Advanced SQL - Common Table Expressions

mjk.space/advanced-sql-cte

January 26, 2018

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.

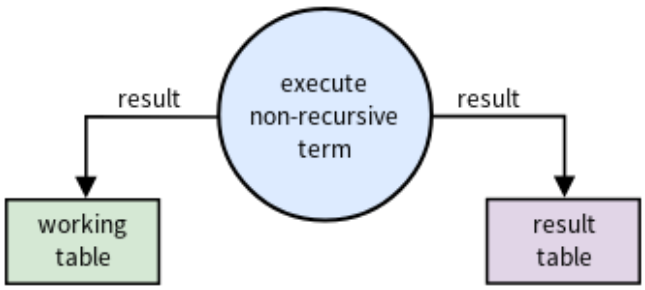
Let’s see how recursive CTE work exactly. Under the hood Postgres uses two temporary tables: working table and result table. The latter is the place that accumulates the final result of a CTE. Technically the whole process is actually iterative, not recursive, but that’s how this operation has been called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

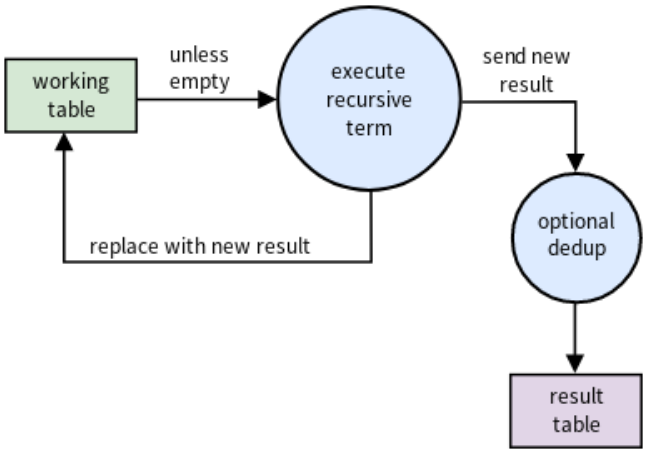
1. Initial step

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.

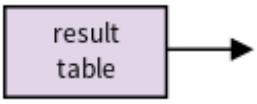




3. Final step

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.