Databases

Seminar 7
CTE, Views
Triggers

Common Table Expressions (CTE)

CTE (Common Table Expression) (a synonym for a subquery; exists exactly for the time of execution of one specific query) is a temporary table with a name that is used in the query.

WITH provides a way to write additional statements for use in large queries. These operators, which are also called Common Table Expressions (CTE), can be represented as definitions of temporary tables that exist only for one query. An additional operator in the WITH clause can be

SELECT, INSERT, UPDATE or DELETE, and the WITH clause itself is attached to the main operator, which

can also be SELECT, INSERT, UPDATE or DELETE.

Syntax

WITH [RECURSIVE] cte_query_name

AS (cte_query)

main_query;

This is a way to specify a temporary set of results for use in DML instructions.

Allows:

- to simplify the query
- to implement recursive queries

Example

Displays sales results only for advanced regions. The WITH clause defines two additional operators regional_sales and top_regions so that the result of regional_sales is used in top_regions, and the result of top_regions is used in the main SELECT query.

```
WITH
    regional_sales AS (
        SELECT
            region,
            SUM(amount) AS total_sales
        FROM
            orders
        GROUP BY
            region),
    top_regions AS (
        SELECT
            region
        FROM
            regional_sales
        WHERE
            total_sales > (SELECT SUM(total_sales) / 10 FROM
regional_sales))
```

This example could be rewritten without WITH, but then we would need two levels of nested SELECT subqueries. In the way shown above, this can be done a little easier.

```
SELECT
    region,
    product,
    SUM(quantity) AS product_units,
    SUM(amount) AS product_sales
FROM
    orders
WHERE
    region IN (SELECT region FROM
top_regions)
GROUP BY
    region,
    product;
```

Recursive queries

The optional RECURSIVE statement turns WITH from just a convenient syntactic construct into a means of implementing what is not possible in standard SQL. Using RECURSIVE, the WITH query can refer to its own result.

```
with recursive <cte_name> (<parameters>) as (
        <recursive base>    -- non-recursive expression
        union all           -- UNION or UNION ALL
        <recursion step>    -- recursive expression, may use the result of the
query
)
<main query>;
```

Example

Query to get the sum of integers from 1 to 100:

```
WITH RECURSIVE t(n) AS (
    VALUES (1)      -- данные, для которых рекурсия не нужна

    UNION ALL

    SELECT
        n + 1    -- рекурсивная часть
    FROM
        t
    WHERE
        n < 100
)
SELECT
    sum(n)
FROM  t;
```

Note!
Strictly speaking, this process is iterative, not recursive, but the SQL Standards Committee has chosen the term RECURSIVE.

Calculating recursive CTE

1. Determine the value of a non-recursive expression. If UNION is specified, then discard the duplicate rows. Place the remaining rows in the result of a recursive query, as well as in a temporary worktable
2. As long as the temporary table is not empty:
    a. Determine the value of a recursive expression by replacing the current contents of the worksheet with a recursive reference to yourself. If UNION is specified, then discard the duplicate rows. Include all remaining rows in the result of a recursive query, as well as in a temporary table
    b. Replace the contents of the worktable with the contents of the intermediate table and clear the intermediate table

Example

```
WITH RECURSIVE
    included_parts(sub_part, part, quantity) AS
(
    SELECT
        sub_part,
        part,
        quantity
    FROM
        parts
    WHERE
        part = 'our_product'

    UNION ALL

    SELECT
        p.sub_part,
        p.part,
        p.quantity
    FROM
        included_parts pr,
        parts p
    WHERE
        p.part = pr.sub_part)
```

Recursive queries are usually used to work with hierarchical or tree-like data structures. A useful example of this is a query that finds all the direct and indirect components of a product using only a table with direct links:

```
SELECT
    sub_part,
    SUM(quantity) AS total_quantity
FROM
    included_parts
GROUP BY
    sub_part;
```

Changing data in WITH

This query actually moves rows from products to products_log.
The DELETE operator in WITH deletes the specified rows from products and returns their contents in the RETURNING clause; and then the main query reads this content and inserts it into the products_log table.

It should be noted that the WITH clause in this case is attached to the INSERT statement, and not to the SELECT nested in INSERT.

This is necessary because WITH can contain operators that modify data only at the top level of the query. However, the usual WITH visibility rules apply, so that the WITH result can be refer to and from the nested SELECT statement.

Example 1:

```
WITH moved_rows AS (
    DELETE FROM
        products
    WHERE
        date >= '2010-10-01' AND
        date < '2010-11-01'
    RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;
```

Operators that modify data in WITH are usually supplemented with the RETURNING[^2] clause, as shown in example 1.

It is important to understand that the temporary table that can be used in the rest of the query is created from the RETURNING result, and not the target table of the operator.

If the operator modifying the data in WITH is not supplemented with the RETURNING clause, a temporary table is not created and it cannot be accessed in the rest of the query.

However, such a request will still be executed. Consider the not very practical example 2.

Example 2:

```
WITH
    t AS (
        DELETE FROM foo
    )
DELETE FROM bar;
```

It will delete all rows from the foo and bar tables. At the same time, the number of involved rows that the client will receive will be counted only by the rows removed from the bar.

Views

Views are a virtual table with contents (columns and rows) determined by the query.

A view is a virtual (logical) table that represents a named query (a synonym for a query) that will be substituted as a subquery when using the view. It is used if it is necessary to frequently make some kind of request with complex logic.

- It is not an independent part of the data set
- It is calculated dynamically based on data stored in real tables
- Changes to the data in the tables are immediately reflected in the content of the views

The view can be used for the following purposes:

- To direct, simplify and customize the perception of information in the database by each user.
- As a security mechanism that allows users to access data through views,

Views

Advantages:

- Security: It is possible to artificially restrict the information to which the user has access.
- Simplicity of queries: when writing queries, we refer to the view, as well as to a regular table.
- Protection against changes: the user does not need to know that the structures / names of the tables have changed. It is enough to update the view.

Disadvantages:

- Performance: A seemingly simple query using a view can actually be very complicated because of the logic “sewn” into the view.
- Control: a view can be based on a view, which in turn is also based on another view, etc.
- Update limit:* not every view can be updated, which is not always obvious to the user.

Syntax

```
CREATE
[ OR REPLACE ] [ TEMP |
TEMPORARY ] [ RECURSIVE ]
  VIEW name [ ( column_name
[, ...] ) ]
  [
WITH (view_option_name [=
view_option_value] [, ...]) ]
  AS query
  [
WITH [ CASCADED | LOCAL ]
CHECK
OPTION ]
```

- CREATE VIEW – create a new view.
- CREATE OR REPLACE VIEW – create or replace an existing view. In case of replacement, all fields of the old view (names, order, data type) must be present in the new view. Only adding new fields is allowed.
- TEMPORARY | TEMP – temporary representation, will exist until the end of the session.
- view_name is the name of the view.
- column_name is a list of view fields. If omitted, the request fields are used.
- query – SELECT or VALUES commands.

View Changes:

https://www.postgresql.org/docs/current/sql-alterview.html

ALTER VIEW [IF EXISTS] name ALTER [COLUMN] column_name SET DEFAULT expression
ALTER VIEW [IF EXISTS] name ALTER [COLUMN] column_name DROP DEFAULT
ALTER VIEW [IF EXISTS] name OWNER TO new_owner
ALTER VIEW [IF EXISTS] name RENAME TO new_name
ALTER VIEW [IF EXISTS] name SET SCHEMA new_schema
ALTER VIEW [IF EXISTS] name SET (view_option_name [= view_option_value] [, ...])
ALTER VIEW [IF EXISTS] name RESET (view_option_name [, ...])
DROP VIEW [IF EXISTS] name [, ...] [CASCADE | RESTRICT]

Creating a view

If you add columns to the table after creating the view, they will not be in the view.

```
CREATE VIEW greeting AS
SELECT 'Hello World';

CREATE VIEW greeting AS
SELECT text 'Hello World' AS hello;


CREATE VIEW comedies AS
SELECT
    *
FROM
    films
WHERE
    kind = 'Comedy';
```

TEMPORARY

The view is created as a temporary one. It is deleted at the end of the session.

```
CREATE TEMP VIEW greeting AS
SELECT 'Hello World';
```

RECURSIVE

The view is created as recursive. Equivalent forms:

```
CREATE RECURSIVE VIEW [ schema
] view_name (column_names
) AS
SELECT ...;
```

```
CREATE VIEW [ schema.
] view_name AS
WITH RECURSIVE view_name
(column_names) AS (SELECT ...)
SELECT column_names
FROM view_name;
```

Recursive representation - example

```
CREATE RECURSIVE VIEW public.nums_1_100 (n) AS
VALUES (1)
UNION ALL
SELECT
    n + 1
FROM
    nums_1_100
WHERE
    n < 100;
```

Types of views

1. horizontal — data restricted by rows:

```
CREATE VIEW V_IT_EMPLOYEE AS
SELECT
    *
FROM
    EMPLOYEE
WHERE
    DEPARTMENT_NM = 'IT';
```

2. vertical — data restricted by columns:

```
CREATE VIEW V_EMP AS
SELECT
    EMP_NM,
    DEPARTMENT_NM
FROM
    EMPLOYEE;
```

Updatable views

A view is called updatable if the UPDATE and DELETE operations are applied to it to change the data in the tables on which this view is built.

Requirements:

- Exactly 1 source in the FROM clause, which is a table or an updatable view
- The request must not contain WITH, DISTINCT, GROUP BY, HAVING, LIMIT or OFFSET
- The request must not contain UNION, INTERSECT, or EXCEPT operators the select-list of the request must not contain aggregate, window, or functions that return sets.

  WITH [CASCADED | LOCAL] CHECK

  OPTION

Sets the behavior of updatable views: checks that do not allow writing data that is invisible through the view

- LOCAL – checks are performed only on the view itself
- CASCADED – checks are performed on the view itself, on the source, and so on along the chain of requests

Updated views – Example 1:

```
CREATE VIEW universal_comedies AS
SELECT
    *
FROM
    comedies
WHERE
    classification = 'U'
WITH LOCAL CHECK OPTION;
```

An attempt to insert or edit a row with classification <> 'U' will result in an error. But at the same time, inserting or editing a row with kind <> 'Comedy' will be successful.

Updated views – Example 2:

```
CREATE VIEW universal_comedies AS
SELECT
    *
FROM
    comedies
WHERE
    classification = 'U'
WITH CASCADED CHECK OPTION;
```

An attempt to insert or edit a row with classification <> 'U' or kind <> 'Comedy' will result in an error.
Columns in the updated view can be either updatable or non-updatable.

Updatable views – Example 3:

```
CREATE VIEW comedies AS
SELECT
    f.*,
    country_code_to_name(f.country_code) AS country,
    (SELECT avg(r.rating) FROM user_ratings r WHERE r.film_id = f.id) AS
avg_rating
FROM
    films f
WHERE
    f.kind = 'Comedy';
```

All columns of the films table are updatable. The columns country and avg_rating are read only. If the view cannot be made updatable, but there is a need for it, use the INSTEAD OF trigger. This is a function that will handle data modification operations – we'll look at it later.

Note

Active use of views is a key aspect of good SQL database design. Views allow you to hide the internal structure of your tables, which may change as the application develops, behind reliable interfaces.

Views can be used almost anywhere that regular tables can be used. And quite often views are created on the basis of other views.

Practice

Assignment on database queries:

Value Generation Task:

1. Write a query to get the sum of numbers from 1 to 100.
2. Write a query to get the sum of an arithmetic progression with a step of 5, starting from 3 and ending at 48.
3. Write a query to get the sum of a geometric progression with a multiplier of 3 and the first element of 1, containing 10 elements, using the limit.

Practice

Hierarchy Construction Task:
Let's assume there is a table containing the hierarchy of departments of a bank.

1. For each dep_id output a row in format «Группа, Отдел, Управление, ...» to which it relates;
2. Display dep_id of 5-th level structural divisions (i.e., «Группа системного анализа, Отдел трансформации и загрузки данных, Управление хранилищ данных и отчетности, Департамент ИТ, Банк»)

```
create table public.department (dep_id, par_dep_id, dep_name) as
select 1, NULL, 'Банк' union
select 2, 1, 'Управление анализа кредитных рисков' union
select 3, 2, 'Отдел риск-менеджмента малого и среднего бизнеса' union
select 4, 2, 'Отдел риск-менеджмента розничного бизнеса' union
select 5, 1, 'Департамент ИТ' union
select 6, 5, 'Управление хранилищ данных и отчетности' union
select 7, 6, 'Отдел очистки и контроля качества данных' union
select 8, 7, 'Группа администрирования хранилищ данных' union
select 9, 7, 'Группа контроля качества данных' union
select 10, 5, 'Отдел отчетности и витрин данных' union
select 11, 5, 'Отдел трансформации и загрузки данных' union
select 12, 11, 'Группа системного анализа' union
select 13, 11, 'Группа разработки';
```