

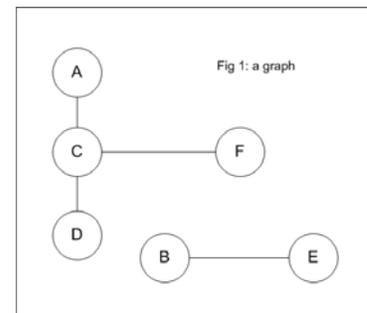
MySQL tree query performance—another look

Peter Brawley, <https://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:



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 have for years addressed this by adding “Common Table Expression” (CTEs) to optimise recursive logic like that of tree traversal. CTEs came to MariaDB with version 10.2.2 and to MySQL with version 8.0.

We benchmarked edgelist, nested sets and CTE treewalk query performance. We got two surprises—Celko’s nested sets perform terribly against thousands of rows, and in MySQL 8.0 CTEs are orders of magnitude faster than both nested sets and procedural edgelist treewalks.

Here is a generic MySQL stored 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 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;
    SELECT parentid AS Parent, group_concat(childid) AS Children
    FROM subtree1000
    GROUP BY parent;
END ;
go
delimiter ;

```

The chart to the right shows how this performs in MySQL 8.0, running on a modern laptop. It walks a small tree of 1k nodes in two tenths of a second, ten times that many nodes in a 2.8 seconds—not too bad, and fairly linear.

A depth-first recursive search on the same data is nearly a hundred times slower with 1k rows, and scales

Breadth-first edge list treewalk in MySQL 8.0				
	Treewalk execution time			
Rows	1k	3k	5k	10k
Secs	0.2	0.9	1.5	2.8

badly. Other depth-first algorithms do as poorly or worse.

Overall, then, a breadth-first non-CTE edge list treewalker can walk a tree of 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 https://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? Not good. With 1k nodes, it's three times slower than the edge list treewalk. With 3k nodes, it's seven times slower. With 5k and 10k nodes it's 30 to 50 times slower.

Nested sets treewalk in MySQL 8.0				
Rows	1k	3k	5k	10k
Secs	0.7	6.4	72.2	76.4

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, for either the joins or the Where or Group By clauses. With and without indexes, 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.

What about CTEs? In earlier 8.0 releases, CTE tree queries didn't scale too well, but with 8.0.29, they scale spectacularly well for all loads we tested. With 10k nodes, our CTE query ran 70 times faster than our non-recursive stored procedure and *about 1,800 times faster than a nested sets query on the same data.*

Listing 5:

```
with recursive treewalk as (  
  select id, 0 as parentid, 0 as level  
  from tree1000  
  where id=1 -- query for subtree root  
  union  
  select -- query for nodes  
  t.id, t.parentid, tw.level+1  
  from tree1000 t  
  join treewalk tw on t.parentid=tw.id  
)  
select parentid, group_concat(id order by id)  
from treewalk  
group by parentid;
```

CTE edge list treewalk in MySQL 8.0				
	Treewalk execution time			
Rows	1k	3k	5k	10k
Secs	0.04	0.05	0.04	0.04

Where does this leave us?

For treewalks in mature MySQL 8.0 releases, CTE queries on edge list tables are the preferred method, by far.

Before MySQL 8.0, a non-recursive stored procedure like `GenericEdgeListTreeWalk()` is much faster than Celko's nested sets model for small and large trees; the bigger the tree, the bigger the performance advantage.

*Peter Brawley,
Artful Software Development
Last updated 19 May 2022*