Directed Graph Representation and Traversal in Relational Databases

Mohammad Beydoun and Ramzi A. Haraty

Department of Computer Science and Mathematics
Lebanese American University
Beirut, Lebanon
rharaty@lau.edu.lb

Abstract. Graph representation in computers has always been a hot topic due to the number of applications that directly benefit from graphs. Multiple methods have emerged in computer science to represent graphs in numerical/logical formats; most of these methods rely heavily on pointers. However, most current business applications also rely heavily on relational databases as a primary source of storing information. Moreover, most databases are relational by nature, and this does not provide the best-fit scenario to represent graphs. In this work, we present a solution to representing a graph in a relational database. Moreover, we will also provide a set of procedures to traverse this graph and determine the connection path between two given nodes. This work was implemented in an online social/travel website which can be found at http://www.tripbot.com [1].

Keywords: graphs and relational databases.

1 Introduction

A graph is a set of nodes linked together with connection paths. The heavy reliance on relational databases as a primary source of information storing has created a gap between classical graph implementation methods and applying those implementations in relational databases. The problem arises from the fact that relational databases do not offer advanced data types, such as pointers, that classical graph algorithms relies heavily upon. Craig Mullins, the author of "The Future of SQL" wrote that "the set-based nature of SQL is not simple to master and is anathema to the OO techniques practiced by Java developers" [2].

Today, SQL has become widespread. Some database engines provided native support for graphs such as DB2, Oracle and Microsoft SQL Server. DB2 uses a 'WITH' operator to process recursive sets; Oracle uses a CONNECT BY operator to represent graphs that are trees in nature, while Microsoft SQL Server has recursive unions. MySQL does not have any native or built in support to handle graphs [3].

This work deals with that problem, especially in MySQL which lacks those graph support functions. This approach is divided into two parts. The first part is carried out by the database management system (DBMS), and the second part is carried out by any

F. Zavoral et al. (Eds.): NDT 2010, Part II, CCIS 88, pp. 443–455, 2010.

programming language capable of consuming data from the DBMS. Graph related applications are very common nowadays, and due to the popularity of MySQL and its open-source community model, applying graph methods to MySQL is becoming very appealing. This work introduces the concept of graphs and graph traversal to MySQL and additional graph concepts and methods can be added accordingly.

This paper is organized as follows: In section 2 we present the literature review and discuss the rise of the need for graph traversal in relational databases. The proposed solution will be presented along with fully working algorithms in section 3. The results of the proposed algorithms will be discussed in section 4. And finally we will branch into our conclusion and future enhancement in section 5.

2 Literature Review

A graph is a set of nodes (vertices) and edges (lines or arcs) that connect them. Graphs are ideal for modeling hierarchies - search trees, relational matrixes and family trees- whose shape and size are variable [3].

In the recent rise of database management system (DBMS) popularity, the need to store and represent graphs in relational databases haunted many developers. Most graph representation and traversal techniques rely on advanced programming features. These features are found in object driven or object based programming languages [4][5].

A DBMS is an engine that stores and retrieves data. Most of its functionalities are oriented towards data storing and data fetching. Most DBMSs uses SQL to manipulate data, while a great tool it is still a declarative programming language that lacks most of the features found in object oriented and object driven languages. Even with the introduction of advanced T-SQL commands, SQL is still weak when it comes to working with graphs. Major DBMS vendors noticed this weakness and some of them developed their own set of proprietary functions to ease working with data of hierarchical nature. Oracle introduced the "START WITH…CONNECT BY" clause. It is primarily used to select data that has a hierarchical relationship. This makes representing trees and traversing them possible.

In DB2, IBM allowed the "WITH" clause to handle recursive-sets by allowing self-references. This is useful to represent and traverses tree-like structures [6]. In DB2 this is implemented using a common table expression. A common table expression is a construct that is similar to a temporary view. However, this construct is only valid for the duration of the single SQL statement in which it is defined. The construct takes the form of a WITH clause at the beginning of a SELECT statement. A common table expression can be used several times in the query that uses it, and it can also be joined to itself by aliasing, which make it very lucractive to implement recursion.

A recursive query in DB2 typically has three parts:

1. A virtual table in the form of a common table expression.
2. An initialization table.
3. A secondary table that does a full inner join with the virtual table.

All of the above tables are merged using UNION ALL. A final SELECT yields the required rows from the recursive output [7].

In MySQL, one does not have access to any similar operators that helps in creating and traversing trees/graphs. Flavio Botelho wrote code that does sequential processing for tree traversal; however, it proved to be slow since it relies heavily on subqueries [8].

3 The Proposed Solution

The method proposed to help graph representation and graph traversal in MySQL is twofold. The first part relates to MySQL on both data representation and data retrieval, while the second part relies on any programming language used (PHP, .NET, etc…). However, in this implementation we provide an example in .NET 2.0 / .NET 3.5 though they can be considered as pseudo-code and can easily be converted to any other programming language.

3.1 MySQL Representation

The table that represents hierarchical information (the graph) is created with the following statement:

Listing 3.1

```
CREATE TABLE `tbl_graph` (
  `Auto_ID` varchar(20) NOT NULL auto_increment,
  `First_Node_ID` varchar(20) default NULL,
  `Second_Node_ID` bigint(20) default NULL,
  `Status_ID` int(11) default NULL,
  `Appear` tinyint(1) NOT NULL default '1',
  `Time_Date` timestamp NULL default NULL on update
CURRENT_TIMESTAMP,
  PRIMARY KEY  (`Auto_ID `)
) ENGINE=MyISAM AUTO_INCREMENT=1 DEFAULT CHARSET=latin1;
```

Auto_ID
This is the primary key of the table; this is the ID of the path between two nodes. It has no impact on the algorithm.

First_Node_ID
This is the ID of a node in the graph; the nodes can be defined in a separate table and referenced by a simple INNER JOIN construct.

Second_Node_ID
This is the ID of a node in the graph that is connection to First_Node_ID; the nodes can be defined in a separate table and referenced by a simple INNER JOIN construct.

Status_ID
This is a status modifier used for business rules (sometimes two nodes can be connected but their connection should not be used to determine the shortest path between two other nodes).

Appear
A Boolean value field to determine if this connection record is active or not.

Time_Date
A timestamp to determine the time of creation of this path. It has no impact on the algorithm. For example, the representation of the following graph would be as shown in Table 1:

Table 1. Same graph in tabular format

Auto_ID	First_Node_ID	Second_Node_ID	Status_ID	Appear	Time_Date
1	A	C	1	1	1/1/2008
2	C	A	1	1	1/1/2008
3	C	D	1	1	1/1/2008
4	D	C	1	1	1/1/2008
5	C	F	1	1	1/1/2008
6	F	C	1	1	1/1/2008
7	B	E	1	1	1/1/2008
8	E	B	1	1	1/1/2008

The table specified above will contains list of paths between the nodes in a give graph. A sample representation of this table can be as follows: Each path is represented in both directions (since this is a directed graph). The Status flag is not used in this example but we will expand on it in the algorithm.

3.2 Algorithm for Traversal (Finding a Path between Two Nodes if It Exists)

The Algorithm in MySQL can be summarized in the following stored procedure:
SP_Fetch_Adjacency (_FromMemberID BIGINT, _ToMemberID BIGINT, _depth BIGINT)
Parameters in use:

_FromNodeID
The node we are starting from.

_ToNodeID
The node we want to end at.

_depth
The maximum number of levels to search for (directly adjacent nodes have a depth of 1, etc…).

The stored procedure will create a HEAP table (a table stored in the DBE internal memory) called Reached. It will contain all the adjacent nodes to the initiating node and recursively fetch all adjacent nodes of those nodes until it hit the maximum depth or we hit the target node.

Listing 3.2

```
1-BEGIN
2-DECLARE depth SMALLINT DEFAULT 0;
3-DECLARE rows SMALLINT DEFAULT 1;
4-DECLARE found SMALLINT DEFAULT 0;
5-DROP TABLE IF EXISTS reached;
6-CREATE TABLE reached (From_Node_ID VARCHAR(20),
To_Node_ID VARCHAR(20), UNIQUE INDEX USING HASH
(From_Node_ID,To_Node_ID))  ENGINE=HEAP;
7-INSERT INTO reached VALUES (0,_FromNodeID);
8-SET depth = _depth;
9-WHILE ((depth > 0) AND (rows > 0)) DO
10-    SET rows = 0;
11-      INSERT  IGNORE  INTO  reached  SELECT  DISTINCT
         e.First_Node_ID,  e.Second_Node_ID FROM tbl_graph
         AS e INNER JOIN reached AS p ON e.First_Node_ID =
         p.To_Node_ID
12-    SET rows = rows + ROW_COUNT();
13-      INSERT  IGNORE  INTO  reached  SELECT  DISTINCT
         e.First_Node_ID,  e.Second_Node_ID  FROM  tbl_graph
         AS e INNER JOIN reached AS p ON e.First_Node_ID =
         p.From_Node_ID
14-    SET rows = rows + ROW_COUNT();
15-      SELECT  COUNT(*)  INTO  Found  FROM  reached  WHERE
         To_Node_ID = _ToNodeID;
16-    IF Found > 0 THEN
17-          SET depth = 0;
18-    ELSE
19-          SET depth = depth -1;
20-    END IF;
21-END WHILE;
22-SELECT * FROM reached;
23-DROP TABLE reached;
24-END
```

Listing 3.2 represents the T-SQL syntax for the algorithm proposed to traverse the relational graph; the T-SQL syntax is compatible with MySQL T-SQL syntax. This procedure is the core of our method since it allows graph navigation in a breadth-first manner. The following section explains the function of each line of code that spans this store procedure.

Lines 2 to 4 are declarative statements that declare the variables in use throughout the algorithm

depth will be used to compare the current depth with the maximum depth.

rows will be used to see if the last iteration returned records, otherwise the procedure will stop.

found will be used to identify if the target node have been reached or not.

Line 5 will drop the heap table if it already exists in the memory.

Line 6 will create the heap table and define its structure. The heap table got a primary key of both nodes combination.

Line 7 will create the starting record in the heap table, from a fictitious root node 0 to the node we are starting from to inner join on this record in the iteration process.

Line 8 will set the depth variable to the depth parameter passed to the stored procedure.

Line 9 will start the while loop and check if the last iteration returned any rows.

Line 10 will reset the rows to 0 for the current iteration.

Line 11 and 13 will try to fetch all adjacent nodes to the nodes in the heap table and insert them into the heap.

Line 12 and 14 will assign the row variable the number of rows returned from the last two fetch iterations.

Line 15-20 checks if the destination node exists in the heap. If it does then it will set the depth directly to 0 to exit the while loop after this iteration.

Line 22 will return the heap to the caller.

Line 23 will delete the heap from memory.

Now we have a tabular result set that we need to do some work around to identify the shortest path between two nodes. The .NET algorithm will consume the heap and traverse it to determine the shortest path between the given two nodes.

Listing 3.3

```
Private strPublicPath As String = "False"
Public Function strFetchConnectionPath(ByVal
lngFromNodeID As Long, ByVal lngToNodeID As Long,
Optional ByVal intDegree As Integer = 3) As String
    Dim strSQL As String = String.Format("CALL
sp_fetch_adjacency({0},{1},{2})", lngFromNodeID,
lngToNodeID, intDegree)
    Dim dtTemp As New Data.DataTable
    Dim drTemp() As Data.DataRow
    dtTemp = fxFunctions.FillDataTable(strSQL)
    drTemp =
dtTemp.Select(String.Format("To_Node_ID={0}",
lngToNodeID))
    If drTemp.Length Then
```

```
        boolBuildConnectionPath(dtTemp,     lngFromNodeID,
lngToNodeID, lngToNodeID.ToString, False, 0)
        If LCase(strPublicPath) <> "false" Then
            Dim aryTmp() As String
            aryTmp = strPublicPath.Split(",")
            strPublicPath = ""
            For Each strTmp As String In aryTmp
                'Code to construct the path in
            Next
        Else
            Return "Those nodes are not connected"
        End If
        Return strPublicPath
    Else
        Return " Those nodes are not connected"
    End If
End Function
```

Listing 3.4

```
Private Function boolBuildConnectionPath(ByRef dtTemp As
Data.DataTable, ByVal lngFromNodeID As Long, ByVal lngToNodeID As
Long, ByRef strPath As String, ByRef boolFound As Boolean, ByVal
intSteps As Integer) As Boolean
    Dim drTemp() As Data.DataRow
    If ((lngFromNodeID <> lngToNodeID) And (dtTemp.Rows.Count > 0))
Then
        drTemp = dtTemp.Select(String.Format("To_Node_ID={0}",
lngToNodeID))
        For Each dr As Data.DataRow In drTemp
            Try
                Dim lngFromNodeIDTmp As Long = dr("From_Node_ID")
                dtTemp.Rows.Remove(dr)
                intSteps += 1
                strPath = strPath & "," & lngFromNodeIDTmp
                boolBuildConnectionPath(dtTemp, lngFromNodeID,
lngFromNodeIDTmp, strPath, False, intSteps)
            Catch ex As Exception
                Return False
            End Try
        Next
    Else
        'compare with previous path length
        ReDim Preserve intPathSteps(intPathIndex)
        If intPathIndex > 0 Then
            For Each intTemp As Integer In intPathSteps
                If intTemp > intSteps Then
                    strPublicPath = strPath
                End If
            Next
        Else
            strPublicPath = strPath
        End If
        intPathSteps(intPathIndex) = intSteps
        intPathIndex += 1
            End If
        End Function
```

We have two functions *strFetchConnectionPath* and *boolBuildConnectionPath* that will be explained next.

The *strFetchConnectionPath* as mentioned earlier will consume the returned result set and check if it contains the destination node. If the destination node exists in the result set then a call to the function *boolBuildConnectionPath* will be done to build the actual shortest path between the two nodes.

The *boolBuildConnectionPath* will build all different paths between the given two nodes. The path steps are stored in an array and the comparison is done at the end of each path creation. At the end, a comma separated string will be returned to indicate the shortest path between two given node if it exists or a "Those nodes are not connected" message if they are not.

Additional methods can easily be added to encapsulate a full graph class, those methods are:

*boolAddNode (intFirstNodeID, intSecondNodeID, boolStatus, boolAppear, dtDate, bool2way) as Integer.*This method will add a node to the graph and link it to a previous node. It is good to note that adding an orphan node 'O' will be done using the following syntax:

boolAddNode('O',NOTHING,1,1,Now(),True).

This will create a node called 'O' that is not linked to any other node in the graph.

The *bool2way* parameter will instruct the function to create the directed paths from the first node to the second node and vice versa. If it is set to true, or a one way direction from the first node to the second if it is set to false.

The *boolAddNode* will return a Boolean flag value presenting the success/failure of the method back to the calling function.

To delete a node link or a node completely we will implement the following methods:

boolDeleteNodeLink (Auto_ID) as Boolean
boolDeleteNodeLink (Node_ID) as Boolean

The first method will delete a row in the graph table according to its Auto_ID value. This is useful to delete a single path between two nodes. The second method is an overload of the first method that will delete all the paths that the specified node is a partner in; hence, deleting the node and all its paths. Similar methods can be added to accommodate for updating the nodes. A point of interest would be to replace the current implementation with nested sets, nested sets means that a whole sub tree or a sub graph can be easily retrieved without looping or recursion but of course this will add upkeep cost to inserts, updates and deletes. Also updating our algorithms a bit we can use it to find 'reach-ability', or where we can go from a certain node. In our specific case, find all friends of a certain individual.

In the next section, we will present an example of a social network. The nodes in the graph represent people, the paths between these nodes represent that those two individuals are friends or 'connected'. For the sake of simplicity we will assume that our graph is not directed; however, our method takes into consideration directed graphs since we represent each direction of a path in a separate row.

Simple Example
Assume the following graph depicted in figure 1:

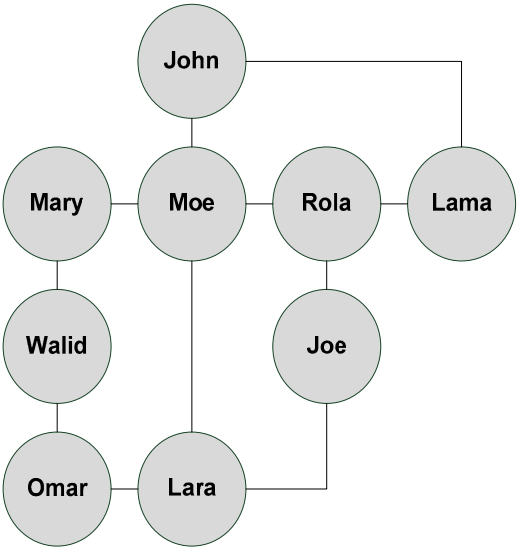

Fig. 1. A small friend's network

The above graph describes a small friends' network and who each person has access to in terms of direct friends. For example, Moe is friend with Mary, Lara, John and Rola. Rola in turn is friend with Lama, Joe and Moe. For the sake of simplicity we will assume that each path is bi-directional. However, our approach will also work with directional paths since those are represented differently in the database; i.e. Rola friend with Lama but Lama is not friend with Rola. The above graph represented in a tabular format in the DBMS would look like Table 2.

This graph is built by calling the *boolAddNode* function:

boolAddNode ('John', 'Moe', True, True, '1/1/2009', True)
→ INSERT INTO FriendsList (First_Node_ID, Second_Node_ID, Status_ID, Appear) VALUES ('John', 'Moe',1,1)
→ INSERT INTO FriendsList (First_Node_ID, Second_Node_ID, Status_ID, Appear) VALUES ('Moe, 'John',1,1)

Table 2. Friends network

Auto_ID	First_Node_ID	Second_Node_ID	Status_ID	Appear	Time_Date
1	John	Moe	1	1	1/1/2009 0:00
2	John	Lama	1	1	1/2/2009 0:00
3	Mary	Moe	1	1	1/3/2009 0:00
4	Mary	Walid	1	1	1/4/2009 0:00
5	Moe	Mary	1	1	1/5/2009 0:00
6	Moe	John	1	1	1/6/2009 0:00
7	Moe	Rola	1	1	1/7/2009 0:00
8	Moe	Lara	1	1	1/8/2009 0:00
9	Rola	Lama	1	1	1/9/2009 0:00
10	Rola	Moe	1	1	1/10/2009 0:00
11	Rola	Joe	1	1	1/11/2009 0:00
12	Lama	John	1	1	1/12/2009 0:00
13	Lama	Rola	1	1	1/13/2009 0:00
14	Joe	Rola	1	1	1/14/2009 0:00
15	Joe	Lara	1	1	1/15/2009 0:00
16	Lara	Moe	1	1	1/16/2009 0:00
17	Lara	Joe	1	1	1/17/2009 0:00
18	Lara	Omar	1	1	1/18/2009 0:00
19	Omar	Lara	1	1	1/19/2009 0:00
20	Omar	Walid	1	1	1/20/2009 0:00
21	Walid	Mary	1	1	1/21/2009 0:00
22	Walid	Omar	1	1	1/22/2009 0:00

Now we would like to test if there exist a path between two nodes, Walid and Joe as an example. By observing the graph we notice that Joe can be reached from Walid using three different routes

Route 1: Walid → Mary → Moe → Role → Joe
Route 2: Walid → Omar → Lara → Joe
Route 3: Walid → Mary → Moe → John → Lama → Rola → Joe

First we need to build the Reached heap, we do this by calling the following MySQL stored procedure defined earlier:

SP_Fetch_Adjacency

Using the following syntax:

SP_Fetch_Adjacency ('Walid','Joe',10)

The result will be a table (Table 3) listing all different routes from 'Walid' to 'Joe' called Reached:

Table 3. The Heap

First_Node_ID	Second_Node_ID
Root	Walid
Walid	Mary
Walid	Omar
Mary	Moe
Mary	Walid
Omar	Lara
Omar	Walid
Moe	Mary
Moe	John
Moe	Rola
Moe	Lara
Lara	Moe
Lara	Joe
Lara	Omar

The function *strFetchConnectionPath* called with the proper argument will build the following heap:

strFetchConnectionPath('Walid','Joe',10) will iterate thru the heap to identify if there exist a path between Walid and Joe using a linear seek (see Table 4).

Table 4. Paths Array

Path Steps	Path
5	Walid, Mary, Moe, Role, Joe
4	Walid, Omar, Lara, Joe
7	Walid, Mary, Moe, John, Lama, Rola, Joe

If a path is found – if the node Joe is found in the heap – then a call to *boolBuild-ConnectionPath* is carried out. *boolBuildConnectionPath* will recursively build all possible paths between the two given node and give priority to the shortest in each iteration. The result will be the shortest path between two given node.

4 Results and Discussion

Graph problems were always computationally heavy, and the proposed method in this paper is not that different. However, our method is built using an existing foundation (the DBMS), and this foundation has been well-established in what relates to fast query execution and fast data access. This gives it the edge over its counterparts and

hopefully will help orient future investment in that direction. In order to evaluate the efficiency of the proposed method, we will split it into the different algorithms used:

Algorithm 1 or the MySQL Stored Procedure used to create the adjacency list (HEAP).
Algorithm 2 or the .NET method: *strFetchConnectionPath.*
Algorithm 3 or the .NET method: *boolBuildConnectionPath.*

Algorithm 1 is the MySQL stored procedure in listing 3.2. This procedure will recursively fetch all adjacent nodes of a given starting node, and then the adjacent nodes of all the nodes fetched in the first iteration and so on. This is executed till either the required node is found or till the maximum depth specified is reached. Since the first encounter of the target node is guaranteed to produce one of the shortest paths available the algorithm will stop at that stage. If we want to return all the possible paths between two different nodes we can easily omit this section and the procedure will run till it reaches the maximum depth specified. Analysis of this algorithm is straightforward, with each step the amount of work is growing exponentially. Depending on the graph type (sparse or dense) this maps to an $O(n^2)$ notation. n being the number of nodes in the graph.

Algorithm 2 will traverse the heap linearly. This maps to an $O(n)$ notation. n being the number of nodes in the graph.

Finally, algorithm 3 will build the connection path in the heap. Also this traversal is linear and at the end another linear comparison will be performed on the result set to find the shortest path. This also maps to an $O(n)$ notation. n being the number of nodes in the graph.

The major performance issue is in algorithm 1. This algorithm is recursive by nature. It is the core of the proposed traversal algorithms and it shares its computational complexity with most of its traversal counterparts. However, since this method relies heavily on SQL fetch mechanisms which are optimized methods for fast data retrieval then it has a slight advantage over simple file read operations. Also the fact that the heap will be stored in memory and directly accessed by the SQL engine is an additional performance boost to the algorithm.

5 Conclusion

Those proposed algorithms are a great start and a first step in identifying future trends of graph usage in relational databases. The world of relational databases and graphs are slowly merging into one. Hence, we see the growth of some graph engines that are used to store relational database information and vice-versa [9]. This growth is starting to materialize in everyday's business problems. On one hand most data-mining systems rely heavily on crawling and traversal from one bit of information to the other. On the other hand data storage systems (DBMS) are here to stay since they provide a resilient and accessible system for fast data retrieval and safe data storage. That is why most of the commercial database management systems are adopting the new trend by implementing proprietary functions into their existing engines. Some of the vendors have scheduled native support for hierarchical data into future releases of their engines. Not forgetting the rise of new data-oriented systems. Such systems focus entirely on data values and representing this data in an optimal manner. Most of

such data is hierarchical or relational and requires extensive use of tree-like structures and graphs [10]. While definitely crude, the proposed method will help orient future progress in this domain, especially when it comes to representing a directed graph easily and traversing it to find the shortest path between two distinct nodes stored in a MySQL database engine.

Acknowledgements. This work was funded by the Lebanese American University.

References

1. Tripbot. Tripbot Friends, http://www.tripbot.com
2. Mulliuns, C.: The Future of SQL. IDUG Solutions Journal (1998)
3. Graph Theory, http://en.wikipedia.org/wiki/Graph_Theory
4. Microsoft. MSDN - Microsoft Developer Network, http://www.msdn.com
5. Oracle Communities, http://www.oracle.com/us/community/index.htm
6. Celko, J.: Joe Celko's Trees and Hierarchies in SQL for Smarties. Morgam Kaufmann, New York (2004)
7. IBM DeveloperWorks, http://www.ibm.com/developerworks/
8. MySQL Reference Manual, http://dev.mysql.com/doc/mysql/en/
9. Team, N. Neo4J Graph Database, http://neo4j.org/
10. Ambler, S.: Agile Database Techniques: Effective Strategies for the Agile Software Developer. John Wiley & Sons, Chichester (2003)