127.87
Rating

Postgres Professional

Разработчик СУБД Postgres Pro

erogov
Feb 27 2020 at 18:23

On recursive queries

25 min 9.9K

Postgres Professional corporate blog, PostgreSQL*, SQL*

Translation

Original author: Egor Rogov

This article deals with writing recursive queries. This topic was brought up routinely, but the discussion was usually limited to simple cases related to trees: to descend from a vertex to the leaves and to ascend from a vertex to the root. We will address a more complicated case of an arbitrary graph.

Let's start with recalling the theory (very briefly since all of it is trivial), and then we will discuss what to do if it is unclear how to approach a real-life problem or if it seems to be clear, but the query persistently fails to work fine.

For an exercise, we will use the airlines demo database and try to write a query to find the shortest route from one airport to another.

A little theory

The good old relational SQL adequately does the job with unordered sets: there is a whole panoply of capabilities for them both in the language and «under the hood» of the DBMS. What SQL dislikes is when one jerks it in a loop row by row rather than have the task performed in one statement. The optimizer gives up and withdraws, leaving you alone face to face with the performance.

A recursive query is just a technique to implement a loop right in SQL. It's not that it is pretty often necessary, but it sometimes happens to be. And here complications arise since a recursive query is not very similar to a usual query and even less to a usual loop.

A general format of a recursive query is as follows:

```
WITH RECURSIVE t AS (
  non-recursive term      (1)
  UNION ALL
  recursive term          (2)
)
SELECT * FROM t;          (3)
```

The non-recursive term (1) is performed first. Then the recursive term (2) is iterated while it returns some rows. The recursive term is called so because it can access the output of the previous iteration that is available under the name of `t`.

In the process, the output of each iteration is placed into a resulting table, which will be available under the same name of `t` when all the query is complete (3). If we replace UNION ALL with UNION, duplicate rows will be eliminated at each iteration.

Using pseudocode, we may represent this as follows:

```
res ← EMPTY;
t ← (non-recursive term);
WHILE t IS NOT EMPTY LOOP
    res ← res UNION ALL t;
    aux ← (recursive term);
    t ← aux;
END LOOP;
t ← res;
```

Here is a simple example:

```
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 rows)
```

And this is enough to recall the basics. If something is unclear as early as at this point, it's time to read the documentation.

Practice

Armed with the theory we can now (in theory) write the above query: search of the shortest route from, say, Ust-Kut (UKX) to Nerungri (CNN).

From the entire demo database, we will need two tables: `airports` and `routes`. Formally, `routes` is a view, but we do not need to think about it. (If you are unaware of the structure of the demo database yet, review its description.)

The query may look somewhat like this:

```
 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;
```

This is, certainly, great, but how can we arrive at it, when there is nothing on the screen but a blinking cursor?

> «At this point in the videotape he always wonders if he's inadvertently set his beer down on the fast-forward button, or something, because the dancers go straight from their vicious Randy parody into something that obviously qualifies as advanced dancing. Randy knows that the steps they are doing are nominally the same as the basic steps demonstrated earlier, but he's damned if he can tell which is which, once they go into their creative mode. There is no recognizable transition, and that is what pisses Randy off, and has always pissed him off, about dancing lessons. Any moron can learn to trudge through the basic steps. That takes all of half an hour. But when that half-hour is over, dancing instructors always expect you'll take flight and go through one of those miraculous time lapse transitions that happen only in Broadway musicals and begin dancing brilliantly. Randy supposes that people who are lousy at math feel the same way: the instructor writes a few simple equations on the board, and ten minutes later he's deriving the speed of light in a vacuum.»
>
> — *Neal Stephenson. «Cryptonomicon»*

For instance, I cannot make up such a query off the cuff. Therefore, let's move gradually.

So, we need to find a route. A route from point A to B is a sequence of flights. The first flight leaves from A to somewhere, from this somewhere to somewhere else and so on, until the last flight ends in the sought point B.

How can we represent such a chain? From the relational standpoint, it is logical to make it a table with the *ordinal number* and *airport* columns. But we need to work with the chain as a single object, so the most reasonable option is to represent it as an array: [*airport*, *airport*, ...]. (If this is complicated for you, read about arrays and functions to manipulate them.)

It is clear what to start the iteration with: with the airport at Ust-Kut.

```
demo=# SELECT ARRAY[airport_code] FROM airports WHERE airport_code = 'UKX';

 array
-------
 {UKX}
(1 row)
```

Why not just ARRAY['UKX']? It makes sense to secure ourselves a little: if we make a typo in the airport code, the query will return nothing.

Now imagine that the result of this initial iteration is stored in a table and we need to do the second iteration. And we can really do so: create and fill the table and write queries with it. But it is easier to make use of WITH:

```
 demo=# WITH p(last_arrival, hops) AS (
  SELECT airport_code,
         ARRAY[airport_code]
  FROM   airports
  WHERE  airport_code = 'UKX'
)
SELECT * FROM p;

 last_arrival | hops
--------------+-------
 UKX          | {UKX}
(1 row)
```

We called the column *hops* to avoid confusion. Besides, we added one more (*last_arrival*) for the last point in our future chain. We could compute the last element of the array (p.hops[cardinality(p.hops)]) instead, but this is not so demonstrable.

Now there is the second iteration:

```
 demo=# WITH p(last_arrival, hops) AS (
  SELECT airport_code,
         ARRAY[airport_code]
  FROM   airports
  WHERE  airport_code = 'UKX'
)
SELECT r.arrival_airport AS last_arrival,
       p.hops || ARRAY[r.arrival_airport] AS hops
FROM   routes r, p
WHERE  r.departure_airport = p.last_arrival;

 last_arrival |   hops
--------------+-----------
 KJA          | {UKX,KJA}
(1 row)
```

What did we do? We got the first iteration (the «p» table) and joined it with routes. We specified a departure airport as the last airport in the chain, and we appended the arrival airport to the chain. It appears that from Ust-Kut only flights to Krasnoyarsk are available.

Now it is more or less clear how to put this together into a recursive query. We add the magic word RECURSIVE and combine the query with the first iteration using UNION ALL. And in the main query, we select the chain that eventually led to the destination airport (CNN).

Is it something like this?

```
 demo=# WITH RECURSIVE p(last_arrival, hops) AS (
  SELECT airport_code,
         ARRAY[airport_code]
  FROM   airports
  WHERE  airport_code = 'UKX'
  UNION ALL
  SELECT r.arrival_airport,
         p.hops || r.arrival_airport
  FROM   routes r, p
  WHERE  r.departure_airport = p.last_arrival
)
SELECT *
FROM   p
WHERE  p.last_arrival = (
         SELECT airport_code FROM airports WHERE airport_code = 'CNN'
);
```

```
ERROR:  recursive query "p" column 2 has type character(3)[] in non-
recursive term but type bpchar[] overall
LINE 3:          ARRAY[airport_code]
                 ^
HINT:  Cast the output of the non-recursive term to the correct type.
```

Hmm. Postgres complains that the second column has the `character(3)[]` type in the non-recursive term, while the resulting type is `bpchar[]`. The `bpchar` (blank-padded char) type is the internal name of the `char` type. Unfortunately, the concatenation of arrays does not retain element types, and therefore, it is required to explicitly cast the types.

```
 demo=# WITH RECURSIVE p(last_arrival, hops) AS (
  SELECT airport_code,
         ARRAY[airport_code]
  FROM   airports
  WHERE  airport_code = 'UKX'
  UNION ALL
  SELECT r.arrival_airport,
         (p.hops || r.arrival_airport)::char(3)[]
  FROM   routes r, p
  WHERE  r.departure_airport = p.last_arrival
)
```

```
SELECT *
FROM   p
WHERE  p.last_arrival = (
         SELECT airport_code FROM airports WHERE airport_code = 'CNN'
);
```

There is no error anymore, but alas — the query has hung. So what shall we do now?

Let's investigate. Let's try to run the query step by step and see what happens at each iteration.

It's clear that we can repeat the trick of pushing the first iteration into the table and gradually combining the query with new iterations, but this is unduly annoying, and error-prone. But there is a better way.

Let's augment our query with one more column for the number of the iteration (the column to be called `level`). It will equal one for the first iteration, and then we will increase it. It won't help in itself, but we can now stop execution of the query wherever we like. We've seen the first two iterations, let's look at the third one now:

```
 demo=# WITH RECURSIVE p(last_arrival, hops, level) AS (
  SELECT airport_code,
         ARRAY[airport_code],
         1
  FROM   airports
  WHERE airport_code = 'UKX'
  UNION ALL
  SELECT r.arrival_airport,
         (p.hops || r.arrival_airport)::char(3)[],
         p.level + 1
  FROM   routes r, p
  WHERE  r.departure_airport = p.last_arrival
  AND    p.level < 3
)
SELECT * FROM p;
```

```
 last_arrival |     hops      | level
--------------+---------------+-------
 UKX          | {UKX}         |     1
 KJA          | {UKX,KJA}     |     2
 UKX          | {UKX,KJA,UKX} |     3
 OVB          | {UKX,KJA,OVB} |     3
 OVB          | {UKX,KJA,OVB} |     3
 NOZ          | {UKX,KJA,NOZ} |     3
 NOZ           | {UKX,KJA,NOZ} |     3
 AER           | {UKX,KJA,AER} |     3
 SVO           | {UKX,KJA,SVO} |     3
 NUX           | {UKX,KJA,NUX} |     3
 UIK           | {UKX,KJA,UIK} |     3
 UIK           | {UKX,KJA,UIK} |     3
 BAX           | {UKX,KJA,BAX} |     3
 KRO           | {UKX,KJA,KRO} |     3
 OVS           | {UKX,KJA,OVS} |     3
(15 rows)
```

And we face something unexpected.

First, some rows are duplicated (for example: {UKX,KJA,OVB}). This is actually correct since there are two different flights from Krasnoyarsk (KJA) to Novosibirsk (OVB):

```
 demo=# SELECT flight_no
FROM   routes
WHERE  departure_airport = 'KJA'
AND    arrival_airport = 'OVB';

 flight_no
-----------
 PG0206
 PG0207
(2 rows)
```

Let's add the flight number to the query in order to distinguish the rows; we will need it anyway.

```
 demo=# WITH RECURSIVE p(last_arrival, hops, flights, level) AS (
  SELECT airport_code,
         ARRAY[airport_code],
         ARRAY[]::char(6)[],
         1
  FROM   airports
  WHERE  airport_code = 'UKX'
  UNION ALL
  SELECT r.arrival_airport,
         (p.hops || r.arrival_airport)::char(3)[],
         (p.flights || r.flight_no)::char(6)[],
         p.level + 1
  FROM   routes r, p
  WHERE  r.departure_airport = p.last_arrival
  AND    p.level < 3
)
SELECT * FROM p;

 last_arrival |     hops      |     flights     | level
--------------+---------------+-----------------+-------
 UKX          | {UKX}         | {}              |     1
 KJA          | {UKX,KJA}     | {PG0022}        |     2
 UKX          | {UKX,KJA,UKX} | {PG0022,PG0021} |     3
 OVB          | {UKX,KJA,OVB} | {PG0022,PG0206} |     3
 OVB          | {UKX,KJA,OVB} | {PG0022,PG0207} |     3
 NOZ          | {UKX,KJA,NOZ} | {PG0022,PG0351} |     3
 NOZ          | {UKX,KJA,NOZ} | {PG0022,PG0352} |     3
 AER          | {UKX,KJA,AER} | {PG0022,PG0501} |     3
 SVO          | {UKX,KJA,SVO} | {PG0022,PG0548} |     3
 NUX          | {UKX,KJA,NUX} | {PG0022,PG0623} |     3
 UIK          | {UKX,KJA,UIK} | {PG0022,PG0625} |     3
 UIK          | {UKX,KJA,UIK} | {PG0022,PG0626} |     3
 BAX          | {UKX,KJA,BAX} | {PG0022,PG0653} |     3
 KRO          | {UKX,KJA,KRO} | {PG0022,PG0673} |     3
 OVS          | {UKX,KJA,OVS} | {PG0022,PG0689} |     3
(15 rows)
```

But another weirdness is worse. One of the rows shows that we can arrive back: {UKX,KJA,UKX}. And this means that we will infinitely fly in a circle, and the query will not terminate. It's this that explains hanging.

What shall we do with it? We need to add a condition that each airport can be visited not more than once (if this condition is not met, the route will not be optimal anyway).

```
 demo=# WITH RECURSIVE p(last_arrival, hops, flights, level) AS (
  SELECT airport_code,
         ARRAY[airport_code],
         ARRAY[]::char(6)[],
         1
  FROM   airports
  WHERE  airport_code = 'UKX'
  UNION ALL
  SELECT r.arrival_airport,
         (p.hops || r.arrival_airport)::char(3)[],
         (p.flights || r.flight_no)::char(6)[],
         p.level + 1
  FROM   routes r, p
  WHERE  r.departure_airport = p.last_arrival
  AND    NOT r.arrival_airport = ANY(p.hops)
  AND    p.level < 3
)
SELECT * FROM p;

 last_arrival |     hops      |     flights     | level
--------------+---------------+-----------------+-------
 UKX          | {UKX}         | {}              |     1
 KJA          | {UKX,KJA}     | {PG0022}        |     2
 OVB          | {UKX,KJA,OVB} | {PG0022,PG0206} |     3
 OVB          | {UKX,KJA,OVB} | {PG0022,PG0207} |     3
 NOZ          | {UKX,KJA,NOZ} | {PG0022,PG0351} |     3
 NOZ          | {UKX,KJA,NOZ} | {PG0022,PG0352} |     3
 AER          | {UKX,KJA,AER} | {PG0022,PG0501} |     3
 SVO          | {UKX,KJA,SVO} | {PG0022,PG0548} |     3
 NUX          | {UKX,KJA,NUX} | {PG0022,PG0623} |     3
 UIK          | {UKX,KJA,UIK} | {PG0022,PG0625} |     3
 UIK          | {UKX,KJA,UIK} | {PG0022,PG0626} |     3
 BAX          | {UKX,KJA,BAX} | {PG0022,PG0653} |     3
 KRO          | {UKX,KJA,KRO} | {PG0022,PG0673} |     3
 OVS          | {UKX,KJA,OVS} | {PG0022,PG0689} |     3
(14 rows)
```

Now it seems to be OK. Ready to run the query without limitations?

```
 demo=# WITH RECURSIVE p(last_arrival, hops, flights, level) AS (
  SELECT airport_code,
         ARRAY[airport_code],
         ARRAY[]::char(6)[],
         1
  FROM   airports
  WHERE  airport_code = 'UKX'
  UNION ALL
  SELECT r.arrival_airport,
         (p.hops || r.arrival_airport)::char(3)[],
         (p.flights || r.flight_no)::char(6)[],
         p.level + 1
  FROM   routes r, p
  WHERE  r.departure_airport = p.last_arrival
  AND    NOT r.arrival_airport = ANY(p.hops)
--  AND    p.level < 3
)
```

```
SELECT *
FROM   p
WHERE  p.last_arrival = (
         SELECT airport_code FROM airports WHERE airport_code = 'CNN'
);
```

Formally everything must be OK… but the query hangs again, and if we arm ourselves with patience, the query can fail with the error «out of space for temporary files».

Why does this happen? Because we have to create all possible routes of any length starting in A, and only at the very end we select those of them that end in B. This is, to say the least, not the most efficient algorithm. To understand «the extent of the disaster», we can start changing the limitations on `level` to see how many rows are returned in each step:

```
1   1
2   2
3   14
4   165
5   1978
6   22322
7   249942
8   2316063
```

And so on, and each next query is considerably slower than the previous one.
Let's think of the number that we expect to get in the answer to the problem. If there were only big cities, it's most likely 2 (with a connection in Moscow). In our situation, it makes sense to anticipate at least a couple more for regional flights to reach a big city. This makes around 4 or 5, or maybe 6. However, the query is not going to stop even at eight: it will reach (if it is strong and healthy enough) something around one hundred until it is unable to extend any chain!

Note that the algorithm is «width-first»: we add *all* routes having length 1, then *all* routes having length 2 and so on. That is, once we find just *any* route from A to B, it will be the shortest one (according to the number of connections). Now the only problem left is how to duly stop the exhaustive search.

The idea is that in each step we set the «route found» indicator if at least one of the just created routes ends in the destination. Then we will be able to write the stopping criterion.

Let's start with adding the destination to the query itself (earlier, it appeared only at the very end, when the results returned were filtered). Let's compute it at the very beginning and just leave it unchanged in the recursive term:

```
 demo=# WITH RECURSIVE p(last_arrival, destination, hops, flights, level)
AS (
  SELECT a_from.airport_code,
         a_to.airport_code,
         ARRAY[a_from.airport_code],
         ARRAY[]::char(6)[],
         1
  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)[],
         p.level + 1
  FROM   routes r, p
  WHERE  r.departure_airport = p.hops[cardinality(p.hops)]
  AND    NOT r.arrival_airport = ANY(p.hops)
  AND    p.level < 3
)
SELECT * FROM p;

 last_arrival | destination |     hops      |     flights     | level
--------------+-------------+---------------+-----------------+-------
 UKX          | CNN         | {UKX}         | {}              |     1
 KJA          | CNN         | {UKX,KJA}     | {PG0022}        |     2
 OVB          | CNN         | {UKX,KJA,OVB} | {PG0022,PG0206} |     3
 OVB          | CNN         | {UKX,KJA,OVB} | {PG0022,PG0207} |     3
 NOZ          | CNN         | {UKX,KJA,NOZ} | {PG0022,PG0351} |     3
 NOZ          | CNN         | {UKX,KJA,NOZ} | {PG0022,PG0352} |     3
 AER          | CNN         | {UKX,KJA,AER} | {PG0022,PG0501} |     3
 SVO          | CNN         | {UKX,KJA,SVO} | {PG0022,PG0548} |     3
 NUX          | CNN         | {UKX,KJA,NUX} | {PG0022,PG0623} |     3
 UIK          | CNN         | {UKX,KJA,UIK} | {PG0022,PG0625} |     3
 UIK          | CNN         | {UKX,KJA,UIK} | {PG0022,PG0626} |     3
 BAX          | CNN         | {UKX,KJA,BAX} | {PG0022,PG0653} |     3
 KRO          | CNN         | {UKX,KJA,KRO} | {PG0022,PG0673} |     3
 OVS          | CNN         | {UKX,KJA,OVS} | {PG0022,PG0689} |     3
(14 rows)
```

Now it is easy to compute the indicator of the found route: it must be set if the last point of the route is the same as the destination at least for one row. To this end, the window function bool_or will be useful (if window functions are something new to you, start with the introduction, which also contains references to a more detailed description).

```
demo=# WITH RECURSIVE p(last_arrival, destination, hops, flights, found,
level) 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,
         1
  FROM   airports a_from, airports a_to
  WHERE  a_from.airport_code = 'UKX'
  AND    a_to.airport_code = 'OVB' -- 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 (),
         p.level + 1
  FROM   routes r, p
  WHERE  r.departure_airport = p.last_arrival
  AND    NOT r.arrival_airport = ANY(p.hops)
  AND    p.level < 3
)
SELECT * FROM p;

 last_arrival | destination |     hops      |     flights     | found |
level
--------------+-------------+---------------+-----------------+-------+
-------
 UKX          | OVB         | {UKX}         | {}              | f     |
1
 KJA          | OVB         | {UKX,KJA}     | {PG0022}        | f     |
2
 OVB          | OVB         | {UKX,KJA,OVB} | {PG0022,PG0206} | t     |
3
 OVB          | OVB         | {UKX,KJA,OVB} | {PG0022,PG0207} | t     |
3
 NOZ          | OVB         | {UKX,KJA,NOZ} | {PG0022,PG0351} | t     |
3
 NOZ          | OVB         | {UKX,KJA,NOZ} | {PG0022,PG0352} | t     |
3
 AER           | OVB          | {UKX,KJA,AER} | {PG0022,PG0501} | t     |
3
 SVO           | OVB          | {UKX,KJA,SVO} | {PG0022,PG0548} | t     |
3
 NUX           | OVB          | {UKX,KJA,NUX} | {PG0022,PG0623} | t     |
3
 UIK           | OVB          | {UKX,KJA,UIK} | {PG0022,PG0625} | t     |
3
 UIK           | OVB          | {UKX,KJA,UIK} | {PG0022,PG0626} | t     |
3
 BAX           | OVB          | {UKX,KJA,BAX} | {PG0022,PG0653} | t     |
3
 KRO           | OVB          | {UKX,KJA,KRO} | {PG0022,PG0673} | t     |
3
 OVS           | OVB          | {UKX,KJA,OVS} | {PG0022,PG0689} | t     |
3
(14 rows)
```

Here we checked how the query will work for the route from Ust-Kut (UKX) to Novosibirsk (OVB), which, as we already know, has length 3. (By the way, to this end, we had to change CNN to OVB only once — every little bit helps.) This does work!

We also compute the indicator in the non-recursive term of the query. We could just write `false`, but this way the query will be more universal and will correctly compute the number of connections from A back to A.

Now we only need to add the stopping criterion, and we can run the query:

```
 demo=# WITH RECURSIVE p(last_arrival, destination, hops, flights, found,
level) 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,
         1
  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 (),
         p.level + 1
  FROM   routes r, p
  WHERE  r.departure_airport = p.last_arrival
  AND    NOT r.arrival_airport = ANY(p.hops)
  AND    NOT p.found
--  AND    p.level < 3
)
SELECT hops, flights
FROM   p
WHERE  p.last_arrival = p.destination;

         hops          |             flights
-----------------------+-------------------------------
 {UKX,KJA,OVB,MJZ,CNN} | {PG0022,PG0206,PG0390,PG0035}
 {UKX,KJA,OVB,MJZ,CNN} | {PG0022,PG0207,PG0390,PG0035}
 {UKX,KJA,SVO,MJZ,CNN} | {PG0022,PG0548,PG0120,PG0035}
 {UKX,KJA,OVB,MJZ,CNN} | {PG0022,PG0206,PG0390,PG0036}
 {UKX,KJA,OVB,MJZ,CNN} | {PG0022,PG0207,PG0390,PG0036}
 {UKX,KJA,SVO,MJZ,CNN} | {PG0022,PG0548,PG0120,PG0036}
 {UKX,KJA,OVS,LED,CNN} | {PG0022,PG0689,PG0686,PG0245}
 {UKX,KJA,SVO,LED,CNN} | {PG0022,PG0548,PG0472,PG0245}
 {UKX,KJA,SVO,LED,CNN} | {PG0022,PG0548,PG0471,PG0245}
 {UKX,KJA,SVO,LED,CNN} | {PG0022,PG0548,PG0470,PG0245}
 {UKX,KJA,SVO,LED,CNN} | {PG0022,PG0548,PG0469,PG0245}
 {UKX,KJA,SVO,LED,CNN} | {PG0022,PG0548,PG0468,PG0245}
 {UKX,KJA,OVB,PEE,CNN} | {PG0022,PG0206,PG0186,PG0394}
 {UKX,KJA,OVB,PEE,CNN} | {PG0022,PG0207,PG0186,PG0394}
 {UKX,KJA,BAX,ASF,CNN} | {PG0022,PG0653,PG0595,PG0427}
 {UKX,KJA,SVO,ASF,CNN} | {PG0022,PG0548,PG0128,PG0427}
 {UKX,KJA,OVS,DME,CNN} | {PG0022,PG0689,PG0544,PG0709}
 {UKX,KJA,OVS,DME,CNN} | {PG0022,PG0689,PG0543,PG0709}
 {UKX,KJA,KRO,DME,CNN} | {PG0022,PG0673,PG0371,PG0709}
 {UKX,KJA,OVB,DME,CNN} | {PG0022,PG0206,PG0223,PG0709}
 {UKX,KJA,OVB,DME,CNN} | {PG0022,PG0207,PG0223,PG0709}
 {UKX,KJA,NUX,DME,CNN} | {PG0022,PG0623,PG0165,PG0709}
 {UKX,KJA,BAX,DME,CNN} | {PG0022,PG0653,PG0117,PG0709}
(23 rows)
```

That's actually all. We arrived at the query shown at the beginning of the article, and it runs instantaneously. We can now remove the «debugging» `level` … or we can leave it.

Let's summarize useful techniques:

- It often helps to represent a «route» as an array. We can concatenate rows instead, but it is not so convenient.
- Preventing infinite loops (also by means of the array).
- Debugging by limiting the number of iterations.
- Sometimes a way to duly stop is needed for better performance.
- Window functions enable working miracles.

For exercises

To reinforce the skill, you can do a few variations on the theme:

1. What is the maximum number of connections needed to fly from any airport to any other?
   Hint 1: the first term of the query must contain all pairs of departure and arrival airports.
   Hint 2: you need to compute the indicator of the found route separately for each pair of airports.

2. Find the route (from UKX to CNN) with the minimum total duration of the flights (not counting the waits for connection flights).
   Hint 1: this route may appear not to have the optimal number of connections.
   Hint 2: you need to think up an indicator to signal that further exhaustive search is useless.

3. Find the route (from UKX to CNN) with the minimum total duration of the flights, *including* the waits for connection flights. Assume that we are ready for the first departure at the moment of `bookings.now() - interval '20 days'` .

When you master the third task, share your success with us (edu@postgrespro.ru)!

Tags: postgresql, postgres, recursive, query, sql, window functions

Hubs: Postgres Professional corporate blog, PostgreSQL, SQL