A Different Type of SQL Recursion with PostgreSQL #1

vbilopav started this conversation in **PostgreSQL**

vbilopav 12 hours ago Maintainer edited

A Different Type of SQL Recursion with PostgreSQL

PostgreSQL offers a powerful procedural programming model out of the box (in addition to the standard SQL).

You can combine that with the standard SQL approach to overcome almost any issue and solve any programming task. Sometimes, the good old procedural approach may be a more straightforward way out of the complex data problems than standard SQL.

This article will try to demonstrate that approach to a complex problem example.

The Problem

Let's say we want to write a function that will return **all related tables** for a table name in a parameter.

We will start with a made-up schema:

```
create table division_status (
    division_status_id int primary key,
    name text
);

create table department_status (
    department_status_id int primary key,
    name text
);

create table divisions (
    division_id int primary key,
    name text,
    owner_user_id int,
    parent_division_id int,
    division_status_id int
);

create table departments (
    department_id int primary key,
    name text,
    owner_user_id int,
    parent_department_id int,
        division_id int,
        department_status_id int
    );

    create table users (
        user_id int primary key,
        name text,
        owner_user_id int,
        department_id int
    );

    alter table users add foreign key (department_id) references departments (department_id);
    alter table users add foreign key (owner_user_id) references users (user_id);

    alter table departments add foreign key (owner_user_id) references users (user_id);
    alter table departments add foreign key (parent_department_id) references departments (department_id);
    alter table departments add foreign key (division_id) references divisions (division_id);
    alter table departments add foreign key (department_status_id) references department_status (department

    alter table divisions add foreign key (owner_user_id) references users (user_id);
    alter table divisions add foreign key (parent_division_id) references divisions (division_id);
    alter table divisions add foreign key (division_status_id) references division_status (division_status_
```

As we can see, we have the following tables:

- `users` - references itself (`users`) and table `departments`
- `departments` references itself, table `users` , table `divisions` , and table `department_status` .
- `divisions` references itself, table `users` and table `division_status` .

So, what we want to achieve is to have a function that will return all related tables directly and indirectly, together with the level of relation.

For example, we want to be able to call a function `select_foreign_tables` for table `users` as a parameter, like this:

```
select * from select_foreign_tables('public', 'users');
```

This should return the following result:

table_schema	table_name	fk_table_schema	fk_table_name	level
public	users	public	departments	1
public	users	public	users	1
public	departments	public	divisions	2
public	departments	public	department_status	2
public	departments	public	users	2
public	departments	public	departments	2
public	divisions	public	divisions	3
public	divisions	public	users	3

table_schema	table_name	fk_table_schema	fk_table_name	level
public	divisions	public	division_status	3

As we can see, each row should also have a level assigned. For example, if table `users` directly references table `departments` , that's level 1. But, `department_status` is level 2 because `users` references `departments` and departments references `department_status` , and so on...

Simple, right? It is just another data-related task. Now, how would we do that?

Let's start with a basic query.

Basic Query

A basic query is the PostgreSQL system query that will query system tables to fetch information about direct references. It's a rather complicated query, and I won't get into it, but for the sake of simplicity, let's create a view out of it:

```
create view fk_tables as
select
    con.schema as table_schema,
    cl2.relname as table_name,
    ns.nspname as fk_table_schema,
    cl.relname as fk_table_name
from
    (select
        unnest(con1.conkey) as parent,
        unnest(con1.confkey) as child,
        con1.confrelid,
        con1.conrelid,
        con1.contype,
        ns.nspname as schema
    from
        pg_class cl
        inner join pg_namespace ns on cl.relnamespace = ns.oid
        inner join pg_constraint con1 on con1.conrelid = cl.oid
    ) con
    inner join pg_attribute att on att.attrelid = con.confrelid and att.attnum = con.child
    inner join pg_class cl on cl.oid = con.confrelid
    inner join pg_attribute att2 on att2.attrelid = con.conrelid and att2.attnum = con.parent
    inner join pg_class cl2 on cl2.oid = att2.attrelid
    inner join pg_namespace ns on cl.relnamespace = ns.oid;

comment on view fk_tables is 'Retuls list of all tables and join related tables for each table.';
```

Fine, it's ugly and dirty; don't even look at it.

Use it as a view, for example:

```
select * from fk_tables where table_schema = 'public' and table_name = 'users';
```

This will return first-level references for the table `users` :

table_schema	table_name	fk_table_schema	fk_table_name
public	users	public	departments
public	users	public	users

Now, all we have to do is run the same query again for each of these FK tables in this results, and we're done.

That's why it's called recursion.

Solution 1

Luckily for us, PostgreSQL has recursive common-table queries as a standard feature, and we can do the first solution with that.

For a recursion seed, we can use the filter on our `fk_tables` view.

```
select
    t.table_schema,
    t.table_name,
    t.fk_table_schema,
    t.fk_table_name
from
    fk_tables t
where
    t.table_schema = 'public'
    and t.table_name = 'users'
```

In the recursive part, we can return the FK table as the base table and join the `fk_tables` view again to get the real FK tables like this:

```
    select
        rec.fk_table_schema as table_schema,
        rec.fk_table_name as table_name,
        t.fk_table_schema,
        t.fk_table_name
    from
        _recursive_cte rec
        inner join fk_tables t on
            rec.fk_table_schema = t.table_schema
            and rec.fk_table_name = t.table_name
```

All together, it should look like this:

```
with recursive _recursive_cte as
(
    select
        t.table_schema,
        t.table_name,
        t.fk_table_schema,
        t.fk_table_name

    from
```

```
            fk_tables t
        where
            t.table_schema = 'public'
            and t.table_name = 'users'

        union

        select
            rec.fk_table_schema as table_schema,
            rec.fk_table_name as table_name,
            t.fk_table_schema,
            t.fk_table_name

        from
            _recursive_cte rec
            inner join fk_tables t on
                rec.fk_table_schema = t.table_schema
                and rec.fk_table_name = t.table_name
    )
    select
        table_schema,
        table_name,
        fk_table_schema,
        fk_table_name
    from
        _recursive_cte
```

Confusing? We have yet to start with the confusing part.

Anyway, this query will return all FK tables for `users` at all levels:

table_schema	table_name	fk_table_schema	fk_table_name
public	users	public	departments
public	users	public	users
public	departments	public	users
public	departments	public	departments
public	departments	public	divisions
public	departments	public	department_status
public	divisions	public	users
public	divisions	public	divisions
public	divisions	public	division_status

However, we still need levels. We can count it manually, but we'd rather have instead the database engine do it.

First naive try it to add level 1 at recursion seed like this:

```
    select
        t.table_schema,
        t.table_name,
            t.fk_table_schema,
            t.fk_table_name,
            1 as level

        from
            fk_tables t
        where
            t.table_schema = 'public'
            and t.table_name = 'users'
```

And have it increased by one in each recursion step:

```
    select
        rec.fk_table_schema as table_schema,
        rec.fk_table_name as table_name,
        t.fk_table_schema,
        t.fk_table_name,
        rec.level + 1 as level
    from
        _recursive_cte rec
        inner join fk_tables t on
            rec.fk_table_schema = t.table_schema
            and rec.fk_table_name = t.table_name
```

This is very naive. We have a `union` of seed query and recursion query. The `union` operator filters out duplicate records by default. From the moment we've introduced a level number that is unique, duplicates won't be filtered out. And since `users` references `departments` and `departments` references `users` again, the query will end up in an infinite loop, which potentially could crash the server.

So this is reckless and even dangerous.

What we have here is a very well-known data problem - the gap and islands problem.

What we want to do is this:

- keep the sort order and
- on each change in `table_schema` and `table_name` - increase the counter by one.

To be able to do this, we first need to add two more calculated fields:

- Row number - we will use this just to keep the sort order.
- Calculated field that is 1 when the table in a row changed from the previous row. Otherwise, it is 0.

Here is what it looks like:

```
    select
        table_schema,
        table_name,
        fk_table_schema,
        fk_table_name,
        row_number() over(),
        case
            when lag(table_schema) over() = table_schema and lag(table_name) over() = table_name
            then 0
        else 1
    end as island_flips
from
    _recursive_cte
```

We can wrap this into another CTE and do the last calculation in the final query:

```
select
    table_schema,
    table_name,
    fk_table_schema,
    fk_table_name,
    island_flips,
    sum(island_flips) over (order by row_number) as level
from
    _cte
order by
    row_number
```

Field `island_flips` will be one if the previous table is different; otherwise, it is 0. Now, if we do a cumulative sum over those values with `sum(island_flips) over (order by row_number) as level` - we will get the correct levels.

Here is the final version of the plain SQL function:

```
create or replace function select_foreign_tables(
    _table_schema text,
    _table_name text
)
returns table (
    table_schema text,
    table_name text,
    fk_table_schema text,
    fk_table_name text,
    level int
)
language sql
as
$$
with recursive _recursive_cte as
(
    select
        t.table_schema,
        t.table_name,
        t.fk_table_schema,
        t.fk_table_name
    from
        fk_tables t
    where
        t.table_schema = _table_schema
        and t.table_name = _table_name

    union

    select
        rec.fk_table_schema as table_schema,
        rec.fk_table_name as table_name,
            t.fk_table_schema,
            t.fk_table_name

        from
            _recursive_cte rec
            inner join fk_tables t on
                rec.fk_table_schema = t.table_schema
                and rec.fk_table_name = t.table_name

    ), _cte as (

        select
            table_schema,
            table_name,
            fk_table_schema,
            fk_table_name,
            row_number() over(),
            case
                when lag(table_schema) over() = table_schema and lag(table_name) over() = table_name
                then 0
                else 1
            end as island_flips
        from
            _recursive_cte
    )
    select
        table_schema,
        table_name,
        fk_table_schema,
        fk_table_name,
        sum(island_flips) over (order by row_number) as level
    from
        _cte
    order by
        row_number
    $$;
```

Do you think this sounds complicated?

It sounds complicated because it is.

Let's try to solve this problem with a completely different approach.

Solution 2

In this approach, we will use an old-school procedural programming approach - instead of using the recursive SQL query - we will use a normal function recursion as you would with any other language.

So, instead of creating a function of the `sql` type - this new function will be of the `plpgsql` type. This allows us to use procedural extensions like "if branching", "loops" and so on... Those are the concepts most developers are familiar with.

The idea is to create a temporary table that we will insert in recursive calls and return values from that temporary table on the last call.

The function skeleton will look like this:

```
create or replace function select_foreign_tables2(
    _table_schema text,
    _table_name text,
    _level int = 1
)
returns table (
    table_schema text,
    table_name text,
    fk_table_schema text,
    fk_table_name text,
    level int
)
language plpgsql
as
$$
begin

    create temp table if not exists _result (
        table_schema text,
        table_name text,
        fk_table_schema text,
        fk_table_name text,
        level int
    ) on commit drop;

    if exists(
        select 1 from _result t where t.table_schema = _table_schema and t.table_name = _table_name
    ) then
        return;
    end if;

    --
    -- Insert  _result table in a loop with recursive calls here
    --

    return query
    select
        t.table_schema,
        t.table_name,
        t.fk_table_schema,
        t.fk_table_name,
        t.level
    from
        _result t;
end;
$$;
```

What we first need to do is to create a temporary table:

```
create temp table if not exists _result (
    table_schema text,
    table_name text,
    fk_table_schema text,
    fk_table_name text,
    level int
) on commit drop;
```

This table will be automatically dropped when transaction commits (or rollbacks). All PostgreSQL functions in procedural language are in transaction by default (notice `begin` and `end`).

When we call this function again in a recursion - it will still be the same transaction. The new transaction is merged into the existing transaction (PostgreSQL doesn't support nested transactions, as far as I know).

However, since it will be called recursively, that temp table may already exist. That's why we also need to check if it exists `create temp table if not exists`.

Next, we must check did we went trough this step already. Meaning, have we processed that table from parameters. If we have, exit immediately:

```
if exists(
    select 1 from _result t where t.table_schema = _table_schema and t.table_name = _table_name
) then
    return;
end if;
```

And now, we are ready to insert the data:

- Loop through the `fk_tables` view for table parameters and for each record:
- Insert into the result.
- Call recursively the same function again but, use `fk_table_schema` and `fk_table_name` as parameters and increase one level.

Here is what that loop looks like:

```
for _row in (
    select
        t.table_schema,
        t.table_name,
        t.fk_table_schema,
        t.fk_table_name
    from
        fk_tables t
    where
        t.table_schema = _table_schema
        and t.table_name = _table_name
) loop

    insert into _result
    select
        _row.table_schema,
        _row.table_name,
        _row.fk_table_schema,
        _row.fk_table_name,
        _level;

    perform sys.select_foreign_tables2(
        _row.fk_table_schema,
        _row.fk_table_name,
        _level + 1
    );
```

```
end loop;
```

And finally, the entire function:

```
create or replace function select_foreign_tables2(
    _table_schema text,
    _table_name text,
    _level int = 1
)
returns table (
    table_schema text,
    table_name text,
    fk_table_schema text,
    fk_table_name text,
    level int
)
language plpgsql
as
$$
declare
    _row record;
begin
    create temp table if not exists _result (
        table_schema text,
        table_name text,
        fk_table_schema text,
        fk_table_name text,
        level int
    ) on commit drop;

    if exists(
        select 1 from _result t where t.table_schema = _table_schema and t.table_name = _table_name
    ) then
        return;
    end if;

    for _row in (
        select
            t.table_schema,
            t.table_name,
            t.fk_table_schema,
            t.fk_table_name
        from
            fk_tables t
        where
            t.table_schema = _table_schema
            and t.table_name = _table_name
    ) loop

        insert into _result
        select
            _row.table_schema,
            _row.table_name,
            _row.fk_table_schema,
            _row.fk_table_name,
            _level;

        perform sys.select_foreign_tables2(
                _row.fk_table_schema,
                _row.fk_table_name,
                _level + 1
            );

        end loop;

        return query
        select
            t.table_schema,
            t.table_name,
            t.fk_table_schema,
            t.fk_table_name,
            t.level
        from
            _result t;
    end;
    $$;
```

Conclusion

Which of these two approaches is better, in your opinion?

Recursive CTE queries are powerful but confusing and limiting. This old-school procedural programming approach may be a much better fit for developers untrained with SQL, and that is, in my opinion, the majority. On the other hand, the procedural approach may be something most developers feel comfortable with.

Leave comments below.

↑ 1

1 comment

Oldest Newest | Top

duki994 4 hours ago

Given the amount of effort and the way recursion is implemented in solution 2, I'd go for solution 2. It's more elegant and more C language-like and more in-line HW would execute recursion (mechanical sympathy).

Though solution 1 is ok, and it's not that convoluted and complex, it's overcomplicated for a simple requirement like this.

It just shows how algebra and lambda calculus theory (procedural, functional and OOP paradigms) can solve some problems in a way more elegant way than set theory (SQL-like DBs and Codd's relational model).

All major DB vendors have procedural extensions and languages associated with them:
T-SQL (MSSQL), plpgsql (PostgreSQL), PL/SQL (Oracle) etc.

Just use them. Portability is not an issue when having similar Turing complete procedural language SQL extensions.

2

0 replies

Category

PostgreSQL

Labels

postgresql programming advanced

2 participants