Adjacency list vs. nested sets: PostgreSQL

This series of articles is inspired by numerous questions asked on the site and on **Stack Overflow**.

> What is better to store hierarchical data: **nested sets** model or **adjacency list** (parent-child) model?

First, let's explain what all this means.

Adjacency list

Hierarchical relations (not to be confused with hierarchical data model) are `0-1:0-N` transitive relations between entities of same domain.

For instance, ancestor-descendant relation is:

- Transitive:
  - If **A** is an ancestor of **B** and **B** is an ancestor of **C**, then **A** is an ancestor of **C**
- Irreflexive:
  - If **A** is an ancestor of **B**, then **B** is never an ancestor of **A**
- `0-1:0-N`
  - **A** can have zero, one or many children. **A** can have zero or one parents.

These relations can be represented by an ordered directed tree.

Tree is a simple directed graph (that with at most one directed edge between two different vertices) and relational model has means to represent simple graphs.

Two vertices are considered related (and therefore their primary keys forming a row in the table) if and only if they are connected with an edge.

This table along with the table defining the vertices identifies a graph completely by defining pairs of vertices connected by the edges. Each record in the table defines a pair of adjacent vertices, that's why this representation is called **adjacency list**.

Adjacency lists can represent any simple directed graphs, not ony hierarchy trees. But due to the fact that this structure is most commonly used to define the parent-child relationships, the terms parent-child model and adjacency list model have almost become synonymous. However, they are not: adjacency list model is much wider and parent-child model is one of its implementations.

Now, since we have a tree here which implies `0-1:0-N` relationship between the vertices, we can define the relation as a *self-relation*: the table defines both the entity and the relationship. Parent is just a one attribute among other attributes with a `FOREIGN KEY` reference to the table itself.

Since multple items can have no parents (and therefore be the roots of their trees), it's sometimes useful to convert this tree into an arborescence: make a single fake root that considered a parent of all entries that have no actual parent.

This is a nice and elegant model, but until recently it had one drawback: it could not be used with **SQL**.

SQL, as we all know, deals with relational tables which can be transformed by the means of relational algebra.

It provides a way to do number of operations including relational multiplication, projection, sum etc.

However, earlier versions or **SQL** lacked recursion which is required to do certain operations efficiently. Namely, recursion is required to operate upon the adjacency list structure.

The most common operations are:

- Find all descendants of a given node
- Find all ancestors of a given node
- Find all descendants of a given node up to a certain depth

The first two operations require recursion.

The third one does too if the depth level should serve as a parameter.

To work around this, the nested sets model was proposed by Michael Kamfonas and popularized by Joe Celko.

Nested sets

The idea of nested sets is quite simple and can be illustrated using one of the most popular ways to store hierarchical data, the **XML**.

How would we store the hierarchies in **XML**?

We would just use one tag to describe every item and nest it accordingly, like this:

view sourceprint?

```
01.<item id="0">
 02. <item id="1">
  03.  <item id="2"/>
  04.  <item id="3">
   05.   <item id="4"/>
  06.  </item>
  07.  <item id="5">
   08.   <item id="6">
    09.    <item id="7"/>
   10.   </item>
  11.  </item>
 12. </item>
 13. <item id="8"/>
14.</item>
```

This is fine, but how to use this structure in a relational table?

As you can see, opening and closing tags of each node are contained on their own lines here. The opening and closing tags may (or may not) be the same for the nodes containing no children (this is not important).

We see that the node ranges never intersect: given two node ranges, their intersection always makes the range of either of the nodes or an empty set.

In other words, a node open within another node should also close within it.

If and only if this holds for each and every node, the nodes make a valid **XML** file and a valid hierarchy.

Now, the nested set model can be described in one sentence:

To store a hierarchy in a nested set model, we just store the **line numbers** of the opening and closing tags of each node as if it were an **XML** file.

Each set of nodes having a common ancestor is nested within the node of this ancestor. That's why this model is called nested sets.

Historically, these line numbers are stored in columns named `lft` and `rgt` (since `LEFT` and `RIGHT` are reserved words in most **SQL** dialects).

That's how this hierarchy would look in a nested sets model:

ID	LFT	RGT
0	1	14
1	2	12
2	3	3
3	4	6
4	5	5
5	7	11
6	8	10
7	9	9
8	13	13

This model is more **SQL** friendly, since the tasks described above can be performed in **SQL** without using recursion.

To find out all ancestors of a given node, we just select all nodes that **contain** its `LFT` boundary (which in a properly built hierarchy implies containing the `RGT` boundary too):

view sourceprint?
```
1.SELECT hp.*
2.FROM  t_hierarchy hc
3.JOIN  t_hierarchy hp
4.ON   hc.lft BETWEEN hp.lft AND hp.rgt
5.WHERE hc.id = ?
```

And to find the descendants, we shoud just reverse the condition, i. e. find all nodes that **are contained** between the current node's boundaries:

view sourceprint?
```
1.SELECT hc.*
2.FROM  t_hierarchy hp
3.JOIN  t_hierarchy hc
4.ON   hc.lft BETWEEN hp.lft AND hp.rgt
5.WHERE hp.id = ?
```

Ironically, selecting all descendants up to a given depth (which is the least problem for the adjacency list as long as the depth is known in design time) is the hardest task for the nested set model. Even obtaning the list of immediate parents and immediate children is not so simple.

However, this is solvable. This is the query to get all descendants up to the third generation (that is node itself, all children and grandchildren):

view sourceprint?
```
01.SELECT hc.*
```

```
02.FROM  t_hierarchy hp
03.JOIN  t_hierarchy hc
04.ON   hc.lft BETWEEN hp.lft AND hp.rgt
05.WHERE  hp.id = ?
      06.         AND
      07.         (
      08.         SELECT COUNT(*)
      09.         FROM  t_hierarchy hn
      10.         WHERE  hc.lft BETWEEN hn.lft AND hn.rgt
            11.                   AND hn.lft BETWEEN hp.lft AND hp.rgt
      12.         ) <= 3
```

Unlike adjacency list model, the depth level can be parametrized in this query which makes it possible to use a single query for all depth level.

Nested set model can be relatively easily queried for, but it's extremely hard to manage.

To insert a new child into a node or make an existing mode a child in adjacency list, everything we need is provide the new value of its parent column. With a single update we can move a whole branch.

To add a new node into a nested set model we should do exactly the same as if we were adding a new node into an **XML** file: all subsequent nodes are moved several lines further. Since the boundaries in **SQL** table represent the line numbers, we should do the same: calculate the offset and make a batch update to all nodes to the right of the updated or inserted one. Very hard to implement and very inefficient.

Now good news.

Three of four major systems (that is **SQL Server**, **Oracle** and **PostgreSQL 8.4**) now support recursion natively.

The fourth one (**MySQL**) does not support it, but it can be emulated to the extent required to run queries against the hierarchical data modelled according to adjacency list model:

- **Hierarchical queries in MySQL**

In this series of articles, we will compare efficiency of the **adjacency list** model to that of the **nested sets** model.

PostgreSQL 8.4 is the system we begin with.

Analysis

PostgreSQL 8.4 supports recursive queries by means of such called **hierarchical CTE's**.

A hierarchical CTE is an analog of this query:

view sourceprint?

```
01.SELECT *
02.FROM  t_hierarchy h1
03.WHERE …
04.
05.UNION ALL
06.
07.SELECT *
08.FROM  t_hierarchy h1
```

```
09.JOIN  t_hierarchy h2
10.ON  …
11.WHERE …
12.
13.UNION ALL
14.
15.SELECT *
16.FROM  t_hierarchy h1
17.JOIN  t_hierarchy h2
18.ON  …
19.JOIN  t_hierarchy h3
20.ON  …
21.WHERE …
22.
23.…
```

with the theoretically unlimited number of `UNION ALL`'s built at runtime and the results of each query cached and called recursively.

To define such a construct, one uses `WITH RECURSIVE` clause.

To compare both methods, we will create a sample table which combines both data models. Each node will have both `parent` and the boundaries (`lft` and `rgt`) defined. Then we will run the three most important queries, which, again, are:

- Find all descendants of a given node
- Find all ancestors of a given node
- Find all descendants of a given node up to a certain depth

Here's the script to create a sample table:

Table creation details

view sourceprint?

```
01.CREATE TABLE t_hierarchy (
      02.         id INT NOT NULL,
      03.         parent INT NOT NULL,
      04.         lft INT NOT NULL,
      05.         rgt INT NOT NULL,
      06.         data VARCHAR(100) NOT NULL,
      07.         stuffing VARCHAR(100) NOT NULL
08.);
09.
10.INSERT
11.INTO  t_hierarchy
12.WITH RECURSIVE
      13.         ini AS
      14.         (
      15.         SELECT 8 AS level, 5 AS children
      16.         ),
      17.         range AS
```

```
        (
        SELECT level, children,
                (
                SELECT SUM(POW(children, n)::INTEGER *
                ((n < level)::INTEGER + 1))
                FROM  generate_series(level, 0, -1) n
                ) width
        FROM  ini
        ),
        q AS
        (
        SELECT s AS id, 0 AS parent, level, children,
                1 + width * (s - 1) AS lft,
                1 + width * s - 1 AS rgt,
                width / children AS width
        FROM  (
                SELECT r.*, generate_series(1,
                children) s
                FROM  range r
                ) q2
        UNION ALL
        SELECT id * children + position, id, level - 1,
        children,
                1 + lft + width * (position - 1),
                1 + lft + width * position - 1,
                width / children
        FROM  (
                SELECT generate_series(1, children) AS
                position, q.*
                FROM  q
                ) q2
        WHERE level > 0
        )
SELECT id, parent, lft, rgt, 'Value ' || id, RPAD('', 100, '*')
FROM  q;

ALTER TABLE t_hierarchy ADD CONSTRAINT pk_hierarchy_id PRIMARY KEY
(id);
CREATE INDEX ix_hierarchy_lft ON t_hierarchy (lft);
CREATE INDEX ix_hierarchy_rgt ON t_hierarchy (rgt);
CREATE INDEX ix_hierarchy_parent ON t_hierarchy (parent);

ANALYZE t_hierarchy;
```

The table contains **8** levels of hierarchy with each node having **5** immediate children. This makes the table **2,441,405** records long.

Each record has a **100**-byte long field `stuffing` which emulates the payload in actual tables.

The fields `parent`, `lft` and `rgt` are indexed.

All descendants

There are lots of descendants, that's why we will select and aggregate the lengths of their `stuffing` fields. Since that field is not indexed, it will emulate selection of all values from an actual table rather well.

Nested sets

view sourceprint?

```
1.SELECT SUM(LENGTH(hc.stuffing))
2.FROM  t_hierarchy hp
3.JOIN  t_hierarchy hc
4.ON   hc.lft BETWEEN hp.lft AND hp.rgt
5.WHERE hp.id = 42
```

View query details

sum
1953100

1 row fetched in 0.0001s (0.0559s)

```
Aggregate  (cost=63042.65..63042.66 rows=1 width=101)
  ->  Nested Loop  (cost=5761.08..62364.46 rows=271274 width=101)
        ->  Index Scan using pk_hierarchy_id on t_hierarchy hp  (cost=0.00..8.54
rows=1 width=8)
              Index Cond: (id = 42)
        ->  Bitmap Heap Scan on t_hierarchy hc  (cost=5761.08..58286.82
rows=271274 width=105)
              Recheck Cond: ((hc.lft >= hp.lft) AND (hc.lft <= hp.rgt))
              ->  Bitmap Index Scan on ix_hierarchy_lft  (cost=0.00..5693.26
rows=271274 width=0)
                    Index Cond: ((hc.lft >= hp.lft) AND (hc.lft <= hp.rgt))
```

Nested sets is particularly good for this kind of query, since it requires a single range scan on the index on `lft`.

This query runs for **50 ms**.

Adjacency list

view sourceprint?

```
01.WITH  RECURSIVE
      02.         q AS
      03.         (
      04.         SELECT id, stuffing
      05.         FROM  t_hierarchy h
      06.         WHERE  id = 42
      07.         UNION ALL
      08.         SELECT hc.id, hc.stuffing
      09.         FROM  q
      10.         JOIN  t_hierarchy hc
      11.         ON   hc.parent = q.id
      12.         )
```

```
13.SELECT SUM(LENGTH(stuffing))
14.FROM  q
```

View query details

sum
1953100

1 row fetched in 0.0001s (0.0985s)

```
Aggregate  (cost=915.24..915.26 rows=1 width=218)
  CTE q
    ->  Recursive Union  (cost=0.00..898.34 rows=751 width=105)
          ->  Index Scan using pk_hierarchy_id on t_hierarchy h  (cost=0.00..8.54
rows=1 width=105)
                Index Cond: (id = 42)
          ->  Nested Loop  (cost=0.00..87.48 rows=75 width=105)
                ->  WorkTable Scan on q  (cost=0.00..0.20 rows=10 width=4)
                ->  Index Scan using ix_hierarchy_parent on t_hierarchy hc
(cost=0.00..8.64 rows=7 width=109)
                      Index Cond: (hc.parent = q.id)
  ->  CTE Scan on q  (cost=0.00..15.02 rows=751 width=218)
```

This query is a trifle less efficient since it requires several index scans instead of a single one. However, the resulting range is of course the same (because the values returned are the same).

This query completes in **98 ms**, or less than twice as long as the nested sets one.

All ancestors

Nested sets

view sourceprint?

```
1.SELECT hp.id, hp.parent, hp.lft, hp.rgt, hp.data
2.FROM  t_hierarchy hc
3.JOIN  t_hierarchy hp
4.ON   hc.lft BETWEEN hp.lft AND hp.rgt
5.WHERE hc.id = 1000000
6.ORDER BY
      7.         hp.lft
```

View query details

id	parent	lft	rgt	data
2	0	585938	1171874	Value 2
12	2	703126	820312	Value 12
63	12	750001	773437	Value 63
319	63	764063	768749	Value 319
1599	319	766875	767811	Value 1599
7999	1599	767437	767623	Value 7999
39999	7999	767549	767585	Value 39999
199999	39999	767571	767577	Value 199999
1000000	199999	767576	767576	Value 1000000

9 rows fetched in 0.0006s (1.8281s)

```
Sort  (cost=109888.38..110566.56 rows=271274 width=29)
  Sort Key: hp.lft
  ->  Nested Loop  (cost=15239.64..85406.72 rows=271274 width=29)
        Join Filter: (hc.lft >= hp.lft)
        ->  Index Scan using pk_hierarchy_id on t_hierarchy hc  (cost=0.00..8.54
rows=1 width=4)
              Index Cond: (id = 1000000)
        ->  Bitmap Heap Scan on t_hierarchy hp  (cost=15239.64..73190.86
rows=813822 width=29)
              Recheck Cond: (hc.lft <= hp.rgt)
              ->  Bitmap Index Scan on ix_hierarchy_rgt  (cost=0.00..15036.18
rows=813822 width=0)
                    Index Cond: (hc.lft <= hp.rgt)
```

This query returns much fewer rows than the previous one (only **9** rows instead of almost **200,000**), but due to its nature it is much more slow and takes almost **2 seconds**.

This is because we search the other way round in this case: instead of looking for indexed value within the range of constants, we need to search the constant against the list of ranges.

Ranges cannot be efficiently indexed using **B-Tree** indexes, that's why **PostgreSQL** uses only part of the condition (`hc.lft <= hp.rgt`), builds a bitmap on it, scans the table using this bitmap and filters the values using the second part of the condition (`hc.lft <= hp.rgt`).

This is quite a costly operations since it requires an index scan (which **PostgreSQL** is not very good at) which returns almost half of all rows.

Adjacency list

view sourceprint?

```
01.WITH  RECURSIVE
      02.         q AS
      03.         (
      04.         SELECT h.*, 1 AS level
      05.         FROM  t_hierarchy h
      06.         WHERE  id = 1000000
      07.         UNION ALL
      08.         SELECT hp.*, level + 1
      09.         FROM  q
      10.         JOIN  t_hierarchy hp
      11.         ON   hp.id = q.parent
      12.         )
13.SELECT id, parent, lft, rgt, data
14.FROM  q
15.ORDER BY
      16.         level DESC
```

View query details

id	parent	lft	rgt	data
2	0	585938	1171874	Value 2
12	2	703126	820312	Value 12

63	12	750001	773437	Value 63
319	63	764063	768749	Value 319
1599	319	766875	767811	Value 1599
7999	1599	767437	767623	Value 7999
39999	7999	767549	767585	Value 39999
199999	39999	767571	767577	Value 199999
1000000	199999	767576	767576	Value 1000000

9 rows fetched in 0.0006s (0.0044s)

```
Sort  (cost=872.98..873.23 rows=101 width=238)
  Sort Key: q.level
  CTE q
    ->  Recursive Union  (cost=0.00..867.59 rows=101 width=134)
          ->  Index Scan using pk_hierarchy_id on t_hierarchy h  (cost=0.00..8.54
rows=1 width=130)
                Index Cond: (id = 1000000)
          ->  Nested Loop  (cost=0.00..85.70 rows=10 width=134)
                ->  WorkTable Scan on q  (cost=0.00..0.20 rows=10 width=8)
                ->  Index Scan using pk_hierarchy_id on t_hierarchy hp
(cost=0.00..8.54 rows=1 width=130)
                      Index Cond: (hp.id = q.parent)
  ->  CTE Scan on q  (cost=0.00..2.02 rows=101 width=238)
```

Now this query is literally instant: only **4 ms** which is within the time measurement error range.

Recursion does very good job here: since the hierarchy is limited, traversing the tree upwards takes only **9** index lookups on the `PRIMARY KEY` and then sorting of **9** values. Both operations are very simple and complete in no time.

Descendants up to a given level

Nested sets

We will run two queries: one with a node close to the root, the second one with a node far from the root.

view sourceprint?

```
01.SELECT hc.id, hc.parent, hc.lft, hc.rgt, hc.data
02.FROM  t_hierarchy hp
03.JOIN  t_hierarchy hc
04.ON   hc.lft BETWEEN hp.lft AND hp.rgt
05.WHERE hp.id = ?
      06.         AND
      07.         (
      08.         SELECT COUNT(*)
      09.         FROM  t_hierarchy hn
      10.         WHERE hc.lft BETWEEN hn.lft AND hn.rgt
            11.                  AND hn.lft BETWEEN hp.lft AND hp.rgt
      12.         ) <= 3
```

View query details for node 42

view sourceprint?

```
01.SELECT hc.id, hc.parent, hc.lft, hc.rgt, hc.data
02.FROM  t_hierarchy hp
03.JOIN  t_hierarchy hc
04.ON   hc.lft BETWEEN hp.lft AND hp.rgt
05.WHERE  hp.id = 42
      06.         AND
      07.         (
      08.         SELECT COUNT(*)
      09.         FROM  t_hierarchy hn
      10.         WHERE  hc.lft BETWEEN hn.lft AND hn.rgt
            11.                   AND hn.lft BETWEEN hp.lft AND hp.rgt
      12.         ) <= 3
```

id	parent	lft	rgt	data
42	8	257814	281250	Value 42
211	42	257815	262501	Value 211
1056	211	257816	258752	Value 1056
1057	211	258753	259689	Value 1057
1058	211	259690	260626	Value 1058
1059	211	260627	261563	Value 1059
1060	211	261564	262500	Value 1060
212	42	262502	267188	Value 212
1061	212	262503	263439	Value 1061
1062	212	263440	264376	Value 1062
1063	212	264377	265313	Value 1063
1064	212	265314	266250	Value 1064
1065	212	266251	267187	Value 1065
213	42	267189	271875	Value 213
1066	213	267190	268126	Value 1066
1067	213	268127	269063	Value 1067
1068	213	269064	270000	Value 1068
1069	213	270001	270937	Value 1069
1070	213	270938	271874	Value 1070
214	42	271876	276562	Value 214
1071	214	271877	272813	Value 1071
1072	214	272814	273750	Value 1072
1073	214	273751	274687	Value 1073
1074	214	274688	275624	Value 1074
1075	214	275625	276561	Value 1075
215	42	276563	281249	Value 215
1076	215	276564	277500	Value 1076

1077	215	277501	278437	Value 1077
1078	215	278438	279374	Value 1078
1079	215	279375	280311	Value 1079
1080	215	280312	281248	Value 1080

31 rows fetched in 0.0156s (120.6055s)

```
Nested Loop  (cost=0.00..6875628456.64 rows=90425 width=29)
  Join Filter: ((SubPlan 1) <= 3)
  ->  Index Scan using pk_hierarchy_id on t_hierarchy hp  (cost=0.00..8.54 rows=1
width=8)
        Index Cond: (id = 42)
  ->  Index Scan using ix_hierarchy_lft on t_hierarchy hc  (cost=0.00..535180.52
rows=271274 width=29)
        Index Cond: ((hc.lft >= hp.lft) AND (hc.lft <= hp.rgt))
  SubPlan 1
    ->  Aggregate  (cost=25343.70..25343.71 rows=1 width=0)
          ->  Index Scan using ix_hierarchy_lft on t_hierarchy hn
(cost=0.00..25333.52 rows=4069 width=0)
                Index Cond: (($0 >= lft) AND (lft >= $1) AND (lft <= $2))
                Filter: ($0 <= rgt)
```

View query details for node 31,415

view sourceprint?

```
01.SELECT hc.id, hc.parent, hc.lft, hc.rgt, hc.data
02.FROM  t_hierarchy hp
03.JOIN  t_hierarchy hc
04.ON   hc.lft BETWEEN hp.lft AND hp.rgt
05.WHERE  hp.id = 31415
      06.         AND
      07.         (
      08.         SELECT COUNT(*)
      09.         FROM  t_hierarchy hn
      10.         WHERE  hc.lft BETWEEN hn.lft AND hn.rgt
            11.                   AND hn.lft BETWEEN hp.lft AND hp.rgt
      12.         ) <= 3
```

id	parent	lft	rgt	data
31415	6282	445651	445687	Value 31415
157076	31415	445652	445658	Value 157076
785381	157076	445653	445653	Value 785381
785382	157076	445654	445654	Value 785382
785383	157076	445655	445655	Value 785383
785384	157076	445656	445656	Value 785384
785385	157076	445657	445657	Value 785385
157077	31415	445659	445665	Value 157077
785386	157077	445660	445660	Value 785386
785387	157077	445661	445661	Value 785387

785388 157077 445662 445662 Value 785388
785389 157077 445663 445663 Value 785389
785390 157077 445664 445664 Value 785390
157078 31415 445666 445672 Value 157078
785391 157078 445667 445667 Value 785391
785392 157078 445668 445668 Value 785392
785393 157078 445669 445669 Value 785393
785394 157078 445670 445670 Value 785394
785395 157078 445671 445671 Value 785395
157079 31415 445673 445679 Value 157079
785396 157079 445674 445674 Value 785396
785397 157079 445675 445675 Value 785397
785398 157079 445676 445676 Value 785398
785399 157079 445677 445677 Value 785399
785400 157079 445678 445678 Value 785400
157080 31415 445680 445686 Value 157080
785401 157080 445681 445681 Value 785401
785402 157080 445682 445682 Value 785402
785403 157080 445683 445683 Value 785403
785404 157080 445684 445684 Value 785404
785405 157080 445685 445685 Value 785405
31 rows fetched in 0.0017s (0.0523s)

```
Nested Loop  (cost=0.00..6875628456.64 rows=90425 width=29)
  Join Filter: ((SubPlan 1) <= 3)
  ->  Index Scan using pk_hierarchy_id on t_hierarchy hp  (cost=0.00..8.54 rows=1
width=8)
        Index Cond: (id = 31415)
  ->  Index Scan using ix_hierarchy_lft on t_hierarchy hc  (cost=0.00..535180.52
rows=271274 width=29)
        Index Cond: ((hc.lft >= hp.lft) AND (hc.lft <= hp.rgt))
  SubPlan 1
    ->  Aggregate  (cost=25343.70..25343.71 rows=1 width=0)
          ->  Index Scan using ix_hierarchy_lft on t_hierarchy hn
(cost=0.00..25333.52 rows=4069 width=0)
                Index Cond: (($0 >= lft) AND (lft >= $1) AND (lft <= $2))
                Filter: ($0 <= rgt)
```

We see that the second query is reasonably fast (completes in **50 ms**).

However, the first query (which is in fact more often used) takes **120.6 seconds**, or more than **2 minutes**!

This is because the query should count all ancestors for all descendants that are within the given node.

It's fast for the nodes that are further from the root (since they don't have lots of descendants), but it may become a real problem when trying to obtain, say, children and grandchildren of a root node.

And this is the task most online catalogs begin their work with: they need to show first-level categories and subcategories. **2 minutes** is way too much for this.

Adjacency list

view sourceprint?

```
01.WITH  RECURSIVE
      02.         q AS
      03.         (
      04.         SELECT id, parent, lft, rgt, data, ARRAY[id] AS
      level
      05.         FROM  t_hierarchy hc
      06.         WHERE  id = ?
      07.         UNION ALL
      08.         SELECT hc.id, hc.parent, hc.lft, hc.rgt, hc.data,
      q.level || hc.id
      09.         FROM  q
      10.         JOIN  t_hierarchy hc
      11.         ON   hc.parent = q.id
      12.         WHERE  array_upper(level, 1) < 3
      13.         )
14.SELECT id, parent, lft, rgt, data
15.FROM  q
16.ORDER BY
      17.         level
```

Note the `ORDER BY` and `level` constructs. They are intended to preserve the tree-like ordering. Arrays are ordered lexicographically in **PostgreSQL** and each `level` contains the breadcrumbs from the root node to the current node.

Here are the query results:

View query details for node 42

view sourceprint?

```
01.WITH  RECURSIVE
      02.         q AS
      03.         (
      04.         SELECT id, parent, lft, rgt, data, ARRAY[id] AS
      level
      05.         FROM  t_hierarchy hc
      06.         WHERE  id = 42
      07.         UNION ALL
      08.         SELECT hc.id, hc.parent, hc.lft, hc.rgt, hc.data,
      q.level || hc.id
      09.         FROM  q
      10.         JOIN  t_hierarchy hc
      11.         ON   hc.parent = q.id
      12.         WHERE  array_upper(level, 1) < 3
      13.         )
14.SELECT id, parent, lft, rgt, data
15.FROM  q
16.ORDER BY
```

17. level

id	parent	lft	rgt	data
42	8	257814	281250	Value 42
211	42	257815	262501	Value 211
1056	211	257816	258752	Value 1056
1057	211	258753	259689	Value 1057
1058	211	259690	260626	Value 1058
1059	211	260627	261563	Value 1059
1060	211	261564	262500	Value 1060
212	42	262502	267188	Value 212
1061	212	262503	263439	Value 1061
1062	212	263440	264376	Value 1062
1063	212	264377	265313	Value 1063
1064	212	265314	266250	Value 1064
1065	212	266251	267187	Value 1065
213	42	267189	271875	Value 213
1066	213	267190	268126	Value 1066
1067	213	268127	269063	Value 1067
1068	213	269064	270000	Value 1068
1069	213	270001	270937	Value 1069
1070	213	270938	271874	Value 1070
214	42	271876	276562	Value 214
1071	214	271877	272813	Value 1071
1072	214	272814	273750	Value 1072
1073	214	273751	274687	Value 1073
1074	214	274688	275624	Value 1074
1075	214	275625	276561	Value 1075
215	42	276563	281249	Value 215
1076	215	276564	277500	Value 1076
1077	215	277501	278437	Value 1077
1078	215	278438	279374	Value 1078
1079	215	279375	280311	Value 1079
1080	215	280312	281248	Value 1080

31 rows fetched in 0.0017s (0.0054s)

```
Sort  (cost=290.87..291.42 rows=221 width=266)
  Sort Key: q.level
  CTE q
    ->  Recursive Union  (cost=0.00..277.84 rows=221 width=61)
          ->  Index Scan using pk_hierarchy_id on t_hierarchy hc  (cost=0.00..8.54
rows=1 width=29)
                Index Cond: (id = 42)
          ->  Nested Loop  (cost=0.00..26.49 rows=22 width=61)
                ->  WorkTable Scan on q  (cost=0.00..0.25 rows=3 width=36)
                      Filter: (array_upper(level, 1) < 3)
```

```
                ->  Index Scan using ix_hierarchy_parent on t_hierarchy hc
(cost=0.00..8.64 rows=7 width=29)
                      Index Cond: (hc.parent = q.id)
  ->  CTE Scan on q  (cost=0.00..4.42 rows=221 width=266)
```

View query details for node 31,415

view sourceprint?

```
01.WITH  RECURSIVE
      02.         q AS
      03.         (
      04.         SELECT id, parent, lft, rgt, data, ARRAY[id] AS
      level
      05.         FROM  t_hierarchy hc
      06.         WHERE  id = 31415
      07.         UNION ALL
      08.         SELECT hc.id, hc.parent, hc.lft, hc.rgt, hc.data,
      q.level || hc.id
      09.         FROM  q
      10.         JOIN  t_hierarchy hc
      11.         ON   hc.parent = q.id
      12.         WHERE array_upper(level, 1) < 3
      13.         )
14.SELECT id, parent, lft, rgt, data
15.FROM  q
16.ORDER BY
      17.         level
```

id	parent	lft	rgt	data
31415	6282	445651	445687	Value 31415
157076	31415	445652	445658	Value 157076
785381	157076	445653	445653	Value 785381
785382	157076	445654	445654	Value 785382
785383	157076	445655	445655	Value 785383
785384	157076	445656	445656	Value 785384
785385	157076	445657	445657	Value 785385
157077	31415	445659	445665	Value 157077
785386	157077	445660	445660	Value 785386
785387	157077	445661	445661	Value 785387
785388	157077	445662	445662	Value 785388
785389	157077	445663	445663	Value 785389
785390	157077	445664	445664	Value 785390
157078	31415	445666	445672	Value 157078
785391	157078	445667	445667	Value 785391
785392	157078	445668	445668	Value 785392

```
785393  157078  445669  445669  Value 785393
785394  157078  445670  445670  Value 785394
785395  157078  445671  445671  Value 785395
157079  31415   445673  445679  Value 157079
785396  157079  445674  445674  Value 785396
785397  157079  445675  445675  Value 785397
785398  157079  445676  445676  Value 785398
785399  157079  445677  445677  Value 785399
785400  157079  445678  445678  Value 785400
157080  31415   445680  445686  Value 157080
785401  157080  445681  445681  Value 785401
785402  157080  445682  445682  Value 785402
785403  157080  445683  445683  Value 785403
785404  157080  445684  445684  Value 785404
785405  157080  445685  445685  Value 785405
31 rows fetched in 0.0017s (0.0054s)
Sort  (cost=290.87..291.42 rows=221 width=266)
  Sort Key: q.level
  CTE q
    ->  Recursive Union  (cost=0.00..277.84 rows=221 width=61)
          ->  Index Scan using pk_hierarchy_id on t_hierarchy hc  (cost=0.00..8.54
rows=1 width=29)
                Index Cond: (id = 31415)
          ->  Nested Loop  (cost=0.00..26.49 rows=22 width=61)
                ->  WorkTable Scan on q  (cost=0.00..0.25 rows=3 width=36)
                      Filter: (array_upper(level, 1) < 3)
                ->  Index Scan using ix_hierarchy_parent on t_hierarchy hc
(cost=0.00..8.64 rows=7 width=29)
                      Index Cond: (hc.parent = q.id)
  ->  CTE Scan on q  (cost=0.00..4.42 rows=221 width=266)
```

As we can see, both queries complete in a little more than **5 ms** (instantly) and this time does not depend on the proximity to the root node.

Summary

We have compared the three most common queries that are usually issued against the hierarachical data:

1. Find all descendants of a given node
2. Find all ancestors of a given node
3. Find all descendants of a given node up to a certain depth

The **nested set** model the fastest for the first query (**0.05 s**), however, **adjacency list** shows good performance and is very fast too (selecting **200,000** rows is a matter of less than **0.1 second**).

For the second query, **adjacency list** is much faster, however, the **nested sets** are still usable.

Finally, for the third query, **nested sets** model shows dependency on the node.

For a node that has few descendants, the query is rather fast, however, for a node close to the root (to

say nothing of the root itself) this query is intolerably slow.

Adjacency list shows superb performance on both nodes.

Conclusion

Given the said above and taking into account that the nested sets model is much harder to manage, we can conclude that **adjacency list model** should be used to manage hierarchical data in **PostgreSQL 8.4**.

Nested sets model was a very smart invention to manage hierarchical data in an environment that allowed no recursion. But now, when recursive queries are finally available, **adjacency list model** is just better.

It yields excellent performance on all three types of queries, outperforms nested sets in two of three most used queries and is extremely simple to manage.

To be continued.