将平面表解析为树的最有效/最优雅的方法是什么？

问 13 年前　活跃 2 个月前　浏览 125,000 次

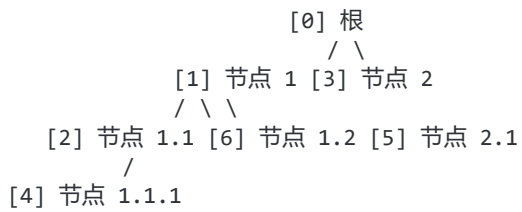
548

474

假设您有一个存储有序树层次结构的平面表：

```
Id   Name         ParentId   Order
 1   'Node 1'            0      10
 2   'Node 1.1'          1      10
 3   'Node 2'            0      20
 4   'Node 1.1.1'        2      10
 5   'Node 2.1'          3      10
 6   'Node 1.2'          1      20
```

这是一个图表，我们有 `[id] Name` . 根节点 0 是虚构的。



您将使用什么简约方法将其作为正确排序、正确缩进的树输出到 HTML（或文本，就此而言）？

进一步假设你只有基本的数据结构（数组和哈希图），没有带有父/子引用的花哨对象，没有 ORM，没有框架，只有你的两只手。该表表示为一个结果集，可以随机访问。

伪代码或简单的英文都可以，这纯粹是一个概念问题。

额外问题：有没有一种从根本上更好的方法来在 RDBMS 中存储这样的树结构？

编辑和补充

回答一位评论者（Mark Bessey的）的问题：根节点不是必需的，因为它永远不会被显示。ParentId = 0 是表达“这些是顶级”的约定。Order 列定义了如何对具有相同父节点的节点进行排序。

我所说的“结果集”可以被描绘成一个哈希图数组（保持在那个术语中）。对于我的例子，本来就已经存在了。有些答案会加倍努力并首先构建它，但这没关系。

树可以任意深。每个节点可以有 N 个子节点。不过，我并没有完全想到“数百万个条目”树。

不要误以为我选择的节点命名（'Node 1.1.1'）是可以依赖的。节点同样可以称为“Frank”或“Bob”，没有暗示命名结构，这只是为了使其可读。

我已经发布了我自己的解决方案，所以你们可以把它撕成碎片。

17 年 5 月 23 日 12:18编辑
社区 机器人
1 1

问08 年 10 月 10 日 16:47
托马拉克
312k 62 494 605

2 “没有带有父/子引用的花哨对象”-为什么不呢？使用 .addChild()、.getParent() 方法创建一个基本的 Node 对象可以让您很好地建模节点关系。 —— 哑光 08 年 10 月 10 日 16:55

2 它是常规（n 个子节点，其中 n 可以 > 2）树还是二叉树（节点可以有 0、1 或 2 个子节点）？ —— 金梅尔 08 年 10 月 10 日 16:59

由于您可以使用哈希图实现适当的节点数据结构，因此这里没有真正的限制，只需要做更多的工作。 —— 斯万特 09 年 4 月 27 日 16:05

.....这正是你所做的。 —— 斯万特 09 年 4 月 27 日 16:08

1 @dreftymac，从技术上讲，传递闭包表是非规范化的。比传统的邻接表设计更难避免数据异常。但作为非规范化设计的典型特征，它使某些类型的查询更快。 —— 比尔·卡文 8 月 19 日 14:21

14 个回答

积极的 | 最老的 | 投票

481

现在MySQL 8.0 支持递归查询，我们可以说所有流行的 SQL 数据库都支持标准语法的递归查询。

```
WITH RECURSIVE MyTree AS (
    SELECT * FROM MyTable WHERE ParentId IS NULL
    UNION ALL
    SELECT m.* FROM MyTABLE AS m JOIN MyTree AS t ON m.ParentId = t.Id
)
SELECT * FROM MyTree;
```

我在 2017 年的Recursive Query Throwdown演示文稿中测试了 MySQL 8.0 中的递归查询。

以下是我 2008 年的原始答案：

有几种方法可以在关系数据库中存储树结构数据。您在示例中显示的内容使用两种方法：

- **邻接表**（“父”列）和
- **路径枚举**（姓名列中的点号）。

另一种解决方案称为**Nested Sets**，它也可以存储在同一个表中。有关这些设计的更多信息，请阅读Joe Celko撰写的“ Smarties 中 SQL 中的树和层次结构”。

我通常更喜欢一种称为**闭包表**（又名“邻接关系”）的设计来存储树结构数据。它需要另一个表，但是查询树非常容易。

我在我的SQL 和 PHP 分层数据演示模型和我的书SQL 反模式：避免数据库编程的陷阱中介绍了闭包表。

```
CREATE TABLE ClosureTable (
  ancestor_id   INT NOT NULL REFERENCES FlatTable(id),
```

将所有路径存储在闭合表中，其中存在从一个节点到另一个节点的直接祖先。为每个节点包含一行以引用自身。例如，使用您在问题中显示的数据集：

```
INSERT INTO ClosureTable (ancestor_id, descendant_id) VALUES
  (1,1), (1,2), (1,4), (1,6),
  (2,2), (2,4),
  (3,3), (3,5),
  (4,4),
  (5,5),
  (6,6);
```

现在你可以得到一个从节点 1 开始的树，如下所示：

```
SELECT f.*
FROM FlatTable f
  JOIN ClosureTable a ON (f.id = a.descendant_id)
WHERE a.ancestor_id = 1;
```

输出（在 MySQL 客户端中）如下所示：

```
+----+
| id |
+----+
|  1 |
|  2 |
|  4 |
|  6 |
+----+
```

换句话说，节点 3 和 5 被排除在外，因为它们是单独层次结构的一部分，而不是从节点 1 下降。

回复：来自 e-satis 的关于直系子女（或直系父母）的评论。您可以向 中添加“ `path_length` ”列， `ClosureTable` 以便更轻松地专门查询直系子代或父代（或任何其他距离）。

```
INSERT INTO ClosureTable (ancestor_id, descendant_id, path_length) VALUES
  (1,1,0), (1,2,1), (1,4,2), (1,6,1),
  (2,2,0), (2,4,1),
  (3,3,0), (3,5,1),
  (4,4,0),
  (5,5,0),
  (6,6,0);
```

然后，您可以在搜索中添加一个术语来查询给定节点的直接子节点。这些 `path_length` 是 1 的后代。

```
SELECT f.*
FROM FlatTable f
  JOIN ClosureTable a ON (f.id = a.descendant_id)
WHERE a.ancestor_id = 1
  AND path_length = 1;

+----+
| id |
+----+
```

来自@ashraf 的评论：“[按名称] 对整棵树进行排序怎么样？”

这是一个示例查询，用于返回节点 1 的所有后代节点，将它们连接到包含其他节点属性（如 ）的 FlatTable `name` ，并按名称排序。

```
SELECT f.name
FROM FlatTable f
JOIN ClosureTable a ON (f.id = a.descendant_id)
WHERE a.ancestor_id = 1
ORDER BY f.name;
```

来自@Nate 的评论：

```
SELECT f.name, GROUP_CONCAT(b.ancestor_id order by b.path_length desc) AS breadcrumbs
FROM FlatTable f
JOIN ClosureTable a ON (f.id = a.descendant_id)
JOIN ClosureTable b ON (b.descendant_id = a.descendant_id)
WHERE a.ancestor_id = 1
GROUP BY a.descendant_id
ORDER BY f.name

+------------+-------------+
| name       | breadcrumbs |
+------------+-------------+
| Node 1     | 1           |
| Node 1.1   | 1,2         |
| Node 1.1.1 | 1,2,4       |
| Node 1.2   | 1,6         |
+------------+-------------+
```

一位用户今天建议进行编辑。SO 版主批准了编辑，但我正在撤销它。

编辑建议上面最后一个查询中的 ORDER BY 应该是 `ORDER BY b.path_length, f.name` ，大概是为了确保排序与层次结构相匹配。但这不起作用，因为它会在“Node 1.2”之后订购“Node 1.1.1”。

如果您希望排序以合理的方式匹配层次结构，这是可能的，但不能简单地按路径长度排序。例如，请参阅我对MySQL Closure Table 分层数据库的回答- 如何以正确的顺序提取信息。

分享 改进这个答案 跟随 20 年 1 月 1 日 18:17编辑 08 年 10 月 10 日 17:58 回答

比尔·卡文
476k 81 619 772

7 这很优雅，谢谢。奖励积分。;-) 不过我看到了一个小缺点——因为它显式*和*隐式地存储了子关系，即使树结构中的一个小变化，你也需要做很多仔细的更新。 —— 托马拉克 08 年 10 月 11 日 9:51

17 确实，在数据库中存储树结构的每种方法都需要一些工作，无论是在创建或更新树时，还是在查询树和子树时。选择您希望更简单的设计：写作或阅读。 —— 比尔·卡文 08 年 10 月 11 日 19:29

2 @buffer，当您为层次结构创建所有行时，可能会产生不一致。Adjacency List(`parent_id`)只有一行来表达每一个父子关系，而Closure Table却有很多。 —— 比尔·卡文 2014 年 5 月 13 日 15:21

1 @BillKarwin 还有一件事，闭包表适用于具有到任何给定节点的多条路径的图（例如，任何叶节点或非叶节

59

如果您使用嵌套集（有时称为修改前序树遍历），您可以使用单个查询以树顺序提取整个树结构或其中的任何子树，但插入成本更高，因为您需要管理通过树结构描述有序路径的列。

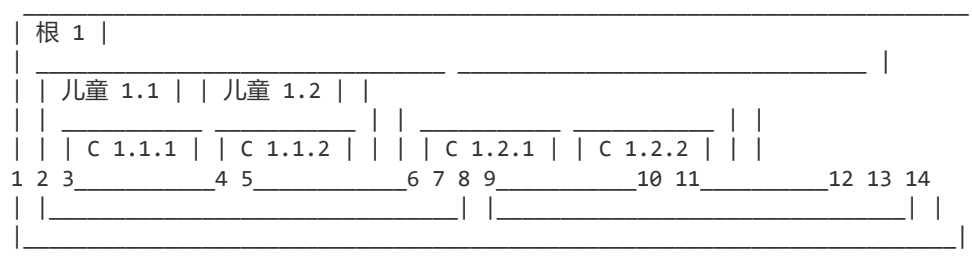
对于django-mptt，我使用了这样的结构：

```
id parent_id tree_id 级别 lft rght
-- --------- ------- ----- --- ----
 1 空 1 0 1 14
 2 1 1 1 2 7
 3 2 1 2 3 4
 4 2 1 2 5 6
 5 1 1 1 8 13
 6 5 1 2 9 10
 7 5 1 2 11 12
```

它描述了一个看起来像这样的树（ `id` 代表每个项目）：

```
1
+-- 2
| +-- 3
| +-- 4
|
+-- 5
    +-- 6
    +-- 7
```

或者，作为嵌套集图，它使 `lft` 和 `rght` 值的工作方式更加明显：



正如你看到的，让整个子树给定节点，在树的顺序，你只需要选择哪个都行 `lft` 和 `rght` 其之间的值 `lft` 和 `rght` 值。检索给定节点的祖先树也很简单。

`level` 为了方便起见，该列有点非规范化，并且该 `tree_id` 列允许您为每个顶级节点重新启动 `lft` 和 `rght` 编号，这减少了受插入、移动和删除影响的列数，因为 `lft` 和 `rght` 列必须是当这些操作发生时进行相应的调整，以创建或缩小差距。当我试图围绕每个操作所需的查询时，我做了一些开发笔记。

就实际使用这些数据来显示树而言，我创建了一个 `tree_item_iterator` 实用函数，对于每个节点，它应该为您提供足够的信息来生成您想要的任何类型的显示。

有关 MPTT 的更多信息：

- SQL 中的树

- 在 MySQL 中管理分层数据

分享 改进这个答案 跟随

11 年 11 月 15 日 15:00编辑
用户212218

08 年 10 月 11 日 12:31 回答
强尼·布坎南
59.1k 16 139 147

10 我希望我们能停止使用缩写像 `lft` 和 `rght` 列名，我的意思是我们没有多少字符类型？一？！
—— 奥鲁斯塔马纳波夫 2018 年 2 月 24 日 11:57

这是因为 'left' 和 'right' 是 SQL 中的保留字 —— 联系我们 2 月 21 日 7:17

24

这是一个相当古老的问题，但由于它有很多观点，我认为值得提出一个替代方案，在我看来非常优雅的解决方案。

为了读取树结构，您可以使用**递归公用表表达式**(CTE)。它提供了一次获取整个树结构的可能性，具有关于节点级别、其父节点和父节点子节点中的顺序的信息。

让我向您展示这在 PostgreSQL 9.1 中是如何工作的。

1. 创建一个结构

```
CREATE TABLE tree (
    id int  NOT NULL,
    name varchar(32)  NOT NULL,
    parent_id int  NULL,
    node_order int  NOT NULL,
    CONSTRAINT tree_pk PRIMARY KEY (id),
    CONSTRAINT tree_tree_fk FOREIGN KEY (parent_id)
      REFERENCES tree (id) NOT DEFERRABLE
);


insert into tree values
  (0, 'ROOT', NULL, 0),
  (1, 'Node 1', 0, 10),
  (2, 'Node 1.1', 1, 10),
  (3, 'Node 2', 0, 20),
  (4, 'Node 1.1.1', 2, 10),
  (5, 'Node 2.1', 3, 10),
  (6, 'Node 1.2', 1, 20);
```

2. 写一个查询

```
WITH RECURSIVE
tree_search (id, name, level, parent_id, node_order) AS (
  SELECT
    id,
    name,
    0,
    parent_id,
    1
  FROM tree
  WHERE parent_id is NULL

  UNION ALL
  SELECT
    t.id,
    t.name
```

```
  FROM tree t, tree_search ts
  WHERE t.parent_id = ts.id
)
SELECT * FROM tree_search
WHERE level > 0
ORDER BY level, parent_id, node_order;
```

结果如下：

```
 id |    name    | level | parent_id | node_order
----+------------+-------+-----------+------------
  1 | Node 1     |     1 |         0 |         10
  3 | Node 2     |     1 |         0 |         20
  2 | Node 1.1   |     2 |         1 |         10
  6 | Node 1.2   |     2 |         1 |         20
  5 | Node 2.1   |     2 |         3 |         10
  4 | Node 1.1.1 |     3 |         2 |         10
(6 rows)
```

树节点按深度级别排序。在最终输出中，我们将在后续行中呈现它们。

对于每个级别，它们在父级中按 parent_id 和 node_order 排序。这告诉我们如何在输出中将它们呈现给父节点的链接节点。

有了这样的结构，用 HTML 制作一个非常好的演示文稿并不困难。

递归 CTE 在**PostgreSQL、IBM DB2、MS SQL Server 和 Oracle**中可用。

如果您想阅读有关递归 SQL 查询的更多信息，您可以查看您最喜欢的 DBMS 的文档或阅读我的两篇涵盖该主题的文章：

- 在 SQL 中实现：递归树遍历
- 了解 SQL 递归查询的强大功能

分享 改进这个答案 跟随

2014 年3 月 13 日 11:19回答

米哈乌·科沃齐耶斯基
596 3 6

18

从 Oracle 9i 开始，您可以使用 CONNECT BY。

```
SELECT LPAD(' ', (LEVEL - 1) * 4) || "Name" AS "Name"
FROM (SELECT * FROM TMP_NODE ORDER BY "Order")
CONNECT BY PRIOR "Id" = "ParentId"
START WITH "Id" IN (SELECT "Id" FROM TMP_NODE WHERE "ParentId" = 0)
```

从 SQL Server 2005 开始，您可以使用递归公用表表达式 (CTE)。

```
WITH [NodeList] (
  [Id]
  , [ParentId]
  , [Level]
  , [Order]
) AS (
  SELECT [Node].[Id]
    , [Node].[ParentId]
    , 0 AS [Level]
    , CONVERT([varchar](MAX), [Node].[Order]) AS [Order]
```

```
  SELECT [Node].[Id]
    , [Node].[ParentId]
    , [NodeList].[Level] + 1 AS [Level]
    , [NodeList].[Order] + '|'
      + CONVERT([varchar](MAX), [Node].[Order]) AS [Order]
  FROM [Node]
    INNER JOIN [NodeList] ON [NodeList].[Id] = [Node].[ParentId]
) SELECT REPLICATE(' ', [NodeList].[Level] * 4) + [Node].[Name] AS [Name]
FROM [Node]
  INNER JOIN [NodeList] ON [NodeList].[Id] = [Node].[Id]
ORDER BY [NodeList].[Order]
```

两者都会输出以下结果。

```
姓名
'节点 1'
'节点 1.1'
'节点 1.1.1'
'节点 1.2'
'节点 2'
'节点 2.1'
```

分享 改进这个答案 跟随

11 年 11 月 15 日 15:00编辑
用户212218

08 年 10 月 10 日 20:06 回答

埃里克·魏瑙
5,012 4 30 30

cte 可用于 sqlserver 和 oracle @Eric Weilnau —— 尼萨尔 2014 年 5 月 25 日 12:38

10

比尔的答案真是太棒了，这个答案增加了一些东西，这让我希望SO支持线程答案。

无论如何，我想支持树结构和 Order 属性。我在每个 Node 中都包含了一个属性，该属性与原始问题中要做的 `leftSibling` 事情相同 `Order` （保持从左到右的顺序）。

```
mysql> desc 节点;
+--------------+--------------+------+-------+------- --+--+
| 领域 | 类型 | 空 | 钥匙 | 默认 | 额外 |
+--------------+--------------+------+-------+------- --+--+
| 身份证 | 整数(11) | 否 | PRI | 空 | 自动增量|
| 姓名 | varchar(255) | 是 | | 空 | |
| 左兄弟 | 整数(11) | 否 | | 0 | |
+--------------+--------------+------+-------+------- --+--+
3 行 (0.00 秒)

mysql> desc 邻接;
+------------+---------+------+-----+---------+--- -------------+
| 领域 | 类型 | 空 | 钥匙 | 默认 | 额外 |
+------------+---------+------+-----+---------+--- -------------+
| 关系 ID | 整数(11) | 否 | PRI | 空 | 自动增量|
| 家长 | 整数(11) | 否 | | 空 | |
| 孩子 | 整数(11) | 否 | | 空 | |
| 路径长度 | 整数(11) | 否 | | 空 | |
+------------+---------+------+-----+---------+--- -------------+
4 行 (0.00 秒)
```

更多细节和 SQL 代码在我的博客上。

分享 改进这个答案 跟随

2012 年6 月 28 日 13:27编辑

莱尼尔·马卡费里
95.4k 42 348 455

10 年 12 月 22 日 4:31 回答

波波波波
59.2k 58 242 344

7

如果可以选择，我会使用对象。我会为每个记录创建一个对象，其中每个对象都有一个 `children` 集合，并将它们全部存储在一个关联数组 (/hashtable) 中，其中 Id 是键。并快速浏览一次集合，将子项添加到相关的子项字段。**简单的。**

但是因为你通过限制使用一些好的 OOP 没有乐趣，我可能会基于以下内容进行迭代：

```
function PrintLine(int pID, int level)
    foreach record where ParentID == pID
        print level*tabs + record-data
        PrintLine(record.ID, level + 1)

PrintLine(0, 0)
```

编辑：这与其他几个条目类似，但我认为它更简洁一些。我要补充的一件事是：这是非常 SQL 密集型的。这是*肮脏的*。**如果可以选择，请走 OOP 路线。**

分享 改进这个答案 跟随

08 年 10 月 10 日 17:36 回答

奥利
221k 61 210 289

这就是我所说的"无框架"的意思——您正在使用 LINQ，不是吗？关于您的第一段：结果集已经存在，为什么要先将所有信息复制到新的对象结构中？（我对这个事实不够清楚，抱歉） —— 托马拉克 08 年 10 月 11 日 10:15

Tomalak - 不，代码是伪代码。当然，您必须将事情分解为正确的选择和迭代器.....以及真正的语法！为什么是面向对象的？因为你可以精确地镜像结构。它让事情变得美好，而且碰巧更有效率（只有一个选择） —— 奥利 08 年 10 月 12 日 9:52

我也没有考虑重复选择。关于 OOP：Mark Bessey 在他的回答中说："你可以用 hashmap 模拟任何其他数据结构，所以这不是一个可怕的限制。"。您的解决方案是正确的，但我认为即使没有 OOP，仍有一些改进的空间。 —— 托马拉克 08 年 10 月 13 日 8:45

6

确实有很好的解决方案可以利用 sql 索引的内部 btree 表示。这是基于 1998 年左右完成的一些伟大研究。

这是一个示例表（在 mysql 中）。

```
CREATE TABLE `node` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `tw` int(10) unsigned NOT NULL,
  `pa` int(10) unsigned DEFAULT NULL,
  `sz` int(10) unsigned DEFAULT NULL,
  `nc` int(11) GENERATED ALWAYS AS (tw+sz) STORED,
  PRIMARY KEY (`id`),
  KEY `node_tw_index` (`tw`),
  KEY `node_pa_index` (`pa`),
  KEY `node_nc_index` (`nc`),
```

树表示所需的唯一字段是：

- tw：从左到右的 DFS 预排序索引，其中 root = 1。
- pa：对父节点的引用（使用 tw），root 为 null。
- sz：包括自身在内的节点分支的大小。
- nc：用作语法糖。它是 tw+sz 并且代表节点的"下一个子节点"的 tw。

这是一个示例 24 个节点填充，按 tw 排序：

```
+-----+---------+----+------+------+------+
| id  | name    | tw | pa   | sz   | nc   |
+-----+---------+----+------+------+------+
|   1 | Root    |  1 | NULL |   24 |   25 |
|   2 | A       |  2 |    1 |   14 |   16 |
|   3 | AA      |  3 |    2 |    1 |    4 |
|   4 | AB      |  4 |    2 |    7 |   11 |
|   5 | ABA     |  5 |    4 |    1 |    6 |
|   6 | ABB     |  6 |    4 |    3 |    9 |
|   7 | ABBA    |  7 |    6 |    1 |    8 |
|   8 | ABBB    |  8 |    6 |    1 |    9 |
|   9 | ABC     |  9 |    4 |    2 |   11 |
|  10 | ABCD    | 10 |    9 |    1 |   11 |
|  11 | AC      | 11 |    2 |    4 |   15 |
|  12 | ACA     | 12 |   11 |    2 |   14 |
|  13 | ACAA    | 13 |   12 |    1 |   14 |
|  14 | ACB     | 14 |   11 |    1 |   15 |
|  15 | AD      | 15 |    2 |    1 |   16 |
|  16 | B       | 16 |    1 |    1 |   17 |
|  17 | C       | 17 |    1 |    6 |   23 |
| 359 | C0      | 18 |   17 |    5 |   23 |
| 360 | C1      | 19 |   18 |    4 |   23 |
| 361 | C2(res) | 20 |   19 |    3 |   23 |
| 362 | C3      | 21 |   20 |    2 |   23 |
| 363 | C4      | 22 |   21 |    1 |   23 |
|  18 | D       | 23 |    1 |    1 |   24 |
|  19 | E       | 24 |    1 |    1 |   25 |
+-----+---------+----+------+------+------+
```

每个树结果都可以非递归地完成。例如，要获取 tw='22' 节点的祖先列表

祖先

```
select anc.* from node me,node anc
where me.tw=22 and anc.nc >= me.tw and anc.tw <= me.tw
order by anc.tw;
+-----+---------+----+------+------+------+
| id  | name    | tw | pa   | sz   | nc   |
+-----+---------+----+------+------+------+
|   1 | Root    |  1 | NULL |   24 |   25 |
|  17 | C       | 17 |    1 |    6 |   23 |
| 359 | C0      | 18 |   17 |    5 |   23 |
| 360 | C1      | 19 |   18 |    4 |   23 |
| 361 | C2(res) | 20 |   19 |    3 |   23 |
| 362 | C3      | 21 |   20 |    2 |   23 |
| 363 | C4      | 22 |   21 |    1 |   23 |
+-----+---------+----+------+------+------+
```

兄弟姐妹和孩子是微不足道的 - 只需使用 pa 字段按 tw 排序。

例如，以 tw = 17 为根的节点集（分支）。

```
select des.* from node me,node des
where me.tw=17 and des.tw < me.nc and des.tw >= me.tw
order by des.tw;
+-----+---------+----+------+------+------+
| id  | name    | tw | pa   | sz   | nc   |
+-----+---------+----+------+------+------+
|  17 | C       | 17 |    1 |    6 |   23 |
| 359 | C0      | 18 |   17 |    5 |   23 |
| 360 | C1      | 19 |   18 |    4 |   23 |
| 361 | C2(res) | 20 |   19 |    3 |   23 |
| 362 | C3      | 21 |   20 |    2 |   23 |
| 363 | C4      | 22 |   21 |    1 |   23 |
+-----+---------+----+------+------+------+
```

补充说明

当读取次数远多于插入或更新次数时，此方法非常有用。

由于树中节点的插入、移动或更新需要对树进行调整，因此在开始操作之前需要锁定表。

插入/删除成本很高，因为需要在插入点之后的所有节点上更新 tw 索引和 sz（分支大小）值，并分别为所有祖先更新。

分支移动涉及到将分支的 tw 值移出范围，因此在移动分支时也需要禁用外键约束。移动分支基本上需要四个查询：

- 将分支移出范围。
- 关闭它留下的间隙。（剩下的树现在已经标准化了）。
- 打开它要去的地方。
- 将分支移动到新位置。

调整树查询

树中间隙的打开/关闭是 create/update/delete 方法使用的一个重要子功能，因此我将其包含在此处。

我们需要两个参数 - 一个表示我们是缩小规模还是扩大规模的标志，以及节点的 tw 索引。因此，例如 tw=18（其分支大小为 5）。让我们假设我们正在缩小规模（删除 tw） - 这意味着我们在以下示例的更新中使用“-”而不是“+”。

我们首先使用（稍微改变的）祖先函数来更新 sz 值。

```
update node me, node anc set anc.sz = anc.sz - me.sz from
node me, node anc where me.tw=18
and ((anc.nc >= me.tw and anc.tw < me.pa) or (anc.tw=me.pa));
```

然后我们需要为那些tw高于要删除的分支的人调整tw。

```
update node me, node anc set anc.tw = anc.tw - me.sz from
```

然后我们需要为那些 pa 的 tw 高于要删除的分支的人调整父级。

```
update node me, node anc set anc.pa = anc.pa - me.sz from
node me, node anc where me.tw=18 and anc.pa >= me.tw;
```

分享 改进这个答案 跟随 7 月 17 日 8:10编辑 17 年 3 月 14 日 8:43 回答

孔乔格
1,528 14 18

5

这写得很快，既不漂亮也不高效（加上它自动装箱很多，在 `int` 和之间转换 `Integer` 很烦人！），但它有效。

它可能违反了规则，因为我正在创建自己的对象，但是嘿，我这样做是为了转移对实际工作的注意力:)

这还假设在您开始构建节点之前，结果集/表已完全读入某种结构，如果您有数十万行，这将不是最佳解决方案。

```
public class Node {

    private Node parent = null;

    private List<Node> children;

    private String name;

    private int id = -1;

    public Node(Node parent, int id, String name) {
        this.parent = parent;
        this.children = new ArrayList<Node>();
        this.name = name;
        this.id = id;
    }

    public int getId() {
        return this.id;
    }

    public String getName() {
        return this.name;
    }

    public void addChild(Node child) {
        children.add(child);
    }

    public List<Node> getChildren() {
        return children;
    }

    public boolean isRoot() {
        return (this.parent == null);
    }

    @Override
    public String toString() {
        return "id=" + id + ", name=" + name + ", parent=" + parent;
```

```
    public static Node build(List<Map<String, String>> input) {

        // maps id of a node to it's Node object
        Map<Integer, Node> nodeMap = new HashMap<Integer, Node>();

        // maps id of a node to the id of it's parent
        Map<Integer, Integer> childParentMap = new HashMap<Integer, Integer>();

        // create special 'root' Node with id=0
        Node root = new Node(null, 0, "root");
        nodeMap.put(root.getId(), root);

        // iterate thru the input
        for (Map<String, String> map : input) {

            // expect each Map to have keys for "id", "name", "parent" ... a
            // real implementation would read from a SQL object or resultset
            int id = Integer.parseInt(map.get("id"));
            String name = map.get("name");
            int parent = Integer.parseInt(map.get("parent"));

            Node node = new Node(null, id, name);
            nodeMap.put(id, node);

            childParentMap.put(id, parent);
        }

        // now that each Node is created, setup the child-parent relationships
        for (Map.Entry<Integer, Integer> entry : childParentMap.entrySet()) {
            int nodeId = entry.getKey();
            int parentId = entry.getValue();

            Node child = nodeMap.get(nodeId);
            Node parent = nodeMap.get(parentId);
            parent.addChild(child);
        }

        return root;
    }
}

public class NodePrinter {

    static void printRootNode(Node root) {
        printNodes(root, 0);
    }

    static void printNodes(Node node, int indentLevel) {

        printNode(node, indentLevel);
        // recurse
        for (Node child : node.getChildren()) {
            printNodes(child, indentLevel + 1);
        }
    }

    static void printNode(Node node, int indentLevel) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < indentLevel; i++) {
            sb.append("\t");
        }
        sb.append(node);

        System.out.println(sb.toString());
    }

    public static void main(String[] args) {
```

```
            resultSet.add(newMap("1", "Node 1", "0"));
            resultSet.add(newMap("2", "Node 1.1", "1"));
            resultSet.add(newMap("3", "Node 2", "0"));
            resultSet.add(newMap("4", "Node 1.1.1", "2"));
            resultSet.add(newMap("5", "Node 2.1", "3"));
            resultSet.add(newMap("6", "Node 1.2", "1"));

            Node root = NodeBuilder.build(resultSet);
            printRootNode(root);

        }

        //convenience method for creating our dummy data
        private static Map<String, String> newMap(String id, String name, String parentId)
    {
            Map<String, String> row = new HashMap<String, String>();
            row.put("id", id);
            row.put("name", name);
            row.put("parent", parentId);
            return row;
        }
    }
```

分享 改进这个答案 跟随

08 年 10 月 10 日 18:25 回答

哑光
134k 64 272 336

当提供大量源代码时，我总是发现很难从特定于实现的部分中过滤出特定于算法的部分。这就是为什么我首先要求一个不特定于语言的解决方案。但它可以完成工作，所以感谢您的时间！ —— 托马拉克 08 年 10 月 11 日 12:19

我现在明白你的意思了，如果主要算法在 NodeBuilder.build() 中不明显 - 我可能可以更好地总结这一点。—— 哑光 08 年 10 月 13 日 20:05

4

假设您知道根元素为零，以下是输出到文本的伪代码：

```
function PrintLevel (int curr, int level)
    //print the indents
    for (i=1; i<=level; i++)
        print a tab
    print curr \n;
    for each child in the table with a parent of curr
        PrintLevel (child, level+1)


for each elementID where the parentid is zero
    PrintLevel(elementID, 0)
```

分享 改进这个答案 跟随

08 年 10 月 10 日 16:59 回答

水箱
8,765 6 37 64

3

您可以使用哈希图模拟任何其他数据结构，因此这不是一个可怕的限制。从上到下扫描，为数据库的每一行创建一个哈希图，每列都有一个条目。将这些哈希映射中的每一个添加到以 id 为键的"主"哈希映射中。如果任何节点具有您尚未看到的"父节点"，请在主哈希图中为它创建一个占位

要打印出来，请执行简单的深度优先遍历数据，并在此过程中跟踪缩进级别。您可以通过为每一行保留一个“子”条目并在扫描数据时填充它来使这更容易。

至于在数据库中存储树是否有“更好”的方法，这取决于您将如何使用数据。我见过具有已知最大深度的系统，这些系统对层次结构中的每个级别使用不同的表。如果树中的级别毕竟不完全相同（顶级类别与叶子不同），这很有意义。

分享 改进这个答案 跟随

08 年 10 月 10 日 17:24 回答

马克·贝西
19.2k 4 46 67

1

如果可以创建嵌套的哈希映射或数组，那么我可以简单地从头开始向下查找表并将每个项目添加到嵌套数组中。我必须跟踪每一行到根节点，以便知道要插入到嵌套数组中的哪个级别。我可以使用备忘录，这样我就不需要一遍又一遍地查找同一个父级。

编辑：我会先将整个表读入一个数组，这样它就不会重复查询数据库。当然，如果您的桌子非常大，这将不实用。

构建结构后，我必须先进行深度遍历并打印出 HTML。

没有更好的基本方法来使用一张表存储这些信息（虽然我可能是错的；），并且希望看到更好的解决方案）。但是，如果您创建一个方案来使用动态创建的 db 表，那么您会以牺牲简单性和 SQL 地狱的风险为代价打开一个全新的世界；）。

分享 改进这个答案 跟随

08 年 10 月 10 日 20:16编辑

08 年 10 月 10 日 17:02 回答

陈
2,103 1 14 18

1 我不想仅仅因为需要新级别的子节点而更改数据库布局。:-) —— 托马拉克 08 年 10 月 11 日 12:06

1

如果元素按树顺序排列，如您的示例所示，您可以使用类似于以下 Python 示例的内容：

```
delimiter = '.'
stack = []
for item in items:
  while stack and not item.startswith(stack[-1]+delimiter):
    print "</div>"
    stack.pop()
  print "<div>"
  print item
  stack.append(item)
```

这样做是维护一个表示树中当前位置的堆栈。对于表中的每个元素，它弹出堆栈元素（关闭匹配的 div），直到找到当前项的父项。然后它输出该节点的开始并将其压入堆栈。

如果你想使用缩进而不是嵌套元素输出树，你可以简单地跳过打印语句来打印 div，并在每个项目之前打印一些等于堆栈大小的倍数的空格。例如，在 Python 中：

您还可以轻松地使用此方法来构造一组嵌套列表或字典。

编辑：我从您的澄清中看出，这些名称并非旨在成为节点路径。这表明了另一种方法：

```
idx = {}
idx[0] = []
for node in results:
  child_list = []
  idx[node.Id] = child_list
  idx[node.ParentId].append((node, child_list))
```

这构造了一个元组数组树（！）。idx[0] 表示树的根。数组中的每个元素都是一个 2 元组，由节点本身及其所有子节点的列表组成。一旦构造完成，您可以保留 idx[0] 并丢弃 idx，除非您想通过它们的 ID 访问节点。

分享 改进这个答案 跟随

08 年 10 月 14 日 12:05编辑

08 年 10 月 10 日 21:45 回答

尼克约翰逊

99.4k 16 126 196

1

要扩展 Bill 的 SQL 解决方案，您基本上可以使用平面数组来做同样的事情。此外，如果您的字符串都具有相同的长度并且您的最大子代数是已知的（例如在二叉树中），您可以使用单个字符串（字符数组）来完成。如果你有任意数量的孩子，这会让事情变得有点复杂……我必须检查我的旧笔记，看看可以做些什么。

然后，牺牲一点内存，特别是如果你的树是稀疏的和/或不平衡的，你可以通过一些索引数学，通过存储你的树随机访问所有字符串，宽度在数组中像这样（对于二进制树）：

```
String[] nodeArray = [L0root, L1child1, L1child2, L2Child1, L2Child2, L2Child3,
L2Child4] ...
```

你知道你的字符串长度，你知道的

我现在正在工作，所以不能花太多时间在这上面，但我有兴趣可以获取一些代码来做到这一点。

我们过去常常用它来搜索由 DNA 密码子组成的二叉树，一个过程构建了树，然后我们将它展平以搜索文本模式，当找到时，虽然索引数学（从上面反向）我们得到节点……非常快速、高效、坚固我们的树很少有空节点，但我们可以在瞬间搜索千兆字节的数据。

分享 改进这个答案 跟随

2012 年10 月 2 日 12:13编辑

淡蓝色

23.8k 24 110 171

08 年 10 月 10 日 18:42 回答

新托邦

7,138 4 47 70

0

考虑使用 nosql 工具（如 neo4j）进行层次结构。例如，像linkedin 这样的网络应用程序使用 couchbase（另一个nosql 解决方案）

但仅将 nosql 用于数据集市级别的查询，而不是存储/维护事务

阅读了 SQL 和“非表”结构的复杂性和性能后，这也是我的第一个想法，nosql。当然，导出等有很多问题。另外，OP 只提到了表格。那好吧。我不是数据库专家，这是显而易见的。 —— 约瑟夫·B 2015 年 12 月 13 日 14:14

高度活跃的问题。为了回答这个问题，赚取 10 点声望（不包括协会奖金）。声誉要求有助于保护此问题免受垃圾邮件和非回答活动的影响。