MySQL tree query performance–another look

Peter Brawley, http://www.artfulsoftware.com/

A *graph* or hierarchy is a set of *nodes* and connecting *edges*. A *tree* is a directed, acyclic graph where one node, the *root*, has no parent, and every other node has exactly one parent. A famous but biologically wrong example would be the tree of life (the "tree" of life is actually more like a web than a tree). More correct examples would be a genealogical tree without spouses or incestuous deviations, a disk directory, or a discussion board.

Once upon a time it was widely agreed that SQL could handle *no* kind of graph effectively. Now several graph models are implemented in SQL. The two most commonly used models are the *edge list* model—often called in the SQL community, somewhat misleadingly, the adjacency list model—and Joe Celko's *nested sets* model.

The edge list model is simplicity itself. The graph in Fig. 1 has six nodes {A, B, C, D, E, F} and four edges {AC, CF, CD, BE}. It consists of the tree of nodes {A, C, D, F} and edges {AC, CD, CF}, and also the minimal tree {B, E}, {BE}. A SQL edge list model needs a table for edges, and optionally a table for nodes, for example:

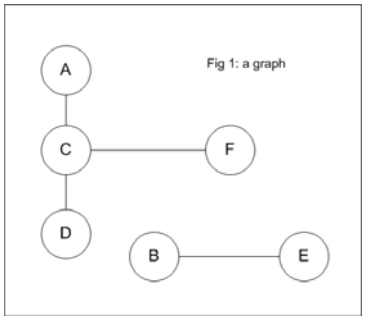

Fig 1: a graph

Listing 1:

```
CREATE TABLE nodes(nodeID CHAR(1) PRIMARY KEY);
CREATE TABLE edges(
  ID CHAR(1) NOT NULL,
  parentID CHAR(1) NOT NULL,
  PRIMARY KEY(ID,parentID)
);
INSERT INTO nodes VALUES('A'), ('B'), ('C'), ('D'), ('E'), ('F');
INSERT INTO edges VALUES ('A','C'), ('C','D'), ('C','F'), ('B','E');
```

If we know in advance how many levels an edge list has, we can write that many joins minus one to retrieve it, but we must write a new query whenever an update changes the depth of the tree. If the target SQL platform implements recursion or loops (as MySQL 5 does, but only in stored routines), we can write a general routine for all edge list trees.

Edge list tree traversal has a reputation for slowness, so Celko and others developed a nested sets model to allow non-recursive tree queries. The model represents a tree using greater-than and less-than arithmetic: starting at the root node, set *left*=1 and *right*=twice tree size, then march down left and up right for all subtrees, assigning sequential integers at each stop, until you arrive back at the root. After this, every node except the root has a *left* value greater than its parent's *left* value, and a *right* value smaller than its parent's *right* value, so many queries become child's play; for example, all the descendants of a given node have *left* and *right* values between the given node's *left* and *right* values.

However the nested sets model has a serious drawback: on average, to update just one node we have to rewrite half the tree! Building buffering gaps into *left* and *right* numeric assignments can ease the problem, but can't get rid of it.

Thus the apparent dilemma: put up with slow and complicated edge list treewalks for unproblematic edge list updates, or put up with slow and complicated nested sets updates for faster nested sets treewalks?

DB2, Oracle and SQL Server appear to resolve this by adding custom syntax for precompiled tree traversal logic. Such enhancements are missing from MySQL, so what's the least painful MySQL solution to the tree dilemma?

To investigate that question, we benchmarked edge list and nested sets treewalk query performance, and got a surprise. At least one assumption underlying the tree dilemma is doubtful.

Here is a generic procedure for retrieving any subtree of parents and their children from an edge list:

Listing 2:

```
DELIMITER go
CREATE PROCEDURE GenericEdgeListTreeWalk(
  edgeTable CHAR(64), edgeIDcol CHAR(64),
  edgeParentIDcol CHAR(64), ancestorID INT
)
BEGIN
  DECLARE r INT DEFAULT 0;
  DROP TABLE IF EXISTS subtree;
  SET @sql = Concat( 'CREATE TABLE subtree ENGINE=MyISAM SELECT ',
                     edgeIDcol,' AS childID, ',
                     edgeParentIDcol, ' AS parentID,',
                     '0 AS level FROM ',
                     edgeTable, ' WHERE ', edgeParentIDcol, '=', ancestorID );
  PREPARE stmt FROM @sql;
  EXECUTE stmt;
  DROP PREPARE stmt;
  ALTER TABLE subtree ADD PRIMARY KEY(childID,parentID);
  REPEAT
    SET @sql = Concat( 'INSERT IGNORE INTO subtree SELECT a.', edgeIDcol,
                       ',a.',edgeparentIDcol, ',b.level+1 FROM ',
                       edgeTable, ' AS a JOIN subtree AS b ON a.',
                       edgeParentIDcol, '=b.childID' );
    PREPARE stmt FROM @sql;
    EXECUTE stmt;
    SET r=Row_Count();  -- save row_count() result before DROP PREPARE loses it
    DROP PREPARE stmt;
  UNTIL r < 1 END REPEAT;
END ;
go
SELECT parentid AS Parent, group_concat(childid) AS Children
FROM subtree1000
GROUP BY parent;
```

The chart to the right shows how this performs in 32-bit MySQL 5.5, running on ordinary hardware. It retrieves a small tree of 1k nodes in a tenth of a second, a 5k tree in a half-second, a 10k tree in a second, and so on. Fairly linear.

Breadth-first treewalk in MySQL 5.5				
	Treewalk execution time			
Rows	1k	3k	5k	10k
Secs	0.1	0.2	0.4	0.7

Is this the best we can do with an edge list? A depth-first recursive search on the same data is nearly a hundred times slower with 1k rows and scales poorly. Other depth-first algorithms do as badly or worse.

Overall, then, the best known edge list treewalker can retrieve more than 10k rows in seconds, though not in milliseconds. It's not exactly a speed demon, but it's not disastrously slow either.

The nested sets model is reputed to be faster. Is it? We store nested sets trees in tables like this:

Listing 3:

```
CREATE TABLE nestedsettree (
  nodeID INT,
  leftedge INT,
  rightedge INT,
  level INT,
  KEY( nodeid, leftedge, rightedge )
);
```

For our generic routine to generate a nested sets tree from any given edge list tree, see *http://www.artfulsoftware.com/mysqlbook/sampler/mysqled1ch20.html#Listing_10*. Note that we're not benchmarking that routine here; we're benchmarking queries on nested sets tree tables.

To retrieve subtrees of parents and children from a nested sets tree, write a three-way join (other query strategies look simpler but are much slower) ...

Listing 4:

```
SELECT Parent, Group_Concat(Child ORDER BY Child) AS Children
FROM (
  SELECT master.nodeID AS Parent, child.nodeID AS Child
  FROM nestedsettree AS master
  JOIN nestedsettree AS parent
  JOIN nestedsettree AS child
    ON child.leftedge BETWEEN parent.leftedge AND parent.rightedge
  WHERE parent.leftedge > master.leftedge
    AND parent.rightedge < master.rightedge
  GROUP BY master.nodeID, child.nodeID
  HAVING COUNT(*)=1
) AS tmp
GROUP BY Parent;
```

Why does it aggregate? For treewalks, nested sets arithmetic requires aggregation.

How is the performance of this nested sets treewalk query? *Ugly*. With 1k nodes, it's 38 times slower than the edge list treewalk. With 3k nodes, it's 89 times slower. With 5k nodes it's 188 times slower, *with 10k nodes it's 310 times slower*.

Nested sets treewalk in MySQL 5.5				
Rows	1k	3k	5k	10k
Secs	3.8	17..8	103	217

How can nested sets queries perform so badly? EXPLAIN tells much of the story: the MySQL query engine cannot master how to use the index, either for

```
… n2.leftedge BETWEEN n1.leftedge AND n1.rightedge …
```

or for the GROUP BY clause. With and without the index, with various column orders in the index, and with all possible FORCE INDEX hints, the JOIN induces a table scan, and the GROUP BY and ORDER BY clauses force table scan and filesort. We can improve this a bit because our routine for generating a nested sets tree from an edge list leaves behind a tree level value for each node, but the improvement we get from that is not large.

Would bolting to Microsoft SQL Server for its built-in recursive queries help? Not enough: SQL Server's WITH ... CTE ... recursive queries are faster than nested sets queries in MySQL, but they're *an order of magnitude slower than our edge list treewalker!*

Where does all this leave us?

First, there's no getting round it. S*ome nested sets queries perform much more slowly, and scale much worse, than functionally equivalent edge list queries on equivalent tree data*. The bigger the tree, the more devastating the difference.

If you like the nested sets model, this is bad news for you, though there's a silver lining: the dilemma of having to choose between fast nested sets queries and efficient edge list updates in MySQL melts away. For some commonly required nested sets queries, there's only slow and slower.

Second, if your tree is large and you want to use nested sets, MySQL needs big help. We don't need to wait for Oracle to do it. MySQL's plugin architecture is open. Three years ago, Maurizio Nardò suggested the need in a MySQL forum. So far as I'm aware, only one group is working on such a computational plugin. Let's hope such help reaches us soon.

And finally, it adds up to a challenge: can you write nested sets queries that perform better in MySQL?

Peter Brawley,
Artful Software Development
15 Aug 2010