNT(*)
student-> FROM objects
student-> GROUP BY type, owner
student-> ORDER BY owner, type;
 owner  |   type    | count
--------+-----------+-------
 user_1 | index     |     4
 user_1 | sequence  |     3
 user_1 | table     |     3
 user_2 | index     |     2
 user_2 | sequence  |     1
 user_2 | table     |     2
 user_3 | table     |     1
(7 rows)
```

Предварительные выводы:

- от перестановки элементов в списке GROUP BY результат не изменяется
- Были написаны и выполнены 4 разных предложения SELECT, чтобы собрать информацию о
  - *общем кол-ве объектов (запрос 1)*
  - *кол-ве владельцев (запрос 1)*
  - *кол-ве типов объектов (запрос 1)*
  - *кол-ве объектов у каждого владельца (запрос 2)*
  - *кол-ве объектов каждого типа (запрос 3)*
  - *кол-ве объектов каждого типа у каждого владельца (запрос 4)*
- Это вообще все вопросы о кол-ве, которые можно задать, рассматривая объекты в разрезе двух выбранных признаков, owner и type, и их сочетаний

ROLLUP

- ROLLUP формирует промежуточные итоги для каждого указанного элемента и общий итог

```
student=> -- запрос 7 -- сколько объектов у каждого владельца
student=> SELECT owner, COUNT(*) ownerobj_count
student-> FROM objects
student-> GROUP BY ROLLUP(owner);
 owner   | ownerobj_count
---------+----------------
         |             16
 user_3  |              1
 user_2  |              5
 user_1  |             10
(4 rows)
```

Добавилась строка с общим количеством объектов.
То есть, одним запросом получены данные, для получения которых ранее понадобилось два запроса, а именно, запросы 1 и 2.

Запрос 8

```
student=> -- запрос 8
student=> SELECT owner, type, COUNT(*) obj_count
student-> FROM objects
student-> GROUP BY ROLLUP(owner, type)
student-> ORDER BY owner, type;
 owner  |   type   | obj_count
--------+----------+-----------
 user_1 | index    |         4
 user_1 | sequence |         3
 user_1 | table    |         3
 user_1 |          |        10
 user_2 | index    |         2
 user_2 | sequence |         1
 user_2 | table    |         2
 user_2 |          |         5
 user_3 | table    |         1
 user_3 |          |         1
        |          |        16
(11 rows)
```

Добавились строки с количеством объектов у каждого владельца и общим количеством объектов.
То есть, одним запросом получены данные, для получения которых ранее понадобилось три запроса, а именно, запросы 1, 2 и 4.

Записать предыдущий запрос иначе

```
student=> -- запрос 9
student=> SELECT NULL, NULL, COUNT(*) FROM objects  -- сколько всего объектов
student-> UNION ALL
student-> SELECT owner, NULL, COUNT(*) -- сколько у каждого владельца
student-> FROM objects GROUP BY owner
student-> UNION ALL
student-> SELECT owner, type, COUNT(*) -- сколько объектов каждого типа у каждого владельца
student-> FROM objects GROUP BY owner, type
student-> ORDER BY 1, 2;
 ?column? | ?column? | count
----------+----------+-------
 user_1   | index    |     4
 user_1   | sequence |     3
 user_1   | table    |     3
 user_1   |          |    10
 user_2   | index    |     2
 user_2   | sequence |     1
 user_2   | table    |     2
 user_2   |          |     5
 user_3   | table    |     1
 user_3   |          |     1
          |          |    16
(11 rows)
```

Поменяем порядок столбцов в списке ROLLUP

```
student=> -- запрос 10
student=> SELECT owner, type, COUNT(*)
student-> FROM objects
student-> GROUP BY ROLLUP(type, owner)
student-> ORDER BY owner, type;
 owner  |   type   | count
--------+----------+-------
 user_1 | index    |     4
 user_1 | sequence |     3
 user_1 | table    |     3
 user_2 | index    |     2
 user_2 | sequence |     1
 user_2 | table    |     2
 user_3 | table    |     1
        | index    |     6
        | sequence |     4
        | table    |     6
        |          |    16
(11 rows)
```

- вместо промежуточных итоговых кол-в объектов у каждого владельца, мы получили промежуточные итоговые кол-ва объектов каждого типа
- запрос 8 формирует промежуточные итоговые кол-ва в разрезе владельцев, потому что столбец owner в списке параметров ROLLUP идет первым
- запрос 10 формирует промежуточные итоговые кол-ва в разрезе типов объектов, поскольку в списке параметров ROLLUP идет первым столбец type
- перестановка столбцов в списке параметров ROLLUP, в отличие от списка GROUP BY, влияет на результат

Возникает вопрос

- А можно ли одним запросом (не прибегая к UNION ALL) получить
  - *общее количество объектов,*
  - *промежуточные итоговые кол-ва в разрезе владельцев,*
  - *промежуточные итоговые кол-ва в разрезе типов объектов,*
  - *кол-ва объектов каждого типа у каждого владельца?*

CUBE

- CUBE формирует результаты для всех возможных перекрестных вычислений

```
student=> SELECT owner, type, COUNT(*)
student-> FROM objects
student-> GROUP BY CUBE(type, owner);
 owner  |   type   | count
--------+----------+-------
        |          |    16
 user_1 | table    |     3
 user_2 | sequence |     1
 user_1 | index    |     4
 user_2 | index    |     2
 user_1 | sequence |     3
 user_3 | table    |     1
 user_2 | table    |     2
        | sequence |     4
        | index    |     6
        | table    |     6
 user_3 |          |     1
 user_2 |          |     5
 user_1 |          |    10
(14 rows)
```

Про CUBE

- В случае CUBE, от порядка столбцов в списке результат снова не зависит. И для (owner, type) и для (type, owner) множество всех возможных подмножеств (подгрупп) одно и то же:
  - *{пустое множество} - соответствует общему итоговому кол-ву объектов*
  - *{owner} - соответствует кол-ву объектов в разрезе владельцев*
  - *{type} - соответствует кол-ву объектов в разрезе типов объектов*
  - *{owner, type} - соответствует кол-ву объектов в разрезе (владелец, тип объекта)*

Итого:

- GROUP BY выполняет агрегирование данных в разрезе списка столбцов
- ROLLUP выполняет агрегирование данных в разрезе некоторых (не всех) подмножеств множества столбцов, причем выбор работающих подмножеств определяется порядком следования столбцов в списке
- CUBE выполняет агрегирование данных в разрезе всех подмножеств множества столбцов

GROUPING SETS

```
student=> -- запрос 12
student=> SELECT owner, type, COUNT(*)
student-> FROM objects
student-> GROUP BY GROUPING SETS ((type), (owner));
 owner  |   type   | count
--------+----------+-------
        | sequence |     4
        | index    |     6
        | table    |     6
 user_3 |          |     1
 user_2 |          |     5
 user_1 |          |    10
(6 rows)
```

GROUPING SETS - формирует результаты нескольких группировок в один набор данных, другими словами, он эквивалентен конструкции UNION ALL к указанным группам

```
student=> -- запрос 13
student=> SELECT owner, COUNT(*) FROM objects GROUP BY owner
student-> UNION
student-> SELECT type, COUNT(*) FROM objects GROUP BY type;
  owner   | count
----------+-------
 user_3   |     1
 user_2   |     5
 table    |     6
 index    |     6
 sequence |     4
 user_1   |    10
(6 rows)
```

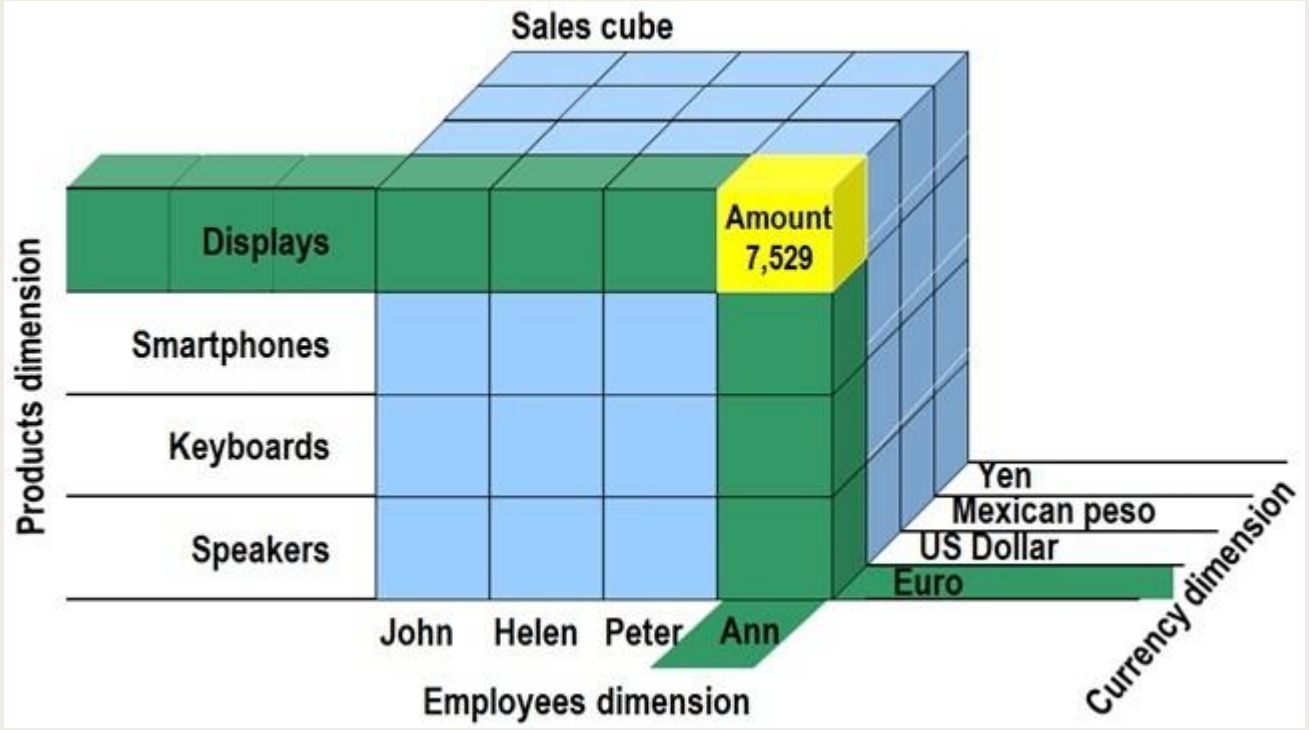
GROUPING SETS

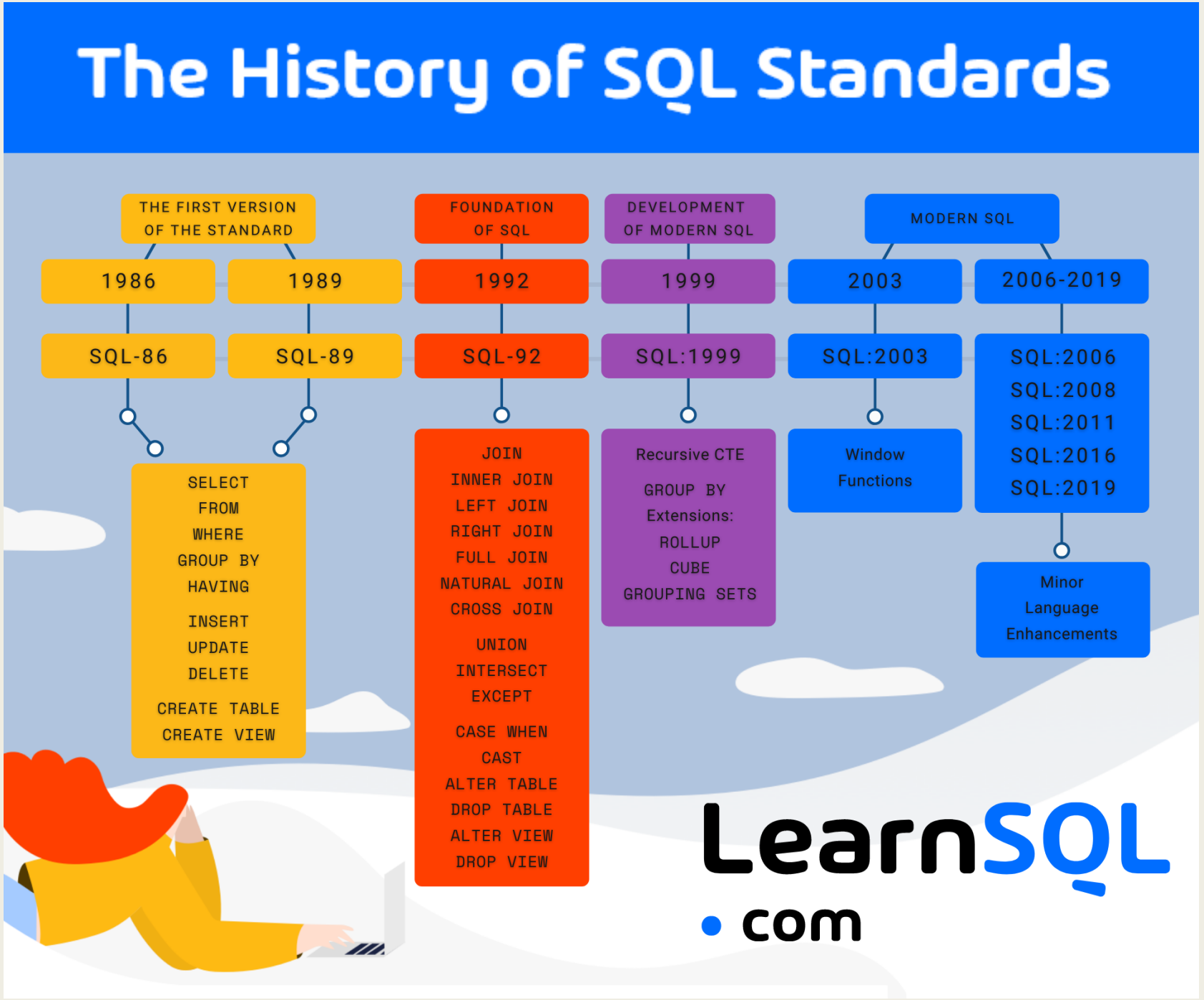
- Для того, чтобы включить в результат запроса общее итоговое кол-во объектов в таблице, в качестве списка нужно указать пустое множество ()

```
student=> -- запрос 14
student=> SELECT owner, type, COUNT(*)
student-> FROM objects
student-> GROUP BY GROUPING SETS ((type), (owner), ());
 owner  |   type   | count
--------+----------+-------
        |          |    16
        | sequence |     4
        | index    |     6
        | table    |     6
 user_3 |          |     1
 user_2 |          |     5
 user_1 |          |    10
(7 rows)
```

ROLLUP, CUBE и GROUPING SETS

- Используются для быстрого и более лёгкого анализа данных
- Чаще всего применяются в разного рода отчётах и аналитике




The History of SQL Standards
THE FIRST VERSION OF THE STANDARD
1986
1989
SQL-86
SQL-89
SELECT
FROM
WHERE
GROUP BY
HAVING
INSERT
UPDATE
DELETE
CREATE TABLE
CREATE VIEW
FOUNDATION OF SQL
1992
SQL-92
JOIN
INNER JOIN
LEFT JOIN
RIGHT JOIN
FULL JOIN
NATURAL JOIN
CROSS JOIN
UNION
INTERSECT
EXCEPT
CASE WHEN
CAST
ALTER TABLE
DROP TABLE
ALTER VIEW
DROP VIEW
DEVELOPMENT OF MODERN SQL
1999
SQL:1999
Recursive CTE
GROUP BY Extensions:
ROLLUP
CUBE
GROUPING SETS
MODERN SQL
2003
SQL:2003
Window Functions
2006-2019
SQL:2006
SQL:2008
SQL:2011
SQL:2016
SQL:2019
Minor Language Enhancements
LearnSQL.com

SQL:2003

- опубликован 1 марта 2004 года
- добавлены оконные функции (Window functions)
- изменения в системе типов данных (исключена поддержка битовых строк BIT и BIT VARYING, добавлен конструктор типов мультимножеств (MULTISET), новый тип BIGINT и тип XML)
- генераторы последовательностей (sequence generators)
- идентифицирующие столбцы (identity columns) и генерируемые столбцы (generated columns)
- оператор MERGE для обновления базы (объединение шагов INSERT и UPDATE при обновлении таблицы)

Оконные функции (пример)

employees

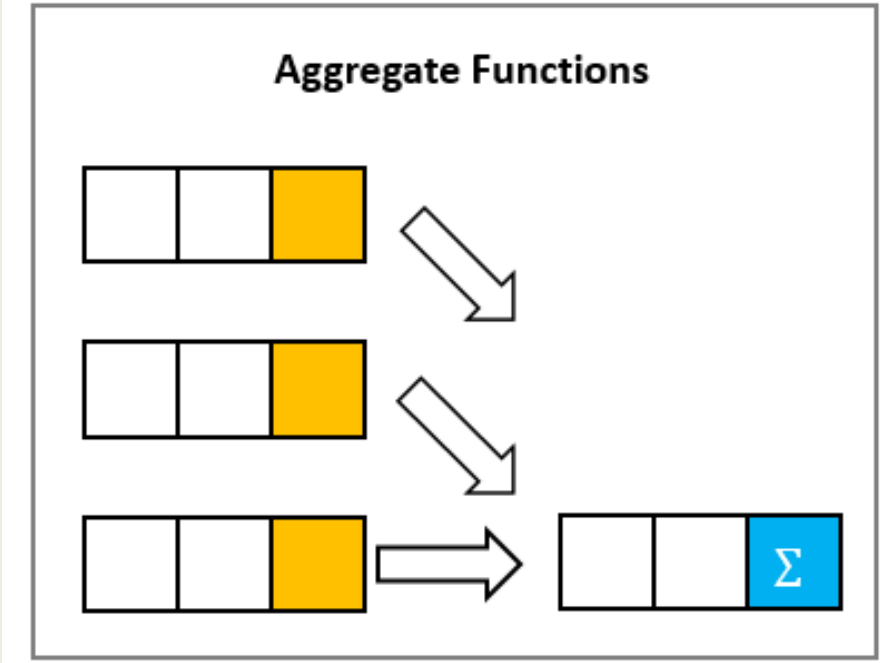
last_name	salary	department
Jones	45000	Accounting
Adams	50000	Sales
Johnson	40000	Marketing
Williams	37000	Accounting
Smith	55000	Sales

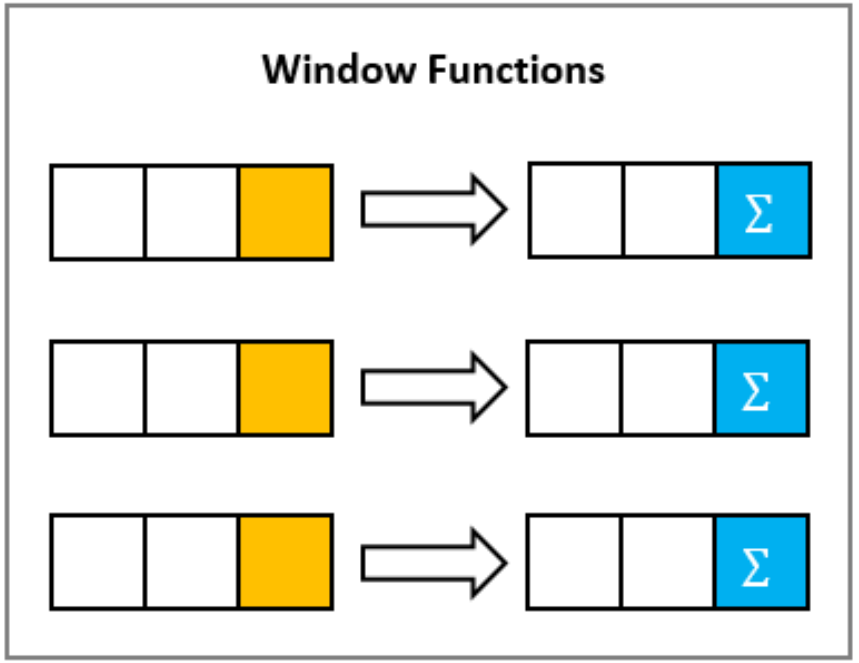
```
SELECT last_name,
       salary,
       department,
       rank() OVER (
        PARTITION BY department
        ORDER BY salary
        DESC
       )
FROM employees;

last_name    salary   department    rank
Jones        45000    Accounting    1
Williams     37000    Accounting    2
Smith        55000    Sales         1
Adams        50000    Sales         2
Johnson      40000    Marketing     1
```

Оконные функции (window functions)

- оконные функции — это не то же самое, что GROUP BY
- они не уменьшают количество строк и не изменяют выборку, а только добавляют некоторую дополнительную информацию о ней.
- можно считать, что SQL сначала выполняет весь запрос (кроме сортировки и limit), а уже потом считает значения окна
- синтаксис: функция OVER окно
- окно — это некоторое выражение, описывающее набор строк, которые будет обрабатывать функция и порядок этой обработки
- окно может быть задано пустыми скобками (), т.е. окном являются все строки результата запроса


Aggregate Functions
Σ


Window Functions
Σ
Σ
Σ

Добавим нумерацию строк

```
student=> SELECT id, owner, type, name, row_number() OVER () AS num FROM objects;
 id | owner  |   type   |           name           | num
----+--------+----------+--------------------------+-----
  1 | user_1 | table    | region                   |   1
  2 | user_1 | table    | job                      |   2
  3 | user_1 | table    | link                     |   3
  4 | user_1 | sequence | region_id_seq            |   4
  5 | user_1 | sequence | job_id_seq               |   5
  6 | user_1 | sequence | link_id_seq              |   6
  7 | user_1 | index    | pk_region                |   7
  8 | user_1 | index    | pk_link                  |   8
  9 | user_1 | index    | ix_link_hash             |   9
 10 | user_1 | index    | pk_job                   |  10
 11 | user_2 | table    | settings_template        |  11
 12 | user_2 | table    | error_pattern            |  12
 13 | user_2 | sequence | settings_template_id_seq |  13
 14 | user_2 | index    | pk_error_pattern         |  14
 15 | user_2 | index    | pk_settings_template     |  15
 16 | user_3 | table    | test_table_1             |  16
(16 rows)
```

row_number() - номер текущей строки в её разделе, начиная с 1

ORDER BY

```
student=> SELECT id, owner, type, name, row_number() OVER (ORDER BY name DESC) AS num
student-> FROM objects
student-> ORDER BY id;
 id | owner  |   type   |           name            | num
----+--------+----------+---------------------------+-----
  1 | user_1 | table    | region                    |   5
  2 | user_1 | table    | job                       |  14
  3 | user_1 | table    | link                      |  12
  4 | user_1 | sequence | region_id_seq             |   4
  5 | user_1 | sequence | job_id_seq                |  13
  6 | user_1 | sequence | link_id_seq               |  11
  7 | user_1 | index    | pk_region                 |   7
  8 | user_1 | index    | pk_link                   |   8
  9 | user_1 | index    | ix_link_hash              |  15
 10 | user_1 | index    | pk_job                    |   9
 11 | user_2 | table    | settings_template         |   3
 12 | user_2 | table    | error_pattern             |  16
 13 | user_2 | sequence | settings_template_id_seq  |   2
 14 | user_2 | index    | pk_error_pattern          |  10
 15 | user_2 | index    | pk_settings_template      |   6
 16 | user_3 | table    | test_table_1              |   1
(16 rows)
```

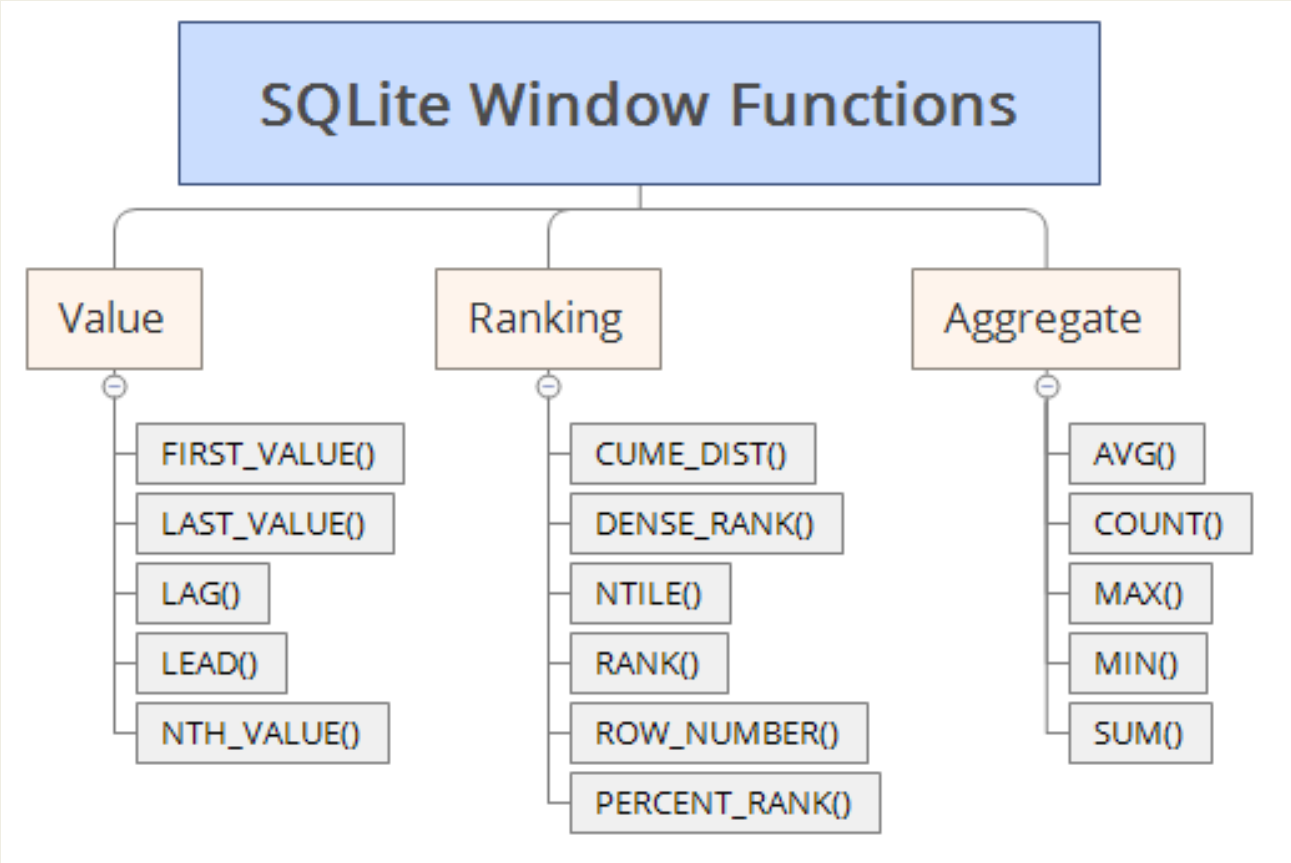
При добавлении ORDER BY меняется порядок обработки строк. Если не добавлять в конце ORDER BY, то столбцы будут упорядочены по num

PARTITION BY

- в оконное выражение можно добавить слово PARTITION BY [expression]
- например row_number() OVER (PARTITION BY owner), тогда подсчет будет идти в каждой группе отдельно
- если не указывать партицию, то партицией является весь запрос

```
student=> SELECT id, owner, type, name, row_number() OVER (PARTITION BY owner) AS num
FROM objects;
 id | owner  |   type   |           name           | num
----+--------+----------+--------------------------+-----
  1 | user_1 | table    | region                   |   1
  2 | user_1 | table    | job                      |   2
  3 | user_1 | table    | link                     |   3
  4 | user_1 | sequence | region_id_seq            |   4
  5 | user_1 | sequence | job_id_seq               |   5
  6 | user_1 | sequence | link_id_seq              |   6
  7 | user_1 | index    | pk_region                |   7
  8 | user_1 | index    | pk_link                  |   8
  9 | user_1 | index    | ix_link_hash             |   9
 10 | user_1 | index    | pk_job                   |  10
 11 | user_2 | table    | settings_template        |   1
 12 | user_2 | table    | error_pattern            |   2
 13 | user_2 | sequence | settings_template_id_seq |   3
 14 | user_2 | index    | pk_error_pattern         |   4
 15 | user_2 | index    | pk_settings_template     |   5
 16 | user_3 | table    | test_table_1             |   1
(16 rows)
```

Оконные функции



value window functions

- first_value(имя_столбца) - значение, вычисленное для первой строки в рамке окна
- last_value(имя_столбца) - значение, вычисленное для последней строки в рамке окна
- lag(имя_столбца, смещение) - возвращает значение для строки, положение которой задаётся смещением от текущей строки к концу раздела
- lead(имя_столбца, смещение) - возвращает значение для строки, положение которой задаётся смещением от текущей строки к началу раздела
- nth_value(имя_столбца, N) - возвращает значение, вычисленное в N-ой строке в рамке окна (считая с 1), или NULL, если такой строки нет

ranking window function

- row_number() - номер текущей строки в её разделе, начиная с 1
- Rank(), dense_rank(), percent_rank() и т.д. – разные виды ранжирования
- Rank() - row_number для первой родственной ей строки
- Percent_rank() - относительный ранг текущей строки: (rank - 1) / (общее число строк - 1)

```
SELECT
        c,
        RANK () OVER (
                ORDER BY c
        ) rank_number
FROM
        ranks;
```

	c character varying (10)	rank_number bigint
1	A	1
2	A	1
3	B	3
4	B	3
5	B	3
6	C	6
7	E	7

Агрегатные функции

- 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 ORDER BY задан, то подсчет в каждой строке идет от начала партиции до этой строки

Пример

```
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
```

```
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
```

```
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
```

Область применения

1. Оконные функции удобно применять для аналитики, отчетов и т.д.

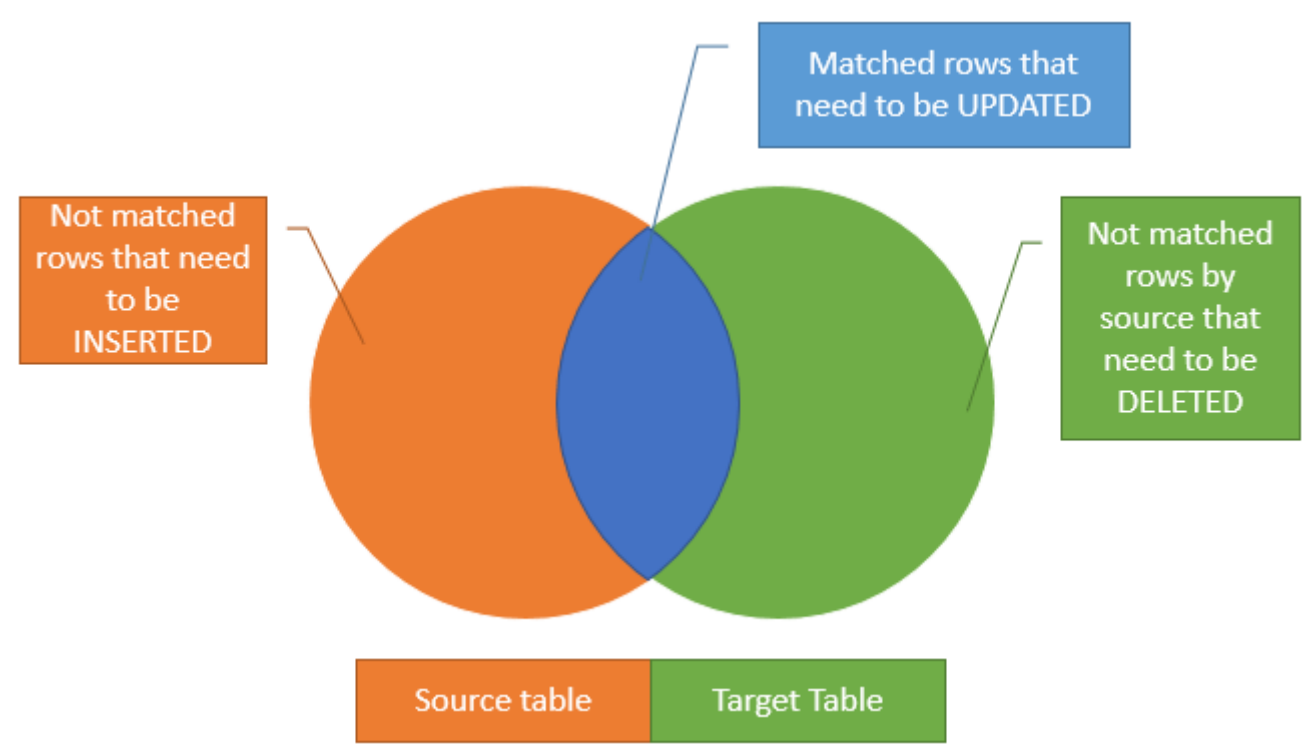
2. Использование для миграции данных: удаление дубликатов, перенумерование записей

3. Оконные функции first/last_value удобны при работе с версионированными или хронологическими данными

Задача

- Дано: 2 таблицы с одинаковыми полями и разными данными. Допустим ФИО и возраст
- Нужно: объединить две таблицы в одну.
- Проблема: что делать с теми личностями, которые имеются в обеих таблицах
- Скорее всего мы захотим, чтобы в итоговой таблице оказались все, а совпадающим личностям обновить информацию.
- Задачу можно решить с помощью сложных запросов, триггеров, но теперь для этого придумали MERGE

MERGE (SQL Server)




```
MERGE target_table USING source_table
ON merge_condition
WHEN MATCHED
    THEN update_statement
WHEN NOT MATCHED
    THEN insert_statement
WHEN NOT MATCHED BY SOURCE
    THEN DELETE;
```

PostgreSQL и стандарт

- Поддерживаемые возможности

https://postgrespro.ru/docs/postgresql/13/features-sql-standard

- Неподдерживаемые возможности

https://postgrespro.ru/docs/postgresql/13/unsupported-features-sql-standard