Postgresql 8.4 Recursive Queries

Posted by Daniel Lyons on August 11, 2009

Apparently I missed the news: PostgreSQL 8.4 came out on July 1st! This release brings two new features and greatly improves one old feature. This post will be about recursive queries using `WITH`.

A classic problem in SQL is how to model trees. In web development you often have to store nested subcategories or model pages as files inside folders. The simplest proper way to model this relationship in SQL is to add a self-referencing column to the table to point to a row's parent row. For example, you might represent the file path `/foo/bar/baz.html` with the following table:

id	name	parent_id
1	/	*NULL*
2	foo	1
3	bar	2
4	baz.html	3

`fs` table

This structure has a number of pleasant features from a relational perspective. For one thing, you can rename a folder by modifying just one row. You can insert a new node anywhere in this hierarchy by just knowing the parent folder's ID. You can also move a node anywhere in the hierarchy by modifying that one ID.

Unfortunately, from an implementation perspective, this structure sucks for doing many common operations: computing a string version of a path, for example, or listing all the descendants under a given node. There are many simpler, less normalized methods that are better at these common operations at the expense of some relational normality. (One I have used is to have outside software maintain a text path in the database.)

With recursive queries it's now possible to recreate the string path, find all descendant nodes, and get all of the parent nodes up to the root. I'm going to show you how to do each of these things with just one query apiece.

All of the magic comes down to the `WITH` statement. This statement is provided both to support recursive queries and also for common table expressions or CTEs. If you find yourself repeating the same table definition in your query you can pull it out and put it above the statement in a `WITH` clause. You can think of this as a way to make a temporary view just for a single SQL statement.

The basic structure you're looking at is this:

```
WITH temp_view(col1, ...) AS (SELECT ...)
SELECT ...;
```

If you use `temp_view` inside the `SELECT` within the `WITH` statement, you have to add `RECURSIVE`:

```
WITH RECURSIVE temp_view(col1, ...) AS (SELECT ...)
SELECT ...;
```

All three of the queries I'm going to show you rely on the recursive flavor in order to do their work. There is a design pattern in the inner `SELECT` whenever a recursive query is written, which is related

to the notion of recursion. You must define a *base case* and an *inductive case*. The base case sets up the initial conditions for the recursion, usually the first set of rows for the result. The inductive case uses the rows which have already been computed to compute more rows and decide whether or not the recursion is complete.

For example, to find the breadcrumbs for a given path, the base case is the initial path. The inductive case is just to find the parent path. If there's no parent path, the recursion is complete. Inside the SQL, you'll do a naïve `SELECT` and `UNION` it with a `SELECT` that uses the recursively defined name. You can read more about this in the (PostgreSQL Documentation) [http://www.postgresql.org/docs/8.4/interactive/queries-with.html]. So the SQL will look like this:

```
WITH RECURSIVE breadcrumb(id, name, parent_id) AS (
  SELECT id, name, parent_id FROM fs WHERE id = 4
  UNION
    SELECT fs.id, fs.name, fs.parent_id FROM fs, breadcrumb
    WHERE breadcrumb.parent_id = fs.id)
SELECT * FROM breadcrumb;
```

The base case gets all of the information for the row we're interested in, which is the row with ID 4. The inductive case takes the previously generate row and gets the row from `fs` with the ID of the parent ID of the row with ID 4, which in this case is the row with ID 3. The next iteration finds the row from `fs` with the ID of the parent ID of the row with ID 3, and so on.

If you're following along at home, run this SQL to set up our initial conditions:

```
CREATE TABLE fs (
  id SERIAL PRIMARY KEY,
  name VARCHAR,
  parent_id INTEGER REFERENCES fs(id)
);

INSERT INTO fs (name, parent_id)
VALUES ('/', NULL), ('foo', 1), ('bar', 2), ('baz.html', 3);
```

Now you should get this output from the above SQL:

id	name	parent_id
4	baz.html	3
3	bar	2
2	foo	1
1	/	*NULL*

Let's work on the other two problems: getting a string path and listing descendants of a given node. We'll use the root node for the sake of fun and add a few extra nodes so that the tree is fuller and tests our query a little better:

```
INSERT INTO fs (name, parent_id)
VALUES ('under foo', 2), ('under bar', 3), ('under bar 2', 3),
       ('baz', 2), ('bazzle.html', 8);
```

Now, to do this query the base case would seem to be the root of the tree. In thinking about what kind of result we want, I think for convenience we might want the current node's name and the full path of

this node in addition to the regular columns. We're going to assume that the root of the tree is the node with ID 1, which gives us this base case:

```
WITH RECURSIVE path(name, path, parent, id, parent_id) AS (
  SELECT name, '/', NULL, id, parent_id FROM fs WHERE id = 1
```

Now the inductive case just takes that case and computes the descendant nodes of that node by looking at their parent_id:

```
  SELECT
    fs.name,
    parentpath.path ||
      CASE parentpath.path
        WHEN '/' THEN ''
        ELSE '/'
      END || fs.name,
    parentpath.path, fs.id, fs.parent_id
  FROM fs, path as parentpath
  WHERE fs.parent_id = parentpath.id
```

Calculating the path, the `CASE` statement you see there is a hack to prevent extra '/'s at the start of the path because of the UNIX convention of paths starting at /. Otherwise you see I'm just taking the name of this node and concatenating the parent path with my name, separated by /'s. So this is the full version:

```
WITH RECURSIVE path(name, path, parent, id, parent_id) AS (
  SELECT name, '/', NULL, id, parent_id FROM fs WHERE id = 1
  UNION
  SELECT
    fs.name,
    parentpath.path ||
      CASE parentpath.path
        WHEN '/' THEN ''
        ELSE '/'
      END || fs.name,
    parentpath.path, fs.id, fs.parent_id
  FROM fs, path as parentpath
  WHERE fs.parent_id = parentpath.id)
SELECT * FROM path;
```

Let's see what it does!

name	path	parent	id	parent_id
/	/		1	
foo	/foo	/	2	1
bar	/foo/bar	/foo	3	2
under foo	/foo/under foo	/foo	5	2
baz	/foo/baz	/foo	8	2
baz.html	/foo/bar/baz.html	/foo/bar	4	3
under bar	/foo/bar/under bar	/foo/bar	6	3
under bar 2	/foo/bar/under bar 2	/foo/bar	7	3

name	path	parent	id	parent_id
bazzle.html	/foo/baz/bazzle.html	/foo/baz	9	8

Looks like it worked! But now that we have this statement it seems like it would be handy to make it a function so you can find the descendants of any particular node. The path might not look right but at least you could get relative children. Let's try it:

```
        CREATE FUNCTION children_of(root_id INTEGER)
        RETURNS TABLE (name VARCHAR, path VARCHAR, parent VARCHAR, id INTEGER,
parent_id INTEGER)
        AS $$
        WITH RECURSIVE path(name, path, parent, id, parent_id) AS (
          SELECT name, '/', NULL, id, parent_id FROM fs WHERE id = $1
          UNION
          SELECT
            fs.name,
            parentpath.path ||
              CASE parentpath.path
                WHEN '/' THEN ''
                ELSE '/'
              END || fs.name,
            parentpath.path, fs.id, fs.parent_id
          FROM fs, path as parentpath
          WHERE fs.parent_id = parentpath.id)
        SELECT * FROM path;
        $$ LANGUAGE 'sql';
```

All I've done here is wrap the previous statement with some stuff to make it into a function and replaced 1 with $1, which makes the value the first parameter of the function call. For extra convenience, if I were using this as the foundation of a CMS or something where I would expect to need the path frequently, I might also make a view:

```
        CREATE VIEW paths AS SELECT * FROM children_of(1);
```

It's worth noting that recursive queries probably do not have excellent performance characteristics (I'm not certain but it seems likely.) If you do run into performance problems with this, you might consider looking into (materialized views) [http://tech.jonathangardner.net/wiki/PostgreSQL/Materialized_Views] which can be implemented manually in PostgreSQL using triggers.

If you prefer, you could instead create a function to find the path of a given ID:

```
        CREATE FUNCTION pathname(id INTEGER) RETURNS VARCHAR AS $$
        SELECT '/' || path FROM
        (WITH RECURSIVE pathto(path, id) AS (
          SELECT name, parent_id FROM fs WHERE id = $1
          UNION
          SELECT fs.name || '/' || pathto.path, parent_id
          FROM fs, pathto WHERE fs.id = pathto.id)
        SELECT * FROM pathto) AS pathto(path, id)
        WHERE id = 1;
        $$ LANGUAGE 'sql';
```

I was then tempted to make an index on this function, but PostgreSQL won't allow computed indices

using functions not marked `IMMUTABLE`, which is a way of certifying to PostgreSQL that it is a pure function. Because this function will return different values depending on the content of a table, it isn't immutable, and the index could get out-of-sync with reality without PostgreSQL knowing it. Intuitively, if you rename a node, all of its children will then have different paths reflecting the new name but it's unclear how you could make PostgreSQL aware of that fact.