Subject: Re: Traversing subtrees
Posted by DannyBoyPoker on Sat, 06 Apr 2013 13:14:43 GMT
View Forum Message <> Reply to Message

I might get some more pushback, about whether that table is normalized. Well, with my change, I managed integrity constraints, to be enforced, and parents and children. One might argue that life is not all about data integrity, and win. But, the table still is not normalized. Let's go back over that, what table. I mean, the single nodes table, where each row specifies one node and its parent. Which, its parent, is NULL for the root node.

Can I give this referential integrity. Yes, I did this by suggesting that childID and parentID can be foreign keys. Like this--I suggested that two tables, one for the nodes, and the other a bridging table for their edges, would be doing things more elaborately. And now, the nodes can be personnel, or assembly parts, or
locations on a map. Also, the edges table could have additional columns for edge properties. And now, whether a graph is connected, directed, whether it is a tree, whatever, one may write a procedure which tells where you can get to from here, wherever 'here' is.

So we have a nodes table, and an edges table like this:

```
CREATE TABLE edges(
  childID CHAR(1) NOT NULL,
  parentID CHAR(1) NOT NULL,
  PRIMARY KEY(childID,parentID)
);
```

So, we select * from edges and get this:

```
| childID | parentID |
+---------+----------+
| A       | C        |
| B       | E        |
| C       | D        |
| C       | F        |
+---------+----------+
```

Maybe I ought to provide a simple approach. Here is a breadth-first search, as a MySQL stored procedure:

```
CREATE PROCEDURE ListReached( IN root CHAR(1) )
BEGIN
  DECLARE rows SMALLINT DEFAULT 0;
  DROP TABLE IF EXISTS reached;
  CREATE TABLE reached (
    nodeID CHAR(1) PRIMARY KEY
  ) ENGINE=HEAP;
  INSERT INTO reached VALUES (root );
```

```
  SET rows = ROW_COUNT();
  WHILE rows > 0 DO
    INSERT IGNORE INTO reached
      SELECT DISTINCT childID
      FROM edges AS e
      INNER JOIN reached AS p ON e.parentID = p.nodeID;
    SET rows = ROW_COUNT();
    INSERT IGNORE INTO reached
      SELECT DISTINCT parentID
      FROM edges AS e
      INNER JOIN reached AS p ON e.childID = p.nodeID;
    SET rows = rows + ROW_COUNT();
  END WHILE;
  SELECT * FROM reached;
  DROP TABLE reached;
END;
```

And, let's CALL ListReached('A'), we get NodeID = A, C, D, F. Which isn't so versatile, what about giving it input parameters which tell it whether to list child, parent, all connections, whether to recognise loops.
That's the normalization part. And, one may add, delete a node. One may also change an edge. And, this an be a connected or treelike graph, or not.

I make a big deal about it, that you have to represent the root node with a NULL, I claim, that this is working with nodes and edges denormalised to one table. Now, supposedly, denormalisation can cost, big time. Is this true, in this case? Yes, and the bigger the table, the bigger the cost. You cannot edit nodes and edges separately. Also, carrying extra node information during edge computation slows performance.

I suggested that 'parent' is a set, you might reply that there is only one parent. Even though, that means that our model cannot model a family, though we speak of parents and children. Your reply, is that when you use a different tree model, you'll have to mess with the data being modelled.

But, now that's fixed.

One will now need some functions, though, like what if I want to return a node description for a parent or child ID in the 'edges' table.
Or, call it the 'familyTree' table (I suggested 'assemblies' before, as well, maybe there's no improving on 'edges').

Here:

```
CREATE FUNCTION NodeDescription( personID SMALLINT )
RETURNS CHAR(20)
BEGIN
  DECLARE NodeDescription CHAR(20) DEFAULT '';
```

```
  SELECT description INTO NodeDescription FROM edges WHERE ID=nodeID;
  RETURN NodeDescription;
END;
```

So, what can this do.

Something like this:

```
SELECT NodeDescription( childID ) AS 'reports directly to Bill Gates'
FROM edges
WHERE parentID = ( SELECT ID FROM edges WHERE description = 'Bill Gates' );
```

And you get, like, three people returned, maybe.

At the same time, the function can be used like this:

```
SELECT nodeDescription(childID) AS subordinate, nodeDescription(parentID) AS superior
FROM edges;
```

And you get a list of each subordinate, and their superior.

I suppose that I stick in a foreign key on delete cascade on update cascade, for the foreign key(parentID), in my 'edges' table.

Then, I can delete from 'edges' a particular node. Suppose, that leaves your company hierarchy, for example.
Then it's just 'delete from edges where id = 2;'

So, we have simple queries to retrieve basic facts about the tree, perhaps? Can I collect parent nodes, with their children?
Easy:

```
SELECT parentID AS Parent, GROUP_CONCAT(childID) AS Children
FROM edges JOIN nodes  ON edges.parentID=f.ID
GROUP BY parentID;
```

A list of parent, and the children of each parent. Can we retrieve subtree statistics. Do a left join. Ask for the root node, with some info, or ask for the leaf nodes, with some info. Maybe it's a list of dead people, and you have their age, or their birth and death date, at least. You can pull the average age of leaf nodes (childless) with a left join.

Inserting a new node requires no revision of existing rows. Add a new node row, then a new edges row. Deletion is a two-step, delete the edges row, delete the node row.

What about traversing subtrees.

It can be done, in a generic version, general purpose. And there are no serious scaling issues. Insert a subtree, point its top edge at an existing node as parent,and..INSERT. This idea, is flexible. There is depth-first subtree traversal, then. What about path enumeration. Doable. Want

the procedure? I'm capable of being less breezy. Seems to me, that this is framework stuff. I'm pulling this actual MySQL (there's more) from a handout that I got in a database class in graduate school. Where, we had to all implement our own database server. Not that I'm a good student--although, prof was mightily amused that I did so providing a command-line interface, as everybody else put a Java Swing interface on it. 'Old school, cool!'
