Advanced SQL - Common Table Expressions

JANUARY 26, 2018

This is the second article in my series discussing advanced SQL concepts. I want to describe features that are well supported in popular database management systems for quite some time, but somehow many people still don't know about their existence. I'd like to explain them with examples, first giving a problem to solve using "plain old" SQL and then showing a better solution using advanced SQL.

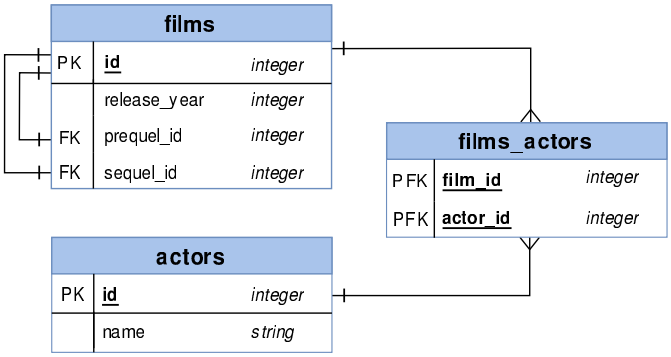
You can find the first article about window functions here.

This time I'd like to discuss **Common Table Expressions (CTE)**.

In this post I'll be using PostgreSQL 10, because it's the most feature-rich open source database available. Common Table Expressions have been available since Postgres 8.4, so any modern version will be fine. They are also supported by other popular RDBMSes.

Problem

This time we'll be working with three tables: `films`, `actors` and a linking table `films_actors` between them:



Database schema

Something that may attract your attention are the columns `prequel_id` and `sequel_id`. They are foreign key referencing to the very same table `films` and pointing respectively to a prequel or sequel of a given film. To make things clear I prepared a set of sample rows for this table:

id	release_year	prequel_id	sequel_id
1	2015	NULL	3
2	2015	NULL	4
3	2015	1	5
4	2016	2	NULL
5	2016	3	NULL
6	2017	NULL	NULL

Films rows

As you can see there are two chains of the prequel-sequel relation (1-3-5 and 2-4) and one film that has no connections at all. I don't think it's necessary to provide the content of the other two tables - let's pretend they have some meaningful data.

Here comes the first example:

Example 1. For each film count number of actors starring in it.

Easy. The result looks like this:

id	actors
1	6
2	2
3	5
4	7
5	2
6	1

Result rows with actors counts

There is no catch here - it's as simple as it looks. All we need to do is to join `films` with `film_actors` and count number of rows:

```
SELECT
  f.id, COUNT(*) AS actors
FROM films f
LEFT JOIN films_actors fa ON f.id = fa.film_id
GROUP BY f.id
```

With that solution in mind let's move on to the next thing.

Example 2. For each film, its prequel and sequel count number of actors starring in them.

This task doesn't look any more complicated than the previous one. Right?

id	prequel_id	sequel_id	actors	prequel_actors	sequel_actors
1	NULL	3	6	NULL	5
2	NULL	4	2	NULL	7
3	1	5	5	6	2
4	2	NULL	7	2	NULL
5	3	NULL	2	5	NULL
6	NULL	NULL	1	NULL	NULL

Result rows with actors counts for prequel and sequel

There is more than one solution, but the most straightforward just uses three identical queries to get information about the film and both its prequel and sequel:

```
SELECT
  films.id, films.prequel_id, films.sequel_id,
  films.actors,
  prequels.actors AS prequel_actors,
  sequels.actors AS sequel_actors

FROM (
  SELECT f.*, COUNT(*) as actors
```

```
  FROM films f
  JOIN films_actors fa ON f.id = fa.film_id
  GROUP BY f.id, f.prequel_id
) films

LEFT JOIN (
  SELECT f.*, COUNT(*) as actors
  FROM films f
  JOIN films_actors fa ON f.id = fa.film_id
  GROUP BY f.id
) prequels ON films.prequel_id = prequels.id

LEFT JOIN (
  SELECT f.*, COUNT(*) as actors
  FROM films f
  JOIN films_actors fa ON f.id = fa.film_id
  GROUP BY f.id
) sequels ON films.sequel_id = sequels.id
```

The query works, but has some problems. It's not easy to read and understand as you have to carefully compare, line by line, all three subqueries to make sure that they do exactly the same thing. Modifying one means also that you need to change others as well. Wouldn't it be nice to write identical parts once and only refer to them somehow?

Moreover this also shows one general disadvantage of SQL - you need to read queries from inside to outside - because that's the order in which they are executed. I think it would look much better if we had them one below the other.

And that's what CTE are mainly about.

Common Table Expressions (CTE)

CTE are a mechanism that allows to define temporary named result sets existing just for one query (you may also think about them as temporary "tables" or "views"). Let's see how they work in practice by solving Example 1 once again:

```
WITH film_with_actors_count AS (
  SELECT f.*, count(*) AS actors
  FROM films f
  JOIN films_actors fa ON f.id = fa.film_id
  GROUP BY f.id
)

SELECT
  f.id, f.actors
FROM film_with_actors_count f
```

CTE are defined using `WITH … AS` clause. Inside them you can put almost any SQL statement you like (not only `SELECT`, but also `INSERT`, `UPDATE` or `DELETE`). Every CTE has a name, so you can easily refer to it in the main query, just like I did in the example above. Fun fact: there is no comma or semicolon between the last CTE definition and the main query.

Of course you can refer to them as many times you want. As a reference take a look at the new solution to Example 2:

```
WITH film_with_actors_count AS (
  SELECT f.*, count(*) AS actors
  FROM films f
  JOIN films_actors fa ON f.id = fa.film_id
  GROUP BY f.id
)

SELECT
  films.id, films.prequel_id, films.sequel_id,
  films.actors,
  prequels.actors AS prequel_actors,
  sequels.actors AS sequel_actors
FROM film_with_actors_count films
LEFT JOIN film_with_actors_count prequels on films.prequel_id = prequels.id
LEFT JOIN film_with_actors_count sequels on films.sequel_id = sequels.id
```

Looks better, doesn't it?

Let's see what other problems CTE can solve. To visualize the first one I'll use the example from my previous article about window functions. This time with a little complication:

Example 3. Return a single film with the greatest number of actors for each release year.

So, only one film from each year and only the one with the most actors:

release_year	id	actors
2015	1	6
2016	4	7
2017	6	1

Films with greatest number of actors for each year

To solve this problem we need to use window functions. Adding a new column with a correct values is just a matter of using `RANK()` over a correctly partitioned and ordered window:

```
SELECT
  f.id, f.release_year, COUNT(*) AS actors,
  RANK() OVER (PARTITION BY release_year ORDER BY COUNT(*) DESC) AS year_rank
FROM films f
LEFT JOIN films_actors fa ON f.id = fa.film_id
GROUP BY f.id
```

That'll give us the following result:

id	release_year	actors	year_rank
1	2015	6	1
2	2015	2	3
3	2015	5	2
4	2016	7	1
5	2016	2	2
6	2017	1	1

Films with year rank

And now we can simply add a `HAVING` clause, right?

```
SELECT
  f.id, f.release_year, COUNT(*) AS actors,
  RANK() OVER (PARTITION BY release_year ORDER BY COUNT(*) DESC) AS year_rank
FROM films f
LEFT JOIN films_actors fa ON f.id = fa.film_id
GROUP BY f.id
HAVING year_rank = 1
```

Unfortunately not:

```
ERROR:  column "year_rank" does not exist
LINE 5: HAVING year_col == 1
```

That's because window functions are not visible in any other clauses in the same query. To overcome this issue we can simple wrap the above query with another query and add necessary filtering there. Or, to make things clearer and simpler, use CTE:

```
WITH films_actors_year_rank AS (
  SELECT
    f.id, f.release_year, COUNT(*) AS actors,
    RANK() OVER (PARTITION BY release_year ORDER BY COUNT(*) DESC) AS year_rank
  FROM films f
  LEFT JOIN films_actors fa ON f.id = fa.film_id
  GROUP BY f.id
)

SELECT f.release_year, f.id, f.actors
FROM films_actors_year_rank f
WHERE year_rank = 1
```

And now it works just fine. Once more a CTE expression was used to improve readability. This is good, but are they really only about making SQL code nicer?

Well, not exactly. In fact there are problems that simply can't be solved without CTE.

Example 4. For each film return number of all its prequels and sequels.

And I'm having such thing in mind:

id	prequel_id	sequel_id	prequels_num	sequels_num
1	NULL	3	0	2
2	NULL	4	0	1
3	1	5	1	1
4	2	NULL	1	0
5	3	NULL	2	0
6	NULL	NULL	0	0

Films with numbers of all their prequels and sequels

We need to count the length of both prequel and sequel chain for each film.

If you think about this problem for a while you may realize that it's not difficult at all to check if a film has a single prequel or sequel by simply looking at its corresponding foreign key column. It's not hard to extend it to the second level either. In other words we can check if the prequel's prequel (or sequel's sequel) exists by doing a self join. Adding more nesting however requires using more subsequent joins. To solve this problem for any length of the prequel/sequel sequence we'd need something more powerful.

Something like a recursion. Wait, what? In SQL? Yes, it's possible.

Recursive CTE

Recursive CTE has an interesting ability to invoke itself. You can put the name of a CTE in its body and therefore make it run recursively. This kind of CTE takes the form of:

```
WITH RECURSIVE cte AS (
-- [non-recursive term]
  UNION ALL
-- [recursive term]
)
```

A very simple working example from Postgres documentation goes as follows:

```
WITH RECURSIVE t(n) AS (
  VALUES (1) -- non-recursive term
UNION ALL
  SELECT n+1 FROM t WHERE n < 100 -- recursive term
)
SELECT * FROM t;
```

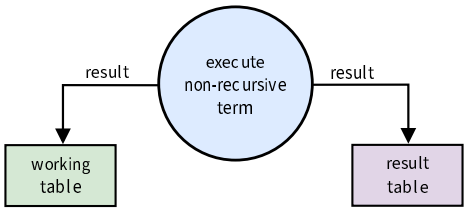
The above query generates numbers from 1 to 100.

As you can see there are few differences between normal and recursive Common Table Expression. First thing is the usage of `RECURSIVE` term in the definition, which enables the recursive mode. Second thing is that the query consists of two separate parts connected with a `UNION ALL` operator. They are called respectively "non-recursive term" and "recursive term". You can make sure that the result table will not have any duplicated by using `UNION` instead of `UNION ALL`. The last thing is the fact that the recursive term query invokes its own CTE - something not possible in the normal mode.

Let's see how recursive CTE work exactly. Under the hood Postgres uses two temporary tables: working table and result table. The latter is the place that accumulates the final result of a CTE. Technically the whole process is actually iterative, not recursive, but that's how this operation has been called by the SQL standards committee. Therefore it can be visualized in three steps:

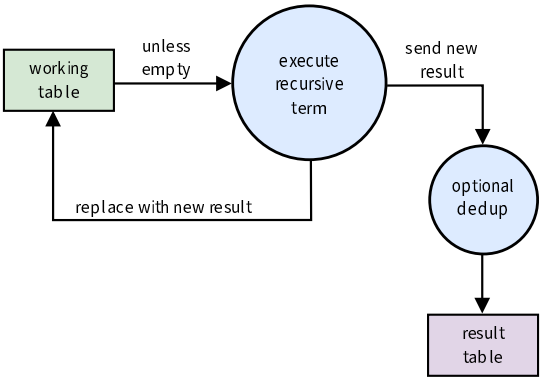
1. Initial step

Initial step is evaluated only once. Executor runs the non-recursive term and puts the result both in working and result tables:



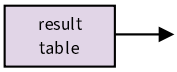
2. Repetitive step

It is evaluated many times, in a loop. Executor runs the recursive term against the content of the working table and then merges its output with the result table. It removes duplicates if the `UNION` operator was used. The result is also used to replace the content of the working table and therefore prepare it for the next step. The whole process repeats as long as the working table is not empty.



3. Final step

Database simply return the content of the result table.



Armed with that knowledge let's get back to Example 4. I asked you to find out how many prequels and sequels each film has.

Let's start with prequels count and do it step by step. First thing is to write the CTE header:

```
WITH RECURSIVE films_with_prequels_number(id, sequel_id, prequels_num) AS (
```

I added the part with column names in brackets here. From now I can skip all aliases, but I need to be careful about putting statements in the right order. The additional column `prequels_num` will hold the number of prequels for a particular film.

Now let's write the non-recursive term, which prepares the first set of rows both for working and result table. Because we're counting prequels we have to select all films that have zero prequels - their `prequel_id` column will be set to `NULL`:

```
SELECT f.id, f.sequel_id, 0 FROM films f WHERE f.prequel_id IS NULL
```

Now we have to choose the linking operator. In our case both `UNION` and `UNION ALL` work identically, because we're not

expecting any duplicates.

```
  UNION ALL
```

Now the hardest part - the recursive term. We need to take the content of working table by recursively selecting from self and joining it together with the films table, effectively replacing each film with its sequel. We also have to remember about incrementing the `prequels_num` column:

```
    SELECT f.id, f.sequel_id, fr.prequels_num + 1
    FROM films_with_prequels_number fr
    JOIN films f ON fr.sequel_id = f.id
  )
```

And that's it. Using inner `JOIN` instead of `LEFT JOIN` ensures that the execution will eventually stop, because it effectively discards all rows that have `NULL` value in its joining column (in this case `sequel_id`). And we don't have cycles here.

Now let's see at the whole query with CTE expressions for both prequels and sequels:

```
  WITH RECURSIVE films_with_prequels_number(id, sequel_id, prequels_num) AS (
    SELECT f.id, f.sequel_id, 0 FROM films f WHERE f.prequel_id IS NULL

    UNION ALL

    SELECT f.id, f.sequel_id, fr.prequels_num + 1
    FROM films_with_prequels_number fr
    JOIN films f ON fr.sequel_id = f.id
  ),

  films_with_sequels_number(id, prequel_id, sequels_num) AS (
    SELECT f.id, f.prequel_id, 0 FROM films f WHERE f.sequel_id IS NULL

    UNION ALL

    SELECT f.id, f.prequel_id, fr.sequels_num + 1
    FROM films_with_sequels_number fr
    JOIN films f ON fr.prequel_id = f.id
  )

  SELECT f.id, f.prequel_id, f.sequel_id, fpn.prequels_num, fsn.sequels_num FROM films f
  LEFT JOIN films_with_prequels_number fpn ON f.id = fpn.id
  LEFT JOIN films_with_sequels_number fsn ON f.id = fsn.id
```

The very last thing we have to do is to write the final query - one that puts everything together. It's as simple as selecting from `films` table and joining it with both CTE.

Bonus example. Find Fibonacci sequence with numbers below 100.

I'll just post the solution here and leave it for a curious reader to analyze ;)

```
WITH RECURSIVE fibo(a, b) AS (
  VALUES (0,1)

  UNION ALL

  SELECT b, a + b
  FROM fibo
  WHERE a + b < 100
)

SELECT b FROM fibo
```

CTE - inconspicuous but powerful

CTE are an interesting SQL feature. They help to organize and simplify complicated queries and also make them easier to maintain by allowing a user to get rid of duplicated parts. In their simplest form however they don't offer anything more, especially nothing in terms of manipulating data.

You may therefore think that they're not very useful. But their true potential lies in the recursive mode. It enables you to do a thing otherwise impossible in pure SQL - write a query that invokes itself, which gives you a lot of new possibilities. For example, you can traverse your relational tables like they were graphs. Recursive CTE might seem hard at first glance, but once you get familiar with them, you will appreciate the power they give.

Share this post