A Model to Represent Directed Acyclic Graphs (DAG) on SQL Databases

Please [Sign up or sign in](https://www.codeproject.com/script/Membership/LogOn.aspx?rp=%2fArticles%2f22824%2fA-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o) to vote.

Addressing the general problem of representing directed acyclic graphs (DAGs) in SQL databases

FROM: <http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o>

Introduction

Over the years that I have spent as a software engineer, I have faced the same challenge of modeling hierarchical data in relational DB tables over and over again. The first one that I remember is the model that I created for the product trees of a processed food company, using FoxPro.

The reason, however, for thinking more on this problem and eventually writing this article was the recurring need for a row-level authorization schema for SQL Server databases. I am, of course, aware of the Microsoft White Paper [Implementing Row- and Cell-Level Security in Classified Databases Using SQL Server 2005](http://www.microsoft.com/technet/prodtechnol/sql/2005/multisec.mspx). I’ll only say that the Microsoft solution is way too complex because they could not make assumptions about the system where the method is to be used. Hence, many possible shortcuts and simplifications were not available to them.

In this article, though, I’ll address only the general problem of representing Directed Acyclic Graphs (DAG) in SQL databases, which I devised as part of the solution to the row-based security problem, as DAGs have a lot more applications than just row-level authorization. I’ll post a second article that will complement this one and will address the specifics of row-based security, along with an efficient implementation on SQL Server.

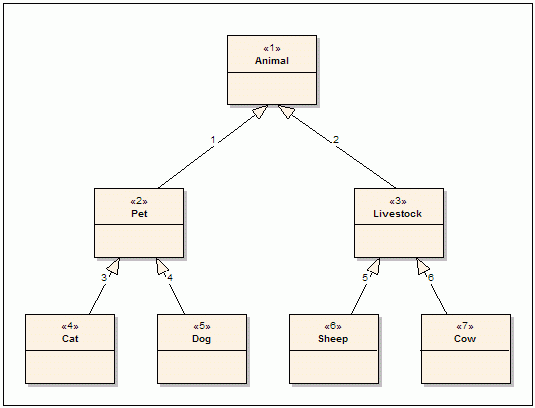
Before I go into further detail, let me explain what a DAG is. Here is an excerpt from Wikipedia on DAG:

“In computer science and mathematics, a directed acyclic graph, also called a DAG, is a directed graph with no directed cycles; that is, for any vertex v, there is no nonempty directed path that starts and ends on v. DAGs appear in models where it doesn't make sense for a vertex to have a path to itself; for example, if an edge u?v indicates that v is a part of u, such a path would indicate that u is a part of itself, which is impossible. Informally speaking, a DAG "flows" in a single direction.”

Please refer to the [article](http://en.wikipedia.org/wiki/Directed_acyclic_graph) on Wikipedia for more details on the topic. In short, DAGs have the following properties:

1. A DAG has directed edges, i.e. the links that connect vertices (nodes) on the graph have a direction. The direction is denoted by an arrow head.
2. Starting from an arbitrary vertex, if we traverse the vertices in the direction of the arrows, it is not possible to **return back** to the originating vertex.

A notable and familiar structure that has the above properties is the inheritance hierarchy in Object-Oriented Programming. [Figure 1](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure1) is a simple (and incomplete) class diagram, which happens to be a **tree**. The arrows in the figure should be read as "is a." A tree is a DAG with one important additional restriction: there may exist **only one path** from one vertex to another, e.g. there exists only one way to go from Dog to Animal: Dog -> Pet -> Animal. This restriction helps us greatly in modeling trees in SQL databases.



**Figure 1: Animal Class Hierarchy**

Arguably, the easiest and an efficient solution to the problem of modeling a tree structure in SQL databases is a method called **Materialized Paths**. This model relies on string markers for the **complete path to each of the vertices from the root vertex**. The marker value is composed of a concatenation of the paths of its ancestors. For example, if the path for the root vertex Animal is A, then the path for Pet is A.P. and for Dog it is A.P.D.. In essence, it is the **path from the root all the way to the current vertex**. [Table 1](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Table1) is an extract from such a table to illustrate the method.

|  |  |  |  |
| --- | --- | --- | --- |
| ***Table 1:  Sample Data from Animal Table*** | | | |
| **ID** | **Path** | **Name** | **Other Columns** |
| 1 | A.P.C. | My cat | ... |
| 2 | A.P.D. | My dog | ... |
| 3 | A.P.D. | My second dog | ... |
| 4 | A.L.S. | White sheep | ... |
| 5 | A.L.C. | Brown cow | ... |

The **Materialized Paths** approach is particularly useful, as it is very easy to construct fast queries with the standard like operator of SQL. For example, to get all the dogs I would write a query like:

Hide   Copy Code

SELECT \*

FROM Animal

WHERE Path LIKE 'A.P.D.%'

Or to get all livestock, I could write:

Hide   Copy Code

SELECT \*

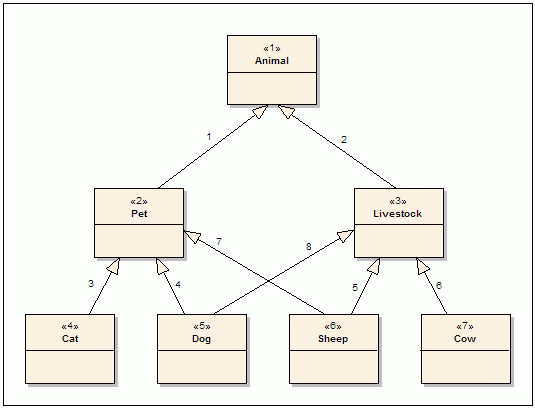
FROM Animal

WHERE Path LIKE 'A.L.%'

Please note that the above queries would continue to work even if the class hierarchy were extended afterwards. For example, if I add two Dog classes, Doberman and Bulldog with paths A.P.D.D. and A.P.D.B., the former query would still return all the dogs.

The Problem

Unfortunately, trees are **too restrictive** in modeling the real world. In reality, an object may have many facets, not only one as is suggested by a tree structure. For example, a closer look at [Figure 1](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure1) would reveal that, arguably, for some nations dogs are actually livestock, too. Moreover, someone might have a lamb as his/her pet. [Figure 2](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure2) shows a modified (and still incomplete) class diagram that incorporates these facts. (Just an example, no arguments please!)



**Figure 2: Modified Animal Class Hierarchy**

Please note that the class diagram in [Figure 2](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure2) is no longer a tree. The difference is that now there exist multiple paths between some vertices. For example, we can use the paths (Dog -> Pet -> Animal) and (Dog -> Livestock -> Animal) to go from Dog to Animal. However, it is still not possible to return back to Dog once we leave the Dog vertex. So, it still conforms to the definition of DAG. In OOP jargon, Dog is said to have multiple ancestors. [The feature is called multiple inheritance (MI). Here is a special kind of MI often referred to as the dreaded diamond.]

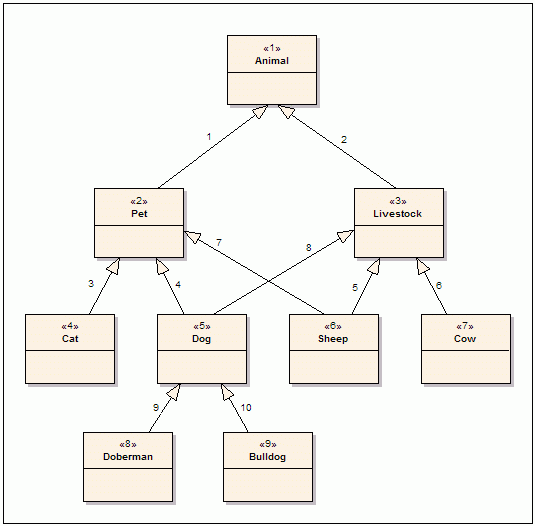
The simple schema of **Materialized Paths** cannot be used to model the class diagrams in [Figure 2](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure2). This is because the **Path** column can only be used to represent a single path on the graph and is hence limited to the modeling of trees. Remember that trees have only one path to a specific vertex.

Class hierarchies with multiple inheritance are not the only familiar cases of DAGs. There are many others, too. For example, in Windows, the NTFS file/folder structure (no, it is really not a tree) and Active Directory user/group hierarchy (which is why I decided to tackle this problem) can be modeled using DAGs.

Solution

There are many publications that address the problem of modeling trees in SQL databases. However, I have yet to see a complete solution and an implementation for the more generalized case of Directed Acyclic Graphs, of which [Figure 2](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure2) is a simple example. So, I decided to solve it myself.

My solution to the problem is based on a modified version of another popular solution of the tree problem. This solution uses a so-called **adjacency list** table, i.e. a helper table that stores the **edges** (the links between vertices) in addition to the table that actually stores the properties of the objects (vertices here). To illustrate the method, let's look at [Figure 3](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure3), which is a further extended version of the class hierarchy in [Figure 2](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure2). Example: the edge Cat -> Pet in [Figure 3](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure3) can be represented by the tuple (EdgeId, Start Vertex, End Vertex) -> (3, 4, 2). [Table 2](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Table2) shows how we represent the DAG in [Figure 3](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure3) with this model.



**Figure 3: Modified Animal Class Hierarchy**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ***Table 2: Representation of Edges on*** [***Figure 3***](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure3) ***Using an Adjacency List Table*** | | | | | | |
| **AnimalType Table** | | |  | **Edge Table** | | |
| **Id** | **Name** | **Other Columns** |  | **EdgeId** | **Start Vertex** | **EndVertex** |
| 1 | Animal | ... |  | 1 | 2 | 1 |
| 2 | Pet | ... |  | 2 | 3 | 1 |
| 3 | Livestock | ... |  | 3 | 4 | 2 |
| 4 | Cat | ... |  | 4 | 5 | 2 |
| 5 | Dog | ... |  | 5 | 6 | 3 |
| 6 | Sheep | ... |  | 6 | 7 | 3 |
| 7 | Cow | ... |  | 7 | 6 | 2 |
| 8 | Doberman | ... |  | 8 | 5 | 3 |
| 9 | Bulldog | ... |  | 9 | 8 | 5 |
|  |  |  |  | 10 | 9 | 5 |

Note that the EdgeId column represents exactly the same edge number shown on [Figure 3](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure3). The Animal table should then be modified as shown in [Table 3](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Table3).

| ***Table 3: Modified Animal Table*** | | | |
| --- | --- | --- | --- |
| **Id** | **AnimalTypeId** | **Name** | **Other Columns** |
| 1 | 4 | My cat | ... |
| 2 | 8 | My dog | ... |
| 3 | 9 | My second dog | ... |
| 4 | 6 | White sheep | ... |
| 5 | 7 | Brown cow | ... |

There are a few notable observations to be made on [Table 2](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Table2) and [Table 3](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Table3):

1. The representation of edges is decoupled from the representation of the vertices, i.e. there remains nothing in the Animal table about its relationship with other animals. Furthermore, the Edge table includes no columns that are specific to the Animal or AnimalType tables. Hence, there is nothing that prevents us using the same Edge table to represent discrete and unrelated graphs, provided that we prevent ID clashes of vertices from different graphs.
2. It is now possible to construct the graph in any order, e.g. you can first create the edge Cat -> Pet and then create Pet -> Animal, etc. without any disturbance to the rest of the graph. This was not the case with the **Materialized Paths** model, where a single change in the graph would trigger a lot of changes to other rows. Think of inserting another vertex between Animal and Pet after the construction of the graph.

Now let us try to use the new structure to get some results. For example, how do I get all the livestock from the tables shown in [Table 2](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Table2) and [Table 3](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Table3)? The SQL query in [Listing 1](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Listing1) would probably do the job:

|  |
| --- |
| ***Listing 1: SQL Query to Get All Livestock*** |
| Hide   Copy Code  SELECT \*  FROM Animal  WHERE AnimalTypeId IN (  SELECT StartVertex  FROM AnimalType  WHERE EndVertex = 3 ) |

This would work for the class diagram in [Figure 2](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure2). Unfortunately, it does not work for [Figure 3](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure3) because not all the descendent classes of livestock are direct descendents. The two subclasses Doberman and Bulldog in [Figure 3](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure3) are also Livestock, but this is not apparent to SQL and results in an incorrect result set. Although we can deduce that Doberman and Bulldog are indeed Livestock from the Edge table, standard SQL does not provide any syntax to extract this information from the Edge table within a query, i.e. SQL lacks the ability to **traverse related records**.

Popular relational databases like Oracle and SQL Server (with the 2005 version) have constructs that address the lack of recursive relation support on the standard SQL: Oracle has the connect by clause and SQL Server 2005 has the with statement along with **Common Table Expressions**. In essence, they both take a procedural approach to the problem and simply traverse the adjacency list table recursively to get answers to recursive questions. The problem with this approach lies with their procedural nature. Since they actually run a procedure behind, there is not much a query optimizer can do when such queries are used, for example, in a join.

However, if we store all the possible paths (called **transitive closure**) in the **adjacency list**, the standard SQL is back to business along with all the powerful query optimizations and with possible use of indices on the **adjacency list** table. This is critical for a system if the number of objects to be represented is considerably high. The Edge table containing the closure of the paths of the graph in [Figure 3](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure3) is shown on the [Table 4](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Table4).

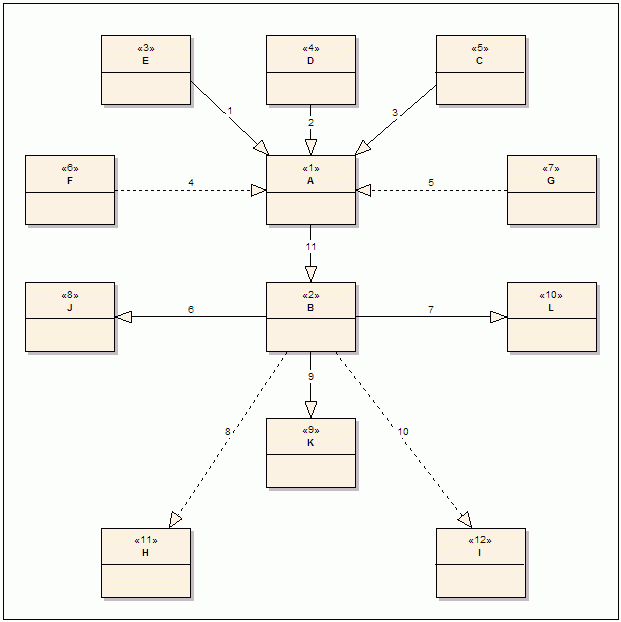
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ***Table 4: Edge Table with Complete List of Paths  (the second trio of columns indicate inferred relations)*** | | | | | | |
| **Edge Id** | **Start Vertex** | **EndVertex** |  | **Edge Id** | **Start Vertex** | **EndVertex** |
| 1 | 2 | 1 |  | 11 | 4 | 1 |
| 2 | 3 | 1 |  | 12 | 5 | 1 |
| 3 | 4 | 2 |  | 13 | 6 | 1 |
| 4 | 5 | 2 |  | 14 | 7 | 1 |
| 5 | 6 | 3 |  | 15 | 8 | 1 |
| 6 | 7 | 3 |  | 16 | 8 | 2 |
| 7 | 6 | 2 |  | 17 | 8 | 3 |
| 8 | 5 | 3 |  | 18 | 9 | 1 |
| 9 | 8 | 5 |  | 19 | 9 | 2 |
| 10 | 9 | 5 |  | 20 | 9 | 3 |

Strictly speaking, the Edges 11 to 20 are redundant, as it is possible to **infer** their existence from the other rows. However, they enable us to use the standard SQL query in [Listing 1](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Listing1) to get all the livestock. As we now put the solution as the maintenance of the transitive closure of the DAG, we'll have to face with the real issue of the maintenance of the Edge table with full transitive closure, which is not trivial.

The solution that I have is based on a recursion, as Oracle or SQL Server also does. However, instead of deferring the recursion until the query time, I do the recursion at the insertion time, assuming that the graph is actually more queried than modified (which is true for all the cases that I have faced so far).

Insertion of a New Edge to the Graph

In general, the connection of two vertices with a new edge has to be assumed to be the connection of two discrete graphs. We cannot assume any order of Edge insertion. Let’s look at [Figure 4](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure4). The new edge 11, (EdgeId, Start Vertex, End Vertex) -> (11, 1, 2), is actually connecting two discrete graphs. The dotted arrows in the figure signify those edges that are not direct connections. Rather, they are **implied** ones that can be **inferred** from existing direct connections and have already been created by the model during previous insertions.



**Figure 4: Adding a New Edge to Connect Two Graphs   
(please note that only parts of two graphs are shown here)**

The graph in [Figure 4](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure4) is not fully shown for the sake of clarity. The following is omitted from the graph:

1. **Outgoing** edges from vertex A and the **incoming** edges to vertex B. This is because they cannot participate in forming new paths between these two DAGs, as they are against the **flow direction**.
2. Most other vertices and edges that are the reason for the formation of the **implied** edges in the diagram, which are shown as dotted arrows.

Adding a new edge to connect two graphs results in the creation of implied edges that:

1. Connect all the vertices that are at the starting point of all **incoming edges** to the start vertex (vertex A in Figure 4) and end vertex (vertex B in [Figure 4](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure4)) of the new edge.
2. Connect the start vertex (vertex A in Figure 4) of the new edge to all of the vertices that are at the ending point of all **outgoing edges** to the end vertex (vertex B in [Figure 4](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure4)).
3. Connect all the vertices that are at the starting point of all incoming edges to the start vertex (vertex A in [Figure 4](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure4)) and all the vertices that are at the end of all outgoing edges to the end vertex (vertex B in [Figure 4](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure4)).

Before I show you the code that does exactly that, we need to modify the Edge table to support deletion operations. The new Edge table definition is as shown in [Table 5](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Table5).

|  |  |  |
| --- | --- | --- |
| ***Table 5: The New Definition of the Edge Table*** | | |
| **Column Name** | **Type** | **Description** |
| Id | Int | Primary key, auto increment |
| EntryEdgeId | Int | The ID of the **incoming edge** to the start vertex that is the creation reason for this **implied** edge; direct edges contain the same value as the Id column |
| DirectEdgeId | Int | The ID of the **direct edge** that caused the creation of this **implied** edge; direct edges contain the same value as the Id column |
| ExitEdgeId | Int | The ID of the **outgoing edge** from the end vertex that is the creation reason for this **implied** edge; direct edges contain the same value as the Id column |
| StartVertex | varchar(36) | The ID of the start vertex (36 is the length of a UUID in string form, if you wonder why) |
| EndVertex | varchar(36) | The ID of the end vertex |
| Hops | Int | Indicates how many vertex hops are necessary for the path; it is zero for direct edges |
| Source | varchar(150) | A column to indicate the context in which the graph is created; useful if we have more than one DAG to be represented within the same application  **CAUTION: you need to make sure that the IDs of vertices from different sources never clash; the best is probably use of UUIDs** |

[Listing 2](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Listing2) is the code that does all this for SQL Server. Porting it to other databases should be trivial. If you do so, please send it to me so that I can include it in this article.

|  |
| --- |
| ***Listing 2: Code for Insertion of a New Edge for SQL Server*** |
|  |
| Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png  Copy Code  SET ANSI\_NULLS ON  GO  SET QUOTED\_IDENTIFIER ON  GO  CREATE PROC AddEdge  @StartVertexId varchar(36),  @EndVertexId varchar(36),  @Source varchar(150)  AS  BEGIN  SET NOCOUNT ON  IF EXISTS(SELECT Id  FROM Edge  WHERE StartVertex = @StartVertexId  AND EndVertex = @EndVertexId  AND Hops = 0)  BEGIN  RETURN 0 *-- DO NOTHING!!!*  END  IF @StartVertexId = @EndVertexId  OR EXISTS (SELECT Id  FROM Edge  WHERE StartVertex = @EndVertexId  AND EndVertex = @StartVertexId)  BEGIN  RAISERROR ('Attempt to create a circular relation detected!', 16, 1)  RETURN 0  END  DECLARE @Id int  INSERT INTO Edge (  StartVertex,  EndVertex,  Hops,  Source)  VALUES (  @StartVertexId,  @EndVertexId,  0,  @Source)  SELECT @Id = SCOPE\_IDENTITY()  UPDATE Edge  SET EntryEdgeId = @Id  , ExitEdgeId = @Id  , DirectEdgeId = @Id  WHERE Id = @Id  *-- step 1: A's incoming edges to B*  INSERT INTO Edge (  EntryEdgeId,  DirectEdgeId,  ExitEdgeId,  StartVertex,  EndVertex,  Hops,  Source)  SELECT Id  , @Id  , @Id  , StartVertex  , @EndVertexId  , Hops + 1  , @Source  FROM Edge  WHERE EndVertex = @StartVertexId  *-- step 2: A to B's outgoing edges*  INSERT INTO Edge (  EntryEdgeId,  DirectEdgeId,  ExitEdgeId,  StartVertex,  EndVertex,  Hops,  Source)  SELECT @Id  , @Id  , Id  , @StartVertexId  , EndVertex  , Hops + 1  , @Source  FROM Edge  WHERE StartVertex = @EndVertexId  *-- step 3: A’s incoming edges to end vertex of B's outgoing edges*  INSERT INTO Edge (  EntryEdgeId,  DirectEdgeId,  ExitEdgeId,  StartVertex,  EndVertex,  Hops,  Source)  SELECT A.Id  , @Id  , B.Id  , A.StartVertex  , B.EndVertex  , A.Hops + B.Hops + 1  , @Source  FROM Edge A  CROSS JOIN Edge B  WHERE A.EndVertex = @StartVertexId  AND B.StartVertex = @EndVertexId  END |

The AddEdge procedure creates one implied edge for each of the possible paths, resulting in many more redundant implied rows than actually necessary. So, there can be many rows with the same Start Vertex and End Vertex values, but with different EntryEdgeIds, ExitEdgeIds and different Hops. Actually, these redundancies can be eliminated if we want to construct the graph once and allow only the addition of new edges. However, by allowing repeated implied rows, we enable our model to support deletion scenarios.

Deletion of an Existing Edge from the Graph

As noted before, deletion has to be supported by the addition operation: the addition operation should tell which **implied** edges it has created. So, we should create a purge list that is composed of:

1. The list of all edges that consists of all the edges that were created during **original add operation**.
2. The list of all edges that consists of all the edges that **depend** on any of the edges described in step 1 and step 2 **recursively**.

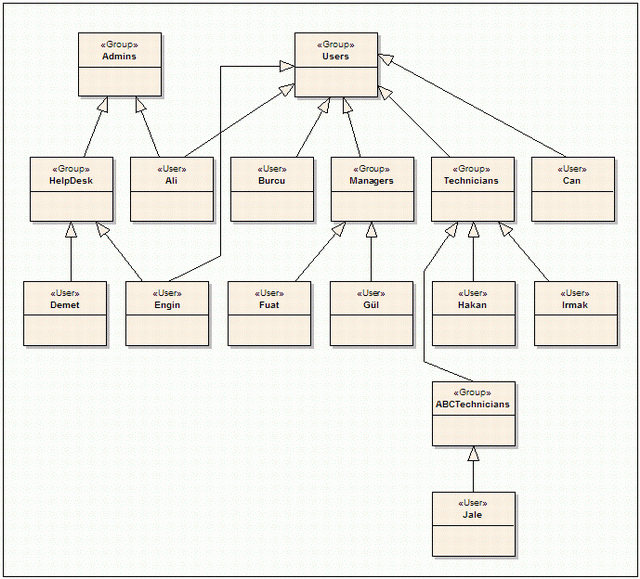
If there were no add operations after the **original add operation**, the list at step 2 would be empty. However, it is possible that there may be other add operations after the original add operation, which may have created implied edges that depend on any of the edges described in step 1. The operation at step 2 has to be recursive, since there may be other add operations that took place after the **second add operation** and so on. [Listing 3](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Listing3) is the code that does this for SQL Server. Again, if you port this code to other databases, please send it to me.

Actually, the addition operation is recursive, too. However, our code does not show any recursive calls or loops in the AddEdge method. The reason for this is that the addition of new direct edges creates all the implied edges, using already-created implied edges. Therefore the add procedure effectively **memorizes** the traversal of the graph in the form of implied edges.

|  |
| --- |
| ***Listing 3: Deletion of an Edge from the Graph, for SQL Server*** |
|  |
| Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png  Copy Code  SET ANSI\_NULLS ON  GO  SET QUOTED\_IDENTIFIER ON  GO  CREATE PROC RemoveEdge  @Id int  AS  BEGIN  IF NOT EXISTS( SELECT Id FROM Edge WHERE Id = @Id AND Hops = 0 )  BEGIN  RAISERROR ('Relation does not exists', 16 ,1)  RETURN  END  CREATE TABLE #PurgeList (Id int)  *-- step 1: rows that were originally inserted with the first*  *-- AddEdge call for this direct edge*  INSERT INTO #PurgeList  SELECT Id  FROM Edge  WHERE DirectEdgeId = @Id  *-- step 2: scan and find all dependent rows that are inserted afterwards*  WHILE 1 = 1  BEGIN  INSERT INTO #PurgeList  SELECT Id  FROM Edge  WHERE Hops > 0  AND ( EntryEdgeId IN ( SELECT Id FROM #PurgeList )  OR ExitEdgeId IN ( SELECT Id FROM #PurgeList ) )  AND Id NOT IN (SELECT Id FROM #PurgeList )  IF @@ROWCOUNT = 0 BREAK  END  DELETE Edge  WHERE Id IN ( SELECT Id FROM #PurgeList)  DROP TABLE #PurgeList  END  GO |

How to Use the Edge Table

The use of the Edge table completely depends on the needs of the application. To illustrate this, let’s take a look at [Figure 5](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure5), which represents an application's roles and players. The arrows here are to be read as “is a member of.”



**Figure 5: An Application’s Roles Defined as a DAG**

As we already have the tools to model DAGs in tables, the construction of the above graph is fairly easy. The SQL Server code in [Listing 4](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Listing4) does just that:

|  |
| --- |
| ***Listing 4: SQL Code to Create the Graph on*** [***Figure 5***](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Figure5)***, for SQL Server*** |
| Hide   Copy Code  EXEC AddEdge 'HelpDesk', 'Admins', 'AD'  EXEC AddEdge 'Ali', 'Admins', 'AD'  EXEC AddEdge 'Ali', 'Users', 'AD'  EXEC AddEdge 'Burcu', 'Users', 'AD'  EXEC AddEdge 'Can', 'Users', 'AD'  EXEC AddEdge 'Managers', 'Users','AD'  EXEC AddEdge 'Technicians', 'Users', 'AD'  EXEC AddEdge 'Demet', 'HelpDesk', 'AD'  EXEC AddEdge 'Engin', 'HelpDesk', 'AD'  EXEC AddEdge 'Engin', 'Users', 'AD'  EXEC AddEdge 'Fuat', 'Managers', 'AD'  EXEC AddEdge 'G l', 'Managers', 'AD'  EXEC AddEdge 'Hakan', 'Technicians', 'AD'  EXEC AddEdge 'Irmak', 'Technicians', 'AD'  EXEC AddEdge 'ABCTechnicians', 'Technicians', 'AD'  EXEC AddEdge 'Jale', 'ABCTechnicians', 'AD' |

The nice thing here is that if we were to change the order of the statements above, the graph would be formed with exactly the same results (except for the IDs of the rows, of course). Now we can start to query our Edge table. For example, to get all the Admins, we can write:

Hide   Copy Code

SELECT StartVertex, Hops

FROM Edge

WHERE EndVertex = 'Admins'

This returns:

Hide   Copy Code

StartVertex Hops

--------------- -----------

HelpDesk 0

Ali 0

Demet 1

Engin 1

(4 row(s) affected)

The results above show that Demet and Engin are Admins, but **indirectly** (Hops value of 1 indicates that). The Hops column actually shows how many **intermediate** vertices have to be **visited** before **reaching** to the desired vertex for that specific path, i.e. **the length of the path**. For example, the following query shows all the memberships of Jale.

Hide   Copy Code

SELECT EndVertex, Hops

FROM Edge

WHERE StartVertex = 'Jale'

This returns:

Hide   Copy Code

EndVertex Hops

--------------- -----------

ABCTechnicians 0

Technicians 1

Users 2

(3 row(s) affected)

Jale is a member of the Users group. She is not **directly** a member, as the Hops count is 2, which also indicates that there are two **intermediate** vertices (ABCTechnicians and Technicians) that have to be visited to traverse from Jale to Users.

Number of Rows on the Edge Table

It is obvious from the code of AddEdge that the number of rows in the Edge table, at least in theory, can be very large as we maintain the **transitive closure**. In reality, the number of **implied** rows in the table is very much dependent on the application. Hence the complexity of the graph. Let us try to find some estimate of the expected number of edges for a **non-existent** average case. Assume that we have two discrete graphs with a total number of v vertices and e edges. **On average**, both of the graphs would then have...

Average number of edges per vertex

...edges per vertex that are **incoming** or **outgoing**. When you look at the code of AddEdge, you can see that the addition of a single new edge would create a total of:

|  |  |  |
| --- | --- | --- |
| Step 1: | Number of edges on step 1 formulae | implied edges |
| Step 2: | Number of edges on step 2 formulae | implied edges |
| Step 3: | Number of edges on step 3 formulae | implied edges per direct edge |

Hence, the total number of edges that a single call to AddEdge would create:

|  |  |
| --- | --- |
| Total number of edges formulae | per direct edge |

Or

|  |  |
| --- | --- |
| Total number of edges formulae | the total edges for the newly connected graph |

[Table 6](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Table6) lists some calculation examples of the formulae above.

|  |
| --- |
| ***Table 6: Sample Average Calculations for Some Number of Vertices and Edges*** |
|  |
| Graph # | v | *e* | *a* | *t* | *T* |
| 1 | 10 | 10 | 0.50 | 2.25 | 23 |
| 2 | 50 | 100 | 1.00 | 4.00 | 400 |
| 3 | 100 | 300 | 1.50 | 6.25 | 1,875 |
| 4 | 500 | 1,000 | 1.00 | 4.00 | 4,000 |
| 5 | 1,000 | 5,000 | 2.50 | 12.25 | 61,250 |

As is apparent from both the formulae and the table, the number of implied edges explodes as the ratio of edges to vertices increases. This is, of course, expected. That high number of edges per vertex simply means that there are many more implied paths to be included in the table.

A Word of Warning

Please note that many assumptions are made to produce the formulae and [Table 6](http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o#Table6). They are here only to give a rough idea of how many rows to expect for the non-existent **average** case. What is certain is that the number of rows expected for a graph is **at least** in the order of **O(e2)**, potentially a lot higher because we assumed a somehow **uniform** distribution of edges among vertices. In theory, the size of the transitive closure set of a fair DAG can be **very large** with this model, well beyond the millions. The maximum number of edges for a given DAG itself is a research topic in Graph Theory, but my practical tests show that there exist DAGs with 100 vertices and 300 edges whose transitive closure would create well beyond 20,000,000 rows with this algorithm.

Fortunately, such **arbitrarily complex** graphs are rare. They exist in areas such as biomolecular research that need supercomputers anyway. In general, as the similarity of the DAG to a **balanced tree** increases, the number of implied rows decreases **considerably**. Moreover, we put the restriction that with the **existence of a single instance** of any implied link, the number of the rows on the table would enormously be reduced (from millions to several thousands) at the cost of **losing capability to delete individual edges** as noted before. So, even complex cases -- i.e. where changes on the network are limited -- can benefit from this model. In this case, the whole graph should be re-created from scratch after each deletion, which is not so bad as it seems.

Conclusion

The use of the **adjacency list** method, augmented with addition of **transitive closure** rows, enables standard SQL to be used with a model of Directed Acyclic Graphs (DAGs). The price to be paid in return for the expressive power of SQL and **very fast** results is some additional disk space and additional **insertion time**, which I find quite reasonable to pay for the problems on my hand.

References

Below is the related work that you may find useful for your own work. Please let me know if you are aware of any other work, so that I'll add to this section.

* Guozhu Dong et al., "Maintaining Transitive Closure of Graphs in SQL" <http://www.comp.nus.edu.sg/~wongls/psZ/dlsw-ijit97-16.ps>

[History](http://www.comp.nus.edu.sg/~wongls/psZ/dlsw-ijit97-16.ps)

* [10 January, 2008 -- Original version posted](http://www.comp.nus.edu.sg/~wongls/psZ/dlsw-ijit97-16.ps)

[License](http://www.comp.nus.edu.sg/~wongls/psZ/dlsw-ijit97-16.ps)

[This article, along with any associated source code and files, is licensed under](http://www.comp.nus.edu.sg/~wongls/psZ/dlsw-ijit97-16.ps) [The Code Project Open License (CPOL)](http://www.codeproject.com/info/cpol10.aspx)