

Trees and Other Hierarchies in MySQL

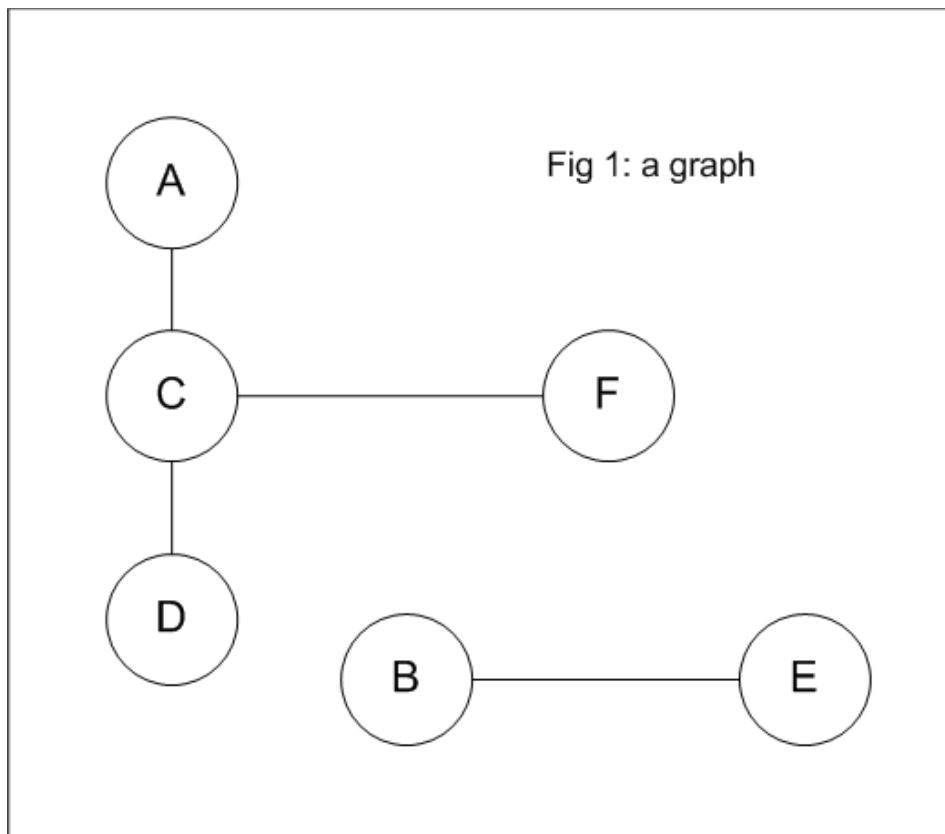
[Graphs and SQL](#) [Edge list](#) [Edge-adjacency list model of a tree](#) [Automate tree drawing](#)
[Nested sets model of a tree](#) [Edge-list model of a network](#) [Parts explosions](#)

Most non-trivial data is hierarchical. Customers have orders, which have line items, which refer to products, which have prices. Population samples have subjects, who take tests, which give results, which have sub-results and norms. Web sites have pages, which have links, which collect hits, which distribute across dates and times. With such data, we know the depth of the hierarchy before we sit down to write a query. The depth of the hierarchy of tables fixes the number of JOINS we need to write.

But if our data describes a family tree, or a browsing history, or a bill of materials, *hierarchical depth depends on the data*. We no longer know how many JOINS it will take to walk the tree. We need a different data model.

That model is the *graph* (Fig 1), which is a set of *nodes* (vertices) and the *edges* (lines or arcs) that connect them. This chapter is about how to model and query graphs in a MySQL database.

Graph theory is a branch of topology. It is the study of geometric relations which aren't changed by stretching and compression—rubber sheet geometry, some call it. Graph theory is ideal for modelling hierarchies—like family trees, browsing histories, search trees and bills of materials—whose shape and size we can't know in advance.



Let the set of nodes in Fig 1 be N , the set of edges be L , and the graph be G . Then G is the tuple or ordered pair $\{N, L\}$:

$N = \{A, B, C, D, E, F\}$
 $L = \{AC, CD, CF, BE\}$
 $G = \{N, L\}$

If the edges are *directed*, the graph is a *digraph* or directed graph. A *mixed graph* has both directed and undirected edges.

Examples of graphs are organisational charts; itineraries; route maps; parts explosions; massively multiplayer games; language rules; chat histories; network and link analysis in a wide variety of fields, for example search engines, forensics, epidemiology and telecommunications; data mining; models of chemical structure hierarchies; and [biochemical processes](#).

Graph characteristics and models

Nodes and edges : Two nodes are *adjacent* if there is an edge between them. Two edges are adjacent if they connect to a common node. In a *complete graph*, all nodes are adjacent to all other nodes.

In a digraph, the number of edges entering a node is its *indegree*; the number leaving is its *outdegree*. A node of indegree zero is a *root node*, a node of outdegree zero is a *leaf node*.

In a *weighted graph*, used for example to solve the travelling salesman problem, edges have a weight attribute. A digraph with weighted edges is a *network*.

Paths and cycles: A connected sequence of edges is a *path*, its length the number of edges traversed. Two nodes are connected if there is a path between them. If there is a path connecting every pair of nodes, the graph is a *connected graph*.

A path in which no node repeats is a *simple path*. A path which returns to its own origin without crossing itself is a *cycle* or circuit. A graph with multiple paths between at least one pair of nodes is *reconvergent*. A reconvergent graph may be *cyclic* or *acyclic*. A unit length cycle is a *loop*.

If a graph's edges intersect only at nodes, it is *planar*. Two paths having no node in common are *independent*.

Traversing graphs: There are two main approaches, *breadth-first* and *depth-first*. Breadth-first traversal visits all a node's siblings before moving on to the next level, and typically uses a *queue*. Depth-first traversal follows edges down to leaves and back before proceeding to siblings, and typically uses a *stack*.

Sparsity: A graph where the size of E approaches the maximum N^2 is *dense*. When the multiple is much smaller than N , the graph is considered *sparse*.

Trees: A *tree* is a connected graph with no cycles. It is also a graph where the indegree of the root node is 0, and the indegree of every other node is 1. A tree where every node is of outdegree ≤ 2 is a *binary tree*. A *forest* is a graph in which every connected component is a tree.

Euler paths: A path which traverses every edge in a graph exactly once is an *Euler path*. An Euler path which is a circuit is an *Euler circuit*.

If and only if every node of a connected graph has even degree, it has an *Euler circuit* (which is why the good people of Königsberg cannot go for a walk crossing each of their seven bridges exactly once). If and only if a connected graph has exactly 2 nodes with odd degree, it has a *non-circuit Euler path*. The degree of an endpoint of a non-cycle Euler path is 1 + twice the number of times the path passes through that node, so it is always odd.

Models for computing graphs

Traditionally, computer science textbooks have offered *edge lists*, *adjacency lists* and *adjacency matrices* as data structures for graphs, with algorithms implemented in languages like C, C++ and Java. More recently other models and tools have been suggested, including query languages customised for graphs.

Edge list: The simplest way to represent a graph is to list its edges: for Fig 1, the edge list is $\{AC, CD, CF, BE\}$. It is easy to add an edge to the list; deletion is a little harder.

Adjacency list: An *adjacency list* is a ragged array: for each node it lists all adjacent nodes. Thus it represents a directed graph of n nodes as a list of n lists where list i contains node j if the graph has an edge from node i to node j .

An undirected graph may be represented by having node j in the list for node i , and node i in the list for node j . Table 1 shows the adjacency list of the graph

Nodes	Adjacent nodes
A	C
B	E

in Fig 1 interpreted as undirected.

C	F,D,A
D	C
E	B
F	C

Adjacency matrix: An *adjacency matrix* represents a graph with n nodes as an $n \times n$ matrix, where the entry at (i,j) is 1 if there is an edge from node i to node j , or zero if there is not.

An adjacency matrix can represent a weighted graph using the weight as the entry, and can represent an undirected graph by using the same entry in both (i,j) and (j,i) , or by using an upper triangular matrix.

There are useful glossaries [here](#) and [here](#).

Graphs and SQL

Often standard SQL has been thought cumbersome for graph problems. Craig Mullins once [wrote](#) that "the set-based nature of SQL is not simple to master and is anathema to the OO techniques practiced by Java developers."

A few years after Mullins wrote that, SQL is everywhere, and it is increasingly applied to graph problems. DB2, Oracle and SQL Server have recursive operators for processing recursive sets, though they all work a little differently. MySQL has no such special tools, though the Open Query group has a [graph engine under development](#). Meanwhile [Joe Celko](#) and [Scott Stephens](#), among others, have published general SQL graph problem solutions that are simpler and smaller than equivalent C++, [C#](#) or Java code. Here we implement some of these ideas in MySQL.

Beware that in ports of *edge list* and *adjacency list* methods to SQL, there has been name slippage. *What's often called the adjacency list model in the SQL world is actually an edge list model*. If you follow the now-common practice in the SQL world of referring to edge lists as adjacency lists, don't be surprised to find that the model isn't quite like the adjacency list in Table 1. Here we waffle. We call them *edge-adjacency lists*.

There are also two newer kinds of models: what Joe Celko called the *nested sets* model—also known as the *interval model*—which uses greater-than/less-than arithmetic to encode tree relationships and modified preorder tree traversal (MPTT) to query them, and Tropashko's *materialised path* model, where each node is stored with its (denormalised) path to the root. So we have four main possibilities for modelling graphs in MySQL:

- *edge-adjacency lists*: based on an adaptation by EF Codd of the logic of linked lists to table structures and queries,
- *adjacency matrices*,
- *nested sets* for trees simplify some queries, but insertion and deletion are cumbersome, and
- *materialised paths*.

Here we work out how to implement edge-adjacency, nested sets and materialised path models— or parts of them—in MySQL 5&6.

The edge list

The edge list is the simplest possible representation of a graph: minimally, a single edges table where each row specifies one node and its parent (which is NULL for the root node), or more elaborately two tables, one for the nodes, the other a bridging table for their [edges](#).

In the real world, the nodes table might be a table of personnel, or assembly parts, or locations on a map. It might have many other columns of data. The edges table might also have additional columns for edge properties. The key integers of both tables might be BIGINTs.

To model [Fig 1](#), though, we keep things as simple as possible:

Listing 1

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

childID	parentID
A	C
B	E
C	D
C	F

Now, without any assumptions whatever about whether the graph is connected, whether it is directed, whether it is a tree, or whatever, how hard is it to write a *reachability* procedure, a procedure which tells us *where we can get to from here*, wherever 'here' is?

A simple approach is a [breadth-first search](#):

1. Seed the list with the starting node,
2. Add, but do not duplicate, nodes which are children of nodes in the list,
3. Add, but do not duplicate, nodes which are parents of nodes in the list,
4. Repeat steps 2 and 3 until there are no more nodes to add.

Here it is as a MySQL stored procedure. It avoids duplicate nodes by defining `reached.nodeID` as a primary key and adding reachable nodes with `INSERT IGNORE`:

Listing 2

```
DROP PROCEDURE IF EXISTS ListReached;
DELIMITER go

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;
go
DELIMITER ;
CALL ListReached('A');
+-----+
| nodeID |
+-----+
| A      |
| C      |
| D      |
| F      |
+-----+

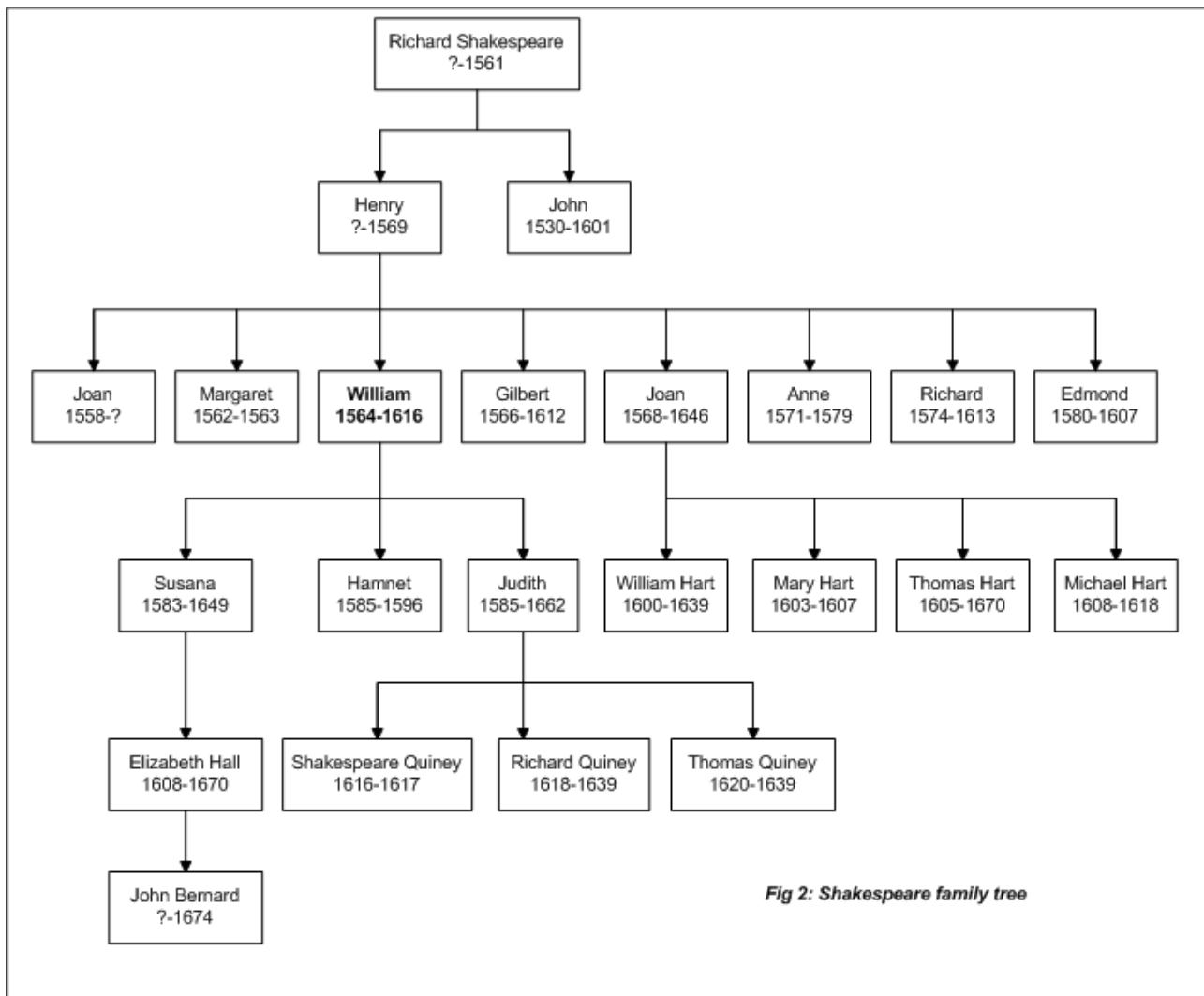
```

To make the procedure more versatile, give it input parameters which tell it whether to list child, parent or all connections, and whether to recognise loops (for example C to C).

To give the model referential integrity, use InnoDB and make `edges.childID` and `edges.parentID` foreign keys. To add or delete a node, add or delete desired single rows in `nodes` and `edges`. To change an edge, edit it. The model does not require the graph to be connected or treelike, and does not presume direction.

The edge list is basic to what SQLers often call the adjacency list model.

Edge-adjacency list model of a tree



Writers in the SQL graph literature often give solutions using single denormalised tables. Denormalisation can cost, big time. The bigger the table, the bigger the cost. You cannot edit nodes and edges separately. Carrying extra node information during edge computation slows performance. With nodes and edges denormalised to one table, you have to represent the root node with a NULL.

To avoid these difficulties, normalise trees like William Shakespeare's family tree (Fig 2) with two tables, a *nodes* table (*family*) containing information about individuals, and an *edges* table (*familytree*) with a row for each parent-child link or *edge*. Later, when we use a different tree model, we won't have to mess with the data being modelled.

Listing 3

```

-- Base data:
CREATE TABLE family (
  ID smallint(6) PRIMARY KEY AUTO_INCREMENT,
  name char(20) default '',
  siborder tinyint(4) default NULL,
  born smallint(4) unsigned default NULL,
  died smallint(4) unsigned default NULL
);
INSERT INTO family VALUES (1, 'Richard Shakespeare', NULL, NULL, 1561);
INSERT INTO family VALUES (2, 'Henry Shakespeare', 1, NULL, 1569);
INSERT INTO family VALUES (3, 'John Shakespeare', 2, 1530, 1601);
INSERT INTO family VALUES (4, 'Joan Shakespeare', 1, 1558, NULL);
INSERT INTO family VALUES (5, 'Margaret Shakespeare', 2, 1562, 1563);
INSERT INTO family VALUES (6, 'William Shakespeare', 3, 1564, 1616);

```

```

INSERT INTO family VALUES (7, 'Gilbert Shakespeare', 4, 1566, 1612);
INSERT INTO family VALUES (8, 'Joan Shakespeare', 5, 1568, 1646);
INSERT INTO family VALUES (9, 'Anne Shakespeare', 6, 1571, 1579);
INSERT INTO family VALUES (10, 'Richard Shakespeare', 7, 1574, 1613);
INSERT INTO family VALUES (11, 'Edmond Shakespeare', 8, 1580, 1607);
INSERT INTO family VALUES (12, 'Susana Shakespeare', 1, 1583, 1649);
INSERT INTO family VALUES (13, 'Hamnet Shakespeare', 1, 1585, 1596);
INSERT INTO family VALUES (14, 'Judith Shakespeare', 1, 1585, 1662);
INSERT INTO family VALUES (15, 'William Hart', 1, 1600, 1639);
INSERT INTO family VALUES (16, 'Mary Hart', 2, 1603, 1607);
INSERT INTO family VALUES (17, 'Thomas Hart', 3, 1605, 1670);
INSERT INTO family VALUES (18, 'Michael Hart', 1, 1608, 1618);
INSERT INTO family VALUES (19, 'Elizabeth Hall', 1, 1608, 1670);
INSERT INTO family VALUES (20, 'Shakespeare Quiney', 1, 1616, 1617);
INSERT INTO family VALUES (21, 'Richard Quiney', 2, 1618, 1639);
INSERT INTO family VALUES (22, 'Thomas Quiney', 3, 1620, 1639);
INSERT INTO family VALUES (23, 'John Bernard', 1, NULL, 1674);

```

-- Table which models the tree:

```

CREATE TABLE familytree (
  childID smallint NOT NULL,
  parentID smallint NOT NULL,
  PRIMARY KEY (childID, parentID);
);
INSERT INTO familytree VALUES
(2, 1), (3, 1), (4, 2), (5, 2), (6, 2), (7, 2), (8, 2), (9, 2),
(10, 2), (11, 2), (12, 6), (13, 6), (14, 6), (15, 8), (16, 8),
(17, 8), (18, 8), (19, 12), (20, 14), (21, 14), (22, 14), (23, 19);

```

(The family PK is auto-increment, but the listing is more reader-friendly when the ID values are shown.)

It will be useful to have a function that returns family.name for a parent or child ID in familytree:

Listing 4

```

-- 5.0.16 OR LATER:
SET GLOBAL log_bin_trust_function_creators=TRUE;

DROP FUNCTION IF EXISTS PersonName;
DELIMITER go

CREATE FUNCTION PersonName( personID SMALLINT )
RETURNS CHAR(20)
BEGIN
  DECLARE pname CHAR(20) DEFAULT '';
  SELECT name INTO pname FROM family WHERE ID=personID;
  RETURN pname;
END;
go
DELIMITER ;

SELECT PersonName( parentID ) AS 'Parent of William'
FROM familytree
WHERE childID = 6;
+-----+
| Parent of William |
+-----+
| Henry Shakespeare |
+-----+
SELECT PersonName( childID ) AS 'Children of William'
FROM familytree
WHERE parentID = ( SELECT ID FROM family WHERE name = 'William Shakespeare' );
+-----+
| Children of William |
+-----+
| Susana Shakespeare |
| Hamnet Shakespeare |
| Judith Shakespeare |

```



```
+-----+
SELECT PersonName(childID) AS child, PersonName(parentID) AS parent
FROM familytree;
```

child	parent
Henry Shakespeare	Richard Shakespeare
John Shakespeare	Richard Shakespeare
Joan Shakespeare	Henry Shakespeare
Margaret Shakespeare	Henry Shakespeare
William Shakespeare	Henry Shakespeare
Gilbert Shakespeare	Henry Shakespeare
Joan Shakespeare	Henry Shakespeare
Anne Shakespeare	Henry Shakespeare
Richard Shakespeare	Henry Shakespeare
Edmond Shakespeare	Henry Shakespeare
Susana Shakespeare	William Shakespeare
Hamnet Shakespeare	William Shakespeare
Judith Shakespeare	William Shakespeare
William Hart	Joan Shakespeare
Mary Hart	Joan Shakespeare
Thomas Hart	Joan Shakespeare
Michael Hart	Joan Shakespeare
Elizabeth Hall	Susana Shakespeare
Shakespeare Quiney	Judith Shakespeare
Richard Quiney	Judith Shakespeare
Thomas Quiney	Judith Shakespeare
John Bernard	Elizabeth Hall

A same-table foreign key can simplify tree maintenance:

Listing 4a

```
create table edges (
  ID int PRIMARY KEY,
  parentid int,
  foreign key(parentID) references edges(ID) ON DELETE CASCADE ON UPDATE CASCADE
) engine=innodb;
insert into edges(ID,parentID) values (1,null),(2,1),(3,1),(4,2);
select * from edges;
```

ID	parentid
1	NULL
2	1
3	1
4	2

```
delete from edges where id=2;
select * from edges;
```

ID	parentid
1	NULL
3	1

Simple queries retrieve basic facts about the tree, for example GROUP_CONCAT() collects parent nodes with their children in correct order:

Listing 5

```
SELECT parentID AS Parent, GROUP_CONCAT(childID ORDER BY siborder) AS Children
FROM familytree t
JOIN family f ON t.parentID=f.ID
GROUP BY parentID;
```

Parent	Children
1	3,2

2	4,5,6,7,8,9,10,11
6	12,13,14
8	18,17,16,15
12	19
14	22,21,20
19	23

Iterate over those comma-separated lists with a bit of application code and you have a hybrid treewalk. The *paterfamilias* is the root node, individuals with no children are the leaf nodes, and queries to retrieve subtree statistics are straightforward:

Listing 6

```
SELECT
  PersonName(ID) AS Paterfamilias,
  IFNULL(born,'?') AS Born,
  IFNULL(died,'?') AS Died
FROM family AS f
LEFT JOIN familytree AS t ON f.ID=t.childID
WHERE t.childID IS NULL;
```

Paterfamilias	Born	Died
Richard Shakespeare	?	1561

```
SELECT
  PersonName(ID) AS Childless,
  IFNULL(born,'?') AS Born,
  IFNULL(died,'?') AS Died
FROM family AS f
LEFT JOIN familytree AS t ON f.ID=t.parentID
WHERE t.parentID IS NULL;
```

Childless	Born	Died
John Shakespeare	1530	1601
Joan Shakespeare	1558	?
Margaret Shakespeare	1562	1563
Gilbert Shakespeare	1566	1612
Anne Shakespeare	1571	1579
Richard Shakespeare	1574	1613
Edmond Shakespeare	1580	1607
Hamnet Shakespeare	1585	1596
William Hart	1600	1639
Mary Hart	1603	1607
Thomas Hart	1605	1670
Michael Hart	1608	1618
Shakespeare Quiney	1616	1617
Richard Quiney	1618	1639
Thomas Quiney	1620	1639
John Bernard	?	1674

```
SELECT ROUND(AVG(died-born),2) AS 'Longevity of the childless'
FROM family AS f
LEFT JOIN familytree AS t ON f.ID=t.parentID
WHERE t.parentID IS NULL;
```

Longevity of the childless
25.86

In striking contrast with Celko's [nested sets model](#), inserting a new item in this model requires *no* revision of existing rows. We just add a new `family` row, then a new `familytree` row with IDs specifying who is parent to whom. Deletion is also a two-step: delete the `familytree` row for that child-parent link, then delete the `family` row for that

child.

Walking an edge list tree

Traversing subtrees is what gives the edge-adjacency list model its reputation for difficulty. We can't know in advance, except in the simplest of trees, how many levels of parent and child have to be queried, so we need recursion or a logically equivalent loop.

It's a natural problem for a stored procedure. Earlier editions showed a brute force [breadth-first](#) algorithm that needed three intermediary tables (it can be found in [Appendix E](#)). Here is a simpler algorithm that just seeds a result table with first-found parent-child pairs, then uses MySQL's INSERT IGNORE to add remaining pairs:

Listing 7

```
DROP PROCEDURE IF EXISTS famsubtree;
DELIMITER go
CREATE PROCEDURE famsubtree( root INT )
BEGIN
  DROP TABLE IF EXISTS famsubtree;
  CREATE TABLE famsubtree
    SELECT childID, parentID, 0 AS level
    FROM familytree
    WHERE parentID = root;
  ALTER TABLE famsubtree ADD PRIMARY KEY(childID,parentID);
  REPEAT
    INSERT IGNORE INTO famsubtree
      SELECT f.childID, f.parentID, s.level+1
      FROM familytree AS f
      JOIN famsubtree AS s ON f.parentID = s.childID;
  UNTIL Row_Count() = 0 END REPEAT;
END ;
go
DELIMITER ;
call famsubtree(1);          -- from the root you can see forever
SELECT Concat(Space(level),parentID) AS Parent, Group_Concat(childID ORDER BY childID) AS Child
FROM famsubtree
GROUP BY parentID;
```

Parent	Child
1	2,3
2	4,5,6,7,8,9,10,11
6	12,13,14
8	15,16,17,18
12	19
14	20,21,22
19	23

Simple and quick. *The logic ports to any edge list.* We can prove that right now by writing a generic version. GenericTree() just needs parameters for the name of the target table, the names of its child and parent ID columns, and the parent ID whose descendants are sought:

Listing 7a: General-purpose edge list tree walker

```
DROP PROCEDURE IF EXISTS GenericTree;
DELIMITER go
CREATE PROCEDURE GenericTree(
  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 the value
    DROP PREPARE stmt;
UNTIL r < 1 END REPEAT;
END ;
go
DELIMITER ;

```

To retrieve details like names and other data associated with node IDs, write a frontend query to join the subtree result table with the required detail table(s), for example:

```

CALL GenericTree('familytree','childID','parentID',1);
SELECT Concat(Repeat( ' ', s.level), a.name ) AS Parent, b.name AS Child
FROM subtree s
JOIN family a ON s.parentID=a.ID
JOIN family b ON s.childID=b.ID;

```

Parent	Child
Richard Shakespeare	Henry Shakespeare
Richard Shakespeare	John Shakespeare
Henry Shakespeare	Joan Shakespeare
Henry Shakespeare	Margaret Shakespeare
Henry Shakespeare	William Shakespeare
Henry Shakespeare	Gilbert Shakespeare
Henry Shakespeare	Joan Shakespeare
Henry Shakespeare	Anne Shakespeare
Henry Shakespeare	Richard Shakespeare
Henry Shakespeare	Edmond Shakespeare
William Shakespeare	Susana Shakespeare
William Shakespeare	Hamnet Shakespeare
William Shakespeare	Judith Shakespeare
Joan Shakespeare	William Hart
Joan Shakespeare	Mary Hart
Joan Shakespeare	Thomas Hart
Joan Shakespeare	Michael Hart
Susana Shakespeare	Elizabeth Hall
Judith Shakespeare	Shakespeare Quiney
Judith Shakespeare	Richard Quiney
Judith Shakespeare	Thomas Quiney
Elizabeth Hall	John Bernard

Is GenericTree() fast? You bet. On standard hardware it processes a 5,000-node tree in less than 0.5 secs—*much faster than a comparable nested sets query on the same tree!* It has no serious scaling issues. And its logic can be used to prune: call GenericTree() then delete the listed rows. Better still, write a generic tree pruner from Listing 7a and a DELETE command. To insert a subtree, prepare a table of new rows, point its top edge at an existing node as parent, and INSERT it.

The edge list treewalk is logically recursive, so how about coding it recursively? Here is a recursive [depth-first PHP](#) treewalk for the `familytree` and `family` tables:

Listing 7b: Recursive edge list subtree in PHP

```
$info = recursivesubtree( 1, $a = array(), 0 );
foreach( $info as $row )
    echo str_repeat( "&nbsp;", 2*$row[4] ), ( $row[3] > 0 ) ? "<b>{$row[1]</b>" : $row[1], "<br/>";

function recursivesubtree( $rootID, $a, $level ) {
    $childcountqry = "(SELECT COUNT(*) FROM familytree WHERE parentID=t.childID) AS childcount";
    $qry = "SELECT t.childid,f.name,t.parentid,$childcountqry,$level " .
        "FROM familytree t JOIN family f ON t.childID=f.ID " .
        "WHERE parentid=$rootID ORDER BY childcount<>0,t.childID";
    $res = mysql_qry( $qry );
    while( $row = mysql_fetch_row( $res ) ) {
        $a[] = $row;
        if( $row[3] > 0 ) $a = recursivesubtree( $row[0], $a, $level+1 );    // down before right
    }
    return $a;
}
```

A query with a subquery, a fetch loop, and a recursive call--that's all there is to it. A nice feature of this algorithm is that it writes result rows in display-ready order. To port this to MySQL, you must have set maximum recursion depth in *my.cnf/ini* or in your client:

Listing 7c: Recursive edge list subtree in MySQL

```
SET @@SESSION.max_sp_recursion_depth=25;
DROP PROCEDURE IF EXISTS recursivesubtree;
DELIMITER go
CREATE PROCEDURE recursivesubtree( iroot INT, ilevel INT )
BEGIN
    DECLARE irows,ichildid,iparentid,ichildcount,done INT DEFAULT 0;
    DECLARE cname VARCHAR(64);
    SET irows = ( SELECT COUNT(*) FROM familytree WHERE parentID=iroot );
    IF ilevel = 0 THEN
        DROP TEMPORARY TABLE IF EXISTS _descendants;
        CREATE TEMPORARY TABLE _descendants (
            childID INT, parentID INT, name VARCHAR(64), childcount INT, level INT
        );
    END IF;
    IF irows > 0 THEN
        BEGIN
            DECLARE cur CURSOR FOR
                SELECT
                    childid,parentid,f.name,
                    (SELECT COUNT(*) FROM familytree WHERE parentID=t.childID) AS childcount
                FROM familytree t JOIN family f ON t.childID=f.ID
                WHERE parentid=iroot
                ORDER BY childcount<>0,t.childID;
            DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
            OPEN cur;
            WHILE NOT done DO
                FETCH cur INTO ichildid,iparentid,cname,ichildcount;
                IF NOT done THEN
                    INSERT INTO _descendants VALUES(ichildid,iparentid,cname,ichildcount,ilevel );
                    IF ichildcount > 0 THEN
                        CALL recursivesubtree( ichildid, ilevel + 1 );
                    END IF;
                END IF;
            END WHILE;
            CLOSE cur;
        END;
    END IF;
    IF ilevel = 0 THEN
        -- Show result table headed by name that corresponds to iroot:
        SET cname = (SELECT name FROM family WHERE ID=iroot);
        SET @sql = CONCAT( 'SELECT CONCAT(REPEAT(CHAR(32),2*level),IF(childcount,UPPER(name),name))',
```

```

        ' AS ', CHAR(39), 'Descendants of ', cname, CHAR(39), ' FROM _descendants');
PREPARE stmt FROM @sql;
EXECUTE stmt;
DROP PREPARE stmt;
END IF;
END;
go
DELIMITER ;
CALL recursivesubtree(1,0);
+-----+
| Descendants of Richard Shakespeare |
+-----+
| HENRY SHAKESPEARE                |
|   Joan Shakespeare                |
|   Margaret Shakespeare            |
| WILLIAM SHAKESPEARE              |
|   SUSANA SHAKESPEARE              |
|     ELIZABETH HALL                |
|       John Bernard                |
|   Hamnet Shakespeare              |
|   JUDITH SHAKESPEARE              |
|     Shakespeare Quiney            |
|       Richard Quiney              |
|         Thomas Quiney             |
|   Gilbert Shakespeare             |
| JOAN SHAKESPEARE                  |
|   William Hart                    |
|   Mary Hart                       |
|   Thomas Hart                     |
|   Michael Hart                    |
|   Anne Shakespeare                |
|   Richard Shakespeare             |
|   Edmond Shakespeare              |
|   John Shakespeare                |
+-----+

```

In MySQL this recursive treewalk can be up to 100 times slower than GenericTree(). Its slowness is comparable to that of a MySQL version of Kendall Willet's depth-first edge list subtree algorithm:

Listing 7d: Depth-first edge list subtree

```

CREATE PROCEDURE depthfirstsubtree( iroot INT )
BEGIN
  DECLARE ilastvisited, inxt, ilastord INT;
  SET ilastvisited = iroot;
  SET ilastord = 1;
  DROP TABLE IF EXISTS descendants;
  CREATE TABLE descendants SELECT childID,parentID,-1 AS ord FROM familytree;
  UPDATE descendants SET ord=1 WHERE childID=iroot;
  this: LOOP
    SET inxt = NULL;
    SELECT MIN(childID) INTO inxt FROM descendants -- go down
    WHERE parentID = ilastvisited AND ord = -1 ;
    IF inxt IS NULL THEN -- nothing down, so go right
      SELECT MIN(d2.childID) INTO inxt
      FROM descendants d1
      JOIN descendants d2 ON d1.parentID = d2.parentID AND d1.childID < d2.childID
      WHERE d1.childID = ilastvisited;
    END IF;
    IF inxt IS NULL THEN -- nothing right. so go up
      SELECT parentID INTO inxt FROM descendants
      WHERE childID = ilastvisited AND parentID IS NOT NULL;
    END IF;
    UPDATE descendants SET ord = ilastord + 1
    WHERE childID = inxt AND ord = -1;
    IF ROW_COUNT() > 0 THEN
      SET ilastord = ilastord + 1;
    END IF;
    IF inxt IS NULL THEN
      LEAVE this;
    END IF;
  END LOOP;
END;

```

```

END IF;
SET ilastvisited = inxt;
END LOOP;
END;

```

One reason Willet's is slower is that MySQL does not permit multiple references to a temporary table in a query. When all algorithms are denied temp tables, though, this algorithm is still slower than recursion, and both are much slower than [GenericTreeQ](#).

Edge list subtree queries perform faster and are easier to write than their reputation suggests. And edge tables are flexible. For a tree describing a parts explosion rather than a family, just add columns for weight, quantity, assembly time, cost, price and so on. Reports need only aggregate column values and sums. We'll [revisit](#) this near the end of the chapter.

Enumerating paths in an edge-adjacency list

Path enumeration in an edge list tree is almost as easy as depth-first subtree traversal:

- create a table for paths,
- seed it with paths of unit length from the tree table,
- iteratively add paths till there are no more to add.

MySQL's INSERT IGNORE command simplifies the code by removing the need for a NOT EXISTS(...) clause in the INSERT ... SELECT statement. Since adjacencies are logically symmetrical, we make path direction the caller's choice, UP OR DOWN:

Listing 8

```

DROP PROCEDURE IF EXISTS ListAdjacencyPaths;
DELIMITER go
CREATE PROCEDURE ListAdjacencyPaths( IN direction CHAR(5) )
BEGIN
  DROP TABLE IF EXISTS paths;
  CREATE TABLE paths(
    start SMALLINT,
    stop SMALLINT,
    len SMALLINT,
    PRIMARY KEY(start,stop)
  ) ENGINE=HEAP;
  IF direction = 'UP' THEN
    INSERT INTO paths
      SELECT childID,parentID,1
      FROM familytree;
  ELSE
    INSERT INTO paths
      SELECT parentID,childID,1
      FROM familytree;
  END IF;
  WHILE ROW_COUNT() > 0 DO
    INSERT IGNORE INTO paths
      SELECT DISTINCT
        p1.start,
        p2.stop,
        p1.len + p2.len
      FROM paths AS p1 INNER JOIN paths AS p2 ON p1.stop = p2.start;
  END WHILE;
  SELECT start, stop, len
  FROM paths
  ORDER BY start, stop;
  DROP TABLE paths;
END;
go

```

```
DELIMITER ;
```

To find the paths from just one node, seed the `paths` table with paths from the starting node, then iteratively search a JOIN of `familytree` and `paths` for edges which will extend existing paths in the user-specified direction:

Listing 8a

```
DROP PROCEDURE IF EXISTS ListAdjacencyPathsOfNode;
DELIMITER go
CREATE PROCEDURE ListAdjacencyPathsOfNode( IN node SMALLINT, IN direction CHAR(5) )
BEGIN
    TRUNCATE paths;
    IF direction = 'UP' THEN
        INSERT INTO paths
            SELECT childID,parentID,1
            FROM familytree
            WHERE childID = node;
    ELSE
        INSERT INTO paths
            SELECT parentID,childID,1
            FROM familytree
            WHERE parentID = node;
    END IF;
    WHILE ROW_COUNT() > 0 DO
        IF direction = 'UP' THEN
            INSERT IGNORE INTO paths
                SELECT DISTINCT
                    paths.start,
                    familytree.parentID,
                    paths.len + 1
                FROM paths
                INNER JOIN familytree ON paths.stop = familytree.childID;
        ELSE
            INSERT IGNORE INTO paths
                SELECT DISTINCT
                    paths.start,
                    familytree.childID,
                    paths.len + 1
                FROM paths
                INNER JOIN familytree ON paths.stop = familytree.parentID;
        END IF;
    END WHILE;
    SELECT start, stop, len
    FROM paths
    ORDER BY start, stop;
END;
go
DELIMITER ;
```

```
CALL ListAdjacencyPathsOfNode(1, 'DOWN');
```

```
+-----+-----+-----+
| start | stop | len |
+-----+-----+-----+
| 1     | 2   | 1   |
| 1     | 3   | 1   |
| 1     | 4   | 2   |
| 1     | 5   | 2   |
| 1     | 6   | 2   |
| 1     | 7   | 2   |
| 1     | 8   | 2   |
| 1     | 9   | 2   |
| 1     | 10  | 2   |
| 1     | 11  | 2   |
| 1     | 12  | 3   |
| 1     | 13  | 3   |
| 1     | 14  | 3   |
| 1     | 15  | 3   |
| 1     | 16  | 3   |
| 1     | 17  | 3   |
```


1	18	3
1	19	4
1	20	4
1	21	4
1	22	4
1	23	5

These algorithms don't bend the brain. They perform acceptably with large trees. Querying edge-adjacency lists for subtrees and paths is less daunting than their reputation suggests.

Automate tree drawing!

Tables of numbers may be the most boring objects on earth. How to bring them alive? The Google Visualization API library has an *'OrgChart' module* that can make edge list trees look like [Fig 2](#), but each instance needs fifty or so lines of specific JavaScript code, plus an additional line of code for each row of data in the tree. Could we autogenerate that code? *Mais oui!* The module needs child node and parent node columns of data, and accepts an optional third column for info that pops up when the mouse hovers. Here is such a query for the Shakespeare family tree ...

Listing 9

```
select concat( node.ID,' ', node.name) as node,
       if( edges.parentID is null, '', concat(parent.ID, ' ',parent.name)) as parent,
       if( node.born is null, 'Birthdate unknown', concat( 'Born ', node.born )) as tooltip
from   family      as node
left join familytree as edges  on node.ID=edges.childID
left join family     as parent on edges.parentID=parent.ID;
```

and here is a PHP function which generates the HTML and JavaScript needed to paint an OrgChart for *any* tree query that returns string columns for node, parent and optionally tooltips:

Listing 9a

```
function orgchart( $qry ) {
    $cols = array(); $rows = array();
    $res = mysql_query( $qry ) or exit( mysql_error() );
    $colcount = mysql_num_fields( $res );
    if( $colcount < 2 ) exit( "Org chart needs two or three columns" );
    $rowcount = mysql_num_rows( $res );
    for( $i=0; $i<$colcount; $i++ ) $cols[] = mysql_fetch_field( $res, $i );
    while( $row = mysql_fetch_row( $res ) ) $rows[] = $row;
    echo "<html>\n<head>\n",
        " <script type='text/javascript' src='https://www.google.com/jsapi'></script>\n",
        " <script type='text/javascript'>\n",
        "   google.load('visualization', '1', {'packages':['orgchart']});\n",
        "   google.setOnLoadCallback(drawChart);\n",
        "   function drawChart() {\n",
        "       var data = new google.visualization.DataTable();\n";
    for( $i=0; $i<$colcount; $i++ ) echo "       data.addColumn('string','{$cols[$i]->name}')\n";
    echo "       data.addRows([\n";
    for( $j=0; $j<$rowcount; $j++ ) {
        $row = $rows[$j];
        $c = (( $j < $rowcount-1 ) ? "," : "" );
        echo "           ['{$row[0]}','{$row[1]}','{$row[2]}']$c\n";
    }
    echo "       ]);\n",
        "       var chart = new google.visualization.OrgChart(document.getElementById('chart_div'));\n",
        "       var options = {'size':'small','allowHtml':'true','allowCollapse':'true'};\n",
        "       chart.draw(data, options);\n",
        "   }\n",
        " </script>\n</head>\n<body>\n",
```

```
" <div id='chart_div'></div>\n",  
"</body>\n</html>";  
}
```

Nested sets model of a tree

Imagine an oval drawn round every leaf and every subtree in Fig 2, and a final oval round the entire tree. The tree is a set. Each subtree is a subset. That's the basic idea of the nested sets model.

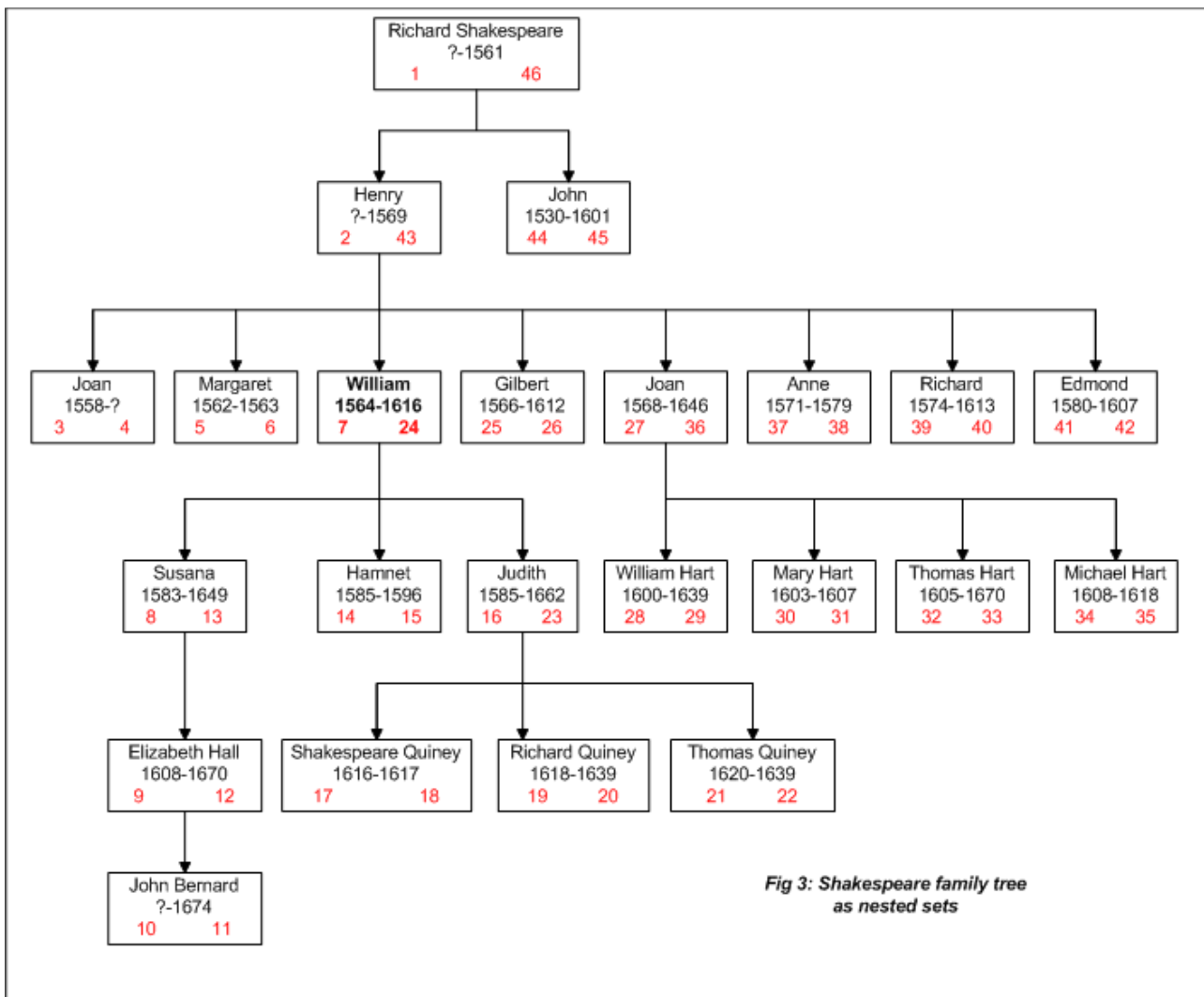
The advantage of the nested sets model is that root, leaves, subtrees, levels, tree height, ancestors, descendants and paths can be retrieved without recursion or application language code. The disadvantages are:

- initial setup of the tree table can be difficult,
- queries for parents (immediate superiors) and children (immediate subordinates) are more complicated than with an edge list model,
- insertion, updates and deletion are extremely cumbersome since they may require updates to much of the tree.

The nested sets model depends on using a [modified preorder tree traversal](#) (MPTT) [depth-first](#) algorithm to assign each node *left* and *right* integers which define the node's tree position. All nodes of a subtree have

- *left* values greater than the subtree parent's *left* value, and
- *right* values smaller than that of the subtree parent's *right* value.

so queries for subtrees are dead simple. If the numbering scheme is integer-sequential as in Fig 3, the root node receives a *left* value of 1 and a *right* value equal to twice the item count.



To see how to code nested sets using MPTT, trace the ascending integers in Fig 3, starting with **1** on the *left* side of the root node (Richard Shakespeare). Following edges downward and leftward, the *left* side of each box gets the next integer. When you reach a leaf (Joan, *left*=3), the *right* side of that box gets the next integer (4). If there is another node to the right on the same level, continue in that direction; otherwise continue up the *right* side of the subtree you just descended. When you arrive back at the root on the *right* side, you're done. Down, right and up.

A serious problem with this scheme jumps out right away: after you've written the Fig 3 tree to a table, what if historians discover an older brother or sister of Henry and John? *Every row in the tree table must be updated!*

Celko and others have proposed alternative numbering schemes to get round this problem, but the logical difficulty remains: inserts and updates can invalidate many or all rows, and no SQL CHECK or CONSTRAINT can prevent it. The nested sets model is not good for trees which require frequent updates, and is pretty much unsupportable for large updatable trees that will be accessed by many concurrent users. But as we'll see in a moment, it can be very useful indeed for reporting a tree.

How to build a nested sets representation from an edge list

Obviously, numbering a tree by hand would be error-prone, seriously impractical for large trees, so it's usually best to code the tree initially as an edge list, then use a stored procedure to translate the edge list representation to nested sets. [Celko](#)'s depth-first pushdown stack method will translate *any* edge list tree into a nested sets tree, though slowly:

1. Create a table `nestedsettree` for the tree: `node`, `leftedge`, `rightedge`, and a stack pointer (`top`),
2. Seed that table with the root node of the edge list tree,
3. Set a `nextedge` counter to 1 plus the left value of the root node, i.e. 2,
4. While that counter is less than the `rightedge` value of the root node ...
 - o insert a row for this parent's smallest unwritten child, and drop down a level, or
 - o if we're out of children, increment `rightedge`, write it to the current row, and back up a level.

This version has been improved to handle edge list trees with or without a row containing the root node and its NULL parent:

Listing 10

```
DROP PROCEDURE IF EXISTS EdgeListToNestedSet;
DELIMITER go
CREATE PROCEDURE EdgeListToNestedSet( edgeTable CHAR(64), idCol CHAR(64), parentCol CHAR(64) )
BEGIN
  DECLARE maxrightedge, rows SMALLINT DEFAULT 0;
  DECLARE trees, current SMALLINT DEFAULT 1;
  DECLARE nextedge SMALLINT DEFAULT 2;
  DECLARE msg CHAR(128);
  -- create working tree table as a copy of edgeTable
  DROP TEMPORARY TABLE IF EXISTS tree;
  CREATE TEMPORARY TABLE tree( childID INT, parentID INT );
  SET @sql = CONCAT( 'INSERT INTO tree SELECT ', idCol, ',', parentCol, ' FROM ', edgeTable );
  PREPARE stmt FROM @sql; EXECUTE stmt; DROP PREPARE stmt;
  -- initialise result table
  DROP TABLE IF EXISTS nestedsettree;
  CREATE TABLE nestedsettree (
    top SMALLINT, nodeID SMALLINT, leftedge SMALLINT, rightedge SMALLINT,
    KEY(nodeID,leftedge,rightedge)
  ) ENGINE=HEAP;
  -- root is child with NULL parent or parent which is not a child
  SET @nulls = ( SELECT Count(*) FROM tree WHERE parentID IS NULL );
  IF @nulls>1 THEN SET trees=2;
  ELSEIF @nulls=1 THEN
    SET @root = ( SELECT childID FROM tree WHERE parentID IS NULL );
    DELETE FROM tree WHERE childID=@root;
  ELSE
    SET @sql = CONCAT( 'SELECT Count(DISTINCT f.', parentcol, ') INTO @roots FROM ', edgeTable,
      ' f LEFT JOIN ', edgeTable, ' t ON f.', parentCol, '=', 't.', idCol,
      ' WHERE t.', idCol, ' IS NULL' );
    PREPARE stmt FROM @sql; EXECUTE stmt; DROP PREPARE stmt;
    IF @roots <> 1 THEN SET trees=@roots;
  ELSE
    SET @sql = CONCAT( 'SELECT DISTINCT f.', parentCol, ' INTO @root FROM ', edgeTable,
      ' f LEFT JOIN ', edgeTable, ' t ON f.', parentCol, '=', 't.',
      idCol, ' WHERE t.', idCol, ' IS NULL' );
    PREPARE stmt FROM @sql; EXECUTE stmt; DROP PREPARE stmt;
  END IF;
END IF;
IF trees<>1 THEN
  SET msg = IF( trees=0, "No tree found", "Table has multiple trees" );
  SELECT msg AS 'Cannot continue';
ELSE -- build nested sets tree
  SET maxrightedge = 2 * (1 + (SELECT + COUNT(*) FROM tree));
  INSERT INTO nestedsettree VALUES( 1, @root, 1, maxrightedge );
```

```

WHILE nextedge < maxrightedge DO
  SET rows=(SELECT Count(*) FROM nestedsettree s JOIN tree t ON s.nodeID=t.parentID AND s.top=current);
  IF rows > 0 THEN
    BEGIN
      INSERT INTO nestedsettree
        SELECT current+1, MIN(t.childID), nextedge, NULL
        FROM nestedsettree AS s
        JOIN tree AS t ON s.nodeID = t.parentID AND s.top = current;
      DELETE FROM tree
        WHERE childID = (SELECT nodeID FROM nestedsettree WHERE top=(current+1));
      SET nextedge = nextedge + 1, current = current + 1;
    END;
  ELSE
    UPDATE nestedsettree SET rightedge=nextedge, top = -top WHERE top=current;
    SET nextedge=nextedge+1, current=current-1;
  END IF;
END WHILE;
-- show result
IF (SELECT COUNT(*) FROM tree) > 0 THEN
  SELECT 'Orphaned rows remain' AS 'Error';
END IF;
DROP TEMPORARY TABLE tree;
END IF;
END;
go
DELIMITER ;
CALL EdgeListToNestedSet( 'familytree', 'childID', 'parentID' );
SELECT
  nodeID, PersonName(nodeID) AS Name,
  ABS(top) AS 'Tree Level', leftedge AS 'Left', rightedge AS 'Right'
FROM nestedsettree
ORDER BY nodeID;

```

nodeID	Name	Tree Level	Left	Right
1	Richard Shakespeare	1	1	46
2	Henry Shakespeare	2	2	43
3	John Shakespeare	2	44	45
4	Joan Shakespeare	3	3	4
5	Margaret Shakespeare	3	5	6
6	William Shakespeare	3	7	24
7	Gilbert Shakespeare	3	25	26
8	Joan Shakespeare	3	27	36
9	Anne Shakespeare	3	37	38
10	Richard Shakespeare	3	39	40
11	Edmond Shakespeare	3	41	42
12	Susana Shakespeare	4	8	13
13	Hamnet Shakespeare	4	14	15
14	Judith Shakespeare	4	16	23
15	William Hart	4	28	29
16	Mary Hart	4	30	31
17	Thomas Hart	4	32	33
18	Michael Hart	4	34	35
19	Elizabeth Hall	5	9	12
20	Shakespeare Quiney	5	17	18
21	Richard Quiney	5	19	20
22	Thomas Quiney	5	21	22
23	John Bernard	6	10	11

Verify the function with a query that generates an edge list tree from a nested sets tree:

Listing 10a:

```

SELECT a.nodeID, b.nodeID AS parent
FROM nestedsettree AS a
LEFT JOIN nestedsettree AS b ON b.leftedge = (
  SELECT MAX( leftedge )
  FROM nestedsettree AS t
  WHERE a.leftedge > t.leftedge AND a.leftedge < t.rightedge
)

```

```
ORDER BY a.nodeID;
```

For a multi-tree version of Listing 10, implementing edge list and nested sets tree models in a single table, see “Multiple nested sets trees in one table” on our [Queries page](#).

Finding a node's parent and children

In an edge list, the parent of a node is the row's parentID, and its children are the rows where that nodeID is parentID. What could be simpler? In comparison, nested sets queries for parents and their children are tortuous and slow. One way to fetch the child nodes of a given node is to INNER JOIN the nested sets tree table AS parent to itself AS child ON child.leftedge BETWEEN parent.leftedge AND parent.rightedge, then scope on the target row's leftedge and rightedge values. In the resulting list, child.nodeID values one level down occur once and are children, grandkids are two levels down and occur twice, and so on:

Listing 11

```
SELECT PersonName(child.nodeID) AS 'Descendants of William', COUNT(*) AS Generation
FROM nestedsettree AS parent
JOIN nestedsettree AS child ON child.leftedge BETWEEN parent.leftedge AND parent.rightedge
WHERE parent.leftedge > 7 AND parent.rightedge < 24      -- William's leftedge, rightedge
GROUP BY child.nodeID;
```

Descendants of William	Generation
Susana Shakespeare	1
Hamnet Shakespeare	1
Judith Shakespeare	1
Elizabeth Hall	2
Shakespeare Quiney	2
Richard Quiney	2
Thomas Quiney	2
John Bernard	3

Therefore `HAVING COUNT(t2.nodeID)=1` scopes listed descendants to the children:

Listing 11a

```
SELECT PersonName(child.nodeID) AS 'Children of William'
FROM nestedsettree AS parent
JOIN nestedsettree AS child ON child.leftedge BETWEEN parent.leftedge AND parent.rightedge
WHERE parent.leftedge > 7 AND parent.rightedge < 24
GROUP BY child.nodeID
HAVING COUNT(child.nodeID)=1
```

Children of William
Susana Shakespeare
Hamnet Shakespeare
Judith Shakespeare

Retrieving a subtree or a subset of parents requires yet another join:

Listing 11b

```
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
WHERE parent in(6,8,12,14)
GROUP BY Parent;

```

Parent	Children
6	12,13,14
8	15,16,17,18
12	19
14	20,21,22

This takes *hundreds* of times longer than a query for the same info from an edge list! An aggregating version of [Listing 19](#) is easier to write, but is an even worse performer:

Listing 11c

```

SELECT p.nodeID AS Parent, Group_Concat(c.nodeID) AS Children
FROM nestedsettree AS p
JOIN nestedsettree AS c
  ON p.leftedge = (SELECT MAX(s.leftedge) FROM nestedsettree AS s
                  WHERE c.leftedge > s.leftedge AND c.leftedge < s.rightedge)
GROUP BY Parent;

```

Parent	Children
1	2,3
2	5,6,7,8,9,10,11,4
6	12,13,14
8	15,16,17,18
12	19
14	20,21,22
19	23

Logic that is reciprocal to that of Listing 11a gets us the parent of a node:

1. retrieve its `leftedge` and `rightedge` values,
2. compute its level,
3. find the node which is one level up and has edge values outside the node's `leftedge` and `rightedge` values.

Listing 12

```

DROP PROCEDURE IF EXISTS ShowNestedSetParent;
DELIMITER go
CREATE PROCEDURE ShowNestedSetParent( node SMALLINT )
BEGIN
  DECLARE level, thisleft, thisright SMALLINT DEFAULT 0;
  -- find node edges
  SELECT leftedge, rightedge
     INTO thisleft, thisright
  FROM nestedsettree
  WHERE nodeID = node;
  -- find node level
  SELECT COUNT(n1.nodeid)
     INTO level
  FROM nestedsettree AS n1
   INNER JOIN nestedsettree AS n2
     ON n2.leftedge BETWEEN n1.leftedge AND n1.rightedge
  WHERE n2.nodeid = node
  GROUP BY n2.nodeid;
  -- find parent
  SELECT
     PersonName(n2.nodeid) AS Parent
  FROM nestedsettree AS n1
   INNER JOIN nestedsettree AS n2

```

```

    ON n2.leftedge BETWEEN n1.leftedge AND n1.rightedge
WHERE n2.leftedge < thisleft AND n2.rightedge > thisright
GROUP BY n2.nodeid
HAVING COUNT(n1.nodeid)=level-1;
END;
go
DELIMITER ;
CALL ShowNestedSetParent(6);
+-----+
| Parent |
+-----+
| Henry Shakespeare |
+-----+

```

Other queries

For some query problems, edge list and nested sets queries are equivalently simple. For example to find the tree root and leaves, compare [Listing 6](#) with:

Listing 13

```

SELECT
    name AS Paterfamilias,
    IFNULL(born,'?') AS Born,
    IFNULL(died,'?') AS Died
FROM nestedsettree AS t
INNER JOIN family AS f ON t.nodeID=f.ID
WHERE leftedge = 1;

```

```

+-----+-----+-----+
| Paterfamilias | Born | Died |
+-----+-----+-----+
| Richard Shakespeare | ? | 1561 |
+-----+-----+-----+

```

```

SELECT
    name AS 'Childless Shakespeares',
    IFNULL(born,'?') AS Born,
    IFNULL(died,'?') AS Died
FROM nestedsettree AS t
INNER JOIN family AS f ON t.nodeID=f.ID
WHERE rightedge = leftedge + 1;

```

```

+-----+-----+-----+
| Childless Shakespeares | Born | Died |
+-----+-----+-----+
| Joan Shakespeare | 1558 | ? |
| Margaret Shakespeare | 1562 | 1563 |
| John Bernard | ? | 1674 |
| Hamnet Shakespeare | 1585 | 1596 |
| Shakespeare Quiney | 1616 | 1617 |
| Richard Quiney | 1618 | 1639 |
| Thomas Quiney | 1620 | 1639 |
| Gilbert Shakespeare | 1566 | 1612 |
| William Hart | 1600 | 1639 |
| Mary Hart | 1603 | 1607 |
| Thomas Hart | 1605 | 1670 |
| Michael Hart | 1608 | 1618 |
| Anne Shakespeare | 1571 | 1579 |
| Richard Shakespeare | 1574 | 1613 |
| Edmond Shakespeare | 1580 | 1607 |
| John Shakespeare | 1530 | 1601 |
+-----+-----+-----+

```

Finding subtrees in a nested sets model requires no twisted code, no stored procedure. To retrieve the `nestedsettree` nodes in William's subtree, just ask for nodes whose `leftedge` values are greater, and whose `rightedge` values are smaller than William's:

Listing 14


```

SELECT PersonName(t.nodeID) AS Descendant
FROM nestedsettree AS s
  JOIN nestedsettree AS t ON s.leftedge < t.leftedge AND s.rightedge > t.rightedge
  JOIN family f ON s.nodeID = f.ID
WHERE f.name = 'William Shakespeare';

```

Finding a single path in the nested sets model is about as complicated as edge list path enumeration (Listings 8, 9):

Listing 15

```

SELECT
  t2.nodeID AS Node,
  PersonName(t2.nodeID) AS Person,
  (SELECT COUNT(*)
   FROM nestedsettree AS t4
   WHERE t4.leftedge BETWEEN t1.leftedge AND t1.rightedge
        AND t2.leftedge BETWEEN t4.leftedge AND t4.rightedge
  ) AS Path
FROM nestedsettree AS t1
  INNER JOIN nestedsettree AS t2 ON t2.leftedge BETWEEN t1.leftedge AND t1.rightedge
  INNER JOIN nestedsettree AS t3 ON t3.leftedge BETWEEN t2.leftedge AND t2.rightedge
WHERE t1.nodeID=(SELECT ID FROM family WHERE name='William Shakespeare')
  AND t3.nodeID=(SELECT ID FROM family WHERE name='John Bernard');

```

Node	Person	Path
6	William Shakespeare	1
12	Susana Shakespeare	2
19	Elizabeth Hall	3
23	John Bernard	4

Displaying the tree

Here the nested sets model shines. The arithmetic that was used to build the tree makes short work of summary queries. For example to retrieve a node list which preserves all parent-child relations, we need just two facts:

- listing order is the order taken in the node walk that created the tree, i.e. leftedge,
- a node's indentation depth is simply the JOIN (edge) count from root to node:

Listing 16

```

SELECT
  CONCAT( SPACE(2*COUNT(parent.nodeid)-2), PersonName(child.nodeid) )
  AS 'The Shakespeare Family Tree'
FROM nestedsettree AS parent
  INNER JOIN nestedsettree AS child
  ON child.leftedge BETWEEN parent.leftedge AND parent.rightedge
GROUP BY child.nodeid
ORDER BY child.leftedge;

```

The Shakespeare Family Tree
Richard Shakespeare
Henry Shakespeare
Joan Shakespeare
Margaret Shakespeare
William Shakespeare
Susana Shakespeare
Elizabeth Hall
John Bernard
Hamnet Shakespeare

```

Judith Shakespeare
  Shakespeare Quiney
    Richard Quiney
    Thomas Quiney
Gilbert Shakespeare
Joan Shakespeare
  William Hart
  Mary Hart
  Thomas Hart
  Michael Hart
Anne Shakespeare
Richard Shakespeare
Edmond Shakespeare
John Shakespeare

```

To retrieve only a subtree, add a query clause which restricts nodes to those whose edges are within the range of the parent node's left and right edge values, for example for William and his descendants...

```
WHERE parent.leftedge >= 7 AND parent.rightedge <=24
```

Node insertions, updates and deletions

Nested sets arithmetic also helps with insertions, updates and deletions, but the problem remains that changing just one node can require changing much of the tree.

Inserting a node in the tree requires at least two pieces of information: the `nodeID` to insert, and the `nodeID` of its parent. The model is normalised so the `nodeID` first must have been added to the parent (`family`) table. The algorithm for adding the node to the tree is:

1. increment `leftedge` by 2 in nodes where the `rightedge` value is greater than the parent's `rightedge`,
2. increment `rightedge` by 2 in nodes where the `leftedge` value is greater than the parent's `leftedge`,
3. insert a row with the given `nodeID`, `leftedge = 1 + parent's leftedge`, `rightedge = 2 + parent's leftedge`.

That's not difficult, but *all* rows will have to be updated!

Listing 17

```

DROP PROCEDURE IF EXISTS InsertNestedSetNode;
DELIMITER go
CREATE PROCEDURE InsertNestedSetNode( IN node SMALLINT, IN parent SMALLINT )
BEGIN
  DECLARE parentleft, parentright SMALLINT DEFAULT 0;
  SELECT leftedge, rightedge
    INTO parentleft, parentright
  FROM nestedsettree
  WHERE nodeID = parent;
  IF FOUND_ROWS() = 1 THEN
    BEGIN
      UPDATE nestedsettree
        SET rightedge = rightedge + 2
      WHERE rightedge > parentleft;
      UPDATE nestedsettree
        SET leftedge = leftedge + 2
      WHERE leftedge > parentleft;
      INSERT INTO nestedsettree
        VALUES ( 0, node, parentleft + 1, parentleft + 2 );
    END
  END IF;
END

```

```

END;
END IF;
END;
go
DELIMITER ;

```

"Sibling" or horizontal order is obviously significant in family trees, but may not be significant in other trees. Listing 17 adds the new node at the left edge of the sibling. To specify another position, modify the procedure to accept a third parameter for the `nodeID` which is to be to the left or right of the insertion point.

Updating a node in place requires nothing more than editing `nodeID` to point at a different parent row.

Deleting a node raises the problem of how to repair links severed by the deletion. In tree models of parts explosions, the item to be deleted is often replaced by a new item, so it can be treated like a simple `nodeID` update. In organisational boss-employee charts, though, does a colleague move over, does a subordinate get promoted, does everybody in the subtree move up a level, or does something else happen? No formula can catch all the possibilities. Listing 18 illustrates how to handle two common scenarios, move everyone up, and move someone over. All possibilities except simple replacement of the `nodeID` involve changes elsewhere in the tree.

Listing 18

```

DROP PROCEDURE IF EXISTS DeleteNestedSetNode;
DELIMITER go
CREATE PROCEDURE DeleteNestedSetNode( IN mode CHAR(7), IN node SMALLINT )
BEGIN
  DECLARE thisleft, thisright SMALLINT DEFAULT 0;
  SELECT leftedge, rightedge
    INTO thisleft, thisright
  FROM nestedsettree
  WHERE nodeID = node;
  IF mode = 'PROMOTE' THEN
    BEGIN
      DELETE FROM nestedsettree
      WHERE nodeID = node;
      UPDATE nestedsettree
        SET leftedge = leftedge - 1, rightedge = rightedge - 1
      WHERE leftedge BETWEEN thisleft AND thisright;
      UPDATE nestedsettree
        SET rightedge = rightedge - 2
      WHERE rightedge > thisright;
      UPDATE nestedsettree
        SET leftedge = leftedge - 2
      WHERE leftedge > thisright;
    END;
  ELSEIF mode = 'REPLACE' THEN
    BEGIN
      UPDATE nestedsettree
        SET leftedge = thisleft - 1, rightedge = thisright
      WHERE leftedge = thisleft + 1;
      UPDATE nestedsettree
        SET rightedge = rightedge - 2
      WHERE rightedge > thisleft;
      UPDATE nestedsettree
        SET leftedge = leftedge - 2
      WHERE leftedge > thisleft;
      DELETE FROM nestedsettree
      WHERE nodeID = node;
    END;
  END IF;
END;
go
DELIMITER ;

```

Nesteds set model summary

Some nested sets queries are quicker than their edge list counterparts, some aren't. Given the concurrency nightmare which nested sets impose for inserts and deletions, it is reasonable to reserve the nested sets model for fairly static trees whose users are mostly interested in querying subtrees. You could think of the nested sets model as a specialised OLAP tool: maintain an OLTP tree in an edge list representation, and build a nested sets OLAP table when certain reports are needed.

If you will be using the nested sets model, you may be converting back and forth with edge list models, so here is a simple query which generates an edge list from a nested sets tree:

Listing 19

```
SELECT
  p.nodeID AS parentID,
  c.nodeID AS childID
FROM nestedsettree AS p
  INNER JOIN nestedsettree AS c
  ON p.leftedge = (SELECT MAX(s.leftedge)
                  FROM nestedsettree AS s
                  WHERE c.leftedge > s.leftedge
                     AND c.leftedge < s.rightedge)
ORDER BY p.nodeID;
```

Edge list model of a non-tree graph

Many graphs are not trees. Imagine a small airline which has just acquired licences for flights no longer than 6,000 km between Los Angeles (LAX), New York (JFK), Heathrow in London, Charles de Gaulle in Paris, Amsterdam-Schiphol, Arlanda in Sweden, and Helsinki-Vantaa. You have been asked to compute the shortest possible one-way routes that do not deviate more than 90° from the direction of the first hop—roughly, one-way routes and no circuits.

Airports are *nodes*, flights are *edges*, routes are *paths*. We will need three tables.

Airports (nodes)

To identify an airport we need its code, location name, latitude and longitude. Latitude and longitude are usually given as degrees, minutes and seconds, north or south of the equator, east or west of Greenwich. To hide details that aren't directly relevant to nodes and edges, code latitude and longitude as simple reals where longitudes west of Greenwich and latitudes south of the equator are negative, whilst longitudes east of Greenwich and latitudes north of the equator are positive:

Listing 20

```
CREATE TABLE airports (
  code char(3) NOT NULL,
  city char(100) default NULL,
  latitude float NOT NULL,
  longitude float NOT NULL,
  PRIMARY KEY (code)
) ENGINE=MyISAM;

INSERT INTO airports VALUES ('JFK', 'New York, NY', 40.75, -73.97);
```

```

INSERT INTO airports VALUES ('LAX', 'Los Angeles, CA', 34.05, -118.22);
INSERT INTO airports VALUES ('LHR', 'London, England', 51.5, -0.45);
INSERT INTO airports VALUES ('HEL', 'Helsinki, Finland', 60.17, 24.97);
INSERT INTO airports VALUES ('CDG', 'Paris, France', 48.86, 2.33);
INSERT INTO airports VALUES ('STL', 'St Louis, MO', 38.63, -90.2);
INSERT INTO airports VALUES ('ARN', 'Stockholm, Sweden', 59.33, 18.05);

```

Flights (edges)

The model attaches two weights to flights: distance and direction.

We need a method of calculating the Great Circle Distance—the geographical distance between any two cities - another natural job for a stored function. The distance calculation

- converts to radians the degree coordinates of any two points on the earth's surface,
- calculates the angle of the arc subtended by the two points, and
- converts the result, also in radians, to surface (circumferential) kilometres (1 radian=6,378.388 km).

Listing 21

```

SET GLOBAL log_bin_trust_function_creators=TRUE; -- since 5.0.16
DROP FUNCTION IF EXISTS GeoDistKM;
DELIMITER go
CREATE FUNCTION GeoDistKM( lat1 FLOAT, lon1 FLOAT, lat2 FLOAT, lon2 FLOAT ) RETURNS float
BEGIN
    DECLARE pi, q1, q2, q3 FLOAT;
    SET pi = PI();
    SET lat1 = lat1 * pi / 180;
    SET lon1 = lon1 * pi / 180;
    SET lat2 = lat2 * pi / 180;
    SET lon2 = lon2 * pi / 180;
    SET q1 = COS(lon1-lon2);
    SET q2 = COS(lat1-lat2);
    SET q3 = COS(lat1+lat2);
    SET rads = ACOS( 0.5*((1.0+q1)*q2 - (1.0-q1)*q3) );
    RETURN 6378.388 * rads;
END;
go
DELIMITER ;

```

That takes care of flight distances. Flight direction is, approximately, the arctangent (ATAN) of the difference between `flights.depart` and `flights.arrive` latitudes and longitudes. Now we can seed the airline's `flights` table with one-hop flights up to 6,000 km long:

Listing 22

```

CREATE TABLE flights (
    id INT PRIMARY KEY AUTO_INCREMENT,
    depart CHAR(3),
    arrive CHAR(3),
    distance DECIMAL(10,2),
    direction DECIMAL(10,2)
) ENGINE=MYISAM;

INSERT INTO flights
SELECT
    NULL,
    depart.code,
    arrive.code,

```

```

ROUND(GeoDistKM(depart.latitude,depart.longitude,arrive.latitude,arrive.longitude),2),
ROUND(DEGREES(ATAN(arrive.latitude-depart.latitude,arrive.longitude-depart.longitude)),2)
FROM airports AS depart
INNER JOIN airports AS arrive ON depart.code <> arrive.code
HAVING Km <= 6000;

```

```
SELECT * FROM flights;
```

id	depart	arrive	distance	direction
1	LAX	JFK	3941.18	8.61
2	LHR	JFK	5550.77	-171.68
3	CDG	JFK	5837.46	-173.93
4	STL	JFK	1408.11	7.44
5	JFK	LAX	3941.18	-171.39
6	STL	LAX	2553.37	-170.72
7	JFK	LHR	5550.77	8.32
8	HEL	LHR	1841.91	-161.17
9	CDG	LHR	354.41	136.48
10	ARN	LHR	1450.12	-157.06
11	LHR	HEL	1841.91	18.83
12	CDG	HEL	1912.96	26.54
13	ARN	HEL	398.99	6.92
14	JFK	CDG	5837.46	6.07
15	LHR	CDG	354.41	-43.52
16	HEL	CDG	1912.96	-153.46
17	ARN	CDG	1545.23	-146.34
18	JFK	STL	1408.11	-172.56
19	LAX	STL	2553.37	9.28
20	LHR	ARN	1450.12	22.94
21	HEL	ARN	398.99	-173.08
22	CDG	ARN	1545.23	33.66

The distances agree approximately with public information sources for flight lengths. For a pair of airports A and B not very near the poles, the error in calculating direction using ATAN(), is small. To remove that error, instead of ATAN() use a formula from spherical trigonometry (for example one of the formulas at <http://www.dynagen.co.za/eugene/where/formula.html>).

Routes (paths)

A route is a *path* along one or more of these *edges*, so `flights:routes` is a 1:many relationship. For simplicity, though, we denormalise our representation of routes with a variation of the [materialised path model](#) to store all the hops of one route as a list of flights in one `routes.route` column. The column `routes.route` is the sequence of airports, from first departure to final arrival, the column `routes.hops` is the number of hops in that route, and the column `routes.direction` is the direction:

Listing 23

```

CREATE TABLE routes (
  id INT PRIMARY KEY AUTO_INCREMENT,
  depart CHAR(3),
  arrive CHAR(3),
  hops SMALLINT,
  route CHAR(50),
  distance DECIMAL(10,2),
  direction DECIMAL(10,2)
) ENGINE=MYISAM;

```

Starting with an empty `routes` table, how do we populate it with the shortest routes between all ordered pairs of airports?

1. Insert all 1-hop flights from the `flights` table.
2. Add in the set of shortest multi-hop routes for all pairs of airports which don't have rows in the `flights` table.

For 1-hop flights we just write

Listing 24

```
INSERT INTO routes
SELECT
  NULL,
  depart,
  arrive,
  1,
  CONCAT(depart,',',arrive),
  distance,
  direction
FROM flights;
```

NULL being the placeholder for the auto-incrementing `id` column.

For multi-hop routes, we iteratively add in sets of all allowed 2-hop, 3-hop, ... n-hop routes, replacing longer routes by shorter routes as we find them, until there is nothing more to add or replace. That also breaks down to two logical steps: add hops to build the set of next allowed routes, and update longer routes with shorter ones.

Next allowed routes

The set of next allowed routes is the set of shortest routes that can be built by adding, to existing routes, flights which leave from the last arrival airport of an existing route, which arrive at an airport which is not yet in the given route, and which stay within $\pm 90^\circ$ of the route's initial compass direction. That is, every new route is a JOIN between routes and flights in which

- `depart = routes.depart,`
- `arrive = flights.arrive,`
- `flights.depart = routes.arrive,`
- `distance = MIN(routes.distance + flights.distance),`
- `LOCATE(flights.arrive,routes.route) = 0,`
- `flights.direction+360 > routes.direction+270 AND`
`flights.direction+360 < routes.direction+450`

This is a natural logical unit of work for a VIEW:

Listing 25

```
CREATE OR REPLACE VIEW nextroutes AS
SELECT
  routes.depart,
  flights.arrive,
  routes.hops+1 AS hops,
  CONCAT(routes.route, ',', flights.arrive) AS route,
  MIN(routes.distance + flights.distance) AS distance,
  routes.direction
FROM routes INNER JOIN flights
  ON routes.arrive = flights.depart
  AND LOCATE(flights.arrive,routes.route) = 0
WHERE flights.direction+360>routes.direction+270
  AND flights.direction+360<routes.direction+450
GROUP BY depart,arrive;
```

How to add these new hops to routes? In standard SQL, this variant on a query by Scott Stephens should do it...

Listing 26

```
INSERT INTO routes
  SELECT NULL,depart,arrive,hops,route,distance,direction FROM nextroutes
  WHERE (nextroutes.depart,nextroutes.arrive) NOT IN (
    SELECT depart,arrive FROM routes
  );
```

but MySQL does not yet support subqueries on the table being updated. We have to use a subquery-less (and faster) version of that logic:

Listing 27

```
INSERT INTO routes
  SELECT
    NULL,
    nextroutes.depart,
    nextroutes.arrive,
    nextroutes.hops,
    nextroutes.route,
    nextroutes.distance,
    nextroutes.direction
  FROM nextroutes
  LEFT JOIN routes ON nextroutes.depart = routes.depart
    AND nextroutes.arrive = routes.arrive
  WHERE routes.depart IS NULL AND routes.arrive IS NULL;
```

Running that code right after the initial seeding from flights gives ...

```
SELECT * FROM routes;
```

id	depart	arrive	hops	route	distance	direction
1	LAX	JFK	1	LAX,JFK	3941.18	8.61
2	LHR	JFK	1	LHR,JFK	5550.77	-171.68
3	CDG	JFK	1	CDG,JFK	5837.46	-173.93
4	STL	JFK	1	STL,JFK	1408.11	7.44
5	JFK	LAX	1	JFK,LAX	3941.18	-171.39
6	STL	LAX	1	STL,LAX	2553.37	-170.72
7	JFK	LHR	1	JFK,LHR	5550.77	8.32
8	HEL	LHR	1	HEL,LHR	1841.91	-161.17
9	CDG	LHR	1	CDG,LHR	354.41	136.48
10	ARN	LHR	1	ARN,LHR	1450.12	-157.06
11	LHR	HEL	1	LHR,HEL	1841.91	18.83
12	CDG	HEL	1	CDG,HEL	1912.96	26.54
13	ARN	HEL	1	ARN,HEL	398.99	6.92
14	JFK	CDG	1	JFK,CDG	5837.46	6.07
15	LHR	CDG	1	LHR,CDG	354.41	-43.52
16	HEL	CDG	1	HEL,CDG	1912.96	-153.46
17	ARN	CDG	1	ARN,CDG	1545.23	-146.34
18	JFK	STL	1	JFK,STL	1408.11	-172.56
19	LAX	STL	1	LAX,STL	2553.37	9.28
20	LHR	ARN	1	LHR,ARN	1450.12	22.94
21	HEL	ARN	1	HEL,ARN	398.99	-173.08
22	CDG	ARN	1	CDG,ARN	1545.23	33.66
23	ARN	JFK	2	ARN,LHR,JFK	7000.89	-157.06
24	CDG	LAX	2	CDG,JFK,LAX	9778.64	-173.93
25	CDG	STL	2	CDG,JFK,STL	7245.57	-173.93
26	HEL	JFK	2	HEL,LHR,JFK	7392.68	-161.17
27	JFK	ARN	2	JFK,LHR,ARN	7000.89	8.32
28	JFK	HEL	2	JFK,LHR,HEL	7392.68	8.32
29	LAX	CDG	2	LAX,JFK,CDG	9778.64	8.61
30	LAX	LHR	2	LAX,JFK,LHR	9491.95	8.61
31	LHR	LAX	2	LHR,JFK,LAX	9491.95	-171.68
32	LHR	STL	2	LHR,JFK,STL	6958.88	-171.68
33	STL	CDG	2	STL,JFK,CDG	7245.57	7.44
34	STL	LHR	2	STL,JFK,LHR	6958.88	7.44

... adding 12 two-hop rows.

Replace longer routes with shorter ones

As we build routes with more hops, it is logically possible that the `nextroutes` view will find shorter routes for an existing `routes` pair of `depart` and `arrive`. Standard SQL for replacing existing `routes` rows with `nextroutes` rows which match `(depart, arrive)` and have shorter `distance` values would be:

Listing 28

```
UPDATE routes SET (hops,route,distance,direction) = (
  SELECT hops, route, distance, direction
  FROM nextroutes
  WHERE nextroutes.depart = routes.depart AND nextroutes.arrive = routes.arrive
)
WHERE (depart,arrive) IN (
  SELECT depart,arrive FROM nextroutes
  WHERE nextroutes.distance < routes.distance
);
```

but MySQL does not support `SET(col1,...)` syntax, and as with [Listing 7](#), MySQL does not yet accept subqueries referencing the table being updated, so we have to write more literal SQL:

Listing 29

```
UPDATE routes, nextroutes
SET
  routes.hops=nextroutes.hops,
  routes.route=nextroutes.route,
  routes.distance=nextroutes.distance,
  routes.direction=nextroutes.direction
WHERE routes.arrive=nextroutes.arrive
  AND routes.depart=nextroutes.depart
  AND nextroutes.distance < routes.distance;
```

Running this code right after the first run of Listing 27 updates no rows. To test the logic of iteration, continue running Listings 27 and 29 until no rows are being added or changed. The final result is:

```
SELECT * FROM ROUTES;
```

id	depart	arrive	hops	route	distance	direction
1	LAX	JFK	1	LAX,JFK	3941.18	8.61
2	LHR	JFK	1	LHR,JFK	5550.77	-171.68
3	CDG	JFK	1	CDG,JFK	5837.46	-173.93
4	STL	JFK	1	STL,JFK	1408.11	7.44
5	JFK	LAX	1	JFK,LAX	3941.18	-171.39
6	STL	LAX	1	STL,LAX	2553.37	-170.72
7	JFK	LHR	1	JFK,LHR	5550.77	8.32
8	HEL	LHR	1	HEL,LHR	1841.91	-161.17
9	CDG	LHR	1	CDG,LHR	354.41	136.48
10	ARN	LHR	1	ARN,LHR	1450.12	-157.06
11	LHR	HEL	1	LHR,HEL	1841.91	18.83
12	CDG	HEL	1	CDG,HEL	1912.96	26.54
13	ARN	HEL	1	ARN,HEL	398.99	6.92
14	JFK	CDG	1	JFK,CDG	5837.46	6.07
15	LHR	CDG	1	LHR,CDG	354.41	-43.52
16	HEL	CDG	1	HEL,CDG	1912.96	-153.46
17	ARN	CDG	1	ARN,CDG	1545.23	-146.34
18	JFK	STL	1	JFK,STL	1408.11	-172.56

19	LAX	STL	1	LAX,STL	2553.37	9.28
20	LHR	ARN	1	LHR,ARN	1450.12	22.94
21	HEL	ARN	1	HEL,ARN	398.99	-173.08
22	CDG	ARN	1	CDG,ARN	1545.23	33.66
23	ARN	JFK	2	ARN,LHR,JFK	7000.89	-157.06
24	CDG	LAX	2	CDG,JFK,LAX	9778.64	-173.93
25	CDG	STL	2	CDG,JFK,STL	7245.57	-173.93
26	HEL	JFK	2	HEL,LHR,JFK	7392.68	-161.17
27	JFK	ARN	2	JFK,LHR,ARN	7000.89	8.32
28	JFK	HEL	2	JFK,LHR,HEL	7392.68	8.32
29	LAX	CDG	2	LAX,JFK,CDG	9778.64	8.61
30	LAX	LHR	2	LAX,JFK,LHR	9491.95	8.61
31	LHR	LAX	2	LHR,JFK,LAX	9491.95	-171.68
32	LHR	STL	2	LHR,JFK,STL	6958.88	-171.68
33	STL	CDG	2	STL,JFK,CDG	7245.57	7.44
34	STL	LHR	2	STL,JFK,LHR	6958.88	7.44
35	ARN	LAX	3	ARN,LHR,JFK,LAX	10942.07	-157.06
36	ARN	STL	3	ARN,LHR,JFK,STL	8409.00	-157.06
37	HEL	LAX	3	HEL,LHR,JFK,LAX	11333.86	-161.17
38	HEL	STL	3	HEL,LHR,JFK,STL	8800.79	-161.17
39	LAX	ARN	3	LAX,JFK,CDG,ARN	10942.07	8.61
40	LAX	HEL	3	LAX,JFK,CDG,HEL	11333.86	8.61
41	STL	ARN	3	STL,JFK,CDG,ARN	8409.00	7.44
42	STL	HEL	3	STL,JFK,CDG,HEL	8800.79	7.44

All that's left to do is to assemble the code in a stored procedure:

Listing 30

```

DROP PROCEDURE IF EXISTS BuildRoutes;
DELIMITER go
CREATE PROCEDURE BuildRoutes()
BEGIN
    DECLARE rows INT DEFAULT 0;
    TRUNCATE routes;
    -- STEP 1, LISTING 24: SEED ROUTES WITH 1-HOP FLIGHTS
    INSERT INTO routes
        SELECT
            NULL,
            depart,
            arrive,
            1,
            CONCAT(depart,',',arrive),
            distance,
            direction
        FROM flights;
    SET rows = ROW_COUNT();
    WHILE (rows > 0) DO
        -- STEP 2, LISTINGS 25, 27: ADD NEXT SET OF ROUTES
        INSERT INTO routes
            SELECT
                NULL,
                nextroutes.depart,
                nextroutes.arrive,
                nextroutes.hops,
                nextroutes.route,
                nextroutes.distance,
                nextroutes.direction
            FROM nextroutes
            LEFT JOIN routes ON nextroutes.depart = routes.depart AND nextroutes.arrive = routes.arrive
            WHERE routes.depart IS NULL AND routes.arrive IS NULL;
        SET rows = ROW_COUNT();
    END WHILE;
END;
go
DELIMITER ;

```

In MySQL 5.0.6 or 5.0.7, BuildRoutes() fails to insert four rows:

+-----+-----+-----+-----+-----+-----+

depart	arrive	route	hops	distance	direction
ARN	LAX	ARN,LHR,JFK,LAX	3	10942.07	-157.06
ARN	STL	ARN,LHR,JFK,STL	3	8409.00	-157.06
HEL	LAX	HEL,LHR,JFK,LAX	3	11333.86	-161.17
HEL	STL	HEL,LHR,JFK,STL	3	8800.79	-161.17

That MySQL bug ([#11302](#)) was corrected in 5.0.9.

Route queries

Route queries are straightforward. How do we check that the algorithm produced no duplicate depart-arrive pairs? The following query should yield zero rows,

Listing 31

```
SELECT depart, arrive, COUNT(*)
FROM routes
GROUP BY depart, arrive
HAVING COUNT(*) > 1;
```

and does. *Reachability* queries are just as simple, for example where can we fly to from Helsinki?

Listing 32

```
SELECT *
FROM routes
WHERE depart='HEL'
ORDER BY distance;
```

id	depart	arrive	hops	route	distance	direction
21	HEL	ARN	1	HEL,ARN	398.99	-173.08
8	HEL	LHR	1	HEL,LHR	1841.91	-161.17
16	HEL	CDG	1	HEL,CDG	1912.96	-153.46
26	HEL	JFK	2	HEL,LHR,JFK	7392.68	-161.17
38	HEL	STL	3	HEL,LHR,JFK,STL	8800.79	-161.17
37	HEL	LAX	3	HEL,LHR,JFK,LAX	11333.86	-161.17

An extended edge list model is simple to implement, gracefully accepts extended attributes for nodes, edge and paths, does not unduly penalise updates, and responds to queries with reasonable speed.

Parts explosions

A bill of materials for a house would include the cement block, lumber, shingles, doors, wallboard, windows, plumbing, electrical system, heating system, and so on. Each subassembly also has a bill of materials; the heating system has a furnace, ducts, and so on. A bill of materials implosion links component pieces to a major assembly. A bill of materials explosion breaks apart assemblies and subassemblies into their component parts.

Which graph model best handles a parts explosion? Combining edge list and "nested sets" algorithms seems a natural solution.

Imagine a new company that plans to make variously sized bookcases, either packaged as do-it-yourself kits of, or assembled from sides, shelves, shelf brackets, backboards,

feet and screws. Shelves and sides are cut from planks. Backboards are trimmed from laminated sheeting. Feet are machine-carved from ready-cut blocks. Screws and shelf brackets are purchased in bulk. Here are the elements of one bookcase:

```
1 backboard, 2 x 1 m
  1 laminate
  8 screws
2 sides 2m x 30 cm
  1 plank length 4m
  12 screws
8 shelves 1 m x 30 cm (incl. top and bottom)
  2 planks
  24 shelf brackets
4 feet 4cm x 4cm
  4 cubes
  16 screws
```

which may be packaged in a box for sale at one price, or assembled as a finished product at a different price. At any time we need to be able to answer questions like

- Do we have enough parts to make the bookcases on order?
- What assemblies or packages would be most profitable to make given the current inventory?

There is no reason to break the normalising rule that item detail belongs in a *nodes* table, and graph logic belongs in an *edges* table. Edges also require a quantity attribute, for example a shelf includes four shelf brackets. Nodes and edges may also have costs and prices:

- item purchase cost,
- item assembly cost,
- assembly cost,
- assembly selling price.

In many parts problems like this one, items occur in multiple assemblies and subassemblies. The graph is not a tree. Also, it is often desirable to model multiple graphs without the table glut that would arise from giving each graph its own edges table. A simple way to solve this problem is to represent multiple graphs (assemblies) in the edges table by giving every row not only `childID` and `parentID` pointers, but a pointer which identifies the root `itemID` of the graph to which the row belongs.

So the data model is just two tables, for items (*nodes*) and for product graphs or assemblies (*edges*). Assume that the company begins with a plan to sell the 2m x 1m bookcase in two forms, assembled and kit, and that the purchasing department has bought quantities of raw materials (laminates, planks, shelf supports, screws, wood cubes, boxes). Here are the nodes (items) and edges (assemblies):

Listing 33

```
CREATE TABLE items (
  itemID INT PRIMARY KEY AUTO_INCREMENT,
  name CHAR(20) NOT NULL,
  onhand INT NOT NULL DEFAULT 0,
  reserved INT NOT NULL DEFAULT 0,
  purchasecost DECIMAL(10,2) NOT NULL DEFAULT 0,
  assemblycost DECIMAL(10,2) NOT NULL DEFAULT 0,
  price DECIMAL(10,2) NOT NULL DEFAULT 0
);
CREATE TABLE assemblies (
  assemblyID INT NOT NULL,
```

```

assemblyroot INT NOT NULL,
childID INT NOT NULL,
parentID INT NOT NULL,
quantity DECIMAL(10,2) NOT NULL,
assemblycost DECIMAL(10,2) NOT NULL,
PRIMARY KEY(assemblyID,childID,parentID)
);
INSERT INTO items VALUES -- inventory
(1,'laminare',40,0,4,0,8),
(2,'screw',1000,0,0.1,0,.2),
(3,'plank',200,0,10,0,20),
(4,'shelf bracket',400,0,0.20,0,.4),
(5,'wood cube',100,0,0.5,0,1),
(6,'box',40,0,1,0,2),
(7,'backboard',0,0,0,3,0),
(8,'side',0,0,0,8,0),
(9,'shelf',0,0,0,4,0),
(10,'foot',0,0,0,1,0),
(11,'bookcase2x30',0,0,0,10,0),
(12,'bookcase2x30 kit',0,0,0,2,0);
INSERT INTO assemblies VALUES
(1,11,1,7,1,0), -- laminate to backboard
(2,11,2,7,8,0), -- screws to backboard
(3,11,3,8,.5,0), -- planks to side
(4,11,2,8,6,0), -- screws to side
(5,11,3,9,0.25,0), -- planks to shelf
(6,11,4,9,4,0), -- shelf brackets to shelf
(7,11,5,10,1,0), -- wood cubes to foot
(8,11,2,10,1,0), -- screws to foot
(9,11,7,11,1,0), -- backboard to bookcase
(10,11,8,11,2,0), -- sides to bookcase
(11,11,9,11,8,0), -- shelves to bookcase
(12,11,10,11,4,0), -- feet to bookcase
(13,12,1,7,1,0), -- laminate to backboard
(14,12,2,7,8,0), -- screws to backboard
(15,12,3,8,0.5,0), -- planks to side
(16,12,2,8,6,0), -- screws to sides
(17,12,3,9,0.25,0), -- planks to shelf
(18,12,4,9,4,0), -- shelf brackets to shelves
(19,12,5,10,1,0), -- wood cubes to foot
(20,12,2,10,1,0), -- screws to foot
(21,12,7,12,1,0), -- backboard to bookcase kit
(22,12,8,12,2,0), -- sides to bookcase kit
(23,12,9,12,8,0), -- shelves to bookcase kit
(24,12,10,12,4,0), -- feet to bookcase kit
(25,12,6,12,1,0); -- container box to bookcase kit

```

Now, we want a parts list, a bill of materials, which will list show parent-child relationships and quantities, and sum the costs. Could we adapt the depth-first "nested sets" treewalk algorithm ([Listing 10](#)) to this problem even though our graph is not a tree and our sets are not properly nested? Yes indeed. We just have to modify the treewalk to handle multiple parent nodes for any child node, and add code to percolate costs and quantities up the graph. Navigation remains simple using `leftedge` and `rightedge` values. This is just the sort of problem the Celko algorithm is good for: reporting!

Listing 34

```

DROP PROCEDURE IF EXISTS ShowBOM;
DELIMITER go
CREATE PROCEDURE ShowBOM( IN root INT )
BEGIN
  DECLARE thischild, thisparent, rows, maxrightedge INT DEFAULT 0;
  DECLARE thislevel, nextedgenum INT DEFAULT 1;
  DECLARE thisqty, thiscost DECIMAL(10,2);

  -- Create and seed intermediate table:
  DROP TABLE IF EXISTS edges;
  CREATE TABLE edges (

```

```

childID smallint NOT NULL,
parentID smallint NOT NULL,
PRIMARY KEY (childID, parentID)
) ENGINE=HEAP;
INSERT INTO edges
SELECT childID,parentID
FROM assemblies
WHERE assemblyRoot = root;
SET maxrightedge = 2 * (1 + (SELECT COUNT(*) FROM edges));
-- Create and seed result table:
DROP TABLE IF EXISTS bom;
CREATE TABLE bom (
level SMALLINT,
nodeID SMALLINT,
parentID SMALLINT,
qty DECIMAL(10,2),
cost DECIMAL(10,2),
leftedge SMALLINT,
rightedge SMALLINT
) ENGINE=HEAP;
INSERT INTO bom
VALUES( thislevel, root, 0, 0, 0, nextedgenum, maxrightedge );
SET nextedgenum = nextedgenum + 1;
WHILE nextedgenum < maxrightedge DO
-- How many children of this node remain in the edges table?
SET rows = (
SELECT COUNT(*)
FROM bom AS s
INNER JOIN edges AS t ON s.nodeID=t.parentID AND s.level=thislevel
);
IF rows > 0 THEN
-- There is at least one child edge.
-- Compute qty and cost, insert into bom, delete from edges.
BEGIN
-- Alas MySQL nulls MIN(t.childid) when we combine the next two queries
SET thischild = (
SELECT MIN(t.childID)
FROM bom AS s
INNER JOIN edges AS t ON s.nodeID=t.parentID AND s.level=thislevel
);
SET thisparent = (
SELECT DISTINCT t.parentID
FROM bom AS s
INNER JOIN edges AS t ON s.nodeID=t.parentID AND s.level=thislevel
);
SET thisqty = (
SELECT quantity FROM assemblies
WHERE assemblyroot = root
AND childID = thischild
AND parentID = thisparent
);
SET thiscost = (
SELECT a.assemblycost + (thisqty * (i.purchasecost + i.assemblycost ))
FROM assemblies AS a
INNER JOIN items AS i ON a.childID = i.itemID
WHERE assemblyroot = root
AND a.parentID = thisparent
AND a.childID = thischild
);
INSERT INTO bom
VALUES(thislevel+1, thischild, thisparent, thisqty, thiscost, nextedgenum, NULL);
DELETE FROM edges
WHERE childID = thischild AND parentID=thisparent;
SET thislevel = thislevel + 1;
SET nextedgenum = nextedgenum + 1;
END;
ELSE
BEGIN
-- Set rightedge, remove item from edges
UPDATE bom
SET rightedge=nextedgenum, level = -level

```

```

WHERE level = thislevel;
SET thislevel = thislevel - 1;
SET nextedgenum = nextedgenum + 1;
END;
END IF;
END WHILE;
SET rows := ( SELECT COUNT(*) FROM edges );
IF rows > 0 THEN
  SELECT 'Orphaned rows remain';
ELSE
  -- Total
  SET thiscost = (SELECT SUM(qty*cost) FROM bom);
  UPDATE bom
  SET qty = 1, cost = thiscost
  WHERE nodeID = root;
  -- Show the result
  SELECT
    CONCAT(Space(Abs(level)*2), ItemName(nodeid,root)) AS Item,
    ROUND(qty,2) AS Qty,
    ROUND(cost, 2) AS Cost
  FROM bom
  ORDER BY leftedge;
END IF;
END;
go
DELIMITER ;

```

```

-- Function used by ShowBOM() to retrieve bom item names:
DROP FUNCTION IF EXISTS ItemName;
SET GLOBAL log_bin_trust_function_creators=TRUE;
DELIMITER go
CREATE FUNCTION ItemName( id INT, root INT ) RETURNS CHAR(20)
BEGIN
  DECLARE s CHAR(20) DEFAULT '';
  SELECT name INTO s FROM items WHERE itemid=id;
  RETURN IF( id = root, UCASE(s), s );
END;
go
DELIMITER ;
CALL ShowBOM(11);

```

```

+-----+-----+-----+
| Item          | Qty  | Cost  |
+-----+-----+-----+
| BOOKCASE2X30 | 1.0  | 327.93 |
|   backboard  | 1.0  | 3.00  |
|   laminate   | 1.0  | 4.00  |
|   screw      | 8.0  | 0.80  |
|   side       | 2.0  | 16.00 |
|   screw      | 6.0  | 0.60  |
|   plank      | 0.5  | 5.00  |
|   shelf      | 8.0  | 32.00 |
|   plank      | 0.3  | 2.50  |
|   shelf bracket | 4.0  | 0.80  |
|   foot       | 4.0  | 4.00  |
|   screw      | 1.0  | 0.10  |
|   wood cube  | 1.0  | 0.50  |
+-----+-----+-----+

```

With `showBOM()` in hand, it's easy to compare costs of assemblies and subassemblies. By adding price columns, we can do the same for prices and profit margins. And now that MySQL has re-enabled prepared statements in stored procedures, it will be relatively easy to write a more general version of `showBOM()`. We leave that to you.

Shorter and sweeter

But `ShowBOM()` is not the small, efficient bit of nested sets reporting code we were hoping

for. There is a simpler solution: hide graph cycles from the edges table by making them references to rows in a nodes table, so we can treat the edges table like a tree; then apply a breadth-first [edge-list subtree algorithm](#) to generate the Bill of Materials. Again assume a cabinetmaking company making bookcases (with a different costing model). For clarity, skip inventory tracking for now. An items table `ww_nodes` tracks purchased and assembled bookcase elements with their individual costs, and an assemblies/edges `ww_edges` table tracks sets of edges that combine to make products.

Listing 35: DDL for a simpler parts explosion

```
DROP TABLE IF EXISTS ww_nodes;
CREATE TABLE ww_nodes (
    nodeID int,
    description CHAR(50),
    cost decimal(10,2)
);
INSERT INTO ww_nodes VALUES (1,'finished bookcase',10);
INSERT INTO ww_nodes VALUES (2,'backboard2x1',1);
INSERT INTO ww_nodes VALUES (3,'laminatex1',8);
INSERT INTO ww_nodes VALUES (4,'screw',.10);
INSERT INTO ww_nodes VALUES (5,'side',4);
INSERT INTO ww_nodes VALUES (6,'plank',20);
INSERT INTO ww_nodes VALUES (7,'shelf',4);
INSERT INTO ww_nodes VALUES (8,'shelf bracket',.5);
INSERT INTO ww_nodes VALUES (9,'feet',1);
INSERT INTO ww_nodes VALUES (10,'cube4cmx4cm',1);
INSERT INTO ww_nodes VALUES (11,'bookcase kit',2);
INSERT INTO ww_nodes VALUES (12,'carton',1);

DROP TABLE IF EXISTS ww_edges;
CREATE TABLE ww_edges (
    rootID INT,
    nodeID int,
    parentnodeID int,
    qty decimal(10,2)
);
INSERT INTO ww_edges VALUES (1,1,null,1);
INSERT INTO ww_edges VALUES (1,2,1,1);
INSERT INTO ww_edges VALUES (1,3,2,1);
INSERT INTO ww_edges VALUES (1,4,2,8);
INSERT INTO ww_edges VALUES (1,5,1,2);
INSERT INTO ww_edges VALUES (1,6,5,1);
INSERT INTO ww_edges VALUES (1,4,5,12);
INSERT INTO ww_edges VALUES (1,7,1,8);
INSERT INTO ww_edges VALUES (1,6,7,.5);
INSERT INTO ww_edges VALUES (1,8,7,4);
INSERT INTO ww_edges VALUES (1,9,1,4);
INSERT INTO ww_edges VALUES (1,10,9,1);
INSERT INTO ww_edges VALUES (1,4,9,1);

INSERT INTO ww_edges VALUES (11,11,null,1);
INSERT INTO ww_edges VALUES (11,2,11,1);
INSERT INTO ww_edges VALUES (11,3,2,1);
INSERT INTO ww_edges VALUES (11,4,2,8);
INSERT INTO ww_edges VALUES (11,5,11,2);
INSERT INTO ww_edges VALUES (11,6,5,1);
INSERT INTO ww_edges VALUES (11,4,5,12);
INSERT INTO ww_edges VALUES (11,7,11,8);
INSERT INTO ww_edges VALUES (11,6,7,.5);
INSERT INTO ww_edges VALUES (11,8,7,4);
INSERT INTO ww_edges VALUES (11,9,11,4);
INSERT INTO ww_edges VALUES (11,10,9,1);
INSERT INTO ww_edges VALUES (11,4,9,11);
INSERT INTO ww_edges VALUES (11,12,11,1);
```

Here is an adaptation of the breadth-first edge list algorithm to retrieve a Bill of Materials for a product identified by a `rootID`:

- Initialise a level-tracking variable to zero.
- Seed a temp reporting table with the rootID of the desired product.
- While rows are being retrieved, increment the level tracking variable and add rows to the temp table whose parentnodeIDs are nodes at the current level.
- Print the BOM ordered by path with indentation proportional to tree level.

Listing 36: A simpler parts explosion

```

DROP PROCEDURE IF EXISTS ww_bom;
DELIMITER go
CREATE PROCEDURE ww_bom( root INT )
BEGIN
  DECLARE lev INT DEFAULT 0;
  DECLARE totalcost DECIMAL( 10,2);
  DROP TABLE IF EXISTS temp;
  CREATE TABLE temp                                     -- initialise temp table with root node
  SELECT
    e.nodeID AS nodeID,
    n.description AS Item,
    e.parentnodeID,
    e.qty,
    n.cost AS nodecost,
    e.qty * n.cost AS cost,
    0 as level,                                       -- tree level
    CONCAT(e.nodeID,') AS path                       -- path to this node as a string
  FROM ww_nodes n
  JOIN ww_edges e USING(nodeID)                     -- root node
  WHERE e.nodeID = root AND e.parentnodeID IS NULL;
  WHILE FOUND_ROWS() > 0 DO
    BEGIN
      SET lev = lev+1;                               -- increment level
      INSERT INTO temp                               -- add children of this level
      SELECT
        e.nodeID,
        n.description AS Item,
        e.parentnodeID,
        e.qty,
        n.cost AS nodecost,
        e.qty * n.cost AS cost,
        lev,
        CONCAT(t.path,',',e.nodeID)
      FROM ww_nodes n
      JOIN ww_edges e USING(nodeID)
      JOIN temp t ON e.parentnodeID = t.nodeID
      WHERE e.rootID = root AND t.level = lev-1;
    END;
  END WHILE;
  WHILE lev > 0 DO                                   -- percolate costs up the graph
    BEGIN
      SET lev = lev - 1;
      DROP TABLE IF EXISTS tempcost;
      CREATE TABLE tempcost                         -- compute child cost
      SELECT p.nodeID, SUM(c.nodecost*c.qty) AS childcost
      FROM temp p
      JOIN temp c ON p.nodeid=c.parentnodeid
      WHERE c.level=lev
      GROUP by p.nodeid;
      UPDATE temp JOIN tempcost USING(nodeID)        -- update parent item cost
      SET nodecost = nodecost + tempcost.childcost;
      UPDATE temp SET cost = qty * nodecost         -- update parent cost
      WHERE level=lev-1;
    END;
  END WHILE;
  SELECT                                             -- list BoM
    CONCAT(SPACE(level*2),Item) AS Item,
    ROUND(nodecost,2) AS 'Unit Cost',

```

```

ROUND(Qty,0) AS Qty,ROUND(cost,2) AS Cost FROM temp
ORDER by path;
END go
DELIMITER ;
CALL ww_bom( 1 );

```

Item	Unit Cost	Qty	Cost
finished bookcase	206.60	1.0	206.60
backboard2x1	9.80	1.0	9.80
laminate2x1	8.00	1.0	8.00
screw	0.10	8.0	0.80
side	25.20	2.0	50.40
screw	0.10	12.0	1.20
plank	20.00	1.0	20.00
shelf	16.00	8.0	128.00
plank	20.00	0.5	10.00
shelf bracket	0.50	4.0	2.00
foot	2.10	4.0	8.40
cube4cmx4cm	1.00	1.0	1.00
screw	0.10	1.0	0.10

Summary

Stored procedures, stored functions and Views make it possible to implement edge list graph models, nested sets graph models, and breadth-first and depth-first graph search algorithms in MySQL 5&6.

Further Reading

Celko J, "Trees and Hierarchies in SQL For Smarties", Morgan Kaufman, San Francisco, 2004.

Codersource.net, "Branch and Bound Algorithm in C#",
http://www.codersource.net/csharp_branch_and_bound_algorithm_implementation.aspx.

Math Forum, "Euler's Solution: The Degree of a Vertex",
<http://mathforum.org/isaac/problems/bridges2.html>

Muhammad RB, "Trees",
<http://www.personal.kent.edu/~rmuhamma/GraphTheory/MyGraphTheory/trees.htm>.

Mullins C, "The Future of SQL", http://www.craigsmullins.com/idug_sql.htm.

Rodrigue J-P, "Graph Theory: Definition and Properties",
<http://people.hofstra.edu/geotrans/eng/ch2en/meth2en/ch2m1en.html>.

Santry P, "Recursive SQL User Defined Functions",
<http://www.wwwcoder.com/main/parentid/191/site/1857/68/default.aspx>.

Shasha D, Wang JTL, and Giugno R, "Algorithmics and applications of tree and graph searching", In Symposium on Principles of Database Systems, 2002, p 39--52.

Stephens S, "Solving directed graph problems with SQL, Part I",
http://builder.com.com/5100-6388_14-5245017.html.

Stephens, S, "Solving directed graph problems with SQL, Part II", http://builder.com.com/5100-6388_14-5253701.html.

Steinbach T, "Migrating Recursive SQL from Oracle to DB2 UDB", <http://www-106.ibm.com/developerworks/db2/library/techarticle/0307steinbach/0307steinbach.html>.

Tropashko V, "Nested Intervals Tree Encoding in SQL", <http://www.sigmod.org/sigmod/record/issues/0506/p47-article-tropashko.pdf>

Van Tulder G, "Storing Hierarchical Data in a Database", <http://www.sitepoint.com/print/hierarchical-data-database>.

Venagalla S, "Expanding Recursive Opportunities with SQL UDFs in DB2 v7.2", <http://www-106.ibm.com/developerworks/db2/library/techarticle/0203venigalla/0203venigalla.html>.

Wikipedia, "Graph Theory", http://en.wikipedia.org/wiki/Graph_theory.

Wikipedia, "Tree traversal", [Wikipedia, "Tree traversal", http://en.wikipedia.org/wiki/Tree_traversal](http://en.wikipedia.org/wiki/Tree_traversal).

Willems K, "SQL Graph Algorithms", <http://willems.org/sqlgraphs.html>.