# PostgreSQL (advanced)

- http://bytes.com/topic/postgresql/answers/66221-recursing-tree-data-pl-pgsql
- http://stackoverflow.com/questions/3440840/postgresql-sql-or-pl-pgsql-query-for-traversing-a-directed-graph-and-returning-al
- http://techportal.ibuildings.com/2009/09/07/graphs-in-the-database-sql-meets-social-networks/
- http://www.storytotell.org/blog/2009/08/11/postgresql84-recursive-queries.html
- http://www.postgresql.org/docs/8.4/static/queries-with.html
- http://stackoverflow.com/questions/53108/is-it-possible-to-make-a-recursive-sql-query
- http://www.depesz.com/index.php/2008/04/11/my-take-on-trees-in-sql/
- http://wiki.postgresql.org/wiki/Getting_list_of_all_children_from_adjacency_tree
- http://www.scribd.com/doc/14683319/Trees-and-More-with-PostgreSQL

## Graphs

```
-- tabulka
CREATE TABLE "graph" (
  "id" serial NOT NULL,
  "link" BIGINT NOT NULL,
  "data" text NULL
)

-- index
-- CREATE INDEX "graph_4d043073083b7" ON "graph" ("id") -- jeste prozkoumat

-- insert (jednoduchy cyklicky graf)
INSERT INTO "graph" ("id", "link", "data") VALUES
('1', '2', 'test a'),
('2', '4', 'test b'),
('3', '1', 'test c'),
('4', '3', 'test d'),
('7', '1', 'oh hai'),
('1', '7', NULL)
```

When working with recursive queries it is important to be sure that the recursive part of the query will eventually return no tuples, or else the query will loop indefinitely. Sometimes, using UNION instead of UNION ALL can accomplish this by discarding rows that duplicate previous output rows. However, often a cycle does not involve output rows that are completely duplicate: it may be necessary to check just one or a few fields to see if the same point has been reached before. The standard method for handling such situations is to compute an array of the already-visited values. For example, consider the following query that searches a table graph using a link field:

```
WITH RECURSIVE search_graph(id, link, DATA, depth) AS (
        SELECT g.id, g.link, g.data, 1
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1
        FROM graph g, search_graph sg
```

```
        WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

This query will loop if the link relationships contain cycles. Because we require a "depth" output, just changing UNION ALL to UNION would not eliminate the looping. Instead we need to recognize whether we have reached the same row again while following a particular path of links. We add two columns path and cycle to the loop-prone query:

```
WITH RECURSIVE search_graph(id, link, DATA, depth, path, cycle) AS (
        SELECT g.id, g.link, g.data, 1, ARRAY[g.id], FALSE
        FROM graph g
    UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1, path || g.id, g.id =
ANY(path)
        FROM graph g, search_graph sg
        WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

```
-- Tady je rekurzivni query lehce upravena pro potreby kyberky:

WITH RECURSIVE search_graph(id, link, DATA, depth, path, cycle) AS (
        SELECT g.id, g.link, g.data, 1, ARRAY[g.id], FALSE
        FROM graph g
        WHERE id = 1      -- tohle je ID nody ze ktery zaciname
    UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1, path || g.id, g.id =
ANY(path)
        FROM graph g, search_graph sg
        WHERE g.id = sg.link AND NOT cycle
)

SELECT * FROM search_graph
    WHERE cycle != TRUE     -- vynechame nodu, ktera zjistila ze cyklime
(zbytecne by tam byla znovu)
    AND depth <= 3          -- hloubku omezime na 3 hrany od startu
    LIMIT 100         -- celkovy pocet vybranych nod omezime na 100
```

## Tree

```
CREATE TABLE tree (node INTEGER, parent INTEGER);
INSERT INTO tree(node,parent) VALUES(1,0);
INSERT INTO tree(node,parent) VALUES(2,0);
INSERT INTO tree(node,parent) VALUES(3,1);
INSERT INTO tree(node,parent) VALUES(4,3);
INSERT INTO tree(node,parent) VALUES(5,4);
INSERT INTO tree(node,parent) VALUES(6,2);
INSERT INTO tree(node,parent) VALUES(7,6);
```

```
INSERT INTO tree(node,parent) VALUES(8,6);
INSERT INTO tree(node,parent) VALUES(9,2);

CREATE OR REPLACE FUNCTION getTree(INTEGER, INTEGER) RETURNS SETOF tree AS '
    DECLARE
        root ALIAS FOR $1;
        depth ALIAS FOR $2;
        tempRow1 tree%ROWTYPE;
        tempRow2 tree%ROWTYPE;
    BEGIN
        -- Using PostgreSQL 7.3.4.
        -- Docs: http://www.postgresql.org/docs/7.3/static/plpgsql.html
        -- See chapter 19, especially 19.6

    FOR tempRow1 IN SELECT node, parent FROM tree WHERE parent = root LOOP
        RETURN NEXT tempRow1;
        FOR tempRow2 IN SELECT node, parent FROM
            getTree(tempRow1.node, depth+1) LOOP
            RETURN NEXT tempRow2;
        END LOOP;
    END LOOP;
    RETURN;
    END;
' LANGUAGE 'plpgsql';

CREATE OR REPLACE FUNCTION getTree(INTEGER) RETURNS SETOF tree AS '
DECLARE
root ALIAS FOR $1;
tempRow tree%ROWTYPE;
BEGIN
FOR tempRow IN SELECT node, parent FROM getTree(root, 0) LOOP
RETURN NEXT tempRow;
END LOOP;
RETURN;
END;
' LANGUAGE 'plpgsql';

SELECT * FROM getTree(0);
node | parent
------+--------
1 | 0
3 | 1
4 | 3
5 | 4
2 | 0
6 | 2
7 | 6
8 | 6
9 | 2
(9 ROWS)
```

```
SELECT * FROM getTree(1);
node | parent
------+--------
3 | 1
4 | 3
5 | 4
(3 ROWS)

SELECT * FROM getTree(2);
node | parent
------+--------
6 | 2
7 | 6
8 | 6
9 | 2
(4 ROWS)
```

From:
https://wiki.spoje.net/ - **SPOJE.NET**

Permanent link:
**https://wiki.spoje.net/doku.php/howto/hosting/pgsql_advanced**

Last update: **2014/01/23 02:35**