QlikView

Hierarchies

QlikView

Contents

Introduction

Hierarchies are an important part of all business intelligence solutions, used to describe dimensions that naturally contain different levels of granularity. Some are simple and intuitive whereas others are complex and demand a lot of thinking to be modeled correctly.

From the top of a hierarchy to the bottom, the members are progressively more detailed. For example, in a dimension that has the levels Market, Country, State and City, the member Americas appears in the top level of the hierarchy, the member U.S.A. appears in the second level, the member California appears in the third level and San Francisco in the bottom level. California is more specific than U.S.A., and San Francisco is more specific than California.

This document tries to describe which types hierarchies there are, define some basic attributes and explain how they should be modeled and loaded into QlikView.

Examples in real life

The time dimension

Year, month, day, hour, minute and second are attributes that form a hierarchy.

The product dimension

A product can often be delivered in different packages, e.g. in different sizes. At the same time a product belongs to a product group. Hence, product group, product and package are attributes that form a hierarchy. A product from the pharmaceutical industry can serve as example:

- Product group: Pain killers
- Product: Paracetamol
- Package: Paracetamol package with 20 x 500 mg.

The geography / customer dimension

A customer is usually associated with a City, a State, a Country and a Market region. These fields form a hierarchy.

The wine districts of the world

A wine always has an origin. It could for instance come from Bordeaux, which is part of France, which in turn is part of Europe and the World. Bordeaux also has sub-levels such as Graves, Sauternes and Médoc, which in turn also have sub-levels such as Haut-Médoc and Barsac. These wine-districts form a hierarchy.

Different types of hierarchies

A hierarchy always consists of a number of members – nodes – that have one parent each. In the general case, each node can have any number of children. This way, a hierarchy often looks like a tree: A starting point – the root – and a structure of nodes that branches out from the root.

Sometimes you encounter "hierarchies" where some nodes have more than one parent. These are strictly speaking really not hierarchies at all, but more general directed graphs.

Just like a real tree, a hierarchy can also have leaves. These are objects that usually are of a different type than the nodes, but still belong to a specific node.

In a database situation, the nodes in a hierarchy typically form the dimension whereas the leaves usually are the transactions in the fact table, associated to one specific node each.

Balanced or Unbalanced?

One property of a hierarchy is whether it is balanced or unbalanced. "Balanced" means that all leaves belong to nodes of the same level.

One good example is the calendar dimension: In the common case, all branches go down to the bottom level, the dates, and all leaves, e.g. orders, invoices or some other transaction type, have dates and are linked to this level.
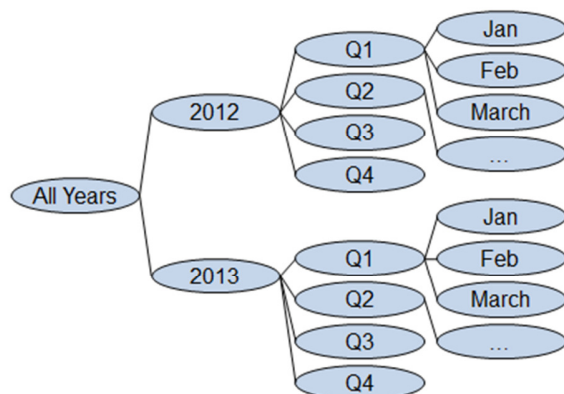
The opposite case is if the hierarchy is unbalanced. In such a case you have consistent parent-child relationships but logically inconsistent levels. This means that all leaves are not connected to the same level: they can be found on different levels.

A good example is the wine districts of the world: A bottle of wine often has its origin written on the label – a specific wine district. But a district can belong to a bigger district, which in turn can belong to an even bigger district. Depending on whether the wine is specified to come from one specific vineyard or is a bulk wine from a larger area, origin can be more or less precise. So, in principle, a wine can be labeled to belong to *any* of the nodes in the tree. It can for instance be an unspecified table wine from Bordeaux or it can be a better wine that comes from one of the sub districts of Bordeaux, e.g. Bas-Médoc.

That the levels are "logically inconsistent" means that a wine district is not necessarily organized or named the same way as in another district. France has, for instance, a different system for denomination of wine districts than Germany.
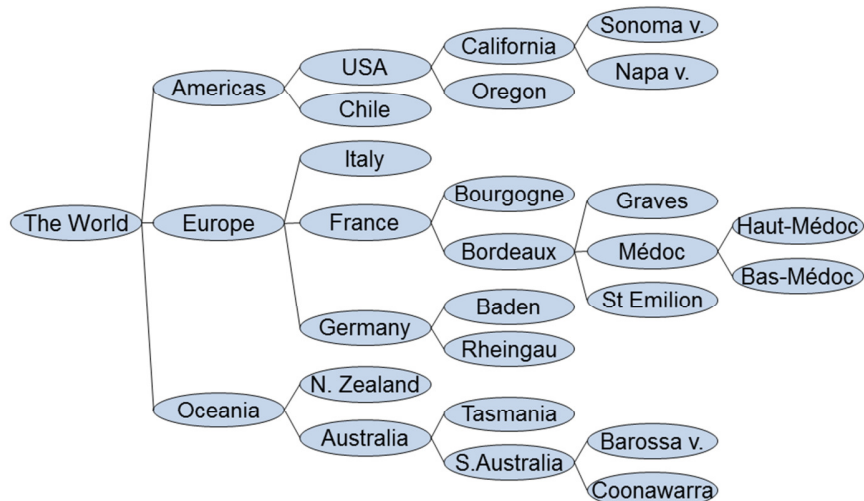
Fix-level or n-level?

Closely related to balanced/unbalanced is the question of number of levels. An unbalanced hierarchy usually has an unknown, maybe even dynamic, number of levels, whereas a balanced hierarchy usually has a fix number of levels, well defined when you decide your data model.

But the two concepts – fix-level vs. balanced – are still not the same thing. There are fix-level, unbalanced hierarchies: An example is if you have a fact table with mixed granularity. Then you have a fix-level calendar hierarchy (Year, Quarter, Month, Day) while at the same time you have transactions that link to different levels in the hierarchy: It could be that the actual numbers are linked to dates, but the budget numbers are linked to months. Hence – an unbalanced hierarchy.

Named levels or non-named levels?

The main question, however, when loading data into QlikView, is whether the levels have specific names or not. If a level has a name, you most likely want to display that level in QlikView as a field with that name. But if you instead have unnamed levels, the situation is quite different. Again, the wine districts can serve as example: A wine district could be referred to as "District" irrespectively if it is a district on the third level or on the fifth level of the hierarchy. In such a case, you do **not** want the different levels in different list boxes.

Ragged or not ragged?

A special case of a hierarchy with named levels is the ragged hierarchy. In a normal, balanced, fix-level hierarchy, all branches of the hierarchy descend to the same level, and each member's logical parent is the level immediately above the member.

But in a ragged hierarchy, this need not be true. In a ragged hierarchy, there may be levels missing in some branches.

One good example is The U.S. states and cities: Cities always belong to a state – except Washington DC. This city does not belong to any state, but still belongs to the U.S. Hence, this is a ragged hierarchy where the state is missing for this node.
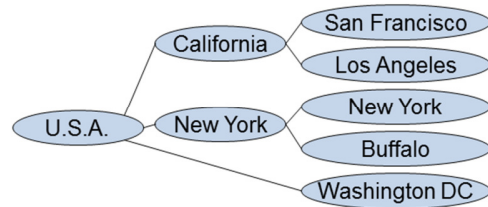
How a hierarchy is stored in a database

Storing hierarchies in a relational model is a common challenge, with multiple solutions. There are several approaches:

- The Horizontal hierarchy
- The Adjacency list model (Adjacent Nodes)
- The Path enumeration method
- The Nested sets model (Tree traversal)
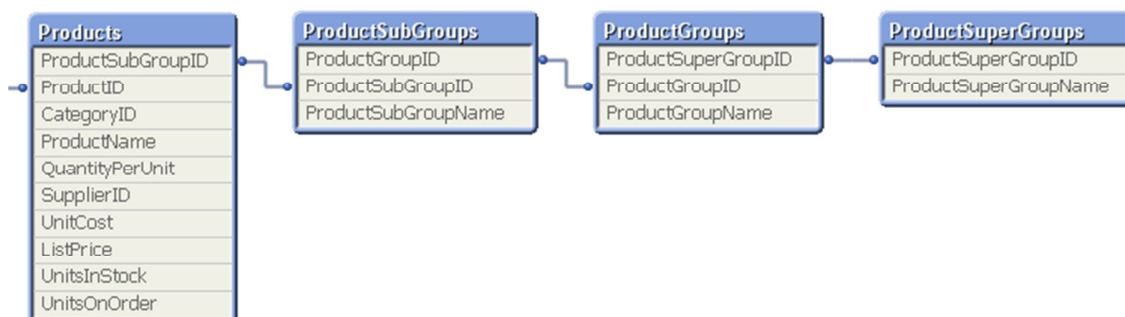- The Ancestor list (Reflexive Transitive Closure)

There is no general rule for how to load a hierarchy. It all depends on what type of hierarchy you have, and how this is stored. You will need to check your data to find out in which form the hierarchy is stored and use the appropriate loading algorithm. Below you will find descriptions for the most common cases.

The Horizontal hierarchy – Each level in its own field

The most common way to store a simple hierarchy is to have names on the levels and store each level in its own field.

ID	Customer	Country	Region
1	Eintrach GS	Germany	Europe
2	La Tienda de la Esquina	Mexico	Americas
3	La Ropa Vieja	Mexico	Americas
4	Dr Jims Trousers	UK	Europe
5	Urras Shop	Sweden	Europe
6	Man Kleider	Germany	Europe
7	Menàge à Trois	France	Europe
8	Las Corbatas	Spain	Europe
9	La Legion Mercenaire	France	Europe
10	Big Foot Shoes	Canada	Americas
11	Shoe Expert	UK	Europe

The hierarchy need not be in one table only, but can be split in several tables, e.g. one table for products and several ones for different levels of product groups.

Products
ProductSubGroupID
ProductID
CategoryID
ProductName
QuantityPerUnit
SupplierID
UnitCost
ListPrice
UnitsInStock
UnitsOnOrder

ProductSubGroups
ProductGroupID
ProductSubGroupID
ProductSubGroupName

ProductGroups
ProductSuperGroupID
ProductGroupID
ProductGroupName

ProductSuperGroups
ProductSuperGroupID
ProductSuperGroupName

The most common case of a horizontal hierarchy is a balanced, fix-level hierarchy with named levels. This means that all the transactions link to the same level in the hierarchy, i.e. to one single field. Typically this field is a date, a customer ID or a product ID. Examples of such hierarchies are:

- Year – Quarter – Month – Day
- Customer – State – Country – Market
- Product Group – Product – Package

Just load the table and create a drill-down group from the appropriate fields in the hierarchy (Document properties – Groups). Then use the drill-down group as a field in charts, tables and list boxes.

However, sometimes you have an unbalanced, fix-level hierarchy with named levels.

You cannot see by looking at the dimensional table that you have an unbalanced, fix-level hierarchy. Instead, you must look at the facts and see whether the transactions always link to the same field or not.

One case where they don't, is when you have both budget and actual numbers in one single fact table. It could be that the budget numbers are per month and country whereas the actual numbers all have a timestamp and a customer ID.

The way you load this in QlikView is by using generic keys. See the Technical Brief "Generic Keys" for more information.

The Adjacency list model

The adjacency list model, or Adjacent nodes table, is by far the most common way to store an unbalanced n-level hierarchy. The principle is that each node is stored in its own record, and that each record has a foreign key pointing out the parent. In other words; there is *exactly one record per node*.

NodeID	ParentID	Name
1		The World
20	1	Americas
79	20	United States
85	79	California
90	85	Napa Valley
178	1	Europe
281	178	France
283	281	Bordeaux
288	283	Médoc
294	288	Haut-Médoc
354	178	Germany
368	354	Rheingau

The ID is the primary key that links the hierarchy to other data, typically transactions or inventory records.

The root is defined by the record that has NULL or blank as parent. Several roots are possible, although it often is preferable to have one common root node.

The main advantage with this model is that it is extremely easy to manage and maintain. If a node is to be added, it does not affect any other node. If an entire sub-tree needs to be moved, the ParentID of the root of the sub-tree is changed, and that is all.

The table completely defines the hierarchy, but since it doesn't explicitly store the information about daughters, it needs to be transformed to be usable in QlikView. This transformation can be made

using one of the two hierarchy-resolving load prefixes in QlikView: Hierarchy and HierarchyBelongsTo. See more below about these.

Path Enumeration

Similar to the adjacency list, is the Path enumeration. It is also a general way to store an unbalanced n-level hierarchy.

NodeID	Path	Name
1	1	The World
20	1/20	Americas
79	1/20/79	United States
85	1/20/79/85	California
90	1/20/79/85/90	Napa Valley
178	1/178	Europe
281	1/178/281	France
283	1/178/281/283	Bordeaux
288	1/178/281/283/288	Médoc
294	1/178/281/283/288...	Haut-Médoc
354	1/178/354	Germany
368	1/178/354/368	Rheingau

However, instead of storing the parent ID explicitly, a path to the node is stored. It has the advantage that some SQL queries are easier to perform. The disadvantage is manageability. It is not as easy to change the structure or to move a tree.

Also this table completely defines the hierarchy, and also here the table needs to be transformed to be usable in QlikView.

The Nested sets model

The nested sets model is an additional, general way to store an unbalanced n-level hierarchy.

LBound	RBound	Name
1	1382	The World
38	313	Americas
155	308	United States
166	189	California
175	176	Napa Valley
354	1195	Europe
559	702	France
562	613	Bordeaux
571	588	Médoc
582	587	Haut-Médoc
705	750	Germany
732	737	Rheingau

Instead of having a key to the parent, each node is associated with a numeric range; a lower bound and an upper bound. All descendants have ranges that are completely enclosed by all ancestors. So by looking at whether a number of a node is enclosed by the bounds of another node, the parent-child relationship can be determined.

Also this table completely defines the hierarchy, and also here the table needs to be transformed to be usable in QlikView.

The nested sets model is not as commonly used as the adjacency list model, since it is not as easy to make changes. However, the nested sets model is usually faster to query on an SQL database than the adjacency list model.

The Ancestor table

The ancestor list, or the Reflexive Transitive Closure table, is not commonly used to store the source data. But it is very common that a database view is defined this way, since it presents the hierarchy in a form that is directly usable in a query.

In this table, every combination of an ancestor and a descendant is listed as a separate record. Hence, it is very easy to find all ancestors or all descendants of a specific node.

| NodeID	Name	BelongsToID	BelongsTo
1	The World	1	The World
20	Americas	1	The World
20	Americas	20	Americas
178	Europe	1	The World
178	Europe	178	Europe
79	United States	1	The World
79	United States	20	Americas
79	United States	79	United States
281	France	1	The World
281	France	178	Europe
281	France	281	France
354	Germany	1	The World

Sometimes it includes records that are self-references, where a node belongs to itself, e.g. 'France' belongs to 'France', sometimes not.

In QlikView, this table can be created using the HierarchyBelongsTo prefix.

Tools in QlikView

The Drill down group

In QlikView you can group fields together to form a drill-down group (Document Properties -> Groups). This means that you can use the group instead of the separate fields in charts and list-boxes. If there is only one value possible in the top field, it will automatically display the field of the next level instead.
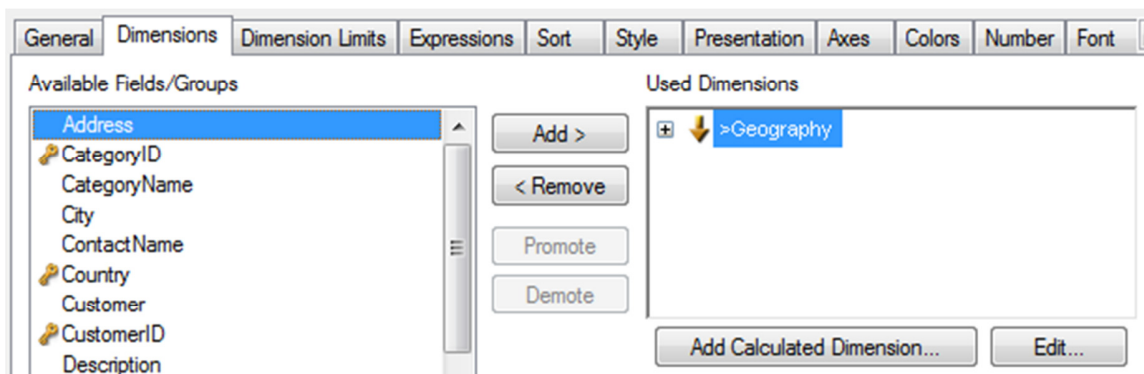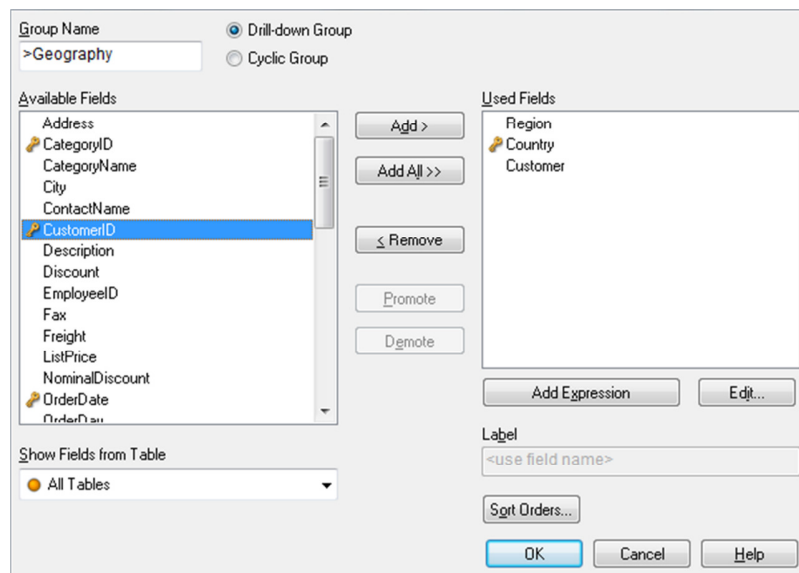
The Hierarchy prefix

The hierarchy prefix is a script command that you put in front of a Load or SELECT statement that loads an adjacent nodes table:

> Hierarchy (NodeID, ParentID, Name)
> **Load** NodeID,
>   ParentID,
>   Name
>   From Examples\Winedistricts.txt ;

NodeID	ParentID	Name
1		The World
20	1	Americas
79	20	United States
85	79	California
90	85	Napa Valley
178	1	Europe
281	178	France
283	281	Bordeaux
288	283	Médoc
294	288	Haut-Médoc
354	178	Germany
368	354	Rheingau

The Load statement needs to have at least three fields: An ID that is a unique key for the node, a reference to the parent and a name.

The reference to the parent should have a match among the node IDs. If it doesn't, or if it is NULL, the node will be considered a root node. The node name has to be there – if your source table lacks node name, you need to create a name using "NodeID as Name".

It is also possible to have additional fields.

The prefix will transform the loaded table into an Expanded Nodes table; a table that has a number of additional columns; one for each level of the hierarchy:

NodeID	ParentID	Name	Name1	Name2	Name3	Name4	Name5	Name6
1		The World	The World	-	-	-	-	-
20	1	Americas	The World	Americas	-	-	-	-
79	20	United States	The World	Americas	United States	-	-	-
85	79	California	The World	Americas	United States	California	-	-
90	85	Napa Valley	The World	Americas	United States	California	Napa Valley	-
178	1	Europe	The World	Europe	-	-	-	-
281	178	France	The World	Europe	France	-	-	-
283	281	Bordeaux	The World	Europe	France	Bordeaux	-	-
288	283	Médoc	The World	Europe	France	Bordeaux	Médoc	-
294	288	Haut-Médoc	The World	Europe	France	Bordeaux	Médoc	Haut-Médoc
354	178	Germany	The World	Europe	Germany	-	-	-
368	354	Rheingau	The World	Europe	Germany	Rheingau	-	-

Note that the resulting Expanded Nodes table has exactly the same number of records as its source table: One per node. This will be true in all well-formed hierarchies. There are however some exceptions, see below under "Data Integrity".

The Expanded Nodes table is very practical since it fulfills a number of requirements for analyzing a hierarchy in a relational model:

• All the node names exist in one and the same column, so that this can be used for searches.

- In addition, the different node levels have been expanded into one field each; fields that can be used in drill-down groups or as dimensions in pivot tables.
- It can be made to contain a path unique for the node, listing all ancestors in the right order.
- It can be made to contain the depth of the node, i.e. the distance from the root.

The HierarchyBelongsTo prefix

Just as the Hierarchy prefix, the HierarchyBelongsTo is a script command that you put in front of a Load or SELECT statement that loads an adjacent nodes table:

> HierarchyBelongsTo (NodeID, ParentID, Name, BelongsToID, BelongsTo)
> **Load** NodeID,
>   ParentID,
>   Name
>   From Examples\Winedistricts.txt ;

Also here, the Load statement needs to have at least three fields: An ID that is a unique key for the node, a reference to the parent and a name.

The prefix will transform the loaded table into an Ancestor table – a reflexive transitive closure table – a table that has every combination of an ancestor and a descendant listed as a separate record. Hence, it is very easy to find all ancestors or all descendants of a specific node.

NodeID	Name	BelongsToID	BelongsTo
1	The World	1	The World
20	Americas	1	The World
20	Americas	20	Americas
178	Europe	1	The World
178	Europe	178	Europe
79	United States	1	The World
79	United States	20	Americas
79	United States	79	United States
281	France	1	The World
281	France	178	Europe
281	France	281	France
354	Germany	1	The World

The Ancestor table is very practical since it fulfills a number of requirements for analyzing a hierarchy in a relational model:

- If the node ID represents the single nodes, the ancestor ID represents the entire trees and sub-trees of the hierarchy.
- All the node names exist both in the role as nodes and in the role as trees, and both can be used for searches.
- It can be made to contain the depth difference between the node depth, and the ancestor depth, i.e. the distance from the root of the sub-tree.

![QlikView logo]

The Tree-view list box

If you have a path to the node, it is possible to display this as a tree by using a list box in Tree-view mode (List box properties -> General -> Show as Tree view). This simplifies navigation and allows you to collapse entire sub-trees.

In the left list box you can see what the field "Path" looks like originally, and in the right you can see how the tree-view displays this information.

The Pivot table

The pivot table is excellent to display hierarchies, especially if you use the "Indent mode". To do this, you need the different levels in different fields.

Data modeling

So, what do you need to do to create a hierarchical data model that supports your analysis? If you have a fix-level hierarchy with named levels, it is straightforward. Just load the data and use drill-down groups, and you are pretty much set.

But what if you have an unbalanced, n-level hierarchy stored in an adjacent nodes table? There is not just one answer to that question. As usual, there is more than one way to solve a problem with QlikView. Below you will find my suggestions for this case.

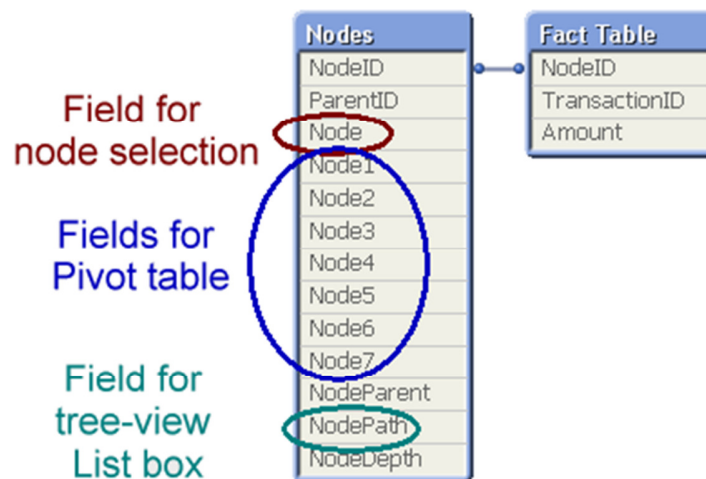### The Expanded Nodes table – to describe the nodes

First, you should create an Expanded nodes table using the Hierarchy prefix.

```
[Nodes]:
Hierarchy (NodeID, ParentID, Node, NodeParent, Node, NodePath, '/', NodeDepth)
Load NodeID,
     ParentID,
     NodeName as Node
From Winedistricts.txt;
```

With the columns listing the different node levels (Node1...Node7), you will be able to create a pivot table showing the proper aggregations from the fact table. Further, the NodePath field can be used in a tree-view list box.

The Ancestor table

The Expanded Nodes table above, however, fails in one aspect: Selections. Yes, you can select nodes, but the most common selection a user wants to make, is an entire sub-tree. I.e. by pointing at a node and clicking it, the user wants a result where all sub-nodes are possible.

To create a field where you can do this, you need to create an Ancestor table. Hence, the second step in creating the data model is to create an ancestor table listing the trees:

```
[TreeBridge]:
HierarchyBelongsTo (NodeID, ParentID, NodeName, TreeID, TreeName)
Load NodeID,
     ParentID,
     NodeName
     From Winedistricts.txt;
```

The TreeName field is perfect for making searches and selections of entire sub-trees. Further, it is also perfect for authorization purposes, e.g. as reducing field in Section Access.

Note that you need to name the node name differently here from in the Expanded Nodes table, to avoid synthetic keys.

## The Expanded Nodes table – to describe the trees

Finally, it is also possible to create a second Expanded Nodes table, but this time for the TreeIDs.

```
[Trees]:
Hierarchy (TreeID, TreeParentID, Tree)
Load NodeID as TreeID,
     ParentID as TreeParentID,
     NodeName as Tree
     From Winedistricts.txt;
```

The advantage with a second Expanded Nodes table is that the fields describing the levels (Tree1...Tree7) are perfect for creating a drill-down group that can be used in different charts.

Note that you need to name the tree name differently here from in the Ancestor table, to avoid synthetic keys.

The final data model may look like the following: You may not need all three tables, so use the ones you need.
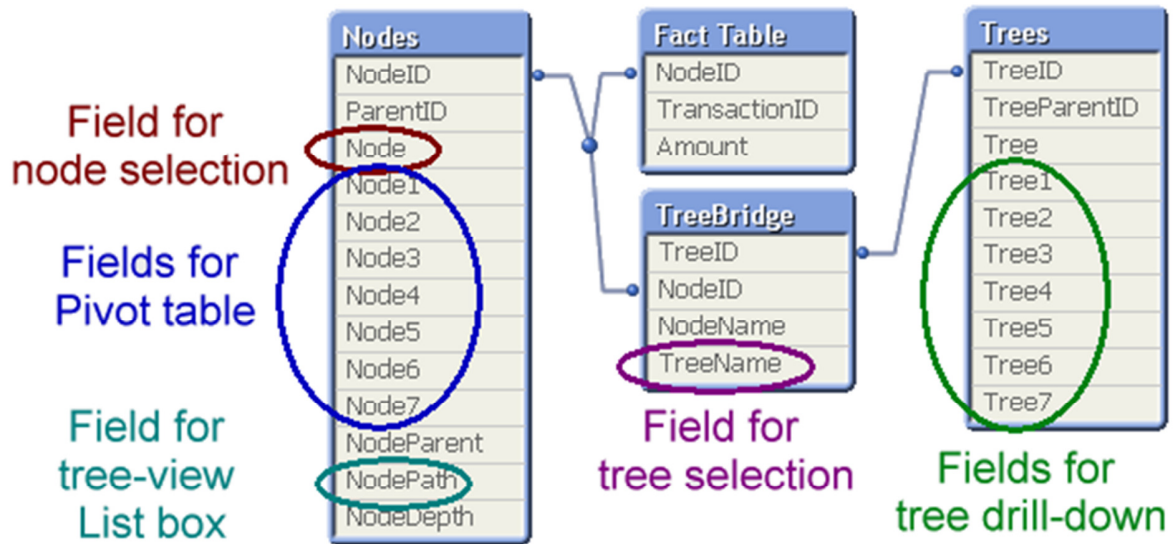
Authorization

It is not uncommon that a hierarchy is used for authorization. One example is an organizational hierarchy. Each manager should obviously have the right to see everything pertaining to their own department, including all its sub-departments. But they should not necessarily have the right to see other departments.
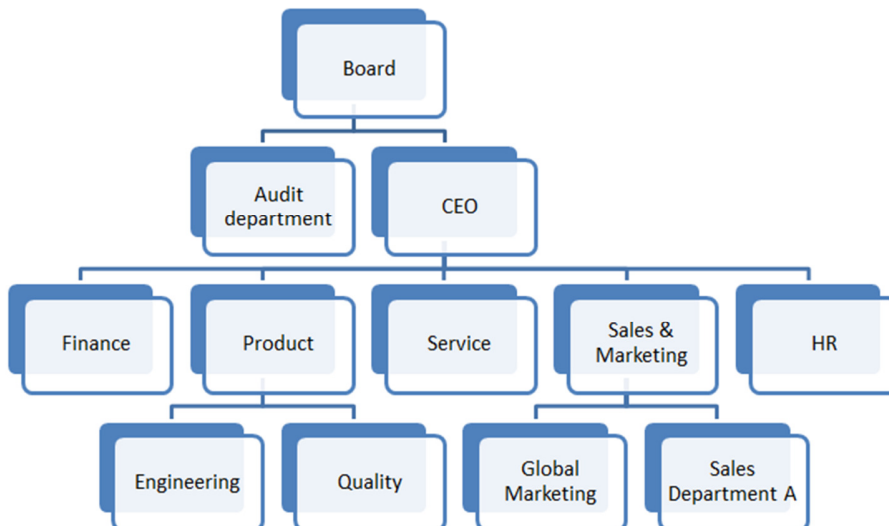
This means that different people will be allowed to see different sub-trees of the organization. The authorization table may look like the following:

ACCESS	NTNAME	Person	Position	Permissions
USER	ACME\BKO	Bill	CPO	HR
USER	ACME\DKN	Diane	CEO	CEO
USER	ACME\DPT	Debbie	Director Engineering	Engineering
USER	ACME\JOO	John	CFO	Finance
USER	ACME\LBY	Les	COO	Sales & Marketing
USER	ACME\SDN	Steve	CTO	Product

In this case, Diane is allowed to see everything pertaining to the CEO and below; Steve is allowed to see the Product organization; and Debbie is allowed to see the Engineering organization only. Hence, this table needs to be matched against sub-trees in the above hierarchy.

### Case 1: The source is in an Adjacent Nodes table

It the hierarchy is stored in an Adjacent Nodes table, the above problem is easy to solve: Just load the Adjacent nodes table as an Ancestor table (see above) using a HierarchyBelongsTo and name the ancestor field "Tree".

Then you need to load an upper case copy of *Tree* and call this new field *PERMISSIONS*. Finally, you need to load the authorization table. These two last steps can be done using the following script lines: (The TempTrees table is the table created by the HierarchyBelongsTo.)

```
Trees:
Load *,
      Upper(Tree) as PERMISSIONS
      Resident TempTrees;
Drop Table TempTrees;

Section Access;
Authorization:
Load  ACCESS,
      NTNAME,
      UPPER(Permissions) as PERMISSIONS
From Organization;
Section Application;
```

When you have done this, you should have a data model that looks like the following:
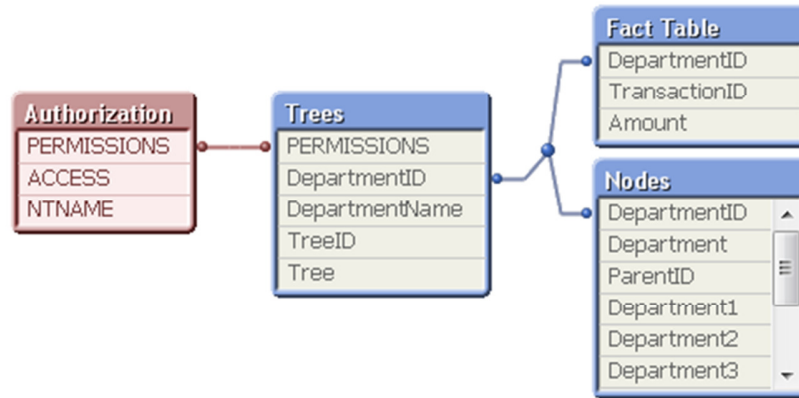
The red table is in Section Access and is invisible in a real application. Should you want to use the publisher for the reduction, you can reduce right away on the Tree field, without loading the Section Access.

This solution will effectively limit the permissions to only the sub-tree as defined in the authorization table.

Case 2: The source is in a Horizontal table

But what if you have the hierarchy in a horizontal hierarchy? Then you cannot use the HierarchyBelongsTo.

DepartmentID	Name	Board level	Top level	Department	Unit
1	Board	Board	-	-	-
2	Audit department	Board	Audit department	-	-
4	CEO	Board	CEO	-	-
5	Finance	Board	CEO	Finance	-
13	Engineering	Board	CEO	Product	Engineering
14	Quality	Board	CEO	Product	Quality
16	Global Marketing	Board	CEO	Sales & Marketing	Global Marketing
17	Sales Department A	Board	CEO	Sales & Marketing	Sales Department A
18	Sales Department B	Board	CEO	Sales & Marketing	Sales Department B
23	Logistics Group	Board	CEO	Service	Logistics Group
26	Salary Group	Board	CEO	HR	Salary Group

A horizontal hierarchy

The solution is not very different from the above one. The only difference is that you need to create the bridging Trees table manually, e.g. by using a loop:

```
Let vHierarchyDefinition = 'Board level,Top level,Department,Unit';
Let vNumberOfLevels    = Len(KeepChar(vHierarchyDefinition,',')) + 1 ;

For vAncestorLevel = 1 to vNumberOfLevels
    Let vAncestor = Subfield(vHierarchyDefinition,',',vAncestorLevel);
    Trees:
    Load distinct
         Upper([$(vAncestor)]) as PERMISSIONS,
         DepartmentID
    Resident [Horizontal Hierarchy];
Next vAncestorLevel
```

Having done this, you will have the following data model:
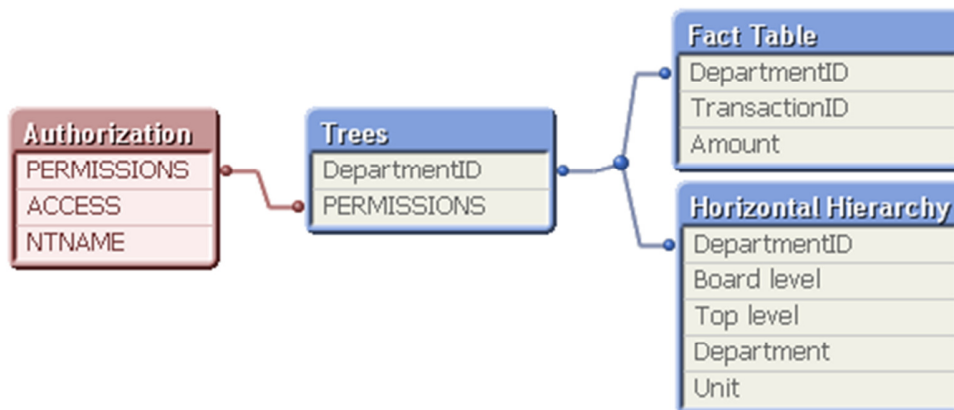
Data integrity

A common problem with unbalanced n-level hierarchies is data integrity. The data may be inconsistent and it could be a good idea to check that the source table contains what you expect it to contain. And maybe correct the source…

1) **Duplicates or Multiple parents**

   Source data may contain duplicate records for single nodes, e.g. that a node is described in two records with different names or that a node has multiple parents.

   You need to determine whether you think this is OK. QlikView can load this type of data, but it often looks inconsistent in the eyes of the user. I would recommend _not_ having duplicates.

Also, it could imply performance problems. Loading such a table will mean duplicate rows in the resulting Expanded Nodes table, so that a daughter node gets not only two records, but four or eight or more, depending on how many of its ancestors have multiple parents.

The following script may help you find the duplicates:

```
Load NodeID,
        Count(NodeID) as NoOfInstances,
        Count(distinct ParentID) as NoOfParents
        From Source
            Group By NodeID;
```

2) **Unlisted parent nodes**

Each node need a record of its own in the Adjacent Nodes table, but it is not uncommon that some parent nodes, especially the root node, are missing from this list.

You need to determine whether you think this is OK. QlikView can load this type of data, but this may lead to the hierarchy having several root nodes and that root nodes aren't selectable.

The following script may help you find the unlisted nodes:

```
Load NodeID
        From Source ;

Load *,
        If(Exists(NodeID,ParentID), 'Existing','Non-existent') as ParentExistence
        From Source ;
```

3) **Circular references**

If the source table contains a circular reference, e.g. node A has node B as parent, node B has node C as parent, and node C has node A as parent, the hierarchy resolution will break up the circle and remove one of the parent IDs. In more complex situations it may result in that some nodes are excluded.

My recommendation is to _always avoid_ such structures.

The following script may help you find the circular references:

```
Tree:
Load NodeID as Level_1
        From Source ;

For nLevel = 0 to 20
        Let nLevelBelow = nLevel + 1 ;
        Left Join (Tree)
```

```
Load ParentID as Level_$(nLevel),
     NodeID as Level_$(nLevelBelow)
     From Source.qvd ;

CountAddedNodes:
Load Count(Level_$(nLevelBelow)) as NoOfAddedNodes Resident Tree;

For nAncestorLevel = 0 to nLevel
    AncestorScan:
    Load Level_$(nLevelBelow) as NodeIDThatAlreadyExists
        Resident Tree
            Where Level_$(nAncestorLevel) = Level_$(nLevelBelow);
Next nAncestorLevel

Let vExitCriterion1 =
    Peek('NoOfAddedNodes',-1,'CountAddedNodes') = 0 ;
Let vExitCriterion2 =
    not IsNull(Peek('NodeIDThatAlreadyExists',-1,'AncestorScan'));

    exit for when vExitCriterion1 or vExitCriterion2
Next nLevel
```

HIC