

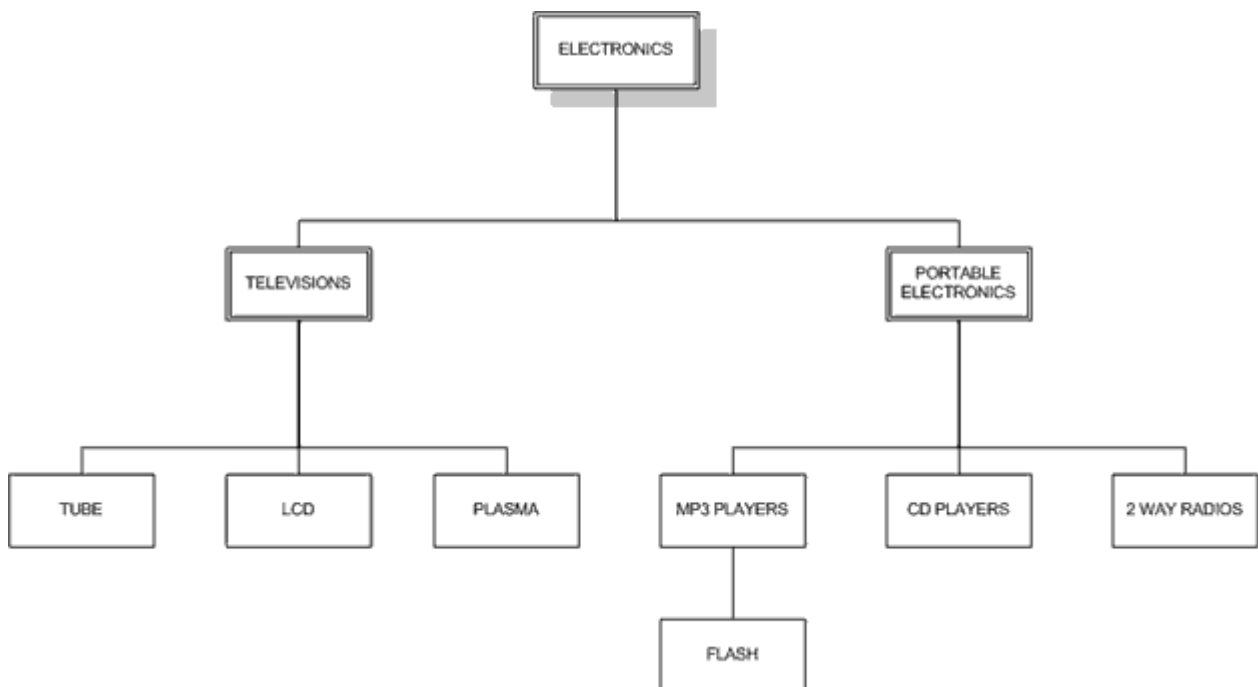
# MYSQL 中分层数据的管理

By Mike Hillyer

## 引言

大多数用户都曾在数据库中处理过分层数据(hierarchical data)，认为分层数据的管理不是关系数据库的目的。之所以这么认为，是因为关系数据库中的表没有层次关系，只是简单的平面化的列表；而分层数据具有父—子关系，显然关系数据库中的表不能自然地表现出其分层的特性。

我们认为，分层数据是每项只有一个父项和零个或多个子项（根项除外，根项没有父项）的数据集合。分层数据存在于许多基于数据库的应用程序中，包括论坛和邮件列表中的分类、商业组织图表、内容管理系统的分类、产品分类。我们打算使用下面一个虚构的电子商店的产品分类：



这些分类层次与上面提到的一些例子中的分类层次是相类似的。在本文中我们将从传统的邻接表(adjacency list)模型出发，阐述 2 种在 MySQL 中处理分层数据的模型。

## 邻接表模型

上述例子的分类数据将被存储在下面的数据表中（我给出了全部的数据表创建、数据插入的

代码，你可以跟着做）：

```
CREATE TABLE category(
category_id INT AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(20) NOT NULL,
parent INT DEFAULT NULL);

INSERT INTO category
VALUES (1, 'ELECTRONICS', NULL), (2, 'TELEVISIONS', 1), (3, 'TUBE', 2),
(4, 'LCD', 2), (5, 'PLASMA', 2), (6, 'PORTABLE ELECTRONICS', 1),
(7, 'MP3 PLAYERS', 6), (8, 'FLASH', 7),
(9, 'CD PLAYERS', 6), (10, '2 WAY RADIOS', 6);

SELECT * FROM category ORDER BY category_id;
```

category_id	name	parent
1	ELECTRONICS	NULL
2	TELEVISIONS	1
3	TUBE	2
4	LCD	2
5	PLASMA	2
6	PORTABLE ELECTRONICS	1
7	MP3 PLAYERS	6
8	FLASH	7
9	CD PLAYERS	6
10	2 WAY RADIOS	6

10 rows in set (0.00 sec)

在邻接表模型中，数据表中的每项包含了指向其父项的指示器。在此例中，最上层项的父项为空值(NULL)。邻接表模型的优势在于它很简单，可以很容易地看出FLASH是MP3 PLAYERS的子项，哪个是portable electronics的子项，哪个是electronics的子项。虽然，在客户端编码中邻接表模型处理起来也相当的简单，但是如果是纯SQL编码的话，该模型会有很多问题。

## 检索整树

通常在处理分层数据时首要的任务是，以某种缩进形式来呈现一棵完整的树。为此，在纯SQL编码中通常的做法是使用自连接(self-join)：

```
SELECT t1.name AS lev1, t2.name as lev2, t3.name as lev3, t4.name as lev4
FROM category AS t1
LEFT JOIN category AS t2 ON t2.parent = t1.category_id
LEFT JOIN category AS t3 ON t3.parent = t2.category_id
LEFT JOIN category AS t4 ON t4.parent = t3.category_id
WHERE t1.name = 'ELECTRONICS';
```

lev1	lev2	lev3	lev4
ELECTRONICS	TELEVISIONS	TUBE	NULL
ELECTRONICS	TELEVISIONS	LCD	NULL
ELECTRONICS	TELEVISIONS	PLASMA	NULL

```

| ELECTRONICS | PORTABLE ELECTRONICS | MP3 PLAYERS | FLASH |
| ELECTRONICS | PORTABLE ELECTRONICS | CD PLAYERS  | NULL  |
| ELECTRONICS | PORTABLE ELECTRONICS | 2 WAY RADIOS | NULL  |
+-----+-----+-----+-----+
6 rows in set (0.00 sec)

```

## 检索所有叶子节点

我们可以用左连接 (LEFT JOIN) 来检索出树中所有叶子节点 (没有孩子节点的节点) :

```

SELECT t1.name FROM
category AS t1 LEFT JOIN category as t2
ON t1.category_id = t2.parent
WHERE t2.category_id IS NULL;

```

```

+-----+
| name |
+-----+
| TUBE |
| LCD  |
| PLASMA |
| FLASH |
| CD PLAYERS |
| 2 WAY RADIOS |
+-----+

```

## 检索单一路径

通过自连接, 我们也可以检索出单一路径:

```

SELECT t1.name AS lev1, t2.name as lev2, t3.name as lev3, t4.name as lev4
FROM category AS t1
LEFT JOIN category AS t2 ON t2.parent = t1.category_id
LEFT JOIN category AS t3 ON t3.parent = t2.category_id
LEFT JOIN category AS t4 ON t4.parent = t3.category_id
WHERE t1.name = 'ELECTRONICS' AND t4.name = 'FLASH';

```

```

+-----+-----+-----+-----+
| lev1 | lev2 | lev3 | lev4 |
+-----+-----+-----+-----+
| ELECTRONICS | PORTABLE ELECTRONICS | MP3 PLAYERS | FLASH |
+-----+-----+-----+-----+
1 row in set (0.01 sec)

```

这种方法的主要局限是你需要为每层数据添加一个自连接, 随着层次的增加, 自连接变得越来越复杂, 检索的性能自然而然的也就下降了。

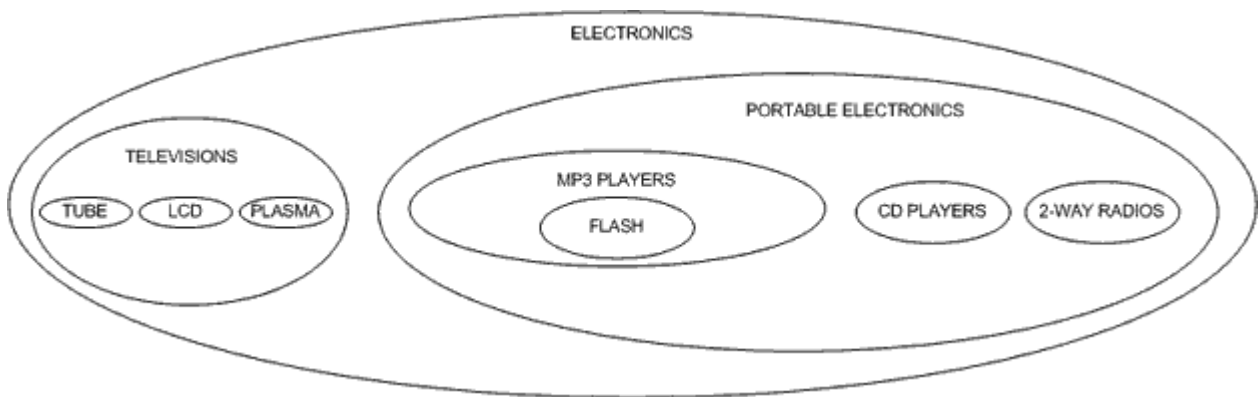
## 邻接表模型的局限性

用纯 SQL 编码实现邻接表模型有一定的难度。在我们检索某分类的路径之前, 我们需要知道

该分类所在的层次。另外，我们在删除节点的时候要特别小心，因为潜在的可能会孤立一棵子树（当删除 portable electronics 分类时，所有他的子分类都成了孤儿）。部分局限性可以通过使用客户端代码或者存储过程来解决，我们可以从树的底部开始向上迭代来获得一颗树或者单一路径，我们也可以删除节点的时候使其子节点指向一个新的父节点，来防止孤立子树的产生。

## 嵌套集合(Nested Set)模型

我想在这篇文章中重点阐述一种不同的方法，俗称为**嵌套集合模型**。在嵌套集合模型中，我们将以一种新的方式来看待我们的分层数据，不再是线与点了，而是嵌套容器。我试着以嵌套容器的方式画出了 electronics 分类图：



从上图可以看出我们依旧保持了数据的层次，父分类包围了其子分类。在数据表中，我们通过使用表示节点的嵌套关系的左值(left value)和右值(right value)来表现嵌套集合模型中数据的分层特性：

```
CREATE TABLE nested_category (
  category_id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(20) NOT NULL,
  lft INT NOT NULL,
  rgt INT NOT NULL
);

INSERT INTO nested_category
VALUES(1, 'ELECTRONICS', 1, 20), (2, 'TELEVISIONS', 2, 9), (3, 'TUBE', 3, 4),
(4, 'LCD', 5, 6), (5, 'PLASMA', 7, 8), (6, 'PORTABLE ELECTRONICS', 10, 19),
(7, 'MP3 PLAYERS', 11, 14), (8, 'FLASH', 12, 13),
(9, 'CD PLAYERS', 15, 16), (10, '2 WAY RADIOS', 17, 18);

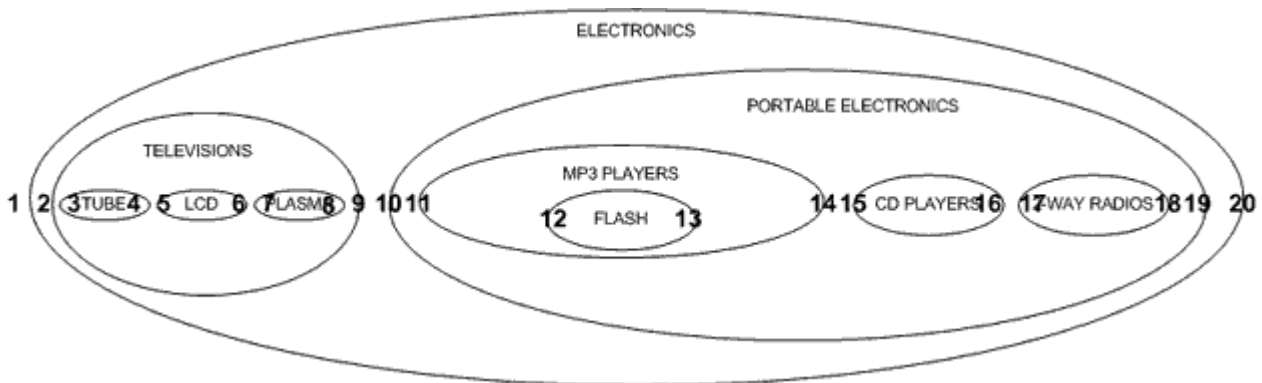
SELECT * FROM nested_category ORDER BY category_id;
```

category_id	name	lft	rgt
1	ELECTRONICS	1	20
2	TELEVISIONS	2	9
3	TUBE	3	4
4	LCD	5	6
5	PLASMA	7	8
6	PORTABLE ELECTRONICS	10	19

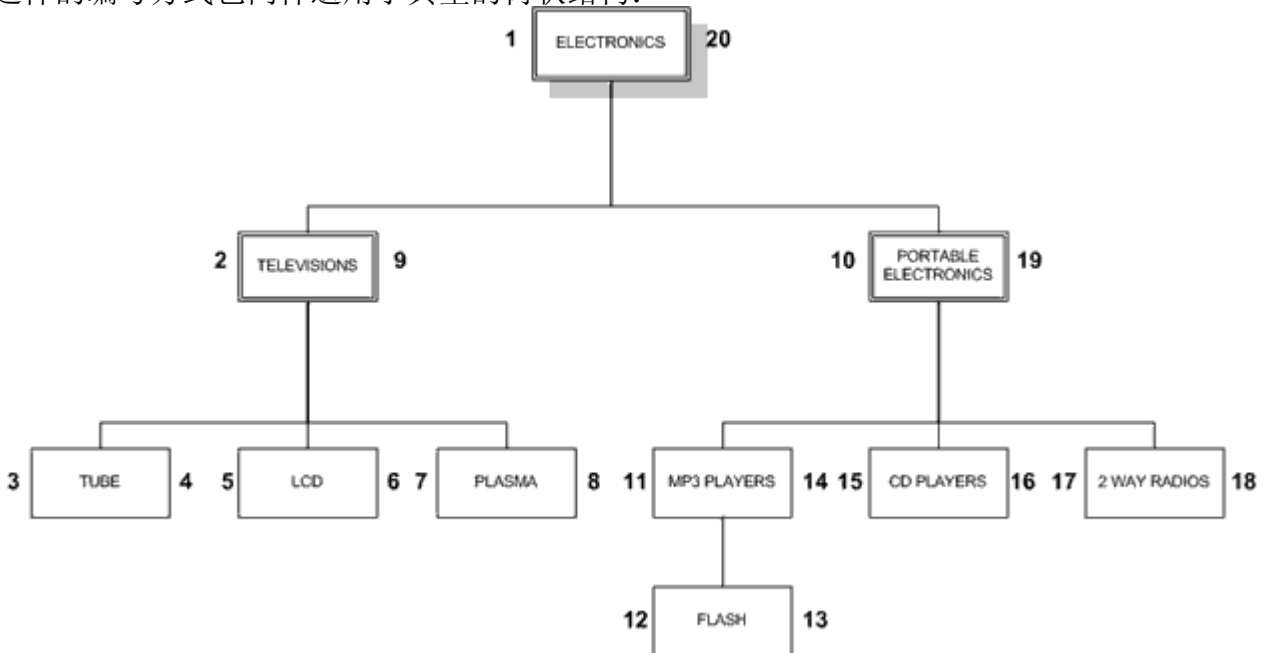
7	MP3 PLAYERS	11	14
8	FLASH	12	13
9	CD PLAYERS	15	16
10	2 WAY RADIOS	17	18

我们使用了 lft 和 rgt 来代替 left 和 right，是因为在 MySQL 中 left 和 right 是保留字。  
<http://dev.mysql.com/doc/mysql/en/reserved-words.html>，有一份详细的 MySQL 保留字清单。

那么，我们怎样决定左值和右值呢？我们从外层节点的最左侧开始，从左到右编号：



这样的编号方式也同样适用于典型的树状结构：



当我们为树状的结构编号时，我们是从左到右，一次一层，为节点赋右值前先从左到右遍历其子节点给其子节点赋左右值。这种方法被称作改进的**先序遍历算法**。

## 检索整树

我们可以通过自连接把父节点连接到子节点上来检索整树，是因为子节点的 `lft` 值总是在其父节点的 `lft` 值和 `rgt` 值之间：

```
SELECT node.name
FROM nested_category AS node,
nested_category AS parent
WHERE node.lft BETWEEN parent.lft AND parent.rgt
AND parent.name = 'ELECTRONICS'
ORDER BY node.lft;
```

```
+-----+
| name  |
+-----+
| ELECTRONICS |
| TELEVISIONS |
| TUBE      |
| LCD       |
| PLASMA    |
| PORTABLE ELECTRONICS |
| MP3 PLAYERS |
| FLASH     |
| CD PLAYERS |
| 2 WAY RADIOS |
+-----+
```

不像先前邻接表模型的例子，这个查询语句不管树的层次有多深都能很好的工作。在 `BETWEEN` 的子句中我们没有去关心 `node` 的 `rgt` 值，是因为使用 `node` 的 `rgt` 值得出的父节点总是和使用 `lft` 值得出的是相同的。

## 检索所有叶子节点

检索出所有的叶子节点，使用嵌套集合模型的方法比邻接表模型的 `LEFT JOIN` 方法简单多了。如果你仔细得看了 `nested_category` 表，你可能已经注意到叶子节点的左右值是连续的。要检索出叶子节点，我们只要查找满足 `rgt=lft+1` 的节点：

```
SELECT name
FROM nested_category
WHERE rgt = lft + 1;
```

```
+-----+
| name  |
+-----+
| TUBE   |
| LCD    |
| PLASMA |
| FLASH  |
| CD PLAYERS |
| 2 WAY RADIOS |
+-----+
```

## 检索单一路径

在嵌套集合模型中，我们可以不用多个自连接就可以检索出单一路径：

```
SELECT parent.name
FROM nested_category AS node,
nested_category AS parent
WHERE node.lft BETWEEN parent.lft AND parent.rgt
AND node.name = 'FLASH'
ORDER BY node.lft;
```

```
+-----+
| name |
+-----+
| ELECTRONICS |
| PORTABLE ELECTRONICS |
| MP3 PLAYERS |
| FLASH |
+-----+
```

## 检索节点的深度

我们已经知道怎样去呈现一棵整树，但是为了更好的标识出节点在树中所处层次，我们怎样才能检索出节点在树中的深度呢？我们可以在先前的查询语句上增加 COUNT 函数和 GROUP BY 子句来实现：

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

```
+-----+-----+
| name | depth |
+-----+-----+
| ELECTRONICS | 0 |
| TELEVISIONS | 1 |
| TUBE | 2 |
| LCD | 2 |
| PLASMA | 2 |
| PORTABLE ELECTRONICS | 1 |
| MP3 PLAYERS | 2 |
| FLASH | 3 |
| CD PLAYERS | 2 |
| 2 WAY RADIOS | 2 |
+-----+-----+
```

我们可以根据 depth 值来缩进分类名字，使用 CONCAT 和 REPEAT 字符串函数<sup>1</sup>：

```
SELECT CONCAT( REPEAT(' ', COUNT(parent.name) - 1), node.name) AS name
FROM nested_category AS node,
nested_category AS parent
```

1 [译注] 缩进在 phpMyAdmin 下显示会有出入，建议在 MySQL 命令行下运行查询语句。

```
WHERE node.lft BETWEEN parent.lft AND parent.rgt
GROUP BY node.name
ORDER BY node.lft;
```

```
+-----+
| name |
+-----+
| ELECTRONICS |
| TELEVISIONS |
| TUBE |
| LCD |
| PLASMA |
| PORTABLE ELECTRONICS |
| MP3 PLAYERS |
| FLASH |
| CD PLAYERS |
| 2 WAY RADIOS |
+-----+
```

当然，在客户端应用程序中你可能会用 depth 值来直接展示数据的层次。Web 开发者会遍历该树，随着 depth 值的增加和减少来添加<li></li>和<ul></ul>标签。

### 检索子树的深度

当我们需要子树的深度信息时，我们不能限制自连接中的 node 或 parent，因为这么做会打乱数据集的顺序。因此，我们添加了第三个自连接作为子查询，来得出子树新起点的深度值：

```
SELECT node.name, (COUNT(parent.name) - (sub_tree.depth + 1)) AS depth
FROM nested_category AS node,
     nested_category AS parent,
     nested_category AS sub_parent,
     (
         SELECT node.name, (COUNT(parent.name) - 1) AS depth
         FROM nested_category AS node,
              nested_category AS parent
         WHERE node.lft BETWEEN parent.lft AND parent.rgt
               AND node.name = 'PORTABLE ELECTRONICS'
         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.name
ORDER BY node.lft;
```

```
+-----+-----+
| name | depth |
+-----+-----+
| PORTABLE ELECTRONICS | 0 |
| MP3 PLAYERS | 1 |
| FLASH | 2 |
| CD PLAYERS | 1 |
| 2 WAY RADIOS | 1 |
+-----+-----+
```

这个查询语句可以检索出任一节点子树的深度值，包括根节点。这里的深度值跟你指定的节点有关。

## 检索节点的直接子节点

可以想象一下，你在零售网站上呈现电子产品的分类。当用户点击分类后，你将要呈现该分类下的产品，同时也需列出该分类下的直接子分类，而不是该分类下的全部分类。为此，我们只呈现该节点及其直接子节点，不再呈现更深层次的节点。例如，当呈现 PORTABLE ELECTRONICS 分类时，我们同时只呈现 MP3 PLAYERS、CD PLAYERS 和 2 WAY RADIOS 分类，而不呈现 FLASH 分类。

要实现它非常的简单，在先前的查询语句上添加 HAVING 子句：

```
SELECT node.name, (COUNT(parent.name) - (sub_tree.depth + 1)) AS depth
FROM nested_category AS node,
     nested_category AS parent,
     nested_category AS sub_parent,
     (
         SELECT node.name, (COUNT(parent.name) - 1) AS depth
         FROM nested_category AS node,
              nested_category AS parent
         WHERE node.lft BETWEEN parent.lft AND parent.rgt
              AND node.name = 'PORTABLE ELECTRONICS'
         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.name
HAVING depth <= 1
ORDER BY node.lft;
```

```
+-----+-----+
| name           | depth |
+-----+-----+
| PORTABLE ELECTRONICS |    0 |
| MP3 PLAYERS      |    1 |
| CD PLAYERS       |    1 |
| 2 WAY RADIOS     |    1 |
+-----+-----+
```

如果你不希望呈现父节点，你可以更改 HAVING depth <= 1 为 HAVING depth = 1。

## 嵌套集合模型中集合函数的应用

让我们添加一个产品表，我们可以使用它来示例集合函数的应用：

```
CREATE TABLE product(
product_id INT AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(40),
category_id INT NOT NULL
```

```
);
```

```
INSERT INTO product(name, category_id) VALUES('20" TV',3),('36" TV',3),
('Super-LCD 42"',4),('Ultra-Plasma 62"',5),('Value Plasma 38"',5),
('Power-MP3 5gb',7),('Super-Player 1gb',8),('Porta CD',9),('CD To go!',9),
('Family Talk 360',10);
```

```
SELECT * FROM product;
```

product_id	name	category_id
1	20" TV	3
2	36" TV	3
3	Super-LCD 42"	4
4	Ultra-Plasma 62"	5
5	Value Plasma 38"	5
6	Power-MP3 128mb	7
7	Super-Shuffle 1gb	8
8	Porta CD	9
9	CD To go!	9
10	Family Talk 360	10

现在，让我们写一个查询语句，在检索分类树的同时，计算出各分类下的产品数量：

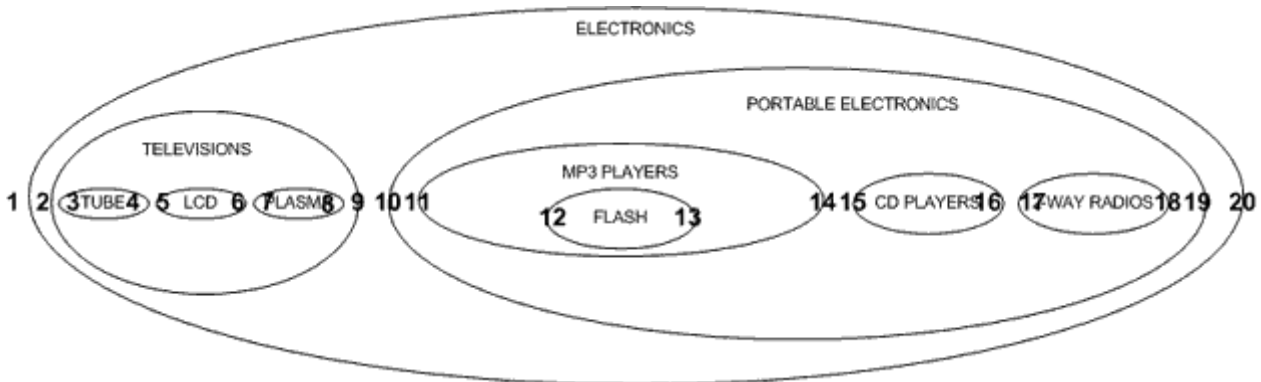
```
SELECT parent.name, COUNT(product.name)
FROM nested_category AS node ,
nested_category AS parent,
product
WHERE node.lft BETWEEN parent.lft AND parent.rgt
AND node.category_id = product.category_id
GROUP BY parent.name
ORDER BY node.lft;
```

name	COUNT(product.name)
ELECTRONICS	10
TELEVISIONS	5
TUBE	2
LCD	1
PLASMA	2
PORTABLE ELECTRONICS	5
MP3 PLAYERS	2
FLASH	1
CD PLAYERS	2
2 WAY RADIOS	1

这条查询语句在检索整树的查询语句上增加了 COUNT 和 GROUP BY 子句，同时在 WHERE 子句中引用了 product 表和一个自连接。

## 新增节点

到现在，我们已经知道了如何去查询我们的树，是时候去关注一下如何增加一个新节点来更新我们的树了。让我们再一次观察一下我们的嵌套集合图：



当我们想要在 TELEVISIONS 和 PORTABLE ELECTRONICS 节点之间新增一个节点，新节点的 lft 和 rgt 的 值为 10 和 11，所有该节点的右边节点的 lft 和 rgt 值都将加 2，之后我们再添加新节点并赋相应的 lft 和 rgt 值。在 MySQL 5 中可以使用存储过程来完成，我假设当前大部分读者使用的是 MySQL 4.1 版本，因为这是最新的稳定版本。所以，我使用了锁表（LOCK TABLES）语句来隔离查询：

```
LOCK TABLE nested_category WRITE;

SELECT @myRight := rgt FROM nested_category
WHERE name = 'TELEVISIONS';

UPDATE nested_category SET rgt = rgt + 2 WHERE rgt > @myRight;
UPDATE nested_category SET lft = lft + 2 WHERE lft > @myRight;

INSERT INTO nested_category(name, lft, rgt) VALUES('GAME CONSOLES', @myRight + 1,
@myRight + 2);

UNLOCK TABLES;
```

我们可以检验一下新节点插入的正确性：

```
SELECT CONCAT( REPEAT( ' ', (COUNT(parent.name) - 1) ), node.name) AS name
FROM nested_category AS node,
nested_category AS parent
WHERE node.lft BETWEEN parent.lft AND parent.rgt
GROUP BY node.name
ORDER BY node.lft;
```

```
+-----+
| name  |
+-----+
| ELECTRONICS |
| TELEVISIONS |
| TUBE     |
| LCD     |
| PLASMA  |
| GAME CONSOLES |
```

```

| PORTABLE ELECTRONICS |
| MP3 PLAYERS         |
| FLASH               |
| CD PLAYERS          |
| 2 WAY RADIOS        |
+-----+

```

如果我们想要在叶子节点下增加节点，我们得稍微修改一下查询语句。让我们在 2 WAY RADIOS 叶子节点下添加 FRS 节点吧：

```

LOCK TABLE nested_category WRITE;

SELECT @myLeft := lft FROM nested_category
WHERE name = '2 WAY RADIOS';

UPDATE nested_category SET rgt = rgt + 2 WHERE rgt > @myLeft;
UPDATE nested_category SET lft = lft + 2 WHERE lft > @myLeft;

INSERT INTO nested_category(name, lft, rgt) VALUES('FRS', @myLeft + 1, @myLeft +
2);

UNLOCK TABLES;

```

在这个例子中，我们扩大了新产生的父节点(2 WAY RADIOS 节点)的右值及其所有它的右边节点的左右值，之后置新增节点于新父节点之下。正如你所看到的，我们新增的节点已经完全融入了嵌套集合中：

```

SELECT CONCAT( REPEAT( ' ', (COUNT(parent.name) - 1) ), node.name) AS name
FROM nested_category AS node,
nested_category AS parent
WHERE node.lft BETWEEN parent.lft AND parent.rgt
GROUP BY node.name
ORDER BY node.lft;

```

```

+-----+
| name                |
+-----+
| ELECTRONICS         |
| TELEVISIONS        |
| TUBE                |
| LCD                 |
| PLASMA              |
| GAME CONSOLES       |
| PORTABLE ELECTRONICS |
| MP3 PLAYERS         |
| FLASH               |
| CD PLAYERS          |
| 2 WAY RADIOS        |
| FRS                 |
+-----+

```

## 删除节点

最后还有个基础任务，删除节点。删除节点的处理过程跟节点在分层数据中所处的位置有关，删除一个叶子节点比删除一个子节点要简单得多，因为删除子节点的时候，我们需要去处理孤立节点。

删除一个叶子节点的过程正好是新增一个叶子节点的逆过程，我们在删除节点的同时该节点右边所有节点的左右值和该父节点的右值都会减去该节点的宽度值<sup>2</sup>：

```
LOCK TABLE nested_category WRITE;

SELECT @myLeft := lft, @myRight := rgt, @myWidth := rgt - lft + 1
FROM nested_category
WHERE name = 'GAME CONSOLES';

DELETE FROM nested_category WHERE lft BETWEEN @myLeft AND @myRight;

UPDATE nested_category SET rgt = rgt - @myWidth WHERE rgt > @myRight;
UPDATE nested_category SET lft = lft - @myWidth WHERE lft > @myRight;

UNLOCK TABLES;
```

我们再一次检验一下节点已经成功删除, 而且没有打乱数据的层次:

```
SELECT CONCAT( REPEAT( ' ', (COUNT(parent.name) - 1) ), node.name) AS name
FROM nested_category AS node,
nested_category AS parent
WHERE node.lft BETWEEN parent.lft AND parent.rgt
GROUP BY node.name
ORDER BY node.lft;
```

```
+-----+
| name          |
+-----+
| ELECTRONICS   |
| TELEVISIONS  |
| TUBE          |
| LCD           |
| PLASMA        |
| PORTABLE ELECTRONICS |
| MP3 PLAYERS   |
| FLASH         |
| CD PLAYERS    |
| 2 WAY RADIOS  |
| FRS           |
+-----+
```

这个方法可以完美地删除节点及其子节点:

```
LOCK TABLE nested_category WRITE;

SELECT @myLeft := lft, @myRight := rgt, @myWidth := rgt - lft + 1
FROM nested_category
```

<sup>2</sup> [译注] 作者的本意是举删除叶子节点的情况，不过他给出的查询语句是通用型的，删除节点及其子节点，跟下例的查询语句是相同的。

```

WHERE name = 'MP3 PLAYERS';

DELETE FROM nested_category WHERE lft BETWEEN @myLeft AND @myRight;

UPDATE nested_category SET rgt = rgt - @myWidth WHERE rgt > @myRight;
UPDATE nested_category SET lft = lft - @myWidth WHERE lft > @myRight;

UNLOCK TABLES;

```

再次验证我们已经成功的删除了一棵子树：

```

SELECT CONCAT( REPEAT( ' ', (COUNT(parent.name) - 1) ), node.name) AS name
FROM nested_category AS node,
nested_category AS parent
WHERE node.lft BETWEEN parent.lft AND parent.rgt
GROUP BY node.name
ORDER BY node.lft;

```

```

+-----+
| name          |
+-----+
| ELECTRONICS  |
| TELEVISIONS  |
| TUBE         |
| LCD         |
| PLASMA      |
| PORTABLE ELECTRONICS |
| CD PLAYERS   |
| 2 WAY RADIOS |
| FRS         |
+-----+

```

有时，我们只删除该节点，而不删除该节点的子节点。在一些情况下，你希望改变其名字为占位符，直到替代名字的出现，比如你开除了一个主管（需要更换主管）。在另外一些情况下，你希望子节点挂到该删除节点的父节点下：

```

LOCK TABLE nested_category WRITE;

SELECT @myLeft := lft, @myRight := rgt, @myWidth := rgt - lft + 1
FROM nested_category
WHERE name = 'PORTABLE ELECTRONICS';

DELETE FROM nested_category WHERE lft = @myLeft;

UPDATE nested_category SET rgt = rgt - 1, lft = lft - 1 WHERE lft BETWEEN @myLeft
AND @myRight;
UPDATE nested_category SET rgt = rgt - 2 WHERE rgt > @myRight;
UPDATE nested_category SET lft = lft - 2 WHERE lft > @myRight;

UNLOCK TABLES;

```

在这个例子中，我们对该节点所有右边节点的左右值都减去了2（因为不考虑其子节点，该节点的宽度为2），对该节点的子节点的左右值都减去了1（弥补由于失去父节点的左值造成的裂缝）。我们再一次确认，那些节点是否都晋升了：

```

SELECT CONCAT( REPEAT( ' ', (COUNT(parent.name) - 1) ), node.name) AS name

```

```
FROM nested_category AS node,
nested_category AS parent
WHERE node.lft BETWEEN parent.lft AND parent.rgt
GROUP BY node.name
ORDER BY node.lft;
```

```
+-----+
| name   |
+-----+
| ELECTRONICS |
| TELEVISIONS |
| TUBE       |
| LCD       |
| PLASMA    |
| CD PLAYERS |
| 2 WAY RADIOS |
| FRS       |
+-----+
```

有时，当删除节点的时候，把该节点的一个子节点挂载到该节点的父节点下，而其他节点挂到该节点父节点的兄弟节点下，考虑到篇幅这种情况不在此处解说了。

## 最后的思考

我希望这篇文章对你有所帮助，SQL 中的嵌套集合的观念大约有十年的历史了，在网上和一些书中都能找到许多相关信息。在我看来，讲述分层数据的管理最全面的，是来自一本名叫《[Joe Celko's Trees and Hierarchies in SQL for Smarties](#)》<sup>3</sup>的书，此书的作者是在高级 SQL 领域倍受尊敬的 Joe Celko。Joe Celko 被认为是嵌套集合模型的创造者，更是该领域的多产作家。我把 Celko 的书当作无价之宝，并极力地推荐它。在这本书中涵盖了在此文中没有提及的一些高级话题，也提到了其他一些关于邻接表和嵌套集合模型下管理分层数据的方法。

在随后的参考书目章节中<sup>4</sup>，我列出了一些网络资源，也许对你研究分层数据的管理会有所帮助，其中包括一些 PHP 相关的资源（处理嵌套集合的 PHP 库）。如果你还在使用邻接表模型，你该去试试嵌套集合模型了，在 [Storing Hierarchical Data in a Database](#)<sup>5</sup> 文中下方列出的一些资源链接中能找到一些样例代码，可以去试验一下。

<sup>3</sup> <http://www.openwin.org/mike/books/index.php/trees-and-hierarchies-in-sql>

<sup>4</sup> [译注] 原文中并没有这个章节，作者忘了？

<sup>5</sup> <http://www.sitepoint.com/article/hierarchical-data-database>

## 关于作者/译者



Mike Hillyer, 本文的作者, MySQL Ab 的技术作家, 生活在加拿大的阿尔伯达省<sup>6</sup>。



Yimin, 本文的译者, 就读于浙江理工大学计算机系。  
我的 Blog: <http://liyimin.net/blog>

---

6 [译注]加拿大西部一省份, 位于不列颠哥伦比亚省和萨斯喀彻温省之间。于 1905 年加入联邦。在六十年代初发现石油和天然气以前, 小麦种植和养牛业为该省经济的主要支柱。埃德蒙顿为该省首府及最大的城市。人口 2, 237, 724