逻辑数据库设计 - 单纯的树(递归关系数据)

相信有过开发经验的朋友都曾碰到过这样一个需求。假设你正在为一个新闻网站开发一个评论功能，读者可以评论原文甚至相互回复。

这个需求并不简单，相互回复会导致无限多的分支，无限多的祖先-后代关系。这是一种典型的递归关系数据。

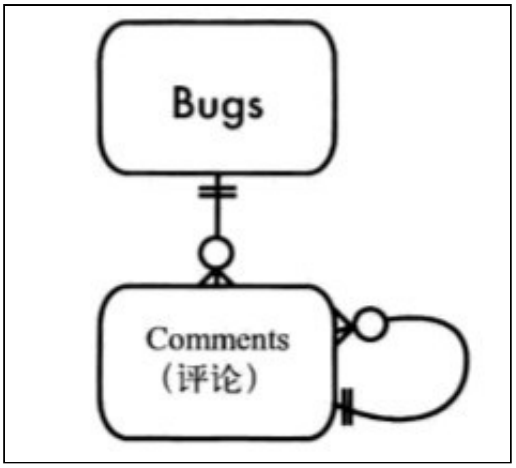
对于这个问题，以下给出几个解决方案，各位客观可斟酌后选择。

一、邻接表:依赖父节点

邻接表的方案如下(仅仅说明问题)：

```
   CREATE TABLE Comments(
      CommentId    int    PK,
      ParentId     int,        --记录父节点
      ArticleId    int,
      CommentBody nvarchar(500),
      FOREIGN KEY (ParentId)  REFERENCES Comments(CommentId)     --自连接，主键外键都在自己表内
      FOREIGN KEY (ArticleId) REFERENCES Articles(ArticleId)
   )
```

由于偷懒，所以采用了书本中的图了，Bugs就是Articles：

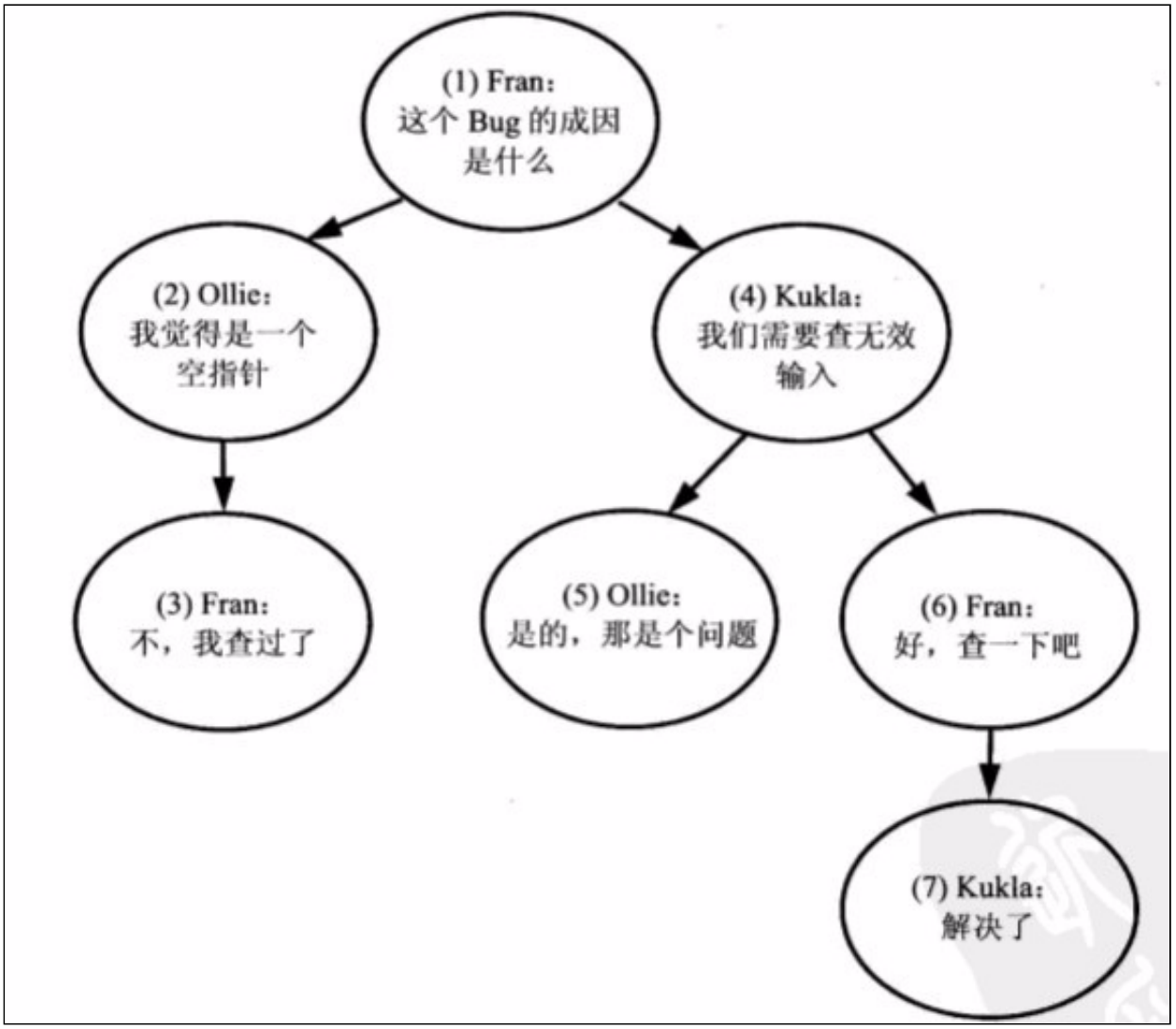


这种设计方式就叫做邻接表。这可能是存储分层结构数据中最普通的方案了。

下面给出一些数据来显示一下评论表中的分层结构数据。示例表：

	CommentId	ParentId	CommentBody
	1	1	这个Bug的成因是什么
	2	1	我觉得是一个空指针
	3	2	不是，我查过了
	4	1	我们需要查无效的输入
	5	4	是的，那是个问题
	6	4	好，查一下吧。
	7	6	解决了
▶*	NULL	NULL	NULL

图片说明存储结构：



邻接表的优缺分析

对于以上邻接表，很多程序员已经将其当成默认的解决方案了，但即便是这样，但它在从前还是有存在的问题的。

分析1：查询一个节点的所有后代(求子树)怎么查呢？

我们先看看以前查询两层的数据的SQL语句：

```
SELECT c1.*,c2.*
FROM Comments c1 LEFT OUTER JOIN Comments2 c2
ON c2.ParentId = c1.CommentId
```

显然，每需要查多一层，就需要联结多一次表。SQL查询的联结次数是有限的，因此不能无限深的获取所有的后代。而且，这种这样联结，执行Count()这样的聚合函数也相当困难。

说了是以前了，现在什么时代了，在SQLServer 2005之后，一个公用表表达式就搞定了,顺带解决的还有聚合函数的问题(聚合函数如Count()也能够简单实用)，例如查询评论4的所有子节点：

```
WITH COMMENT_CTE(CommentId,ParentId,CommentBody,tLevel)
AS
(
    --基本语句
    SELECT CommentId,ParentId,CommentBody,0 AS tLevel FROM Comment
    WHERE ParentId = 4
    UNION ALL  --递归语句
    SELECT c.CommentId,c.ParentId,c.CommentBody,ce.tLevel + 1 FROM Com
ment AS c
    INNER JOIN COMMENT_CTE AS ce    --递归查询
    ON c.ParentId = ce.CommentId
)
SELECT * FROM COMMENT_CTE
```

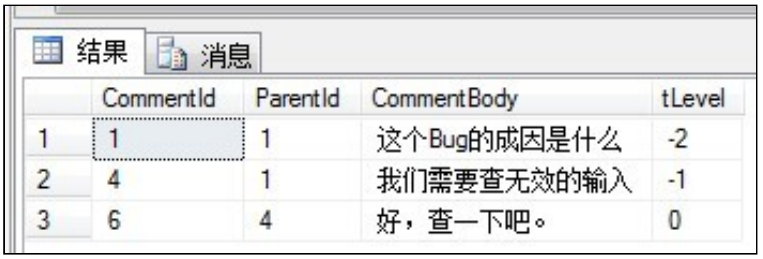
显示结果如下：

	CommentId	ParentId	CommentBody	tLevel
1	5	4	是的，那是个问题	0
2	6	4	好，查一下吧。	0
3	7	6	解决了	1

那么查询祖先节点树又如何查呢？例如查节点6的所有祖先节点：

```
WITH COMMENT_CTE(CommentId,ParentId,CommentBody,tLevel)
AS
(
    --基本语句
    SELECT CommentId,ParentId,CommentBody,0 AS tLevel FROM Comment
    WHERE CommentId = 6
    UNION ALL
    SELECT c.CommentId,c.ParentId,c.CommentBody,ce.tLevel - 1  FROM Co
mment AS c
    INNER JOIN COMMENT_CTE AS ce    --递归查询
    ON ce.ParentId = c.CommentId
    where ce.CommentId <> ce.ParentId
)
SELECT * FROM COMMENT_CTE ORDER BY CommentId ASC
```

结果如下：



结果 消息

	CommentId	ParentId	CommentBody	tLevel
1	1	1	这个Bug的成因是什么	-2
2	4	1	我们需要查无效的输入	-1
3	6	4	好，查一下吧。	0

再者，由于公用表表达式能够控制递归的深度，因此，你可以简单获得任意层级的子树。

```
    OPTION(MAXRECURSION 2)
```

看来哥是为邻接表平反来的。

分析**2**：当然，邻接表也有其优点的，例如要添加一条记录是非常方便的。

```
    INSERT INTO Comment(ArticleId,ParentId)...      --仅仅需要提供父节点Id就能够添加了。
```

分析**3**：修改一个节点位置或一个子树的位置也是很简单.

```
UPDATE Comment SET ParentId = 10 WHERE CommentId = 6    --仅仅修改一个节点的ParentId，其后面的子代节点自动合理。
```

分析**4**：删除子树

想象一下，如果你删除了一个中间节点，那么该节点的子节点怎么办(它们的父节点是谁)，因此如果你要删除一个中间节点，那么不得不查找到所有的后代，先将其删除，然后才能删除该中间节点。

当然这也能通过一个ON DELETE CASCADE级联删除的外键约束来自动完成这个过程。

分析**5**：删除中间节点，并提升子节点

面对提升子节点，我们要先修改该中间节点的直接子节点的ParentId，然后才能删除该节点：

```
    SELECT ParentId FROM Comments WHERE CommentId = 6;       --搜索要删除节点的父节点，假设返回4
    UPDATE Comments SET ParentId = 4 WHERE ParentId = 6;     --修改该中间节点的子节点的ParentId为要删除中间节点的ParentId
    DELETE FROM Comments WHERE CommentId = 6;                --终于可以删除该中间节点了
```

由上面的分析可以看到，邻接表基本上已经是很强大的了。

二、路径枚举

路径枚举的设计是指通过将所有祖先的信息联合成一个字符串，并保存为每个节点的一个属性。

路径枚举是一个由连续的直接层级关系组成的完整路径。如"/home/account/login",其中home是account的直接父亲，这也就意味着home是login的祖先。

还是有刚才新闻评论的例子，我们用路径枚举的方式来代替邻接表的设计：

```
CREATE TABLE Comments(
    CommentId    int    PK,
    Path         varchar(100),        --仅仅改变了该字段和删除了外键
    ArticleId    int,
    CommentBody nvarchar(500),
    FOREIGN KEY (ArticleId) REFERENCES Articles(ArticleId)
)
```

简略说明问题的数据表如下：

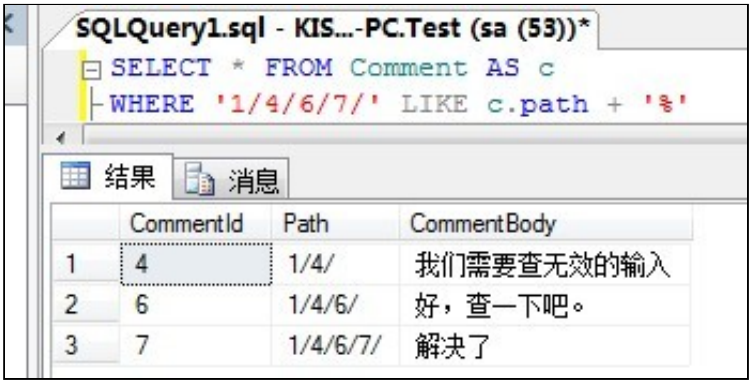
CommentId	Path	CommentBody
1	1/	这个Bug的成因是什么
2	1/2/	我觉得是一个空指针
3	1/2/3	不是，我查过了
4	1/4/	我们需要查无效的输入
5	1/4/5/	是的，那是个问题
6	1/4/6/	好，查一下吧。
7	1/4/6/7/	解决了

路径枚举的优点：

对于以上表，假设我们需要查询某个节点的全部祖先，SQL语句可以这样写(假设查询7的所有祖先节点)：

```
SELECT * FROM Comment AS c
WHERE '1/4/6/7/' LIKE c.path + '%'
```

结果如下：

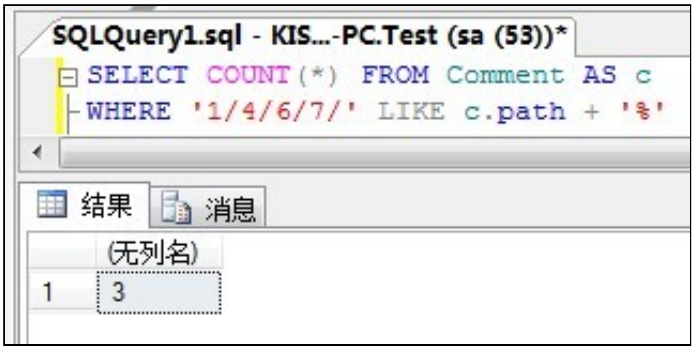


假设我们要查询某个节点的全部后代，假设为4的后代：

```
SELECT * FROM Comment AS c
WHERE c.Path LIKE '1/4/%'
```

结果如下：

一旦我们可以很简单地获取一个子树或者从子孙节点到祖先节点的路径，就可以很简单地实现更多查询，比如计算一个字数所有节点的数量(COUNT聚合函数)



插入一个节点也可以像和使用邻接表一样地简单。可以插入一个叶子节点而不用修改任何其他的行。你所需要做的只是复制一份要插入节点的逻辑上的父亲节点路径，并将这个新节点的Id追加到路径末尾就可以了。如果这个Id是插入时由数据库生成的，你可能需要先插入这条记录，然后获取这条记录的Id，并更新它的路径。

路径枚举的缺点：

1、数据库不能确保路径的格式总是正确或者路径中的节点确实存在(中间节点被删除的情况，没外键约束)。

2、要依赖高级程序来维护路径中的字符串，并且验证字符串的正确性的开销很大。

3、VARCHAR的长度很难确定。无论VARCHAR的长度设为多大，都存在不能够无限扩展的情况。

路径枚举的设计方式能够很方便地根据节点的层级排序，因为路径中分隔两边的节点间的距离永远是1，因此通过比较字符串长度就能知道层级的深浅。

三、嵌套集

嵌套集解决方案是存储子孙节点的信息，而不是节点的直接祖先。我们使用两个数字来编码每个节点，表示这个信息。可以将这两个数字称为nsleft和nsright。

还是以上面的新闻-评论作为例子，对于嵌套集的方式表可以设计为：

```
CREATE TABLE Comments(
    CommentId    int    PK,
    nsleft       int,    --之前的一个父节点
     nsright      int,    --变成了两个
    ArticleId    int,
    CommentBody nvarchar(500),
```

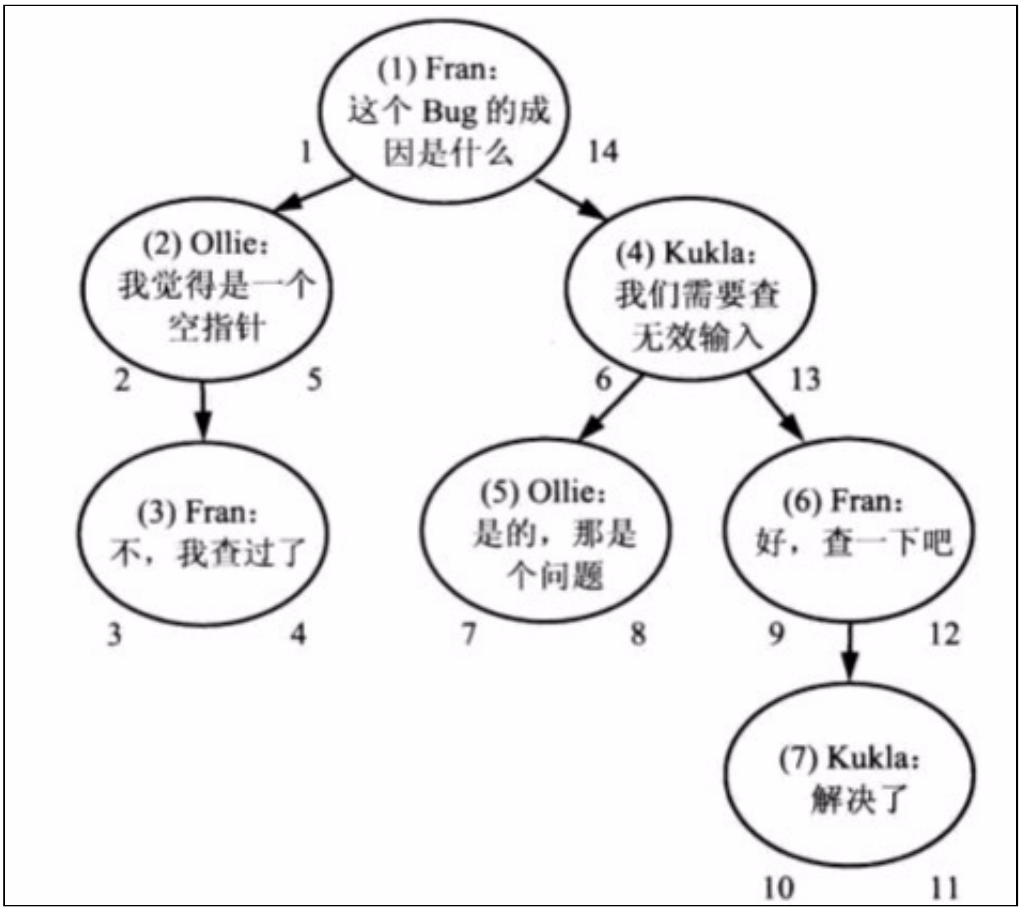
```
    FOREIGN KEY (ArticleId) REFERENCES Articles(ArticleId)
)
```

nsleft值的确定：nsleft的数值小于该节点所有后代的Id。

nsright值的确定：nsright的值大于该节点所有后代的Id。

当然，以上两个数字和CommentId的值并没有任何关联，确定值的方式是对树进行一次深度优先遍历，在逐层入神的过程中依次递增地分配nsleft的值，并在返回时依次递增地分配nsright的值。

采用书中的图来说明一下情况：



一旦你为每个节点分配了这些数字，就可以使用它们来找到给定节点的祖先和后代。

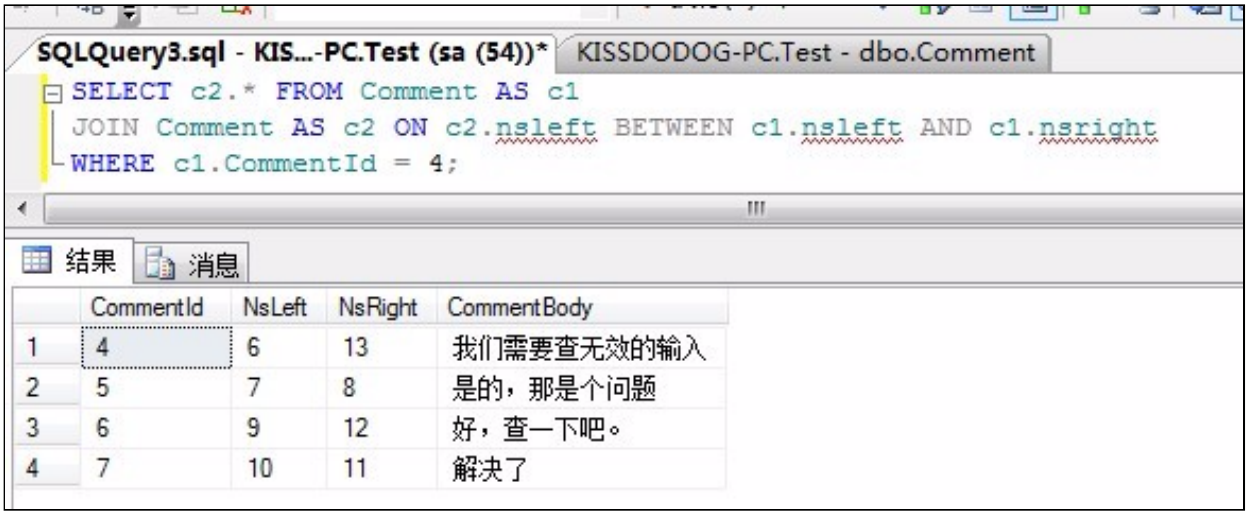
嵌套集的优点：

我觉得是唯一的优点了，查询祖先树和子树方便。

例如，通过搜索那些节点的ConmentId在评论4的nsleft与nsright之间就可以获得其及其所有后代：

```
SELECT c2.* FROM Comments AS c1
JOIN Comments AS c2    ON cs.neleft BETWEEN c1.nsleft AND c1.nsright
WHERE c1.CommentId = 1;
```

结果如下：



	CommentId	NsLeft	NsRight	CommentBody
1	4	6	13	我们需要查无效的输入
2	5	7	8	是的，那是个问题
3	6	9	12	好，查一下吧。
4	7	10	11	解决了

通过搜索评论6的Id在哪些节点的nsleft和nsright范围之间，就可以获取评论6及其所有祖先：

```
SELECT c2.* FROM Comment AS c1
JOIN Comment AS c2 ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.CommentId = 6;
```

	CommentId	NsLeft	NsRight	CommentBody
1	1	1	14	这个Bug的成因是什么
2	4	6	13	我们需要查无效的输入
3	6	9	12	好，查一下吧。

这种嵌套集的设计还有一个优点，就是当你想要删除一个非叶子节点时，它的后代会自动地代替被删除的节点，称为其直接祖先节点的直接后代。

嵌套集设计并不必须保存分层关系。因此当删除一个节点造成数值不连续时，并不会对树的结构产生任何影响。

嵌套集缺点：

1、查询直接父亲。

在嵌套集的设计中，这个需求的实现的思路是，给定节点c1的直接父亲是这个节点的一个祖先，且这两个节点之间不应该有任何其他的节点，因此，你可以用一个递归的外联结来查询一个节点，它就是c1的祖先，也同时是另一个节点Y的后代，随后我们使y=x就查询，直到查询返回空，即不存在这样的节点，此时y便是c1的直接父亲节点。

比如，要找到评论6的直接父节点：老实说，SQL语句又长又臭，行肯定是行，但我真的写不动了。

2、对树进行操作，比如插入和移动节点。

当插入一个节点时，你需要重新计算新插入节点的相邻兄弟节点、祖先节点和它祖先节点的兄弟，来确保它们的左右值都比这个新节点的左值大。同时，如果这个新节点是一个非叶子节点，你还

要检查它的子孙节点。

够了，够了。就凭查直接父节点都困难，这个东西就很冷门了。我确定我不会使用这种设计了。

四、闭包表

闭包表是解决分层存储一个简单而又优雅的解决方案，它记录了表中所有的节点关系，并不仅仅是直接的父子关系。

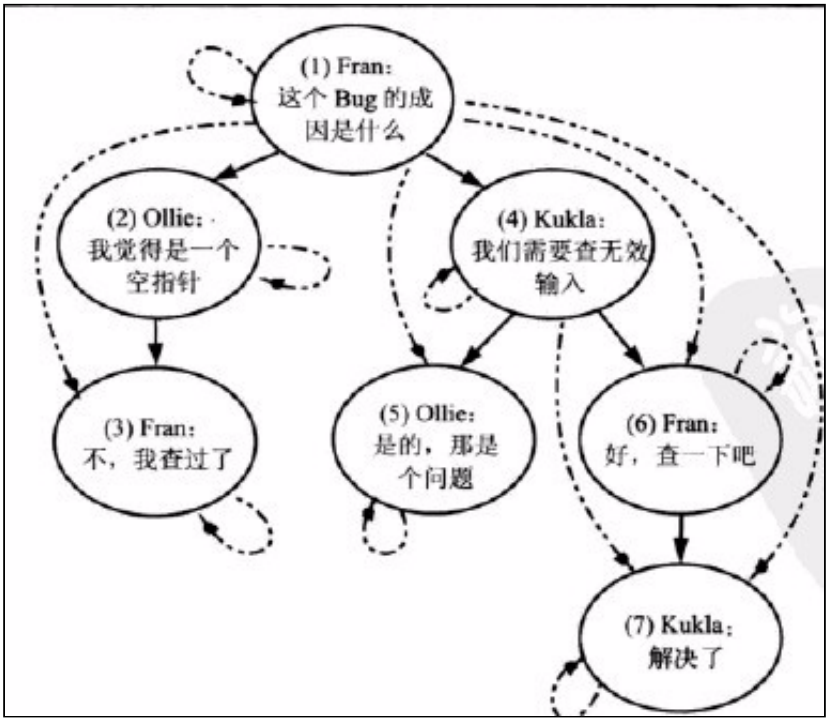
在闭包表的设计中，额外创建了一张TreePaths的表(空间换取时间)，它包含两列，每一列都是一个指向Comments中的CommentId的外键。

```
CREATE TABLE Comments(
    CommentId int PK,
    ArticleId int,
    CommentBody int,
    FOREIGN KEY(ArticleId) REFERENCES Articles(Id)
)
```

父子关系表：

```
CREATE TABLE TreePaths(
    ancestor    int,
    descendant int,
    PRIMARY KEY(ancestor,descendant),     --复合主键
    FOREIGN KEY (ancestor) REFERENCES Comments(CommentId),
    FOREIGN KEY (descendant) REFERENCES Comments(CommentId)
)
```

在这种设计中，Comments表将不再存储树结构，而是将书中的祖先-后代关系存储为TreePaths的一行，即使这两个节点之间不是直接的父子关系；同时还增加一行指向节点自己，理解不了？就是TreePaths表存储了所有祖先-后代的关系的记录。如下图：



Comment表：

	CommentId	CommentBody
▶	1	这个Bug的成因…
	2	我觉得是一个…
	3	不是，我查过了
	4	我们需要查无…
	5	是的，那是个…
	6	好，查一下吧。
	7	解决了
*	NULL	NULL

TreePaths表：

KISSDODOG-PC.Test - dbo.TreePaths

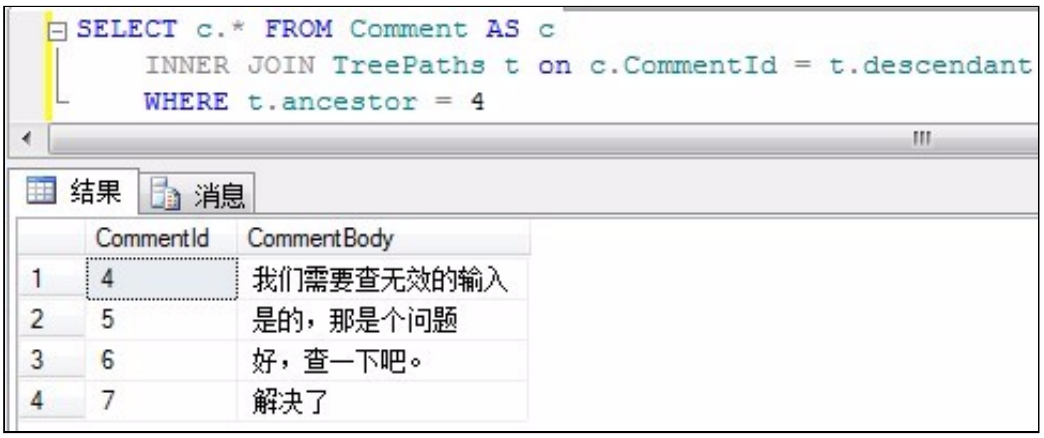
	ancestor	descendant
▶	1	1
	1	2
	1	3
	1	4
	1	5
	1	6
	1	7
	2	2
	2	3
	3	3
	4	4
	4	5
	4	6
	4	7
	5	5
	6	6
	6	7
	7	7
*	NULL	NULL

优点：

1、查询所有后代节点**(**查子树**)**：

```
SELECT c.* FROM Comment AS c
    INNER JOIN TreePaths t on c.CommentId = t.descendant
    WHERE t.ancestor = 4
```

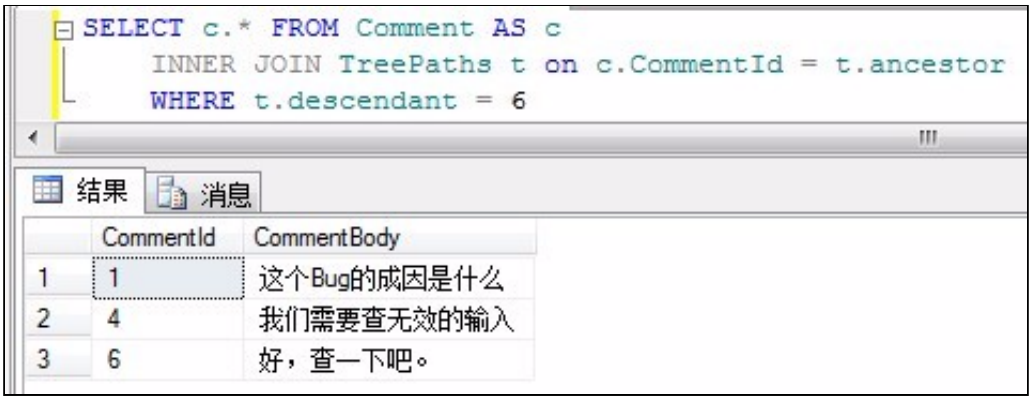
结果如下：



2、查询评论**6**的所有祖先**(**查祖先树**)**：

```
SELECT c.* FROM Comment AS c
    INNER JOIN TreePaths t on c.CommentId = t.ancestor
    WHERE t.descendant = 6
```

显示结果如下：



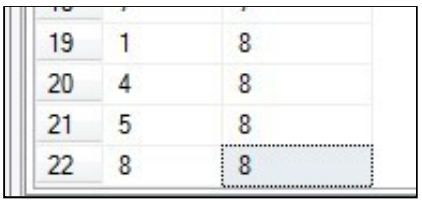
3、插入新节点：

要插入一个新的叶子节点，应首先插入一条自己到自己的关系，然后搜索TreePaths表中后代是评论5的节点，增加该节点与要插入的新节点的"祖先-后代"关系。

比如下面为插入评论5的一个子节点的TreePaths表语句：

```
INSERT INTO TreePaths(ancestor,descendant)
    SELECT t.ancestor,8
    FROM TreePaths AS t
    WHERE t.descendant = 5
    UNION ALL
    SELECT 8,8
```

执行以后：



至于Comment表那就简单得不说了。

4、删除叶子节点：

比如删除叶子节点7,应删除所有TreePaths表中后代为7的行：

```
DELETE FROM TreePaths WHERE descendant = 7
```

5、删除子树：

要删除一颗完整的子树，比如评论4和它的所有后代，可删除所有在TreePaths表中的后代为4的行，以及那些以评论4的后代为后代的行：

```
DELETE FROM TreePaths
WHERE descendant
IN(SELECT descendant FROM TreePaths WHERE ancestor = 4)
```

另外，移动节点，先断开与原祖先的关系，然后与新节点建立关系的SQL语句都不难写。

另外，闭包表还可以优化，如增加一个path_length字段，自我引用为0，直接子节点为1，再一下层为2，一次类推，查询直接自子节点就变得很简单。

总结

其实，在以往的工作中，曾见过不同类型的设计，邻接表，路径枚举，邻接表路径枚举一起来的都见过。

每种设计都各有优劣，如果选择设计依赖于应用程序中哪种操作最需要性能上的优化。

下面给出一个表格，来展示各种设计的难易程度：

设计	表数量	查询子	查询树	插入	删除	引用完整性
邻接表	1	简单	简单	简单	简单	是
枚举路径	1	简单	简单	简单	简单	否
嵌套集	1	困难	简单	困难	困难	否
闭包表	2	简单	简单	简单	简单	是

1、邻接表是最方便的设计，并且很多软件开发者都了解它。并且在递归查询的帮助下，使得邻接表的查询更加高效。

2、枚举路径能够很直观地展示出祖先到后代之间的路径，但由于不能确保引用完整性，使得这个设计比较脆弱。枚举路径也使得数据的存储变得冗余。

3、嵌套集是一个聪明的解决方案，但不能确保引用完整性，并且只能使用于查询性能要求较高，而其他要求一般的场合使用它。

4、闭包表是最通用的设计，并且最灵活，易扩展，并且一个节点能属于多棵树，能减少冗余的计算时间。但它要求一张额外的表来存储关系，是一个空间换取时间的方案。
