How to make sense of your hierarchical content

Seeing the Wood for the Trees — Categories, Menus and More.

Jonathan Guthrie

In pretty much everything we do online, we're dealing with organizing content, and mostly it's hierarchical content. Site content, menu systems, product categories and endless subcategories, forums, examples are everywhere we look.

We can just ignore it, put physical files in folders and let the client author hard files and put them in the right place, letting the user navigate something you hope is not broken because the client, one of your staff, or heaven forbid, you yourself, put something in the wrong directory. That's the first approach many of us started with, and yet once you get above just a few pages it becomes a nightmare to maintain.

XML is a perfect example of a data model that is hierarchical in nature, but not a lot of people store dynamic content as XML in situations where it's read from and written to frequently, and for very valid reasons.

Relational databases are essentially flat data with relationships, and yet while creating a flexible hierarchical data model can be challenging, using a database to store and retrieve data fits well with the way most of us work.

Multi-table approach

A system I used in the early days of content management was to have a separate table for each level of navigation. I had primary, secondary and tertiary nav tables, and while it made many things drop dead easy, it was severely limited in flexibility, and I had to have an additional table for content.

Primary	Secondary	Tertiary	Content
ID	ID	ID	ID
Name	Name	Name	Page Name
Display_order	Primary	Primary	Primary
Status	Display_order	Secondary (optional)	Secondary
	Status	Display_order	Tertiary
		Status	Content
			Status

It's a workable system, if limited. It has the advantage of being simple to understand, but laden with snafus such as...

"Can I have another level in my menus please"

"Sorry, that's all you get" became a frequent story during deployment of new sites, and we often justified it successfully with "Best Practice Guidelines" and all that garbage. Then again Best

Practice is only Best Practice as long as you have need of it, or no-one else has developed a better practice.

Getting the data out

To generate the menus or list of categories, you can either nest your queries (inlines) or try the one-hit SQL join. Either way gets ugly, and as much as it pains me to admit, the nesting of queries/inlines is often the only way to truly control sense of place etc.

Should I be seeing this?

It's wise in any CMS or product catalogue to allow content to be worked on while not actively viewable by Joe Average, so you need a status flag. However unless you join your SQL to ascertain the status of the content, you have to store status in the nav tables. That gets painful to maintain, and the entire situation can escalate out of control.

Data Integrity (Referential)

If you have a page that's attached to a tertiary nav row, what happens if the tertiary row's deleted? You get an orphaned content row. It makes your job as a developer harder because at every step you have to ensure the integrity of multiple table's data, and there's no easy way to lock that down on the database side.

The Adjacency Model

There are quite a few other variations that rely on multiple tables and on single tables, but we're going to focus on just 2 more. The easier of the two to understand and hence the more popular one is the Adjacency Model.

If you look at a typical off-license, the product catalogue might go something like this: (abbreviated!)

```
Alcoholic Beverages
  Beers
  Wines
    White Wines
      Chardonnay
        Oaked
        Unoaked
Sauvignon Blanc
Pinot Gris
Riesling
Gewurztraminer
Viognier
Semillon
    Red Wines
    Champagne
    Sparkling
    Sweet Dessert
    Ports/Sherries
  Spirits
Non-Alcoholic Beverages
Snacks
```

Immediately you can see that the multi-table model will barf on this - we need something much more adaptive.

In the adjacency model you want to store the element's immediate parent element's id. This enables an easy inheritance reference.

Id	Name	Parent	Status
1	Alcoholic Beverages	0	1
2	Beers	1	1
3	Wines	1	1
4	White Wines	3	1
5	Chardonnay	4	1
6	Sauvignon Blanc	4	1
7	Pinot Gris	4	1
8	Riesling	4	1
9	Gewurztraminer	4	0
10	Viognier	4	1
11	Semillon	4	1
12	Red Wines	3	1
13	Champagne	3	1
14	Sparkling	3	1
15	Sweet Dessert	3	1
16	Ports/Sherries	3	1
17	Spirits	1	1
18	Non-Alcoholic Beverages	0	1
19	Snacks	0	1
20	Oaked	5	1
21	Unoaked	5	1

As you can see, one table handles all categories. Now you can simply have products that have a single parent reference, or if this were a CMS then your body content, template reference etc, could live in the row with your category. Even a custom non-alphabetical display order can be handled neatly.

Simplicity is also one of it's strengths: one can see what the immediate parent is at a glance.

So lets look at some of the ways to work with this data. Please bear in mind that there will be other ways to do this, an exhaustive display is outside the scope of this presentation!

Displaying the tree

When Jane Average is browsing your site, if she's not searching for the product she already knows she wants, she often starts at the top and follows a path to the category she's interested in, so we'd like to get the full tree from the database.

Assuming she's selected "Alcoholic Beverages":

```
SELECT c1.name AS cat1, c2.name as cat2, c3.name as cat3, c4.name as cat4
FROM adjacency AS c1
  LEFT JOIN adjacency AS c2 ON c2.parent = c1.id
  LEFT JOIN adjacency AS c3 ON c3.parent = c2.id
  LEFT JOIN adjacency AS c4 ON c4.parent = c3.id
WHERE c1.id = 1;
```

Returns:

Cat1	Cat2	Cat3	Cat4
Alcoholic Beverages	Beers	[Null]	[Null]
Alcoholic Beverages	Wines	White Wines	Chardonnay
Alcoholic Beverages	Wines	White Wines	Sauvignon Blanc
Alcoholic Beverages	Wines	White Wines	Pinot Gris
Alcoholic Beverages	Wines	White Wines	Riesling
Alcoholic Beverages	Wines	White Wines	Gewurztraminer
Alcoholic Beverages	Wines	White Wines	Viognier
Alcoholic Beverages	Wines	White Wines	Semillon
Alcoholic Beverages	Wines	Red Wines	[Null]
Alcoholic Beverages	Wines	Champagne	[Null]
Alcoholic Beverages	Wines	Sparkling	[Null]
Alcoholic Beverages	Wines	Sweet Dessert	[Null]
Alcoholic Beverages	Wines	Ports/Sherries	[Null]
Alcoholic Beverages	Spirits	[Null]	[Null]

Now there's a few obvious problems here:

- it's not returning Wines as a distinct row because it has at least 1 child, hence Beers returning and not Wines.
- If you try to do this query originating at the top of the tree you get some funky unintended results.
- Note that the categories Oaked and Unoaked are missing from Chardonnay, they're judged 5th level and we only requested data down to 4. You have to know in advance, at the code stage, how many levels deep the client's going to want to show Jane Average.

Around about now it's obvious the solution is Lasso. Create a recursive tag that simply calls itself until the whole tree's assembled.

For example:

```
define_type('adjacency',-priority='replace',-namespace='demo_');
    define_tag('drill', -required='id',-optional='level');
      local('out' = string);
      inline(
-database='summitDemo',
-SQL='SELECT * FROM adjacency WHERE parent = '#id';');
        records;
          #out += #level'. '(field('name'))'<br \>';
          #out += demo_adjacency->(drill(
```

```
-id=(field('id')),
-level=#level+1));
       /records;
     /inline;
     return(@#out);
   /define_tag;
  /define_type;
  // action and output the above tag
  demo_adjacency->(drill(-id=0,-level=0));
```

So you could in theory do that from any point in the tree and get all siblings and children. You could also build in a depth limiter which would stop it crawling more than 2 levels deep so that you only returned what was appropriate for a shallow top level menu system.

Difficulties with the Adjacency Model

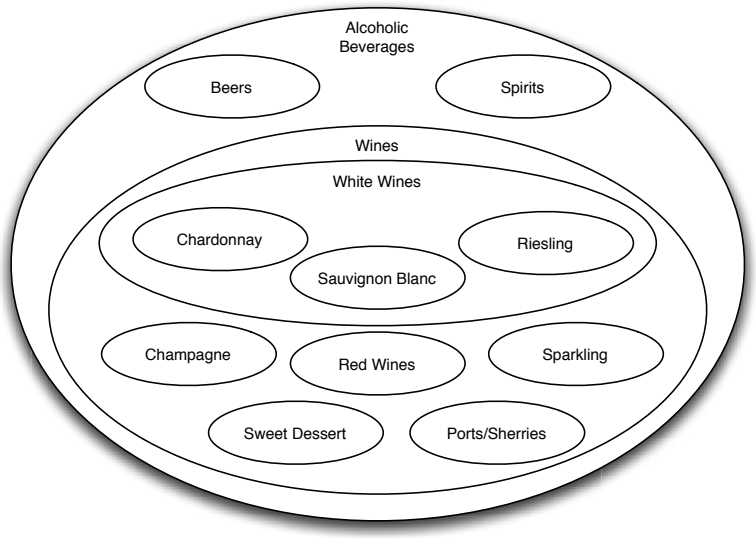
Where the adjacency model falls down is that you have to build in exactly how you want to present it into the recursive presentation tag, unless you want to make it into a nested array but when you think about it if you do that you're no further ahead at all.

Referential integrity is not easily taken care of unless once again you do these checks in lasso. It would be all too easy to accidentally orphan nodes, however if that does happen, it's a piece of cake to fix by a bit of manually editing.

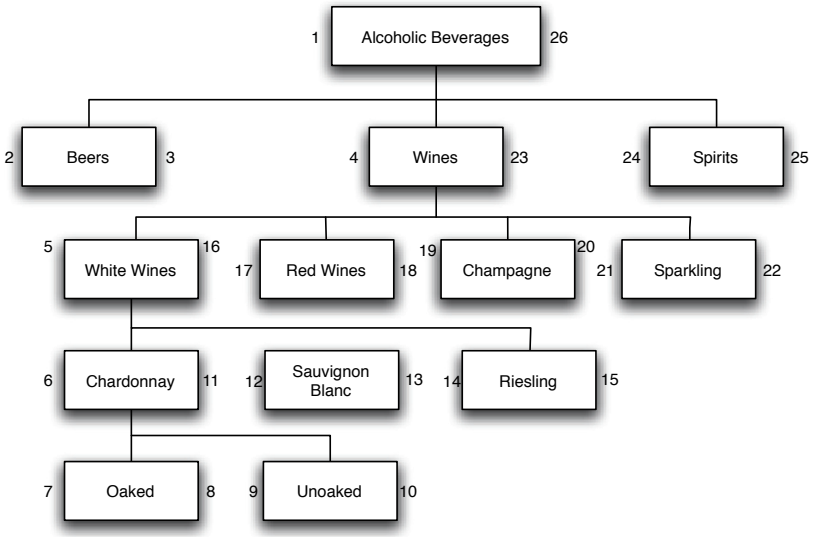
My main objection to this model is that it forces us to use a recursive approach. Not only is it slow, but it will be prone to getting bogged down with extremely large data sets, as I found out with a certain scientific research application.

The Nested Set Model

The previous examples are so simple (easy to understand) because they're either constrained or linear. However rather than conceptualizing data as linear objects, think of hierarchical data as groups, or nested sets. A group that is not fully contained in a parent group might need to be re-classified.



So what we need to do is work out a way to represent these groups, or sets. If we represent this data in a tree diagram, to quote Mike Hillyer in an article on mysql.com, "When working with a tree, we work from left to right, one layer at a time, descending to each node's children before assigning a right-hand number and moving on to the right. This approach is called the modified preorder tree traversal algorithm."



About now I feel I need to offer up a warning: This method requires MySQL 4.1 (or a datasource that utilizes subqueries and variables) and a clear, sober head. Some of what I am presenting here is very close to the first major article I absorbed on the subject and I make no apologies for similarities... however I will be providing LP8-specific examples that are in use IRL. The full article by Mike Hillyer can be found at http://dev.mysql.com/tech-resources/articles/hierarchical-data.html

The database would look something like this:

```
CREATE TABLE `nestedset` (
  `id` int(11) NOT NULL auto_increment,
  `name` varchar(64) collate utf8_bin NOT NULL default '',
  `lft` int(11) NOT NULL default '0',
  `rgt` int(11) NOT NULL default '0',
  `status` int(2) NOT NULL default '1',
  PRIMARY KEY  (`id`))
insert into `nestedset` values('1','Alcoholic Beverages','1','38','1'),
 ('2','Beers','2','3','1'),
 ('3','Wines','4','35','1'),
 ('4','White Wines','5','24','1'),
 ('5','Chardonnay','6','11','1'),
 ('6','Sauvignon Blanc','12','13','1'),
 ('7','Pinot Gris','14','15','1'),
 ('8','Riesling','16','17','1'),
 ('9','Gewurztraminer','18','19','1'),
 ('10','Viognier','20','21','1'),
 ('11','Semillon','22','23','1'),
 ('12','Red Wines','25','26','1'),
 ('13','Champagne','27','28','1'),
 ('14','Sparkling','29','30','1'),
 ('15','Sweet Dessert','31','32','1'),
 ('16','Ports/Sherries','33','34','1'),
 ('17','Spirits','36','37','1'),
```

```
('18','Non-Alcoholic Beverages','39','40','1'),
('19','Snacks','41','42','1'),
('20','Oaked','7','8','1'),
('21','Unoaked','9','10','1');
```

Note ids 20 and 21, they belong inside Chardonnay, and the lft and rgt values of Unoaked and Oaked neatly fit inside Chardonnay's lft and rgt.

Selecting the ordered data, all the data is returned ordered.

```
SELECT node.id, node.name, node.lft, node.rgt
FROM nestedset AS node
WHERE node.status = 1
ORDER BY node.lft;
```

In multi-table and adjacency models the path root through child has to be a known maximum depth. In the Nested Set model it's not required.

```
SELECT parent.id, parent.name
FROM nestedset AS node,nestedset AS parent
WHERE node.lft BETWEEN parent.lft AND parent.rgt AND node.id = 20
ORDER BY node.lft;
1     Alcoholic Beverages
3     Wines
4     White Wines
5     Chardonnay
20      Oaked
```

Remember the adjacency model and our nested cTag that burrowed though the hierarchy, returning a "formatted" string, nested and with depth? We're about to annihilate it with one foul query.

```
SELECT node.id, node.name, (COUNT(parent.name) - 1) AS depth
FROM nestedset AS node, nestedset AS parent
WHERE node.lft BETWEEN parent.lft AND parent.rgt
GROUP BY node.name
ORDER BY node.lft;
```

ID	Name	Depth
1	Alcoholic Beverages	0
2	Beers	1
3	Wines	1
4	White Wines	2
5	Chardonnay	3
20	Oaked	4
21	Unoaked	4
6	Sauvignon Blanc	3
7	Pinot Gris	3
8	Riesling	3
9	Gewurztraminer	3
10	Viognier	3
11	Semillon	3
12	Red Wines	2
13	Champagne	2
14	Sparkling	2
15	Sweet Dessert	2
16	Ports/Sherries	2
17	Spirits	1
18	Non-Alcoholic Beverages	0
19	Snacks	0

Yeah I know this is supposed to be a lasso paper, but imagine what we can do when the hard part's taken care of in SQL and we can concentrate on writing code rather than untying the knots we've tied ourselves in with recursive tags and so on!

Tying this in with Lasso

Adding a new node

It's relatively easy to create new nodes in multi-table and adjacency models, but it's more complex in the nested set model. Fortunately the hard work has been done by those with higher IQ's, and I've adapted some of these to cTags.

So we want to get the right value of the item we're inserting AFTER, and shuffle those rows to the RIGHT up by 2 to make way for the new row, then inserting the new row with appropriate lft and rgt values. Simple once you get used to it!

```
define_tag('addSibling',
    -Required='cattable',
    -Required='txt',
    -Optional='othersmap',
    -Required='id',
    -Description='Adds entry after another child, same level'
      );

  local('xtraFields' = string);
```

```
  local('xtraValues' = string);
  if(local_defined('othersmap') && local('othersmap')->(IsA('Map')));
    iterate(#othersmap,local('temp'));
      #xtraFields += ','+#temp->name;
      #xtraValues += ","'+#temp->value+'"';
    /iterate;
  /if;
  local('sSQL' = '
    LOCK TABLE '+#cattable+' WRITE;

    SELECT @myRight := rgt FROM '+#cattable+' WHERE id = '+#id+';

    UPDATE '+#cattable+' SET rgt = rgt + 2 WHERE rgt > @myRight;
    UPDATE '+#cattable+' SET lft = lft + 2 WHERE lft > @myRight;

    INSERT INTO '+#cattable+'(name, lft, rgt'+#xtraFields+') VALUES("'+(encode_
sql(#txt))+'", @myRight + 1, @myRight + 2'+#xtraValues+');

    UNLOCK TABLES;
  ');
  inline($gv_sql,-SQL=#sSQL);
  /inline;
/define_tag;
USAGE:
xs_cat->(addSibling(
  -cattable='nestedset',-txt=#txt,-othersmap=(map('f1'=#f1)),-id=#id));
```

Points to note:

- The table name is dynamic as we use this in situations where there may be more than one nested set table in a given client solution.
- "Othersmap" is a method of getting the "other" data into the row at creation time. Remember, this is a generic tag that gets used in a number of solutions.
- Note the use of multiple statements in the executed block.
- The LOCK TABLES is used to ensure no other query either gets in the way or gets wrong data.

The tag to add a nested child is almost identical, see the tag "addChild" in the cTags file supplied for this presentation on your Lasso 2006 Summit CD.

However this time we select the LEFT value of the item we're inserting INTO, and again shuffle those rows to the RIGHT up by 2 to make way for the new row, then inserting the new row with appropriate lft and rgt values. The biggest difference is that the shuffling is relative to the LEFT of the identified row rather than the right.

Deleting nodes

Remember "Referential Data Integrity"? We've pretty much got that sorted here too, especially if the row also includes the content data. If you delete the navigation item, you're deleting the content. You can also set it up so that if you're deleting a row that has child nodes, then the child nodes are history too.

```
define_tag('deleteNode',
      -Required='cattable',
      -Required='id'
      );
  local('sSQL' = '
    LOCK TABLE '+#cattable+' WRITE;
```

```
    SELECT @myLeft := lft, @myRight := rgt, @myWidth := rgt - lft + 1
    FROM '+#cattable+'
    WHERE id = '+#id+';

    DELETE FROM '+#cattable+' WHERE lft BETWEEN @myLeft AND @myRight;

    UPDATE '+#cattable+' SET rgt = rgt - @myWidth WHERE rgt > @myRight;
    UPDATE '+#cattable+' SET lft = lft - @myWidth WHERE lft > @myRight;

    UNLOCK TABLES;
  ');
  inline($gv_sql,-SQL=#sSQL);
  /inline;
/define_tag;
USAGE:
xs_cat->(deleteNode(-cattable='category',-id=#id));
```

The actual DELETE is not on an ID but on lft BETWEEN the 2 values. If the node has child nodes, they're zapped too. Dangerous, but effective. I often disallow deletion of nodes if they have child nodes.

Returning Full Hierarchy including depth

This tag returns SQL only, due to the complexities of what we'd want to do with it - it's not appropriate to always do the action in an inline, so this simply returns fully assembled SQL.

```
define_tag('fullCatSQL',
    -Required='cattable',
    -Optional='xtraReturn',
    -Optional='xtraWhere',
    -Optional='depth'
    );
  (!(local_defined('xtraReturn'))) ? local('xtraReturn' = string);
  (!(local_defined('xtraWhere'))) ? local('xtraWhere' = string);
  if(!(local_defined('depth')));
    local('depthComp' = string);
  else(integer(#depth) > 0);
    local('depthComp' = 'HAVING depth <= '+#depth);
  else;
    local('depthComp' = string);
  /if;
  return('
SELECT
  node.id, node.name, (COUNT(parent.name) - 1) AS depth '+#xtraReturn+'
FROM '+#cattable+' AS node,
'+#cattable+' AS parent
WHERE node.lft BETWEEN parent.lft AND parent.rgt '+#xtraWhere+'
GROUP BY node.id
'+#depthComp+'
ORDER BY node.lft');
    /define_tag;
```

This SQL essentially is selecting all those rows in the table that conform to #xtrawhere, which could be as simple as "node.status = 1". The order of node.lft sets us up for the order we've specified when we numbered it all, and the #depthComp helps us restrict the query to a set number of levels below root.

Note the "(COUNT(parent.name) - 1) AS depth". This and the join to the crawling up the tree to count the levels, something I've seen some people doing manually each nested statement (shudder), or caching it which is reasonably sensible if you have no alternative.

The xtraReturn and xtraWhere are used to broaden the usefulness. An example of an xtraReturn is as follows:

```
  (
     SELECT COUNT(*)
    FROM asset, category AS subc
    WHERE asset.category_id = subc.id
    AND subc.lft BETWEEN node.lft AND node.rgt
  )AS chqty,
  (
  SELECT COUNT(asset.name) FROM asset WHERE asset.category_id = node.id
  )AS qty,
  (
  SELECT COUNT(*) - 1
  FROM category AS nnode
  WHERE nnode.lft BETWEEN node.lft AND node.rgt
  )AS nchild
* chqty counts the number of items belonging to that category;
* qty counts number of items belonging to that node only;
* nchild counts the number of child nodes to that category.
```

Once you call this tag you've got all you need to execute an inline and you've got your menu data, category data, or forum list... you get the idea?

Returning just the subtree

OK, selecting a full nested list from the root is useful, but that SQL doesn't give us what we often need: the subtree descending from a given node, with either absolute depth or relative depth.

This tag is slightly more involved. It requires all the normal parameters plus 'relative', which is Boolean true or false.

```
define_tag('subTreeSQL',
      -Required='cattable',
      -Required='id',
      -Optional='depth',
      -Optional='relative',
      -Optional='xtraReturn',
      -Optional='xtraWhere'
      );
  (!(local_defined('xtraReturn'))) ? local('xtraReturn' = string);
  (!(local_defined('xtraWhere'))) ? local('xtraWhere' = string);
  (!(local_defined('relative')) || (local('relative') == false)) ?
    local('relative' = '1') | local('relative' = '(sub_tree.depth + 1)');
  if(!(local_defined('depth')));
    local('depthComp' = string);
  else(integer(#depth) > 0);
    local('depthComp' = 'HAVING depth <= '+#depth);
  else;
    local('depthComp' = string);
  /if;
  //(sub_tree.depth + 1) - makes the depth relative to the one requested
  //HAVING depth <= 1 - limits how many subs it pulls in
  local('out' = 'SELECT node.id, node.name, (COUNT(parent.name) - '+#relative+') AS
depth '+#xtraReturn+'
    FROM '+#cattable+' AS node,
      '+#cattable+' AS parent,
      '+#cattable+' AS sub_parent,
      (
      SELECT node.name, (COUNT(parent.name) - 1) AS depth
      FROM '+#cattable+' AS node,
      '+#cattable+' AS parent
      WHERE node.lft BETWEEN parent.lft AND parent.rgt
      AND node.id = '+#id+'
```

```
      GROUP BY node.name
      ORDER BY node.lft
      )AS sub_tree
    WHERE node.lft BETWEEN parent.lft AND parent.rgt
      AND node.lft BETWEEN sub_parent.lft AND sub_parent.rgt
      AND sub_parent.name = sub_tree.name '+#xtraWhere+'
    GROUP BY node.id
    '+#depthComp+'
    ORDER BY node.lft
    ;');
  return(#out);
/define_tag;
```

The joined 'parent' is the same as the full tree retrieval - it gets the bubble-up, but the 'sub_parent' join and 'sub_tree' subquery join set us up to get the nested levels. It might seem overly complex but consider the alternative: if you try to restrict the "full tree" query to a given id as a parent node, then it won't even return the children, it will only return that node. If we try to restrict it using the lft and rgt values of that node (which is what we actually are doing here anyway), it doesn't accurately retrieve the subtree.

Moving a node

Here's where I regretted ever getting into this, but by now I was hooked. Moving a node was simply not documented properly anywhere online. It was described, but no one could really succinctly show it. The "step by step" instructions proved that some of these guys found it confusing as well. I admit I made copious notes trying to work the theory out on paper, and still got it wrong - it took trial and error in a couple of places before I understood what I was actually doing!

```
define_tag('moveNode',
    -Required='cattable',
    -Required='id'
    );
  local('id2' = 0);
// (A)
  // get immediate prior sibling
      local('sSQL' = (
    'SELECT node.id, node.name,
     (COUNT(parent.name) - (sub_tree.depth + 1)) AS depth,node.lft,node.rgt
    FROM '+#cattable+' AS node,
    '+#cattable+' AS parent,
    '+#cattable+' AS sub_parent,
    (SELECT node.name, (COUNT(parent.name) - 1) AS depth
    FROM '+#cattable+' AS node,
    '+#cattable+' AS parent
    WHERE node.lft BETWEEN parent.lft AND parent.rgt
    AND node.id =
        (SELECT parent.id
          FROM '+#cattable+' AS node,
          '+#cattable+' AS parent
          WHERE node.lft BETWEEN parent.lft AND parent.rgt
          AND node.id = '+#id+' AND parent.id != '+#id+'
          ORDER BY parent.lft DESC LIMIT 1)
    GROUP BY node.name
    ORDER BY node.lft
    )AS sub_tree
    WHERE node.lft BETWEEN parent.lft AND parent.rgt
    AND node.lft BETWEEN sub_parent.lft AND sub_parent.rgt
    AND sub_parent.name = sub_tree.name
    GROUP BY node.id
    HAVING depth = 1
    ORDER BY node.lft ASC;'
```

```
      ));
      inline($gv_sql,-SQL=#sSQL);
        records;
          if(integer(field('id')) != #id);
            #id2 = integer(field('id'));
          else;
            loop_abort;
          /if;
        /records;
      /inline;

      if(#id2 == 0);
// (B)
        // here we are trying to ascertain if it's a root node!
        #sSQL = '
        SELECT node.id, node.name, (COUNT(parent.name) - 1) AS depth
        FROM '+#cattable+' AS node,
        '+#cattable+' AS parent
        WHERE node.lft BETWEEN parent.lft AND parent.rgt
        AND node.id = '+#id+'
        GROUP BY node.id
        ORDER BY node.lft';
        inline($gv_sql,-SQL=#sSQL);
          records;
            if(integer(field('depth')) == 0);
              // yay, it's a root node!!!
              #sSQL = '
                SELECT node.id, node.name,
                 (COUNT(parent.name)-1) AS depth
                FROM '+#cattable+' AS node,
                '+#cattable+' AS parent
                WHERE node.lft BETWEEN
                  parent.lft AND parent.rgt
                GROUP BY node.id
                HAVING depth = 0
                ORDER BY node.lft
              ';
              inline($gv_sql,-SQL=#sSQL);
                records;
                if(integer(field('id')) != #id);
                  #id2 = integer(field('id'));
                else;
                  loop_abort;
                /if;
                /records;
              /inline;
            /if;
          /records;
        /inline;
      /if;
      if(#id2 > 0);
      #sSQL = ('
LOCK TABLE '+#cattable+' WRITE;
-- 1
SELECT @myLeft := lft, @myRight := rgt, @myWidth := rgt - lft + 1 FROM '+#cattable+'
WHERE id = '+#id2+';
-- 2
UPDATE '+#cattable+' SET rgt = (rgt*-1), lft = (lft*-1) WHERE lft BETWEEN @myLeft AND
@myRight;
-- 3
UPDATE '+#cattable+' SET rgt = rgt - @myWidth WHERE rgt > @myRight;
UPDATE '+#cattable+' SET lft = lft - @myWidth WHERE lft > @myRight;
-- 4
SELECT @myLeft2 := lft, @myRight2 := rgt, @myWidth2 := rgt - lft + 1 FROM
'+#cattable+' WHERE id = '+#id+';
-- 5
UPDATE '+#cattable+' SET rgt = rgt + @myWidth WHERE rgt > @myRight2;
UPDATE '+#cattable+' SET lft = lft + @myWidth WHERE lft > @myRight2;
```

```
-- 6
SELECT @x := lft FROM '+#cattable+' WHERE id = '+#id+';
SELECT @y := rgt FROM '+#cattable+' WHERE id = '+#id+';
-- 8
UPDATE '+#cattable+' SET rgt = (rgt - (@y - @x + 1)) WHERE rgt < 0;
UPDATE '+#cattable+' SET lft = (lft - (@y - @x + 1)) WHERE lft < 0;
-- 9
UPDATE '+#cattable+' SET rgt = rgt * -1 WHERE rgt < 0;
UPDATE '+#cattable+' SET lft = lft * -1 WHERE lft < 0;
UNLOCK TABLES;
      ');
      inline($gv_sql,-SQL=#sSQL);
      /inline;
    /if;
  /define_tag;
USAGE:
xs_cat->(moveNode(-cattable='category',-id=#id));
```

The in-SQL comments are kept there for my own reference as to the various logical steps that they represent.

In the placeholder comments, (A) and (B) are in the lasso, --1 and so on are SQL comments.

A few notes about the process:

- This particular tag is meant fro moving a node UPWARDS in the ORDER, not changing level, although in this code can be adapted to move a node and all it's children from one part of the tree to any other part of the tree with a little effort.
- (A) - In order to move a node up the order you need to know what it's immediate previous sibling is, proved to be complex to solve in SQL, so I've captured all child nodes with depth 1 of the parent of the node we're moving (ie, siblings!). In Lasso we then set the #id2 to the column id until we hit the moving node and we abort the loop.
- (B) Special case is if the node we're moving is at depth 0, therefore no parents, so the (A) code blows apart and #id2 will remain at zero.

  1: We retrieve the lft, rgt and width of the node we're exchanging "places" with.

  2: Make the entire node+children we're exchanging with, negative, as a placeholder.

  3: Collapse the rest of the tree to assume the removed node's place. Note the similarity with the delete node code.

  4: Retrieve lft, rgt, width from the node we desire to move "up"

  5: Make space for the exchanged node (that which we negative-valued in step 2) by shifting everything to the right of our moving node outwards.

  6: Reselecting the lft and rgt of the moving node, partly out of consistency with other tags that do similar things.

  7: yes you are right, there's no 7 out of respect to those very same similar tags

  8 & 9: We take every lft and rgt value that is less than zero and make them positive with the newly calculated lft and rgt that fit into the gap we created. I had some issues with doing this as one step in the SQL so accepted a minute performance hit with doing this as 2 sets of statements.

It looks hideous, but apart from ascertaining the immediate prior sibling's id, it's pure SQL.

Summary

I've played with a lot of data models over the years, and have looked at (and worked with) the way some large content management systems - you know the type, that cost hundreds of thousands to license and implement, manage their categorized data. I believe we can learn a lot from they way these guys have missed the mark in certain areas.

Where we need to be prepared to jump through hoops is on the back end - we can afford a speed hit there, not on the front-end. Surprisingly, I found the "big solution providers" were constantly seeming to convince their clients it was easy to maintain, and yet in reality the front-end code was often clumsy and slow.

The easier we can categorize our data and make it cohesive, the better we're setting ourselves and our clients up to win. When the client wins, they will recommend us to others and so we win, as we will have work walking in our doors.

I believe the more we talk to each other about how we're dealing with our data, hierarchical content being just one piece in the spectrum, the more we will be able to move forward as a community.

I hope this exposé of the way I've begun dealing with this area will open doors for many of you and promote active discussion. One thing's for sure, we live in a constantly evolving environment - by the time my son is earning his keep by coding, these techniques will probably be well outdated. Collaborative development and discussion will keep us all ahead of the curve.

References

Managing Hierarchical Data in MySQL by Mike Hillyer http://dev.mysql.com/tech-resources/articles/hierarchical-data.html

MySQL 4.1 User Variables, http://dev.mysql.com/doc/refman/4.1/en/variables.html

More links at http://xserve.co.nz/hd_links.lasso

Author: Jonathan Guthrie

xServe Limited, Wellington NZ. jono@xserve.co.nz

For Lasso Summit, Ft Lauderdale, Miami, February 2006