



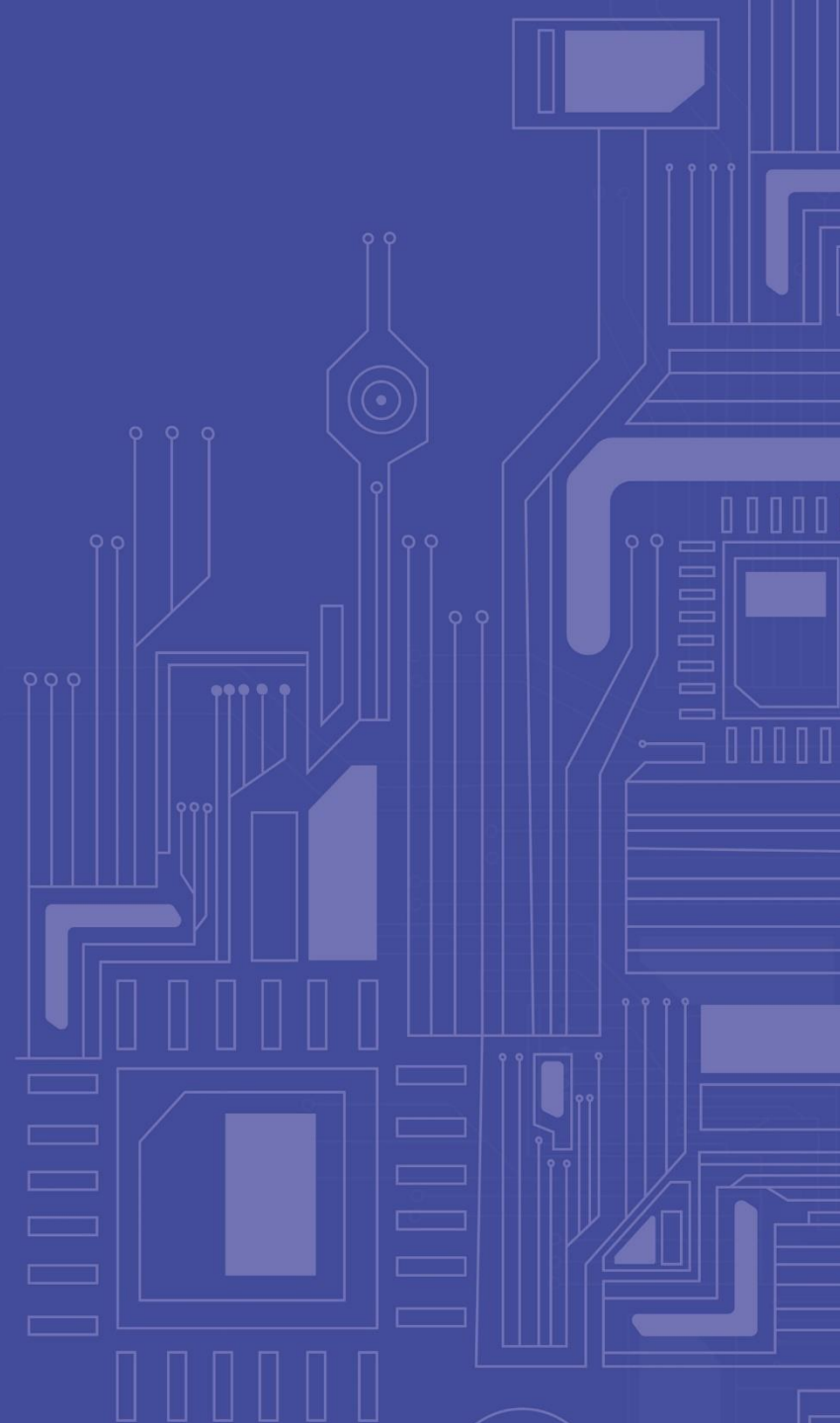
МИНОБРНАУКИ  
РОССИИ



Передовые  
инженерные  
школы

# СИСТЕМЫ УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ

*Лекция 13*



- Рекурсивные запросы
  - Пример рекурсии из новейшей практики (PostgreSQL)
    - Операция обновления представлений
    - Базовые средства манипулирования данными

# РЕКУРСИВНЫЕ ЗАПРОСЫ



**Обход дерева в ширину.** При этом способе обхода непосредственные потомки обходятся слева направо, до того как производится переход к потомкам следующего уровня родства.

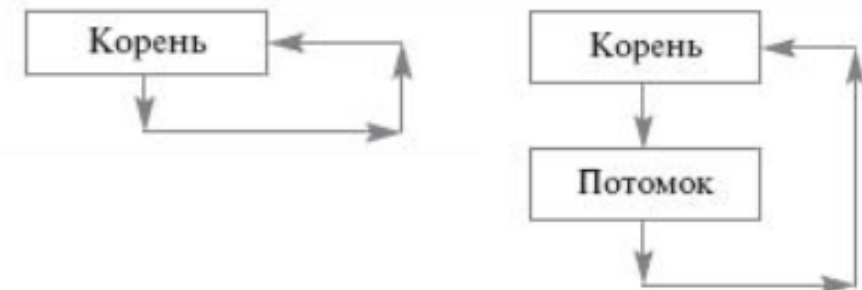
При обходе в ширину дерева узлы будут обходиться в следующем порядке:

Корень-Потомок1-Потомок2-Потомок3-П1.1-П1.2-П1.3-П2.1-П2.2-П2.3-П3.1-П3.2-П3.3.



**Обход дерева в глубину.** При этом способе обхода на каждом шаге производится переход к самому левому текущему потомку. При обходе в глубину дерева порядок обхода узлов будет следующим: Корень-Потомок1-П1.1-П1.2-П1.3-Потомок2-П2.1-П2.2-П2.3-Потомок3-П3.1-П3.2-П3.3.

**Цикл в ориентированном графе.** В теории графов ориентированный граф называется циклическим в том и только в том случае, когда хотя бы один узел графа одновременно является и предком, и потомком (т. е. для этого узла имеется и выходящая, и входящая дуги).



Начиная с SQL:1999 узлами графа рекурсии являются строки, входящие в результат рекурсивного запроса, а дуги соответствуют способам обработки текущих строк, которые ведут к добавлению к результату новых строк.

**Прямая рекурсия.** По определению, некоторый элемент использует прямую рекурсию в том и только в том случае, когда он обращается сам к себе без посредников.



Пример не прямой рекурсии в графовой форме.



**Линейная рекурсия.** При линейно рекурсивном вызове элемент прямо рекурсивно обращается сам к себе не более одного раза. В SQL:1999 в определении любой виртуальной таблицы с рекурсией допускается не более одной ссылки на саму себя (в разделе FROM и/или в подзапросах).

Графовый пример рекурсии, не являющейся линейной.



# МОНОТОННОСТЬ, ВЗАИМНАЯ РЕКУРСИЯ, ОТРИЦАНИЕ

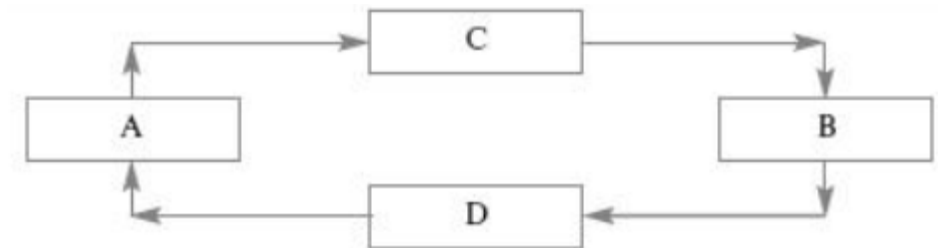


**Монотонность.** *Монотонной прогрессией* называется последовательность неубывающих или невозрастающих значений. Например, последовательность натуральных чисел  $\{1, 2, \dots, n, \dots\}$  является монотонной. В SQL:1999 свойство монотонности поддерживается в том смысле, что число строк результата рекурсивного запроса не уменьшается на каждом шаге рекурсии.


**Взаимная рекурсия.** Элементы A и B связаны отношением взаимной рекурсии, если A прямо или косвенно вызывает B, и B прямо или косвенно вызывает A.

**Отрицание.** В контексте SQL отрицанием называется любое действие, приводящее к уменьшению числа строк в результате запроса. Свойствами отрицания обладают операции над (мульти)множествами EXCEPT и INTERSECT, спецификация DISTINCT, условие NOT EXISTS и т.д.

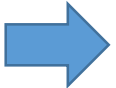
Графовый пример взаимной рекурсии (элемент A вызывает элемент B через элемент C, а элемент B вызывает элемент A через элемент D).




В стандарте SQL не запрещается использование отрицания в рекурсивных запросах. Возможной проблемы нарушения монотонности удастся избежать за счет того, что отрицание разрешается применять только к тем таблицам, которые являются полностью известными (или вычисленными) к моменту применения отрицания. В процессе вычисления таблицы применение к ней отрицания не допускается.



**Начальный источник рекурсии.** При выполнении рекурсивных вычислений обычно (хотя и не всегда) имеется некоторое начальное значение. В SQL этим начальным источником рекурсии является одна или несколько строк, удовлетворяющих некоторым начальным условиям. На основе этих строк в процессе рекурсивного вычисления производятся дополнительные строки, образующие окончательный результат.



**Стратификация.** В SQL рекурсивный запрос обычно состоит из «рекурсивной» и «нерекурсивной» частей. В процессе стратификации («расслоения») запроса выполнение этих двух частей разделяется. В более сложных рекурсивных запросах может содержаться несколько рекурсивных частей и более одной нерекурсивной части. В этом случае в процессе стратификации будет обнаружено большее число слоев.



**Семантика фиксированной точки.** В контексте SQL:1999 семантика фиксированной точки означает, что решение о завершении рекурсивного запроса принимается тогда, когда становится невозможно добавить к результату какие-либо дополнительные строки.

# СИНТАКСИС РЕКУРСИИ С РАЗДЕЛОМ WITH



Полный синтаксис раздела WITH выглядит следующим образом:

```
with_clause ::= WITH [ RECURSIVE ]
with_element_comma_list
with_element ::= query_name [ (column_name_list) ]
                AS ( query_expression ) [ search_or_cycle_clause ]
search_or_cycle_clause ::= search_clause
                        | cycle_clause
                        | search_clause cycle_clause
search_clause ::= SEARCH recursive_search_order SET
                sequence_column_name
recursive_search_order ::= DEPTH FIRST BY
                        order_item_comma_list
                        | BREATH FIRST BY order_item_comma_list
cycle_clause ::= CYCLE cycle_column_name_comma_list
                SET cycle_mark_column_name TO value_expression
                DEFAULT value_expression
                USING path_column_name
```



# КЛАССИЧЕСКИЙ ПРИМЕР РЕКУРСИИ (1/2)



Для иллюстрации возможностей рекурсивных запросов с разделом WITH и пояснения смысла конструкций SEARCH и CYCLE воспользуемся классическим примером «разборки деталей» (в данном случае мы будем разбирать автомобиль). Предположим, что данные о конструктивных элементах автомобиля хранятся в таблице CAR, определенной следующим образом:



```
CREATE TABLE CAR (CONTAINING_PART VARCHAR (10),  
CONTAINED_PART VARCHAR (10),  
NUMBER_OF_PARTS INTEGER,  
PART_COST DECIMAL (6,2));
```

У автомобиля имеется один конструктивный элемент верхнего уровня – полностью собранный автомобиль. Этот элемент не является составной частью какого-либо другого элемента, и для его строки значением столбца CONTAINING\_PART является текстовая строка длины 0. В любой другой строке таблицы CAR, соответствующей некоторому неатомарному конструктивному элементу **e**, столбец CONTAINING\_PART содержит идентификационный номер элемента **e1**, в который входит элемент **e**, столбец NUMBER\_OF\_PARTS – число экземпляров элемента **e**, входящих в **e1**, а столбец CONTAINED\_PART – идентификационный номер самого элемента **e**. В любой строке таблицы CAR, соответствующей некоторому атомарному конструктивному элементу, значением столбца CONTAINED\_PART является строка длины 0, а в столбце PART\_COST сохраняется цена атомарного конструктивного элемента (для неатомарных элементов значение этого столбца равно нулю).

# КЛАССИЧЕСКИЙ ПРИМЕР РЕКУРСИИ (2/2)



## ЗАДАЧА

Требуется разобрать автомобиль, начиная с элемента самого верхнего уровня, и для каждого конструктивного элемента получить его номер, общее число используемых экземпляров этого элемента, а также, если элемент является атомарным, общую стоимость используемых экземпляров.

```
WITH RECURSIVE PARTS (PART_NUMBER,  
    NUMBER_OF_PARTS, COST) AS  
    (SELECT CONTAINED_PART, 1, 0.00 (a)  
    FROM CAR  
    WHERE CONTAINING_PART = ''  
    UNION ALL  
    SELECT CAR.CONTAINED_PART, CAR.NUMBER_OF_PARTS,  
        CAR.NUMBER_OF_PARTS * CAR.PART_COST  
    FROM CAR, PARTS  
    WHERE PARTS.PART_NUMBER = CAR.CONTAINING_PART)  
SELECT PART_NUMBER, SUM(NUMBER_OF PARTS),  
    SUM(COST) (b)  
FROM PARTS  
GROUP BY PART_NUMBER;
```

При вычислении раздела FROM основного запроса (b) начнется выполнение рекурсивного выражения запросов (a), определенного в разделе WITH. На первом шаге рекурсии будет выполнена часть данного выражения, предшествующая операции UNION ALL и образующая начальный источник рекурсии. В результате будет произведено исходное состояние виртуальной таблицы PARTS, в котором, в нашем случае, появится единственная строка, соответствующая автомобилю целиком. На следующем шаге к таблице PARTS будут добавлены строки, соответствующие конструктивным элементам второго уровня (для автомобиля это, по-видимому, двигатель, колеса, шасси и т.д.).

Этот процесс будет продолжаться до тех пор, пока мы не дойдем до атомарных конструктивных элементов и не достигнем, тем самым, фиксированной точки. Поскольку в рекурсивном запросе содержится операция UNION ALL, в результирующей таблице могут появляться строки-дубликаты. Наличие строки-дубликата вида <part\_no, number, cost> означает, что элемент с номером part\_no входит в одном и том же числе экземпляров в несколько конструктивных элементов более высокого уровня.

В предыдущем примере не определялся порядок, в котором строки добавляются к частичному результату рекурсивного запроса. иногда требуется, чтобы иерархия обходилась в глубину или в ширину. Соответствующая возможность обеспечивается конструкцией SEARCH. При указании требования обхода в глубину гарантируется, что каждый элемент-предок появится в результате раньше своих потомков и своих братьев справа. Если указывается требование обхода иерархии в ширину, в результате все братья одного уровня появляются раньше, чем какой-либо их потомок.

В списке столбцов сортировки раздела SEARCH должны указываться имена столбцов виртуальной таблицы, определенной в разделе WITH. Поскольку в данном случае мы хотим, чтобы в результате сначала появлялись все конструктивные элементы одного уровня (CONTAINING\_PART), а затем все их подэлементы (CONTAINED\_PART), в список выборки рекурсивного запроса PARTS добавлен столбец CONTAINING\_PART, который не используется нигде, кроме раздела SEARCH. В разделе SET к результирующей таблице рекурсивного запроса добавлен столбец, который мы назвали ORDER\_COLUMN. Название соответствует природе столбца, потому что при выполнении рекурсивного запроса в этот столбец автоматически заносятся значения, характеризующие порядок генерируемых строк в соответствии с выбранным способом обхода иерархии. Чтобы строки результата основного запроса появлялись в должном порядке, в этом запросе требуется наличие раздела ORDER BY с указанием столбца, определенного в разделе SET.

**Вариант запроса, в котором содержится раздел SEARCH с требованием обхода иерархии элементов автомобиля в ширину.**

```
WITH RECURSIVE PARTS (ASSEMBLY, PART_NUMBER,
    NUMBER_OF_PARTS, COST) AS
    (SELECT CONTAINING_PART, CONTAINED_PART, 1, 0.00
    FROM CAR
    WHERE CONTAINING_PART = ''
    UNION ALL
    SELECT CAR.CONTAINING_PART, CAR.CONTAINED_PART,
    CAR.NUMBER_OF_PARTS, CAR.NUMBER_OF_PARTS *
    CAR.PART_COST
    FROM CAR, PARTS
    WHERE PARTS.PART_NUMBER = CAR.CONTAINING_PART)
SEARCH BREADTH FIRST
BY CONTAINING_PART, CONTAINED_PART
SET ORDER_COLUMN
SELECT PART_NUMBER, NUMBER_OF PARTS, COST
FROM PARTS
ORDER BY ORDER_COLUMN;
```

Иногда сами данные, хранимые в таблицах базы данных, могут иметь циклическую природу.

## ПРИМЕР

Представим себе, например, компанию, в которой существует совет директоров, являющийся высшим органом управления компанией. Обычным случаем является тот, когда по крайней мере один из членов совета директоров является простым служащим этой же компании (например, он может входить в совет директоров как представитель профсоюза). Назовем данного члена совета директоров EMP\_DIR. Как член совета директоров, EMP\_DIR «управляет» деятельностью президента компании. С другой стороны, как служащий компании, EMP\_DIR находится в прямом или косвенном подчинении у президента компании. Такое положение может привести к зацикливанию выполнения рекурсивных запросов.

Раздел CYRCLE обеспечивает некоторую возможность распознавать подобные ситуации. Если у пользователя имеется полная уверенность в отсутствии циклов в данных, к которым адресуется рекурсивный запрос, то использование раздела CYRCLE не требуется.

➔ Подход к распознаванию зацикленных запросов, принятый в SQL, состоит в том, что распознаются данные, которые уже участвовали ранее в формировании результата рекурсивного запроса. При наличии раздела CYRCLE при добавлении к результату строк, удовлетворяющих условию запроса, такие строки помечаются указанным значением, которое означает, что эти строки уже вошли в результат. При попытке добавления к результату каждой новой строки проверяется, не находится ли она уже в результате, т. е. не помечена ли она этим указанным в разделе CYRCLE значением. Если это действительно так, то считается, что имеет место цикл, и дальнейшее выполнение рекурсивного запроса прекращается.



## Синтаксис раздела CYRCLE.

```
cycle_clause ::= CYCLE cycle_column_name_comma_list  
               SET cycle_mark_column_name TO value_expression_1  
               DEFAULT value_expression_2  
               USING path_column_name
```

В списке `cycle_column_name_comma_list` указываются имена одного или нескольких столбцов, которые используются для идентификации новых строк результата на основе строк, уже входящих в результат. Например, в примере столбец `CONTAINED_PART` связывает конструктивный элемент автомобиля с входящими в его состав подэлементами (через значения их столбцов `CONTAINING_PART`).

Раздел `SET` приводит к образованию нового столбца результирующей таблицы. Для строк, которые попадают в результат первый раз, в столбец `cycle_mark_column_name` заносится значение выражения `value_expression_2`. В повторно заносимых строках значение столбца – `value_expression_1`. Типом данных этого столбца является тип символьных строк длины один, так что в качестве `value_expression_1` и `value_expression_2` разумно использовать константы '0' и '1' или 'Y' и 'N'.

Раздел `USING` приводит к образованию еще одного дополнительного столбца результата с именем `path_column_name`. Типом данных столбца является `ARRAY`, причем кардинальность этого типа предполагается достаточно большой, чтобы сохранить информацию обо всех строках, попавших в результат. Элементы массива имеют «строчный тип» (`row type`), содержащий столько столбцов, сколько их указано в списке раздела `CYRCLE`. Каждый элемент массива соответствует строке результата, и в его столбцах содержится копия значений соответствующих столбцов этой строки.

```
WITH RECURSIVE PARTS (PART_NUMBER,  
    NUMBER_OF_PARTS, COST) AS  
    (SELECT CONTAINED_PART, 1, 0.00  
     FROM CAR  
     WHERE CONTAINING_PART = ''  
    UNION ALL  
     SELECT CAR.CONTAINED_PART, CAR.NUMBER_OF_PARTS,  
           CAR.NUMBER_OF_PARTS * CAR.PART_COST  
     FROM CAR, PARTS  
     WHERE PARTS.PART_NUMBER = CAR.CONTAINING_PART)  
CYRCLE CONTAINED_PART  
SET CYCLEMARK TO 'Y' DEFAULT 'N'  
USING CYRCLEPATH  
SELECT PART_NUMBER, SUM(NUMBER_OF PARTS), SUM(COST)  
FROM PARTS  
ORDER BY PART_NUMBER;
```

Имена столбцов CYCLEMARK и CYRCLEPATH выбраны произвольным образом – требуется только, чтобы имена этих столбцов отличались от имен столбцов рекурсивного запроса. При выполнении запроса строки, удовлетворяющие его условию, накапливаются в результирующей таблице. Но, кроме того, эти строки «кэшируются» в столбце CYRCLEPATH. При попытке добавления к результату новой строки на основе текущего содержимого столбца CYRCLEPATH проверяется, не содержится ли она уже в результате.

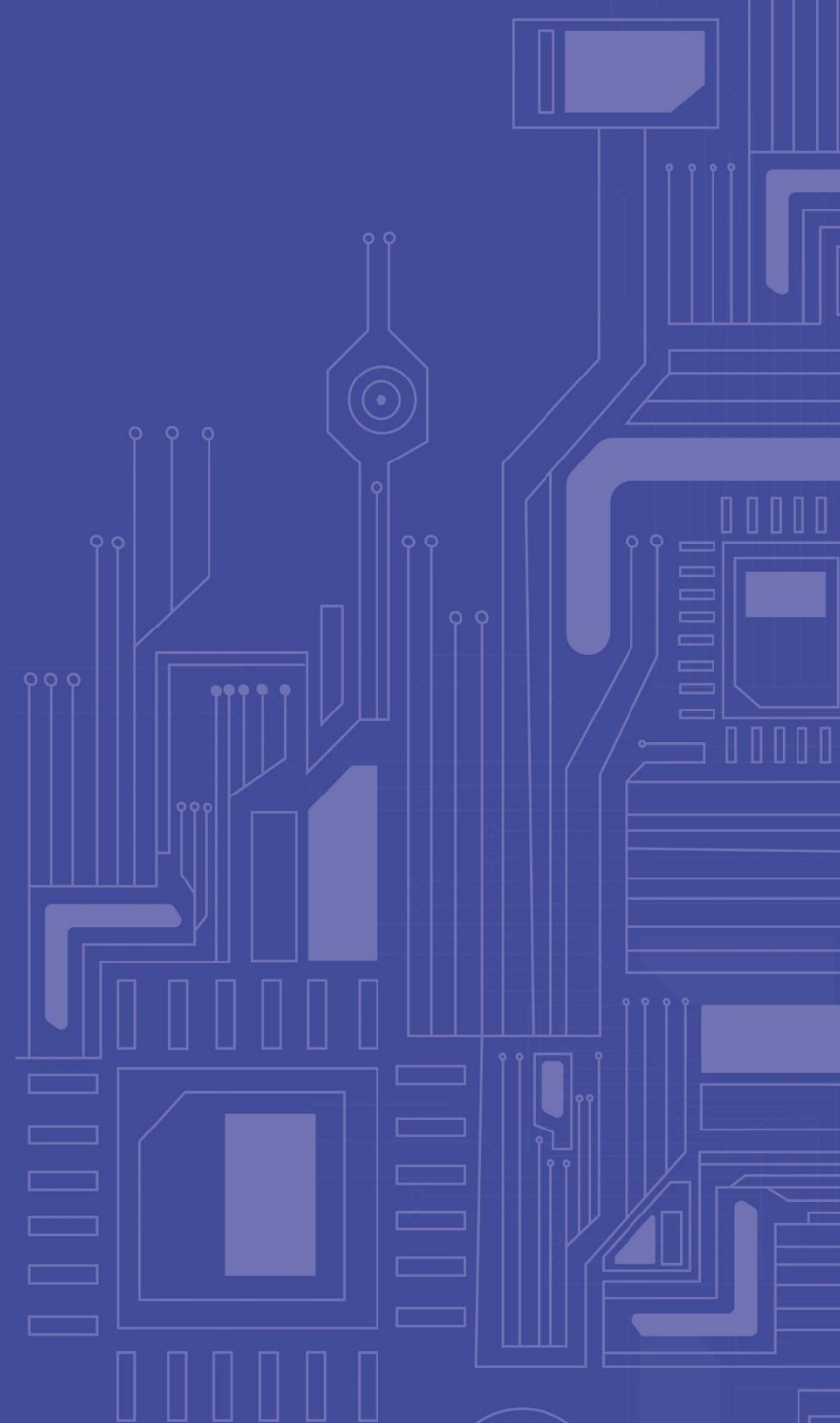
Если не содержится, то данные об этой строке добавляются к столбцу CYRCLEPATH (к массиву добавляется новый элемент), в столбец CYCLEMARK этой строки заносится значение 'N', и строка добавляется к результату. Иначе в столбец CYCLEMARK соответствующей строки результата заносится значение 'Y', означающее, что от этой строки начинается цикл.

Рекурсивным называется *представление*, в определяющем выражении запроса которого используется имя этого же представления. В представлениях может использоваться и прямая, и взаимная рекурсия. Синтаксис оператора определения рекурсивного запроса выглядит следующим образом:

```
CREATE RECURSIVE VIEW table_name  
  [ column_name_comma_list ]  
  AS query_expression
```

Хотя для того, чтобы представление было рекурсивным, требуется рекурсивность определяющего выражения запроса (т.е. в нем должна присутствовать спецификация RECURSIVE); наличие избыточного ключевого RECURSIVE в определении рекурсивного представления является обязательным. Как говорят авторы стандарта, это сделано для того, чтобы избежать случайного появления непредусмотренных рекурсивных представлений.

# ПРИМЕР РЕКУРСИИ ИЗ НОВЕЙШЕЙ ПРАКТИКИ

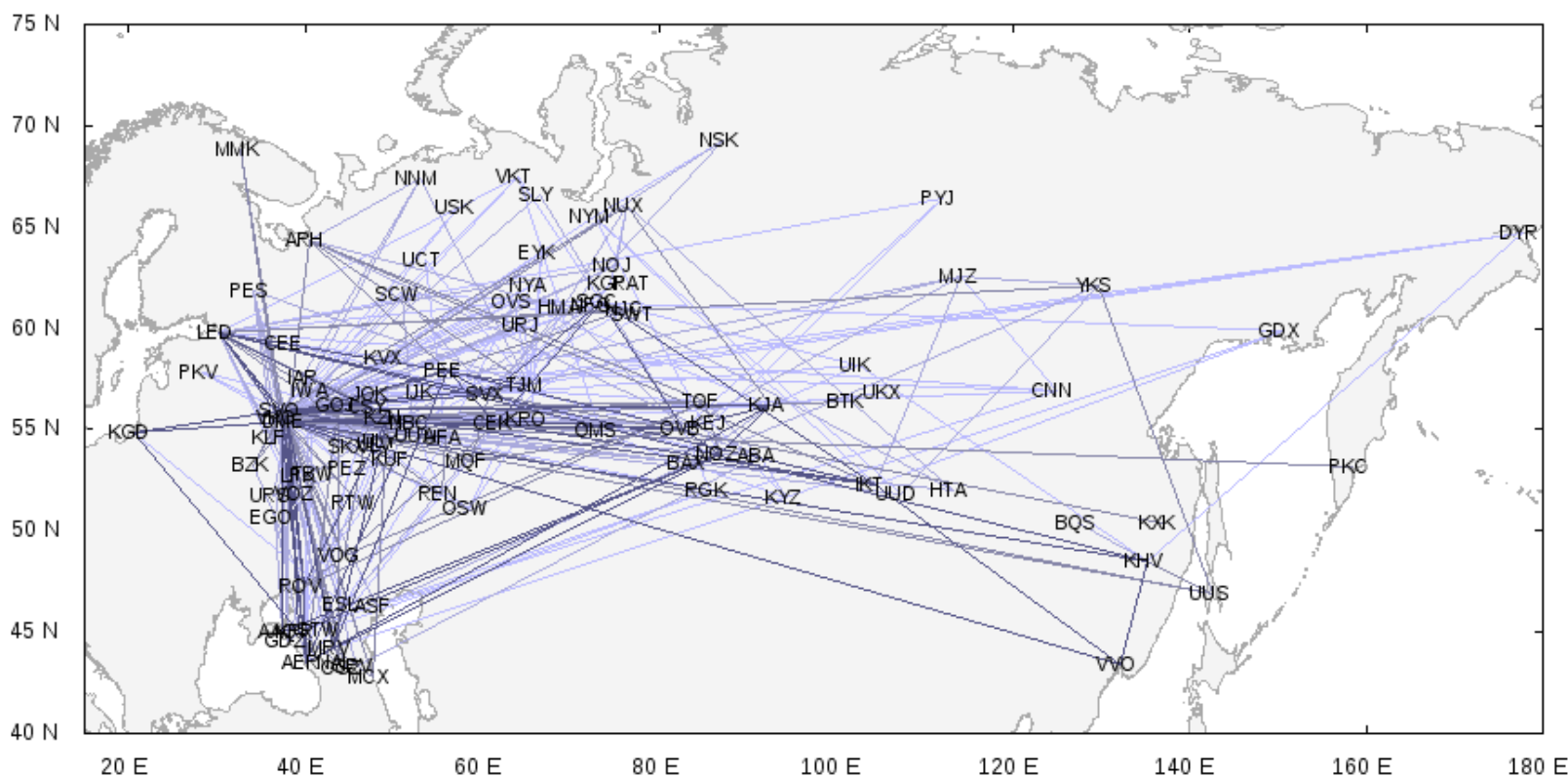




<https://habr.com/ru/companies/postgrespro/articles/318398/> - статья с примером на рекурсию.

<https://habr.com/ru/companies/postgrespro/articles/316428/> - учебная база PostgreSQL.

## База данных «Авиаперевозки»



<https://edu.postgrespro.ru/bookings.pdf>

- полное описание базы данных и инструкция по скачиванию.

- *Как реальное время полета отличается от запланированного?*
- *Обычно полеты с запада на восток длинные (вылетаем ночью, прилетаем утром следующего дня), а с востока на запад — короткие (прилетаем в тот же день почти в то же время). А что происходит в демо-базе?*
- *Как распределено время бронирования и время регистрации по отношению к дате и времени рейса?*
- *Сколько человек обычно входит в одно бронирование?*
- *Бывают ли пассажиры, летящие туда-обратно? Всегда ли маршрут «туда» совпадает с маршрутом «обратно»?*

Старый добрый реляционный SQL хорошо справляется с неупорядоченными множествами: для них и в самом языке, и «под капотом» СУБД имеется целый арсенал средств. Чего SQL не любит, так это когда его дергают в цикле строчка за строчкой, вместо того, чтобы выполнить задачу одним оператором. Оптимизатор опускает руки и отходит в сторону, и вы остаетесь один на один с производительностью.

Так вот рекурсивный запрос — это такой способ устроить цикл прямо внутри SQL. Не то, чтобы это было нужно очень часто, но иногда все-таки приходится. И тут начинаются сложности, поскольку рекурсивный запрос не очень похож ни на обычный запрос, ни тем более на обычный цикл.

Сначала выполняется нерекурсивная часть (1). Затем рекурсивная часть (2) выполняется до тех пор, пока она возвращает какие-либо строки. Рекурсивная часть называется так потому, что она может обращаться к результату выполнения предыдущей итерации, который доступен ей под именем *t*.

Попутно результат выполнения каждой итерации складывается в результирующую таблицу, которая будет доступна под тем же именем *t* после того, как весь запрос отработает (3). Если вместо UNION ALL написать UNION, то на каждой итерации будут устраняться дубликаты строк.

Общая схема рекурсивного запроса такова:

```
WITH RECURSIVE t AS (  
    нерекурсивная часть      (1)  
  
    UNION ALL  
  
    рекурсивная часть        (2)  
)  
  
SELECT * FROM t;              (3)
```

# ПРОСТОЙ ПРИМЕР



```
demo=# WITH RECURSIVE t(n,factorial) AS (  
  
VALUES (0,1)  
  
UNION ALL  
  
SELECT t.n+1, t.factorial*(t.n+1) FROM t WHERE t.n < 5  
  
)  
  
SELECT * FROM t;
```



n		factorial
---	--	-----------

-----+-----		
-------------	--	--

0		1
---	--	---

1		1
---	--	---

2		2
---	--	---

3		6
---	--	---

4		24
---	--	----

5		120
---	--	-----

(6 строк)

# ТОТ САМЫЙ ПРИМЕР С РЕКУРСИЕЙ



## Поиск кратчайшего пути из Усть-Кута (UKX) в Нерюнгри (CNN)

```
WITH RECURSIVE p(last_arrival, destination, hops, flights, found) AS (
```

```
  SELECT a_from.airport_code,
```

```
         a_to.airport_code,
```

```
         ARRAY[a_from.airport_code],
```

```
         ARRAY[]::char(6)[],
```

```
         a_from.airport_code = a_to.airport_code
```

```
  FROM   airports a_from, airports a_to
```

```
  WHERE  a_from.airport_code = 'UKX'
```

```
  AND    a_to.airport_code = 'CNN'
```

```
  UNION ALL
```

```
    SELECT r.arrival_airport,
```

```
           p.destination,
```

```
           (p.hops || r.arrival_airport)::char(3)[],
```

```
           (p.flights || r.flight_no)::char(6)[],
```

```
           bool_or(r.arrival_airport = p.destination) OVER ()
```

```
  FROM   routes r, p
```

```
  WHERE  r.departure_airport = p.last_arrival
```

```
  AND    NOT r.arrival_airport = ANY(p.hops)
```

```
  AND    NOT p.found
```

```
)
```

```
SELECT hops,
```

```
       flights
```

```
FROM   p
```

```
WHERE  p.last_arrival = p.destination;
```



# БАЗОВЫЕ СРЕДСТВА МАНИПУЛИРОВАНИЯ ДАННЫМИ



К базовым средствам манипулирования данными языка SQL относятся «поисковые» варианты операторов **UPDATE** и **DELETE**. Эти варианты называются поисковыми, потому что при задании соответствующей операции задается логическое условие, налагаемое на строки адресуемой оператором таблицы, которые должны быть подвергнуты модификации или удалению. Кроме того, в такую категорию языковых средств входит оператор **INSERT**, позволяющий добавлять строки в существующие таблицы.

## Оператор INSERT для вставки строк в существующие таблицы

Общий синтаксис оператора INSERT выглядит следующим образом:

```
INSERT INTO table_name
    { [ (column_comma_list) ] query_expression
    | DEFAULT VALUES
```

Даже если ограничиться простейшей составляющей этой конструкции (simple\_table), то мы имеем следующие возможности:

```
simple_table ::= query_specification
    | table_value_constructor
    | TABLE table_name
```

```
query_expression ::= [ with_clause ] query_expression_body
query_expression_body ::= { non_join_query_expression
    | joined_table }
non_join_query_expression ::= non_join_query_term
    | query_expression_body
    { UNION | EXCEPT } [ ALL | DISTINCT ]
    [ corresponding_spec ] query_term
query_term ::= non_join_query_term | joined_table
non_join_query_term ::= non_join_query_primary
    | query_term INTERSECT [ ALL | DISTINCT ]
    [ corresponding_spec ] query_primary
query_primary ::= non_join_query_primary | joined_table
non_join_query_primary ::= simple_table
    | (non_join_query_expression)
simple_table ::= query_specification
    | table_value_constructor
    | TABLE table_name
corresponding_spec ::= CORRESPONDING
    [ BY column_name_comma_list ]
```

# ВСТАВКА ВСЕХ СТРОК УКАЗАННОЙ ТАБЛИЦЫ



Стандарт допускает вставку в указанную таблицу всех строк некоторой другой таблицы (вариант `table_name`). Эта другая таблица может быть как базовой, так и представляемой.

- В определении представления не должны присутствовать ссылки на таблицу, в которую производится вставка.
- При использовании данного варианта оператора вставки число столбцов вставляемой таблицы должно совпадать с числом столбцов таблицы, в которую производится вставка, или с числом столбцов, указанных в списке `column_commalist`, если этот список задан.
- Типы данных соответствующих столбцов вставляемой таблицы и таблицы, в которую производится вставка, должны быть совместимыми.
- Если в операции задан список `column_commalist` и в нем содержатся не все имена столбцов таблицы, в которую производится вставка, то в оставшиеся столбцы во всех строках заносятся значения столбцов по умолчанию.
- Если для какого-либо из оставшихся столбцов значение по умолчанию не определено, при выполнении операции вставки фиксируется ошибка.

**ПРИМЕР** Предположим, что в базе данных EMP-DEPT-PRO имеется еще одна промежуточная таблица EMP\_TEMP, в которой временно хранятся данные о служащих, проходящих испытательный срок. Пусть эта таблица имеет следующий заголовок:

EMP\_TEMP:

EMP_NO	:	EMP_NO
EMP_NAME	:	VARCHAR
EMP_BDATE	:	DATE



```
INSERT INTO EMP (EMP_NO, EMP_NAME, EMP_BDATE) TABLE EMP_TEMP;
```

В столбцах EMP\_NO, EMP\_NAME, EMP\_BDATE будут данные, взятые из таблицы EMP\_TEMP, а в столбцах EMP\_SAL, DEPT\_NO, PRO\_NO будут значения по умолчанию.

EMP\_NO не должен нарушать ограничения целостности первичного ключа.



# ВСТАВКА ЯВНО ЗАДАННОГО ЧИСЛА СТРОК



Набор вставляемых строк задается явно с использованием синтаксической конструкции `table_value_constructor`.

```
table_value_constructor ::=
    VALUES row_value_constructor_comma_list
row_value_constructor ::= row_value_constructor_element
    | [ ROW ] (row_value_constructor_element_comma_list)
    | row_subquery
row_value_constructor_element ::= value_expression
    | NULL | DEFAULT
```

## Варианты вставки:

```
INSERT INTO EMP
ROW (2445, 'Brown', '1985-04-08', 16500.00, 630, 772);
```

```
INSERT INTO EMP
ROW ( 2445, DEFAULT, NULL, DEFAULT, NULL, NULL);
```

<=>

```
INSERT INTO EMP (EMP_NO) 2445;
```

Одной из разновидностей `value_expression_primary` является **scalar\_subquery**

```
INSERT INTO EMP VALUES
    ROW (2445, (SELECT EMP_NAME
                FROM EMP
                WHERE EMP_NO = 2555),
        '1985-04-08',
        (SELECT EMP_SAL
         FROM EMP
         WHERE EMP_NO = 2555),
        NULL, NULL ),
    ROW (2446, (SELECT EMP_NAME
                FROM EMP
                WHERE EMP_NO = 2556),
        '1978-05-09',
        (SELECT EMP_SAL
         FROM EMP
         WHERE EMP_NO = 2556),
        NULL, NULL );
```



После выполнения этой операции в таблице EMP появятся две новые строки для служащих с уникальными идентификаторами 2445 и 2446, причем первому из них будет присвоено имя и размер заработной платы служащего с уникальным идентификатором 2555, а второму – аналогичные данные о служащем с уникальным идентификатором 2556.

# ВСТАВКА СТРОК РЕЗУЛЬТАТА ЗАПРОСА



Это вариант оператора вставки, когда набор вставляемых строк определяется через спецификацию запроса.

**ПРИМЕР** Требуется сохранить в отдельной таблице DEPT\_SUMMARY сведения о числе служащих каждого отдела, их максимальной, минимальной и суммарной заработной плате.

DEPT\_SUMMARY:

DEPT_NO : DEPT_NO
DEPT_EMP_NO : INTEGER
DEPT_MAX_SAL : SALARY
DEPT_MIN_SAL : SALARY
DEPT_TOTAL_SAL : SALARY

```
INSERT INTO DEPT_SUMMARY
  (SELECT DEPT_NO, COUNT(*), MAX (EMP_SAL),
    MIN (EMP_SAL), SUM (EMP_SAL)
   FROM EMP
  GROUP BY DEPT_NO);
```

# ОПЕРАТОР UPDATE



Общий синтаксис оператора UPDATE выглядит следующим образом:

```
UPDATE table_name SET update_assignment_commalist
WHERE conditional_expression
update_assignment ::= column_name =
{ value_expression | DEFAULT | NULL }
```

Как это работает:

1. Для всех строк таблицы с именем `table_name` вычисляется булевское выражение `conditional_expression`. Строки, для которых значением этого булевского выражения является `true`, считаются подлежащими модификации (обозначим множество таких строк через  $T_m$ );
2. Каждая строка  $s$  (из  $T_m$ ) подвергается модификации таким образом, что значение каждого столбца этой строки, указанного в списке `update_assignment_commalist`, заменяется значением, указанным в правой части соответствующего элемента списка модификации. Значения столбцов строки  $s$ , не указанные в списке модификации, остаются неизменными.

## ПРИМЕР

Для всех служащих, работающих в отделах, заработная плата менеджеров которых превышает 30000 руб., установить размер заработной платы, на 1000 руб. превышающий средний размер заработной платы соответствующего отдела, а номера проектов, в которых участвуют эти служащие, сделать неопределенными.

```
UPDATE EMP SET EMP_SAL = (SELECT AVG (EMP1_SAL)
FROM EMP EMP1
WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
+ 1000.00, PRO_NO = NULL
WHERE (SELECT EMP1.EMP_SAL
FROM EMP EMP1, DEPT
WHERE EMP.DEPT_NO = DEPT.DEPT_NO
AND DEPT_MNG = EMP1.EMP_NO AND) > 30000.00;
```

# ОПЕРАТОР DELETE

Общий синтаксис оператора UPDATE выглядит следующим образом:

```
DELETE FROM table_name  
WHERE conditional_expression
```

В некотором смысле оператор DELETE является частным случаем оператора UPDATE (или, наоборот, действие оператора UPDATE представляет собой комбинацию действий операторов DELETE и INSERT).

Как это работает:

1. Для всех строк таблицы с именем `table_name` вычисляется булевское выражение `conditional_expression`. Строки, для которых значением этого булевского выражения является `true`, считаются подлежащими удалению (обозначим множество таких строк через  $T_d$ );
2. Каждая строка  $s$  (из  $T_d$ ) удаляется из указанной таблицы.

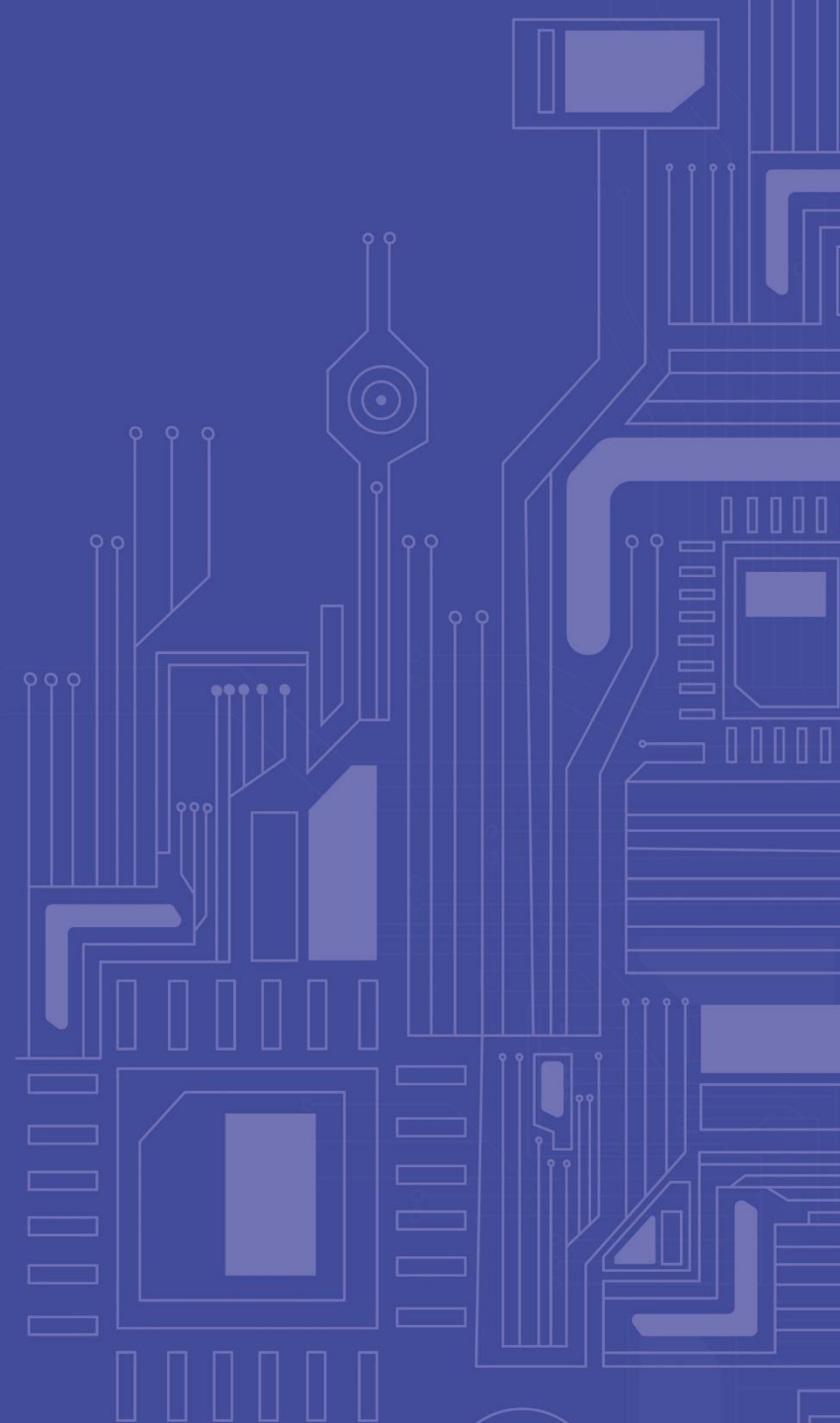
**ПРИМЕР 1** Удалить из таблицы EMP все строки, относящиеся к служащим, которые участвуют в проекте с номером 772.

```
DELETE FROM EMP WHERE PRO_NO = 772;
```

**ПРИМЕР 2** Удалить из таблицы EMP все строки, относящиеся к служащим, размер заработной платы которых превышает размер заработной платы менеджеров их отделов.

```
DELETE FROM EMP WHERE EMP_SAL >  
(SELECT EMP1.EMP_SAL  
FROM EMP EMP1, DEPT  
WHERE EMP.DEPT_NO = DEPT.DEPT_NO  
AND DEPT.DEPT_MNG = EMP1.EMP_NO);
```

# ОПЕРАЦИЯ ОБНОВЛЕНИЯ ПРЕДСТАВЛЕНИЙ



# ТРЕБОВАНИЯ СТАНДАРТА SQL:1992



*Представление* – это сохраняемое в каталоге базы данных выражение запросов, обладающее собственным именем и, возможно, собственными именами столбцов.

Синтаксис создания представления:

```
create_view ::= CREATE [ RECURSIVE ] VIEW table_name  
               [ column_name_comma_list ]  
               AS query_expression  
               [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Операция обновления над представлением должна однозначно отображаться в одну или несколько операций обновления над теми постоянно хранимыми базовыми таблицами, над которыми прямо или косвенно определено данное представление.

В стандарте SQL/92 спецификация запроса считалась допускающей операции обновления в том и только в том случае, когда выполнялись следующие условия:

- в разделе `SELECT` спецификации запроса отсутствует ключевое слово `DISTINCT` (т.е. не требуется удаление строк-дубликатов из результата запроса);
- все элементы списка выборки раздела `SELECT` являются именами столбцов, и ни одно имя столбца не встречается в этом списке более одного раза;
- в разделе `FROM` присутствует только одна ссылка на таблицу, и она указывает либо на базовую таблицу, либо на порождаемую таблицу, допускающую операции обновления;
- прямые или косвенные ссылки на базовую таблицу, прямо или косвенно идентифицируемую ссылкой на таблицу в разделе `FROM`, не встречаются в разделе `FROM` ни одного подзапроса, участвующего в разделе `WHERE` спецификации запроса;
- в спецификации запроса отсутствуют разделы `GROUP BY` и `HAVING`.



# ПРИМЕР ДЛЯ СТАНДАТА SQL:1992



```
SELECT EMP_SAL
FROM EMP
WHERE EMP_NAME = (SELECT EMP_NAME
                   FROM EMP
                   WHERE EMP_NO = 4425 )
                   AND DEPT_NO <> 630;
```

Предположим, что с данной спецификацией запроса связано представление с именем **EMPSAL**. Тогда операция

UPDATE EMPSAL SET EMP\_SAL = EMP\_SAL – 1000.00;

эквивалентна операции

```
UPDATE EMP SET EMP_SAL = EMP_SAL - 1000.00
WHERE EMP_NAME = (SELECT EMP_NAME
                  FROM EMP
                  WHERE EMP_NO = 4425 )
                  AND DEPT_NO <> 630;
```

Операция

DELETE FROM EMPSAL WHERE EMP\_SAL > 20000.00;

эквивалентна операции

```
DELETE EMPSAL
WHERE EMP_SAL > 20000.00 AND
      EMP_NAME = (SELECT EMP_NAME
                  FROM EMP
                  WHERE EMP_NO = 4425 )
                  AND DEPT_NO <> 630;
```

В стандарте SQL:1999 правила применимости операций обновления к спецификации запроса существенно уточнены.

## Критерии применимости операций обновления

Введены понятия потенциальной применимости операций обновления, применимости операций обновления, простой применимости операций обновления и применимости операции вставки. К спецификации запроса потенциально применимы операции обновления в том и только в том случае, когда выполняются следующие условия:

- в разделе `SELECT` спецификации запроса отсутствует ключевое слово `DISTINCT`;
- элемент списка выборки раздела `SELECT`, состоящий из ссылки на некоторый столбец, не может присутствовать в этом списке более одного раза;
- в спецификации запроса отсутствуют разделы `GROUP BY` и `HAVING`.

Если выражение запросов отвечает условиям потенциальной применимости операций обновления и в его разделе `FROM` присутствует только одна ссылка на таблицу, то к каждому столбцу выражения запроса, соответствующему одному столбцу таблицы из раздела `FROM`, применимы операции обновления. Если выражение запроса отвечает условиям потенциальной применимости операций обновления, но в его разделе `FROM` присутствуют две или более ссылки на таблицы, то операции обновления применимы к столбцу выражения запросов только при выполнении следующих условий:

- столбец порождается из столбца только одной таблицы из раздела `FROM`;
- эта таблица используется в выражении запросов таким образом, что сохраняются свойства ее первичного и всех возможных ключей.



Другими словами, к столбцу таблицы, которая отвечает условиям потенциальной применимости операций обновления, применимы операции обновления только в том случае, когда этот столбец может быть однозначно сопоставлен с единственным столбцом единственной таблицы, участвующей в выражении запроса, и каждая строка выражения запроса может быть однозначно сопоставлена с единственной строкой данной таблицы.

Выражение запросов удовлетворяет условию *применимости операций обновления*, если по крайней мере к одному столбцу выражения запросов применимы операции обновления. Выражение запросов удовлетворяет условию *простой применимости операций обновления*, если в разделе FROM выражения запросов содержится ссылка только на одну таблицу, и все столбцы выражения запросов удовлетворяют условию применимости операций обновления.

Выражение запросов удовлетворит условию *применимости операций вставки*, если оно удовлетворяет условию применимости операций обновления; каждая из таблиц, от которых зависит это выражение (т.е. таблиц, на которые имеются ссылки в разделе FROM), удовлетворяет условию применимости операций вставки и выражение запросов не содержит операций UNION, INTERSECT и EXCEPT. Конечно, это определение базируется на том факте, что для любой базовой таблицы условие применимости операции вставки удовлетворяется.

Приведенный набор правил считается достаточно грубым. В стандарте SQL:1999 он уточняется набором дополнительных правил, устанавливающих восприимчивость различных языковых конструкций к операциям обновления и вставки. В основе этих правил лежит понятие *функциональной зависимости*.

# ОБНОВЛЕНИЕ VIEW В POSTGRESQL



In PostgreSQL, a view is a named query stored in the database server. A view can be updatable if it meets certain conditions. This means that you can insert, update, or delete data from the underlying tables via the view.

A view is updatable when it meets the following conditions:

1. The defining query of the view must have exactly one entry in the FROM clause, which can be a table or another updatable view.
2. The defining query must not contain one of the following clauses at the top level:
3. The selection list of the defining query must not contain any:
  - [Window functions](#)
  - [Set-returning function](#)
  - [Aggregate functions](#)

• GROUP BY	• DISTINCT
• HAVING	• UNION
• LIMIT	• INTERSECT
• OFFSET FETCH	• EXCEPT

An updatable view may contain both updatable and non-updatable columns. If you attempt to modify a non-updatable column, PostgreSQL will raise an error.

When you execute a modification statement such as INSERT, UPDATE, or DELETE to an updatable view, PostgreSQL will convert this statement into the corresponding statement of the underlying table.

If you have a WHERE condition in the defining query of a view, you still can update or delete the rows that are not visible through the view. However, if you want to avoid this, you can use the WITH CHECK OPTION to define the view.

## ПРИМЕР

EMP

EMP_NO	DEPT_NO	EMP_BDATE	EMP_SAL
2440	1	1950	15000.00
2441	1	1950	16000.00
2442	1	1960	14000.00
2443	1	1960	19000.00
2444	2	1950	17000.00
2445	2	1950	16000.00
2446	2	1960	14000.00
2447	2	1960	20000.00
2448	3	1950	18000.00
2449	3	1950	13000.00
2450	3	1960	21000.00
2451	3	1960	22000.00

Предположим, что в базе данных имеется представление RICH\_EMP, определенное следующим образом:

```
CREATE VIEW RICH_EMP AS  
SELECT *  
FROM EMP  
WHERE EMP_SAL > 18000.00;
```

Как видно, в таблице EMP содержится строка, которая соответствует служащему с номером 2447, получающему зарплату в размере 20000 руб. Естественно, эта строка будет присутствовать в виртуальной таблице RICH\_EMP. Поэтому можно было бы выполнить, например, операцию

```
UPDATE RICH_EMP  
SET EMP_SAL = EMP_SAL - 3000  
WHERE EMP_NO = 4452;
```

Но если выполнение такой операции действительно допускается, то в результате строка, соответствующая служащему с номером 2447, исчезнет из виртуальной таблицы RICH\_EMP!

Чтобы избежать такого противоречивого поведения представляемых таблиц, нужно включать в определение представления раздел **WITH CHECK OPTION**. При наличии этого раздела до реального выполнения операций модификации или вставки строк через представление для каждой строки будет проверяться, что она соответствует условиям представления.

В полном виде синтаксис раздела WITH CHECK OPTION может включать ключевые слова CASCADED или LOCAL.

Предположим, что представление V2 определяется над представлением V1 следующим образом:

Пусть над V2 выполняется некоторая операция O обновления базы данных. Тогда:

```
CREATE VIEW V2 AS  
  SELECT ...  
  FROM V1  
  WHERE ...  
  [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

- если представление V2 определялось без раздела WITH CHECK OPTION, то при выполнении операции O будут проверяться все условия, определяющие ограничения целостности V1 (если в определении V1 присутствовал раздел WITH CHECK OPTION), но никаким образом не будут учитываться условия выборки, содержащееся в выражении запросов представления V2;
- если в определении представления V2 содержался раздел WITH LOCAL CHECK OPTION, то при выполнении операции O будут проверяться все условия, определяющие ограничения целостности V1, и все условия, содержащееся в выражении запросов представления V2;
- наконец, если в определении представления V2 содержался раздел WITH CASCADED CHECK OPTION, то при выполнении операции O будут проверяться все условия, определяющие ограничения целостности V1 (так, как если бы в определении V1 присутствовал раздел WITH CASCADED CHECK OPTION). Тем самым, будут проверяться все ограничения целостности, установленные для всех базовых таблиц, на которых основывается определение V1; все условия всех представлений, определенных над этими базовыми таблицами; и, конечно, все условия, содержащиеся в выражении запросов представления V2.



# ПРИМЕРЫ WITH CHECK POINT (1/5)



Чтобы пояснить результаты действия раздела WITH CHECK OPTION, допустим, что в базе данных присутствуют определения двух представлений MIDDLE\_RICH\_EMP и MORE\_RICH\_EMP:

```
CREATE VIEW MIDDLE_RICH_EMP AS
  SELECT *
    FROM EMP
   WHERE EMP_SAL < 20000.00
  [ WITH [ CASCADED | LOCAL ] CHECK OPTION ];

CREATE VIEW MORE_RICH_EMP AS
  SELECT *
    FROM MIDDLE_RICH_EMP
   WHERE EMP_SAL > 18000.00
  [ WITH [ CASCADED | LOCAL ] CHECK OPTION ];
```

В каждом из представлений MIDDLE\_RICH\_EMP и MORE\_RICH\_EMP может отсутствовать или присутствовать (в одном из двух видов) раздел WITH CHECK OPTION. В совокупности возможен один из девяти случаев:

MORE_RICH_EMP	none	LOCAL	CASCADED
MIDDLE_RICH_EMP			
none	Случай 1	Случай 2	Случай 3
LOCAL	Случай 4	Случай 5	Случай 6
CASCADED	Случай 7	Случай 8	Случай 9

MIDDLE_RICH_EMP			
EMP_NO	DEPT_NO	EMP_BDATE	EMP_SAL
2440	1	1950	15000.00
2441	1	1950	16000.00
2442	1	1960	14000.00
2443	1	1960	19000.00
2444	2	1950	17000.00
2445	2	1950	16000.00
2446	2	1960	14000.00
2448	3	1950	18000.00
2449	3	1950	13000.00

MORE_RICH_EMP			
EMP_NO	DEPT_NO	EMP_BDATE	EMP_SAL
2443	1	1960	19000.00

# ПРИМЕРЫ WITH CHECK POINT (2/5)



```
UPDATE MORE_RICH_EMP  
SET EMP_SAL = EMP_SAL + 7000.00;
```

U1

```
UPDATE MORE_RICH_EMP  
SET EMP_SAL = EMP_SAL - 7000.00;
```

U2

**Случай 1.** Ни в одном из представлений не содержится раздел WITH CHECK OPTION.

Первый неожиданный результат состоит в том, что после выполнения операции U1 тело представления MORE\_RICH\_EMP оказывается пустым. Действительно, у единственной строки таблицы EMP (со значением EMP\_NO, равным 2443), одновременно удовлетворяющей условиям обоих представлений, столбец EMP\_SAL принимает значение 26000.00. После этого строка перестает удовлетворять условию представления MIDDLE\_RICH\_EMP и исчезает из результирующей таблицы MORE\_RICH\_EMP. Этот результат может быть особенно неожиданным для пользователей базы данных, которым известно, что условие представления MORE\_RICH\_EMP имеет вид  $EMP\_SAL > 18000.00$ , и соблюдение этого условия должно сохраняться при увеличении размера зарплаты.

Выполнение операции U2 также приведет к опустошению тела MORE\_RICH\_EMP (в базовой таблице EMP не останется ни одной строки, удовлетворяющей условию этого представления). Возможно, это будет достаточно естественно для пользователей представления MORE\_RICH\_EMP, которым известно условие представления, но те, кто работает с представлением MIDDLE\_RICH\_EMP, с удивлением обнаружат в теле результирующей таблицы новые строки.

**Случай 2.** В определении представления MIDDLE\_RICH\_EMP содержится раздел WITH LOCAL CHECK OPTION, а в определении MORE\_RICH\_EMP раздел WITH CHECK OPTION отсутствует.

В этом случае, в соответствии с первыми двумя правилами проверки корректности выполнения операций обновления над представлениями, операция U1 должна быть отвергнута системой (поскольку ее выполнение нарушает условие представления MIDDLE\_RICH\_EMP). Но заметим, что такое поведение системы будет совершенно неожиданным и непонятным для тех пользователей базы данных, которым известно только определение «верхнего» представления MORE\_RICH\_EMP, поскольку операция U1 явно не может нарушить видимое ими ограничение.

С другой стороны, операция U2 будет успешно выполнена и по-прежнему приведет к опустошению тела результирующей таблицы представления MORE\_RICH\_EMP.

**Случай 3.** В определении представления MIDDLE\_RICH\_EMP содержится раздел WITH CASCADED CHECK OPTION, а в определении MORE\_RICH\_EMP раздел WITH CHECK OPTION отсутствует.

В этой ситуации будут проверяться условия, содержащиеся в определении представления MIDDLE\_RICH\_EMP, а также все ограничения целостности таблицы EMP и всех других представлений, определенных над этой базовой таблицей. В результате операция U1 будет отвергнута системой, а операция U2 будет «успешно» выполнена. Другими словами, повторится Случай 2.

**Случай 4.** В определении представления MIDDLE\_RICH\_EMP раздел WITH CHECK OPTION отсутствует, а в определении MORE\_RICH\_EMP содержится раздел WITH LOCAL CHECK OPTION.

Понятно, что в этом варианте операция U2 не сработает (ее выполнение не будет допущено условием «ограничения целостности» представления MORE\_RICH\_EMP). Но операция U1 (увеличение размера зарплаты служащих) будет успешно выполнена, поскольку она не противоречит локальным ограничениям представления MORE\_RICH\_EMP.

**Случай 5.** В определениях представлений MIDDLE\_RICH\_EMP и MORE\_RICH\_EMP содержится раздел WITH LOCAL CHECK OPTION.

Выполнение обеих операций U1 и U2 будет справедливо отвергнуто. На первый взгляд все в порядке. Но если над представлением MORE\_RICH\_EMP будет определено еще одно представление V, то мы можем получить ситуацию Случая 2, где V будет играть роль MORE\_RICH\_EMP, а MIDDLE\_RICH\_EMP – роль MORE\_RICH\_EMP.

**Случай 6.** В определении представления MIDDLE\_RICH\_EMP содержится раздел WITH CASCADED CHECK OPTION, а в определении MORE\_RICH\_EMP содержится раздел WITH LOCAL CHECK OPTION.

Снова, если над представлением MORE\_RICH\_EMP будет определено еще одно представление V, то мы можем попасть в ситуацию Случая 2, где V будет играть роль MORE\_RICH\_EMP, а MIDDLE\_RICH\_EMP – роль MORE\_RICH\_EMP.



**Случай 7.** В определении представления MIDDLE\_RICH\_EMP раздел WITH CHECK OPTION отсутствует, а в определении MORE\_RICH\_EMP содержится раздел WITH CASCADED CHECK OPTION.

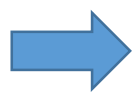
Если над представлением MORE\_RICH\_EMP будет определено еще одно представление V, то мы можем попасть в ситуацию Случая 3, где V будет играть роль MORE\_RICH\_EMP, а MIDDLE\_RICH\_EMP – роль MORE\_RICH\_EMP.

**Случай 8.** В определении представления MIDDLE\_RICH\_EMP содержится раздел WITH LOCAL CHECK OPTION, а в определении MORE\_RICH\_EMP – раздел WITH CASCADED CHECK OPTION.

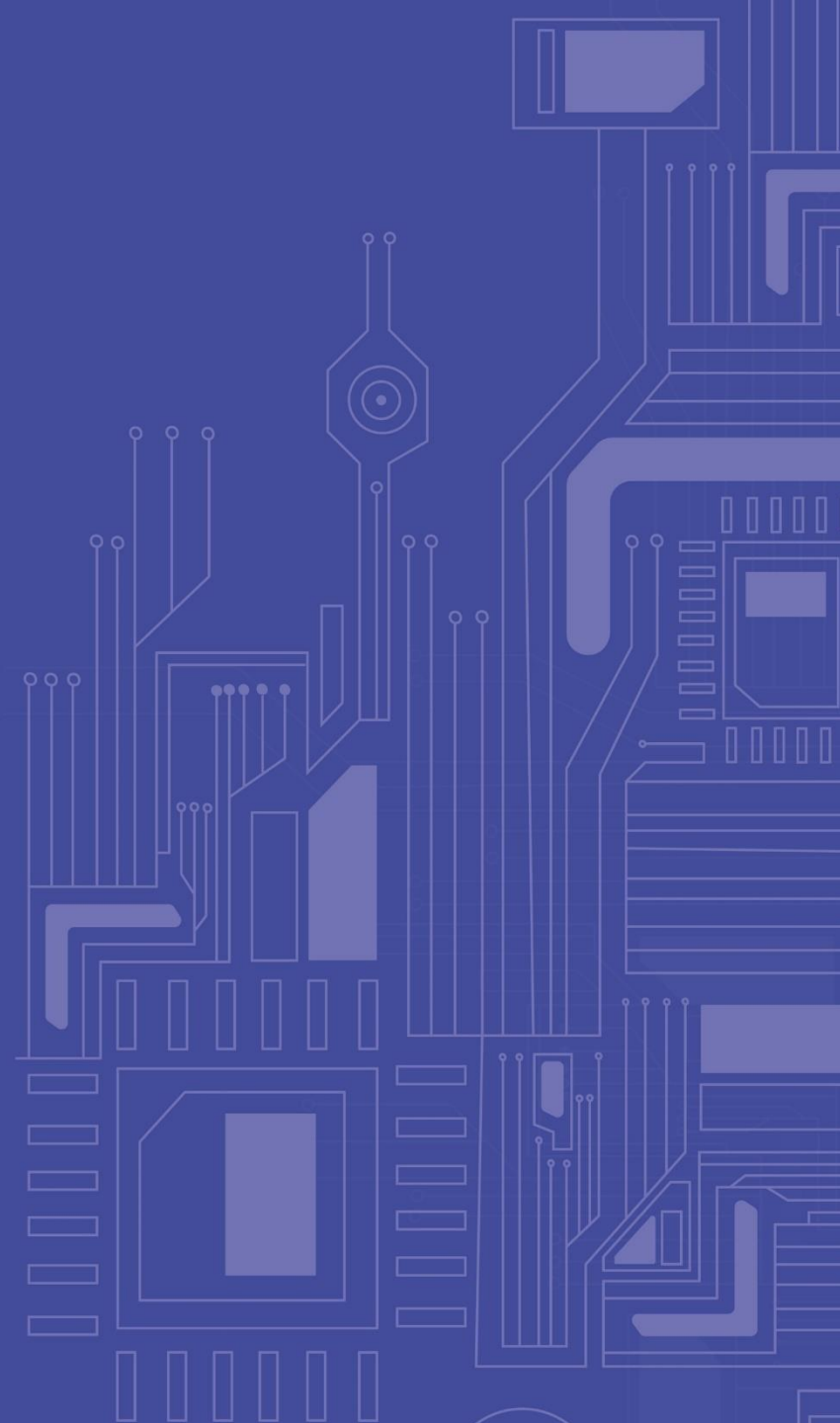
Если над представлением MORE\_RICH\_EMP будет определено еще одно представление V, то мы можем получить ситуацию Случая 3, где V будет играть роль MORE\_RICH\_EMP, а MIDDLE\_RICH\_EMP – роль MORE\_RICH\_EMP.

**Случай 9.** В определениях представлений MIDDLE\_RICH\_EMP и MORE\_RICH\_EMP содержится раздел WITH CASCADED CHECK OPTION.

Только в этом случае операции обновления будут выполняться корректно, независимо от того, имеются ли в базе данных представления, определенные над MORE\_RICH\_EMP или между MORE\_RICH\_EMP, MIDDLE\_RICH\_EMP и EMP.



Очевидный вывод из приведенного анализа заключается в том, что единственным способом обеспечить корректность выполнения операций обновления через представления (допускающие операции обновления) является включение в определение каждого представления раздела WITH CASCADED CHECK OPTION.



# СЕМИНАР



1. На Git лежит два файла .csv: users(1) и log(1). Надо скачать.
2. В этих файлах лежит информация о пользователях, делавших ставки, времени захода на платформу и выигрышах/проигрышах.
3. В log(1).csv есть данные идентификатора пользователя, времени посещения, дате посещения, размере ставки и размере выигрыша.
4. Данные из файла log(1).csv надо будет загрузить в таблицу LOG с колонками user\_id, time, bet, win.
5. Файл users(1).csv в кодировке koi8\_r. Перед обработкой надо перекодировать, либо научиться записывать данные из этой кодировки.
6. Данные из файла users(1).csv надо загрузить в таблицу USERS со столбцами user\_id, email, geo.
7. Данные грязные, их надо почистить. Например, надо удалить строки с ошибками в user\_id, оставить только значения вида user\_N, где N - значение идентификатора.
8. Теперь надо ответить на несколько вопросов по бизнесу:
  - a) Сколько раз человеку надо прийти, чтобы сделать ставку?
  - b) Каков средний выигрыш в процентах?
  - c) Каков баланс по каждому пользователю?
  - d) Какие города самые выгодные?
  - e) В каких городах самая высокая ставка?
  - f) Сколько в среднем времени проходит от первого посещения сайта до первой попытки?