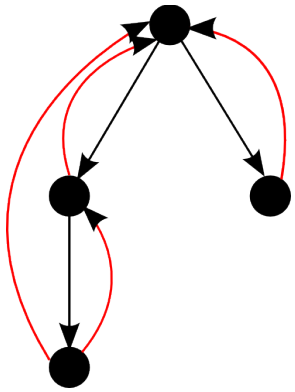
1 The memberof plugin

The purpose of this plugin is to compute direct and indirect memberships of user or group entities which are marked to be a member of other group entities.

The plugin computes and stores two attributes that are read-only outside of the ldb and its plugins. These attributes are **memberof** and **memberuid**. The former one is a transitive closure of relationship opposite to that defined by attribute *member*. That means if group *A* has attribute member with value *B* and group *B* has member attribute with value *C*, than *C* will have memberof attribute with values *A* and *B*. Visually it can be displayed on following graph:



The *memberuid* on the other hand contains a list of all member users of a particular group, including those which are not directly members of that group, but are members of one of groups in the group tree rooted in that particular group.

2 The memberofRefCount attribute

The **memberofRefCount** was designed to solve high computational demands of deleting groups / memberships in more complex group trees – in that case the algorithm had to scan and recompute the entire tree, which could potentially take a long time.

The basic idea of reference counting is to create the memberofRefCount as internal, read-only attribute, where *a number of paths leading from group A to group B in the group membership graph* will be stored.

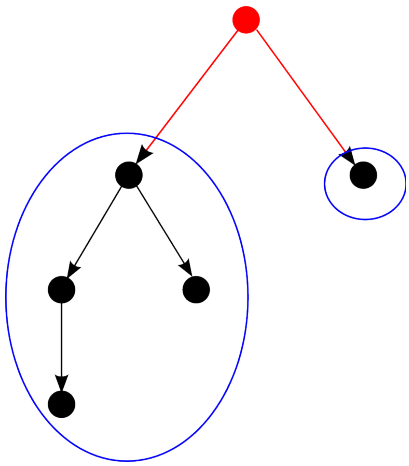
The attribute has following syntax:

`memberofrefcount: DN:RC`

DN is distinguished name of ancestor and RC is the number of paths leading to it. In the following text, a *value of memberofrefcount linking* ***`A`*** will refer to the value of RC with DN being **`A`**.

2.1 Member subtrees

In following text, the concept of member subtrees is widely used. When adding or removing a relationship between group P (parent) and C (child), a member subtree is a group tree with root being the group C. Such subtree will contain the group C and all of its descendants (i.e. all nodes that have group C stated in their memberof attribute). This is graphically displayed on following graph. Added group with values of its member attribute is red, each subtree is marked by blue circle.



2.2 Inverse member subtree

Inverse member subtree follows the same concept as ordinary member subtree. It just doesn't follow the member attribute, but rather memberof attribute. Therefore it contains all direct and indirect ancestors of the group. Since the memberof attribute is transitively closed, all it takes to construct this tree is to extract all values of memberof attribute of the root node.

2.3 Root refcount

Consider an entity B which is a part of member subtree rooted in group A. The root reference count is a value of memberofrefcount linking A to B. The root refcount won't be used to refer to one particular relationship, it will rather designate this relationship for every node in the subtree.

2.4 Hash tables

In current implementation, hash tables are widely used. This makes duplicate detection and similar operations computationally much simpler than lists/arrays. The key in the hash table is always the DN and a number is used as value – it can either store the RC number or, in cases we just want some duplicate checking, it is just 1.

3 Adding a new group with memberships

After group **A** is stored in the database, its inverse member subtree is stored in a form of hash table.

Then we iterate through all values of member attribute of the added node A. For each of them we store the the member subtree and then we update the subtree.

The update consists of iterating through all nodes in the subtree. For each node a root refcount is computed. Then all its current values of memberof and memberofrefcount attributes are inspected and if any of them is the in ancestor hash table, the value is udated as follows:

$$RC_{new} = RC_{old} + RC_{root} * RC_{hash}$$

RC_{old} – the original value of RC stored in the memberofrefcount attribute

RC_{root} – the previously computed root refcount

RC_{hash} – tha value stored in the ancestors hash table, designating RC from A to that particular parent

After all values in original memberof attribute are inspected and refcounts updated, those ancestors that weren't processed during it (i.e. the node wasn't their descendant before) are added, all computed values are stored and the next node in the member subtree is processed.

4 Deleting a group with memberships

This operation is similar to the adding operations. First, some information about the deleted node **A** are looked up. After is it deleted, all memberships of its direct parents are modified so they does not contain it any more.

Then the process is almost the same as adding. A member subtree is loaded, **A** being its root. The root itself is of course excluded from the subtree. Basically, all entities that contain the node **A** in their memberof attribute are looked up.

For each descendant a root refcount (linking it to **A**) is fetched and then refcount for each ancestor of **A** is decreased as following:

$$RC_{new} = RC_{old} - RC_{root} * RC_{hash}$$

Semantics of all RCs are the same as stated in section 2.5. If $RC_{new} > 0$, it is added to the replace message (i.e. if it is zero, the value will be deleted).

5 Updating memberships between existing groups

The approach here is based on previously described approaches. If we are simply adding some memberships, the process is the same as it is when adding new node – member subtree is loaded for each new membership and all nodes in it are updated accordingly.

The same situation is when deleting some memberships. The only difference is that the algorithm asks for each member subtree individually, there isn't just one query for all descendants of the parent.

If a replacement is to be performed, a hash table with items to be removed (initially all original values) is constructed. Then it is cross-referenced with the list of to-be-added values. During this operation, the original table is pruned and another table, containing values to be added, is created. Adding and deleting operations then follow as described before.

6 Design problems - loops

The first issue raises when considering circular membership. In any loop is created, all values are originally computed correctly. Where memberof existed before, another one starts existing through the created loop and the refcount is incremented. Where it didn't exist, a new one is created through the loop. The problem now is when a membership is deleted, we somehow need to detect how many references to substract from existing memberships. Even in simple loop 0-1-2-3-4-5-6-0 it is possible to get badly burned – consider deleting one of memberships existing before the loop was created. In that case some memberofrefcounts have to be decreased by 2 and some of them by 1. Many more examples exist even with one loop

7 Loop resolving solution

Brute force tree analysis is the only option that will cover every corner case. It is very similar to what is done now. We would basically traverse the entire tree and rebuild all memberofrefcount elements from scratch.

Please note three things:

1. This is done **only** if a circular graph is detected between parent and child and **only** if a delete operation takes place
2. Only one part of the membership graph would be inspected. Hopefully only a small amount of groups will be connected like this on the server and the rest of them won't be affected.
3. When having a loop in the membership graph, the user will experience significant performance slowdown compared to all other cases. But again, if there is a loop in the graph, there are much more serious issues going on in the environment.