# Proximity 4.3 Cookbook

## Recipes for Using and Extending Proximity Functionality

# Proximity 4.3 Cookbook: Recipes for Using and Extending Proximity Functionality

Published November 15, 2007


This effort is or has been supported by AFRL, DARPA, NSF, and LLNL/DOE under contract numbers F30602-00-2-0597, F30602-01-2-0566, HR0011-04-1-0013, EIA9983215, and W7405-ENG-48 and by the National Association of Securities Dealers (NASD) through a research grant with the Univeristy of Massachusetts. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied, of AFRL, DARPA, NSF, LLNL/DOE, NASD, the University of Massachusetts Amherst, or the U.S. Government.

General inquiries regarding Proximity should be directed to:

Knowledge Discovery Laboratory
c/o Professor David Jensen, Director
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003-9264

# Table of Contents

# Chapter 1. Introduction

The *Proximity Cookbook* provides example scripts for common or prototypical Proximity tasks. These scripts are designed to serve as models for you to modify to meet your specific needs. Each “recipe” includes extensive discussion of the classes, methods, and data structures used in its accompanying script.

The *Proximity Cookbook* is currently in development and this edition represents a preliminary release of several example scripts. We welcome your suggestions for additional examples to include in future editions. Please contact us at

*proximity@kdl.cs.umass.edu*

with your comments and recommendations.

# Chapter 2. Database Manipulation

# Creating Shortcut Links

## Problem

You need to create direct links between all pairs of objects that are connected to a common third item. A typical example of this task is building the actor collaboration network in the IMDB: You want to create a direct link between two actors if they have both acted in the same movie (i.e., if they both have an ActedIn link to the same Movie object).

Suppose for example that there are two movies in your database, and three different actors:

movie 300 linked to actors 30 and 40
movie 400 linked to actors 30, 40, and 50

You would then like to create a link between the following pairs of actors (with one link in each direction):

30 and 40, and 40 and 30 (because they were together in the two movies)
30 and 50, and 50 and 30 (because they were together in the second movie)
40 and 50, and 50 and 40 (because they were together in the second movie)

## Solution

It is possible to solve this problem with QGraph, with a query containing three vertices: a main *actor* vertex connected to a central *movie* vertex (via an *acted-in* edge), which in turn is connected to an annotated vertex named *collaborator*. This query will create a subgraph for each actor and each movie that she or he has been in, with the list of all the people who have acted in the same film. You can then use the Add-links feature of QGraph to create a new link for each distinct pair of *actor* and *collaborator*, adding the constraint that *actor* < > *collaborator* to avoid self links.

Even though the QGraph approach works, it is very inefficient: QGraph pays a high performance penalty in exchange for its support of very complex queries. Sometimes it is more convenient to simulate the execution of the query manually, to speed up the processing and avoid unnecessary steps.

In this particular case, all we need to do is find the links of the type we are interested in (ActedIn) and simply join them with themselves based on equal values of the movie endpoint (which in the IMDB are stored as the `o2_id` end—that is, the ActedIn links go from actors to movies). We begin by getting all the ActedIn links:

```
actedInLinks = DB.getLinks("link_type = 'ActedIn'")
printNST(actedInLinks)
```

```
[SHOWING head,link_id,o1_id,o2_id,link_type WHERE * LIMIT *]
[ 0@0,    1@0,    30@0,   300@0,   "ActedIn"    ]
[ 1@0,    2@0,    40@0,   300@0,   "ActedIn"    ]
[ 2@0,    3@0,    30@0,   400@0,   "ActedIn"    ]
[ 3@0,    4@0,    40@0,   400@0,   "ActedIn"    ]
[ 4@0,    5@0,    50@0,   400@0,   "ActedIn"    ]
```

A join of the table with itself based on equal `o2_id` produces the table shown below:

```
collaboratedLinks = actedInLinks.join(actedInLinks , "o2_id = o2_id")
printNST(collaboratedLinks)
```

```
[SHOWING head,A.link_id,A.o1_id,A.o2_id,A.link_type,B.link_id,B.o1_id,B.o2_id,\
B.link_type WHERE * LIMIT *]
[ 0@0,   1@0,   30@0,   300@0,   "ActedIn",   1@0,   30@0,   300@0,   "ActedIn"   ]
[ 1@0,   1@0,   30@0,   300@0,   "ActedIn",   2@0,   40@0,   300@0,   "ActedIn"   ]
[ 2@0,   2@0,   40@0,   300@0,   "ActedIn",   1@0,   30@0,   300@0,   "ActedIn"   ]
[ 3@0,   2@0,   40@0,   300@0,   "ActedIn",   2@0,   40@0,   300@0,   "ActedIn"   ]
[ 4@0,   3@0,   30@0,   400@0,   "ActedIn",   3@0,   30@0,   400@0,   "ActedIn"   ]
[ 5@0,   3@0,   30@0,   400@0,   "ActedIn",   4@0,   40@0,   400@0,   "ActedIn"   ]
[ 6@0,   3@0,   30@0,   400@0,   "ActedIn",   5@0,   50@0,   400@0,   "ActedIn"   ]
[ 7@0,   4@0,   40@0,   400@0,   "ActedIn",   3@0,   30@0,   400@0,   "ActedIn"   ]
[ 8@0,   4@0,   40@0,   400@0,   "ActedIn",   4@0,   40@0,   400@0,   "ActedIn"   ]
[ 9@0,   4@0,   40@0,   400@0,   "ActedIn",   5@0,   50@0,   400@0,   "ActedIn"   ]
[ 10@0,  5@0,   50@0,   400@0,   "ActedIn",   3@0,   30@0,   400@0,   "ActedIn"   ]
[ 11@0,  5@0,   50@0,   400@0,   "ActedIn",   4@0,   40@0,   400@0,   "ActedIn"   ]
[ 12@0,  5@0,   50@0,   400@0,   "ActedIn",   5@0,   50@0,   400@0,   "ActedIn"   ]
```

We remove the self links, that is, those rows where the two actors (`o1_id`s) are the same, creating a new NST that contains only the two actor IDs (`A.o1_id` and `B.o1_id`):

```
noLoops=collaboratedLinks.filter("A.o1_id != B.o1_id", "A.o1_id, B.o1_id")
printNST(noLoops)
```

```
[SHOWING head,A.o1_id,B.o1_id WHERE * LIMIT *]
[ 1@0,    30@0,   40@0    ]
[ 2@0,    40@0,   30@0    ]
[ 5@0,    30@0,   40@0    ]
[ 6@0,    30@0,   50@0    ]
[ 7@0,    40@0,   30@0    ]
[ 9@0,    40@0,   50@0    ]
[ 10@0,   50@0,   30@0    ]
[ 11@0,   50@0,   40@0    ]
```

The problem with this table is that we have more than one row per pair of collaborating actors, one for each movie they collaborated in. In particular, the pairs (30,40) and (40,30) appear twice, once for the first movie and another time for the second. We want to create a single link per pair of collaborating actors, so we use the `distinct()` method to get the correct rows from the table above:

```
noRepeats=noLoops.distinct("A.o1_id, B.o1_id")
printNST(noRepeats)
```

```
[SHOWING head,A.o1_id,B.o1_id WHERE * LIMIT *]
[ 0@0,   30@0,   40@0    ]
[ 1@0,   40@0,   30@0    ]
[ 2@0,   30@0,   50@0    ]
[ 3@0,   40@0,   50@0    ]
[ 4@0,   50@0,   30@0    ]
[ 5@0,   50@0,   40@0    ]
```

That is exactly what we want! We're now ready to save the new links permanently, which we can easily do with the `createLinks()` method from the `DB` class. This method requires that the passed in NST has columns named "from" and "to", so we rename them first:

```
DB.createLinks(noRepeats.renameColumns("from, to"))
```

And that's it. You now have six new links, connecting the actors that have collaborated in the same movie.

## Discussion

The combination of NST operations with `createLinks()` gives you a lot of power to add new links to your database. The `createLinks()` method becomes particularly handy with its ability to also create attributes on those new links: for every column named `attr_*` in the NST you pass in, it will create (add) the corresponding values to the attribute for the new links, and create the attribute first if it does not already exist. We can, for example, set the link_type of the new actor-to-actor links to be CollaboratedWith:

```
noRepeats.renameColumns("from, to")
noRepeats.addConstantColumn("attr_link_type", "str", "CollaboratedWith")
DB.createLinks(noRepeats);
```

What's more, the attribute columns don't necessarily have to be constants, but can instead hold any arbitrary value. For example, if you wanted to also save, as attributes on the links, the names of the two actors, you could add to the noRepeats NST two columns, say `attr_actor1Name` and `attr_actor2Name` (by joining the NST with the attribute table that holds the name of actors), before passing it to `DB.createLinks()`.

The general lesson to keep in mind is that `createLinks()` allows you add new links to the database from the contents of any NST. You can create shortcut links as in the example above, self-loop links, links between objects for which there is no path, or in fact any kind of links you want. It all depends on the contents of the arbitrary NST that you pass in.

## Supporting Files

Download the following XML data file to create the database used above to demonstrate this recipe:

- Database: `DBManip_ShortcutLinks_DB.xml`

You may also want to examine the unit test file for this recipe at `$PROX_HOME/test/java/kdl/prox/cookbook/CreateLinksTest.java`.

# Creating Attributes from Arbitrary Data

## Problem

`AddAttribute` is a powerful class that allows you to create new attributes based on complex expressions. It provides if/then capabilities, aggregation of values, comparisons, and regular arithmetic operations. But what if you want to create an attribute based on an expression that `AddAttribute` doesn't support?

## Solution

We'll use an example to show how to create an arbitrary attribute. Suppose, for instance, that you want to create a subgraph attribute that counts the distinct kinds of items in each subgraph of a given container. If your container has the following elements:

| item_id | subg_id | name |
|---|---|---|
| 1 | 0 | Movie |
| 2 | 0 | Movie |
| 3 | 0 | Studio |
| 4 | 0 | Actor |
| 5 | 0 | Actor |
| 6 | 1 | Movie |
| 7 | 1 | Studio |
| 8 | 1 | Director |
| 9 | 1 | Actor |

then you want to create an attribute that says that the first subgraph has three different kinds of items (Movie, Studio, Actor) and that the second subgraph has four (Movie, Studio, Director, Actor), like this:

| subg_id | value |
|---|---|
| 0 | 3 |
| 1 | 4 |

You can create a new attribute by first computing an NST with the desired rows, and then saving that NST using methods in the `Attributes` class. In this case, you can create the attribute NST using the `addDistinctCountColumn()` method. Assume that the container is named studio-clusters and get its objects NST.

```
studioClusters = DB.getContainer("studio-clusters")
containerObjects = studioClusters.getObjectsNST()
```

Use the `addDistinctCountColumn()` method to find, for every `subg_id`, the count of distinct values of name and save it as the `cnt` column.

```
containerObjects.addDistinctCountColumn("subg_id", "name", "cnt")
```

The containerObjects NST will now have a new column with the same value for all the rows with a common `subg_id`.

```
printNST(containerObjects)
```

```
[SHOWING head,item_id,subg_id,name,cnt WHERE * LIMIT *]
[ 0@0,   3@0,   0@0,   "studio",     3    ]
[ 1@0,   7@0,   1@0,   "studio",     4    ]
[ 2@0,   1@0,   0@0,   "movie",      3    ]
[ 3@0,   2@0,   0@0,   "movie",      3    ]
[ 4@0,   6@0,   1@0,   "movie",      4    ]
[ 5@0,   4@0,   0@0,   "actor",      3    ]
[ 6@0,   5@0,   0@0,   "actor",      3    ]
[ 7@0,   9@0,   1@0,   "actor",      4    ]
[ 8@0,   8@0,   1@0,   "director",   4    ]
```

However, attribute tables normally have two columns (called `id` and `value`, such that the first one stores the ID of the item or subgraph that the second one applies to), and a single row for each unique ID (except for multi-valued attributes, but that is not the case in this example). Use the `projectDistinct()` method to get the distinct (`subg_id`, `cnt`) pairs. The `AS` keyword renames the columns in the resulting NST to the expected `id` and `value` without requiring a separate method call. The new attribute will get its values from a projection of distinct (`subg_id`, `cnt`) pairs, stored in a new NST with columns named `id` and `value`.

```
attrNST = containerObjects.projectDistinct("subg_id AS id, cnt AS value")
printNST(attrNST)
```

```
[SHOWING head,id,value WHERE * LIMIT *]
[ 0@0,   0@0,    3     ]
[ 1@0,   1@0,    4     ]
```

At this point, you are ready to create the new attribute. The `Attributes` class provides several public methods to manipulate its internal tables. In particular, the `defineAttributeWithData()` method allows you to define a new attribute based on an arbitrary NST that you pass to it, which is what you need in this case. Save the new NST as a subgraph attribute inside the container, with the name `distinct_types` and of type `int`.

```
studioClusters.getSubgraphAttrs().defineAttributeWithData("distinct_types","int",attrNST)
```

And you're done. Just remember to remove the `cnt` column that you added to the objectsNST:

```
containerObjects.removeColumn("cnt")
```

## Discussion

The combination of NST operations with the `defineAttributeWithData()` method give you a lot of power to create all kinds of attributes. The example above focused on the creation of a standard single-column, single-value attribute, but in fact you can create any kind of attribute. Just remember to match the structure and content of your NST to the kind of attribute you want to create, and to make the first column of the NST be of type `oid` and be named `id`.

## Supporting Files

Download the following XML data file and query to create the database and container used above to demonstrate this recipe:

- Database: `DBManip_AttrsFromData_DB.xml`
- Query: `DBManip_AttrsFromData_Query.qg2.xml`

You may also want to examine the unit test file for this recipe at `$PROX_HOME/test/java/kdl/prox/cookbook/AddAttributeTest.java`.

# Aggregation in NSTs

## Problem

You have a set-valued attribute, where a single object is associated with more than one value, and you want to summarize that information with a single value per object ID. For example, an attribute that stores the historical salaries of company employees could be summarized by finding the maximum amount ever received by each individual. In a SQL database, you can get such a summary using a combination of `GROUP BY id` and an aggregation method (`max` in this case) on the `salary` column. How can you do this in Proximity?

## Solution

The NST class provides a powerful but (until now) less well known `aggregate()` method. Given a table with a *grouping column* (*`G`*, typically the `id` column) and a changing *value column* (*`V`*), this method lets you apply a specified aggregation function on *`V`* to groups of rows with the same value in *`G`*. For example, suppose that your multi-set salary attribute has the following content:

```
objAttrs = prox.objectAttrs
salaryNST = objAttrs.getAttrDataNST("salary")
printNST(salaryNST)
```

```
[SHOWING head, id, value WHERE * LIMIT *]
[ 1@0,    1@0,    10    ]
[ 2@0,    1@0,    20    ]
[ 3@0,    2@0,     5    ]
[ 4@0,    2@0,     7    ]
[ 5@0,    3@0,    10    ]
[ 6@0,    3@0,    10    ]
```

You can apply the `max` aggregator to find sets of equal `id` (the *`G`* column) and get their highest `value` (*`V`* column):

```
maxSalaryNST = salaryNST.aggregate("max", "id", "value")
printNST(maxSalaryNST)
```

```
[SHOWING head, id, value WHERE * LIMIT *]
[ 0@0,    1@0,    20    ]
[ 1@0,    2@0,     7    ]
[ 2@0,    3@0,    10    ]
```

The `aggregate()` method is fast because it takes advantage of MonetDB's `{operator}` syntax, a built-in construct that very efficiently groups a BAT's rows by their `HEAD` values and then applies the specified operator to each set. Furthermore, most of MonetDB's aggregation operators are implemented as cache-conscious and type-specific C code, which allows MonetDB to use highly optimized code for the most common operations.

Several aggregation operators are available:

- min, max, mode
- sum, prod
- avg, variance
- count
- card
- size

Most of these operators are self-explanatory. For example, `avg` computes the average of values within common values of the *`G`* column:

```
avgSalaryNST = salaryNST.aggregate("avg", "id", "value")
printNST(avgSalaryNST)
```

```
[SHOWING head, id, value WHERE * LIMIT *]
[ 0@0,    1@0,    15    ]
[ 1@0,    2@0,     6    ]
[ 2@0,    3@0,    10    ]
```

The definitions of the last two aggregation operators, `card` and `size`, are less obvious. The `card` operator counts the number of `DISTINCT` values associated with a given group, and the `size` operator returns a count of the number of `true` values within a given group (on boolean columns). For example,

```
distinctSalaryNST = salaryNST.aggregate("card", "id", "value")
printNST(distinctSalaryNST)
```

```
[SHOWING head, id, value WHERE * LIMIT *]
[ 0@0,    1@0,    2    ]
[ 1@0,    2@0,    2    ]
[ 2@0,    3@0,    1    ]
```

and

```
tempNST = salaryNST.copy()
tempNST.addConditionColumn("value >= 10", "gte10")
gteNST = tempNST.aggregate("size", "id", "gte10")
printNST(gteNST)
```

```
[SHOWING head, id, gte10 WHERE * LIMIT *]
[ 0@0,    1@0,    2    ]
[ 1@0,    2@0,    0    ]
[ 2@0,    3@0,    2    ]
```

It is also possible to write new aggregation operators directly in MIL, and some packages in Proximity take advantage of that ability, but that's the subject of another recipe.

## Discussion

There are many cases when you need to get summaries of tables with more than one row per group. The `aggregate()` method makes this easy and efficient. For example, to compute the out-degree of all the objects in the system, all you have to do is:

```
DB.getLinkNST().aggregate("count", "o1_id", "o2_id")
```

In general, `aggregate()` provides an efficient way to compute aggregates over groups.

Keep in mind that `aggregate()` always returns an NST of two columns. The first column has the same name and type as the grouping column *`G`*, while the second column has the name of the value column *`V`*, but its type corresponds to the operator used. For example, the call to compute the average of salaries returns an NST where the second column is of type `dbl`, even though the original value column was of type `int`.

And remember that the new NST uses the old column name for the column containing the aggregated value. In the out-degree example, above, the `o2_id` column in the new NST holds the out degree for each of the `o1_id` IDs. You can use the `renameColumn()` method to change the name of this column, if you want to provide a more meaningful column name.

## Supporting Files

Download the following files to demonstrate this recipe:

- Database: `DBManip_Aggregation_DB.xml`

You may also want to examine the unit test file for this recipe at `$PROX_HOME/test/java/kdl/prox/cookbook/AggregateTest.java`.

Also, the `aggregators` package in `model2` makes extensive use of the techniques described above.

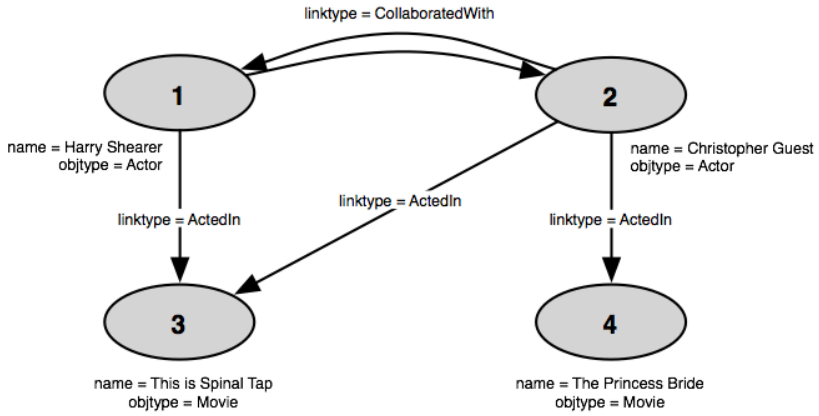
# Creating a Degree Attribute

## Problem

When working with relational data, you often want to know the degree of a node in the graph (an object in Proximity). This recipe demonstrates how to calculate the degree of an object and store that value as an attribute on the object.

## Solution

Suppose you have a database like the one shown in the following fragment:




The example database contains actors and movies. Actors are linked to the movies they've appeared in by ActedIn links, and actors that have appeared in the same movie are connected by a pair of CollaboratedWith links, one in each direction. In the fragment shown above, both Christopher Guest and Harry Shearer appeared in *This is Spinal Tap*, so there is a pair of CollaboratedWith links connecting the actors.

The recipe in "Aggregation in NSTs" showed us how to use the `aggregate()` method to summarize multiple pieces of information for a single object. In the current example, we can think of each linked object as a piece of information for the target object, which we summarize by counting all such linked objects, once again using the versatile `aggregate()` method.

The pieces of information we need—the different objects linked to our target node—are contained in the links NST.

```
printNST(prox.linkNST)
```

```
[SHOWING head,link_id,o1_id,o2_id WHERE * LIMIT *]
[ 1@0,    1@0,    1@0,    3@0     ]
[ 2@0,    2@0,    2@0,    3@0     ]
[ 3@0,    3@0,    2@0,    4@0     ]
[ 4@0,    4@0,    1@0,    2@0     ]
[ 5@0,    5@0,    2@0,    1@0     ]
```

The links NST contains one row per link. To get the degree of an object, we need to count the number of rows that show that object as the start node for a link (rows having the target object ID in the `o1_id` column) and add that to the number of rows that show that object as the terminating node for a link (rows having the target object ID in the `o2_id` column).

We can accomplish this more easily by employing a neat trick involving adding inverse "links" to our link data. (Because these inverses are not added to the links NST, they are not actually added to the database.) First, we create an NST that contains the inverses of all the links in the database.

```
inverseLinks = prox.linkNST.project("o2_id, o1_id")
printNST(inverseLinks)
```

```
[SHOWING head,o2_id,o1_id WHERE * LIMIT *]
[ 1@0,    3@0,    1@0     ]
[ 2@0,    3@0,    2@0     ]
[ 3@0,    4@0,    2@0     ]
[ 4@0,    2@0,    1@0     ]
[ 5@0,    1@0,    2@0     ]
```

Then we create an NST that combines the actual link data with the data from the inverse-link NST we just created. (The steps to create the biDirLinks NST have been broken up into two lines for clarity and formatting purposes; you can string the method calls together on one line if you prefer.)

```
biDirLinks = prox.linkNST.project("o1_id, o2_id")
biDirLinks.insertRowsFromNST(inverseLinks)
printNST(biDirLinks)
```

```
[SHOWING head,o1_id,o2_id WHERE * LIMIT *]
[ 1@0,    1@0,    3@0     ]
[ 2@0,    2@0,    3@0     ]
[ 3@0,    2@0,    4@0     ]
[ 4@0,    1@0,    2@0     ]
[ 5@0,    2@0,    1@0     ]
[ 6@0,    3@0,    1@0     ]
[ 7@0,    3@0,    2@0     ]
[ 8@0,    4@0,    2@0     ]
[ 9@0,    2@0,    1@0     ]
[ 10@0,   1@0,    2@0     ]
```

Now we can aggregate over the IDs in the `o1_id` column to get a total count of links going out of or into each object. That is, we group on `o1_id` and count the number of (non-distinct) `o2_id` values for each value of `o1_id`.

```
degreeNST = biDirLinks.aggregate("count", "o1_id", "o2_id")
printNST(degreeNST)
```

```
[SHOWING head,o1_id,o2_id WHERE * LIMIT *]
[ 0@0,    1@0,    3     ]
[ 1@0,    2@0,    4     ]
[ 2@0,    3@0,    2     ]
[ 3@0,    4@0,    1     ]
```

The `aggregate()` method reuses the column names specified in the method call. Thus, the column containing the group identifier is labeled `o1_id` and the column containing the counts is labeled `o2_id`. Note that the `count` operator gives us a count of *all* linked objects, not of distinct linked objects.

The resulting NST contains the data we want to use for the new degree attribute. As we saw in the recipe “Creating Attributes from Arbitrary Data”, we just need to rename the columns of the NST so that they have the expected names `id` and `value`, then use that NST to define the values for the new degree attribute.

```
objectAttrs = prox.objectAttrs
degreeNST.renameColumns("id, value")
objectAttrs.defineAttributeWithData("degree", "int", degreeNST)
```

## Discussion

There are many variations on simple degree that we might find useful in developing models. Calculating these values involve simple modifications to the above procedure.

### Degree for selected links

Sometimes we might want to limit our degree calculations to include only links of a specified type. Suppose we only wanted to know the degree of ActedIn links. To get this value we limit the links included in the bi-directional NST to those of the specified type. We start by creating an NST containing all the links of the desired type.

```
actedInLinks = DB.getLinks("linktype = 'ActedIn'")
printNST(actedInLinks)
```

```
[SHOWING head,link_id,o1_id,o2_id,linktype WHERE * LIMIT *]
[ 0@0,    1@0,    1@0,    3@0,    "ActedIn"    ]
[ 1@0,    2@0,    2@0,    3@0,    "ActedIn"    ]
[ 2@0,    3@0,    2@0,    4@0,    "ActedIn"    ]
```

Now create the inverse links and combine them with the links in the actedInLinks NST to obtain an NST of all ActedIn links and their inverses. Use this new bi-directional NST to calculate and define the degree attribute as before.

### Degree for selected objects

Similarly, we might only care about the degree of objects of a specific type. To limit our degree calculations to the specified objects, we start by creating an NST that holds the IDs of the objects that we're interested in, in this case, actors.

```
actorObjects = DB.getObjects("objtype = 'Actor'")
printNST(actorObjects)
```

```
[SHOWING head,id,objtype WHERE * LIMIT *]
[ 0@0,    1@0,    "Actor"    ]
[ 1@0,    2@0,    "Actor"    ]
```

Now we can join that NST with the (full) bi-directional link NST we created earlier. The join with the bi-directional links NST returns rows where the originating object is an actor.

```
actorBiDirLinks = biDirLinks.join(actorObjects,"o1_id = id", "o1_id, o2_id")
```

Note that this limits the objects for which we calculate degree to Actor objects but does not similarly restrict the type of the linked objects. If we wanted to limit our degree calculations to only actor-to-actor links, we can do so by joining the actorObjects NST with the links NST to create an NST where `o2_id` only contains actors, then using that in place of the links NST when creating our bi-directional NST.

### In- or out-degree only

We needed the trick of creating inverse "links" above because we wanted to include both link directions in our degree calculations. But suppose we only care about the number of links terminating at the target object (in degree) or the number of links originating from the target object (out degree)? This is even simpler—we can skip creating the bi-directional NST and use the links NST directly.

To calculate the out degree, we group on the originating object (`o1_id`) and count the number rows for each ID in `o1_id`.

```
outDegree = prox.linkNST.aggregate("count", "o1_id", "o2_id")
printNST(outDegree)
```

```
[SHOWING head,o1_id,o2_id WHERE * LIMIT *]
[ 0@0,    1@0,    2    ]
[ 1@0,    2@0,    3    ]
```

And to get the in degree, we group on the terminating object (`o2_id`).

```
inDegree = prox.linkNST.aggregate("count", "o2_id", "o1_id")
printNST(inDegree)
```

```
[SHOWING head,o2_id,o1_id WHERE * LIMIT *]
[ 0@0,    3@0,    2    ]
[ 1@0,    4@0,    1    ]
[ 2@0,    2@0,    1    ]
[ 3@0,    1@0,    1    ]
```

We can now proceed as before by renaming the columns and using `defineAttributeWithData()` to create and populate the new attribute.

### Degree for unique links

All the above examples count multiple links connecting objects separately, but what if we want to instead count the number of unique objects that an object is linked to?

The `aggregate()` method solves this nicely with the `card` aggregation operator. The `card` operator counts the number of distinct values for a given group, giving us exactly the values we want.

```
uniqueDegree = biDirLinks.aggregate("card", "o1_id", "o2_id")
```

## Supporting Files

Download the following XML data file to create the database used above to demonstrate this recipe:

- Database: `DBManip_AddDegreeAttr_DB.xml`

You may also want to examine the unit test file for this recipe at `$PROX_HOME/test/java/kdl/prox/cookbook/DegreeAttributeTest.java`.

# Chapter 3. Data Import

# Importing Data with Mismatched IDs

## Problem

You have a table in a MySQL database that you would like to import as a Proximity attribute. However, you realize that the IDs from that table no longer correspond to the object IDs in Proximity: The original IDs were replaced by new Proximity OIDs during the import. But, luckily, you saved the old ones in an attribute.

## Solution

Suppose you have an attribute in a MySQL database called class_label that you want to import, with the following contents:

| member_id | class_label |
|---|---|
| 15 | + |
| 25 | + |
| 35 | - |

The `member_id` column is, in fact, a foreign key that references the `id` of another table—let's call it the persons table. Let's assume that when you originally imported the data, the rows from the persons table were assigned object IDs starting from 200. Let's also assume that you saved the original `persons.id` column as an attribute in Proximity called persons_id. The contents of that attribute might look like this:[1]

```
printNST(DB.getObjectAttrs().getAttrDataNST("persons_id"))
```

```
[SHOWING head,id,value WHERE * LIMIT *]
[ 1@0,   200@0,   15    ]
[ 2@0,   201@0,   16    ]
[ 3@0,   202@0,   25    ]
[ 4@0,   205@0,   35    ]
```

Your task is to combine the `id` from the persons_id attribute with the class_label from the new, imported data to create a new attribute with the rows shown below:

| id | value |
|---|---|
| 200 | + |
| 202 | + |
| 205 | - |

Start by saving the MySQL table into a text file that Proximity can read. There are many ways to do this, for example:

```
$ echo 'select * from class_label;' | mysql -u usrname -p dbname > class_label.txt
```

Now you can go to Proximity and read that file directly into a new NST.

```
importedNST = NST("old_member_id, value", "int, str").fromfile("datafile")
```

where *`datafile`* includes the full path to the file. See the "Supporting Files" section, below, for an example data file you can use to test this recipe.

[1] You can use the "shortcut" `prox.objectAttrs` in place of `DB.getObjectAttrs()` in Proximity scripts and in the interactive Proximity Python interpreter.

The resulting NST is shown below.

```
printNST(importedNST)
```

```
[SHOWING head,old_member_id,value WHERE * LIMIT *]
[ 1@0,   15,   "+"    ]
[ 2@0,   25,   "+"    ]
[ 3@0,   35,   "-"    ]
```

If you were to save this NST as an attribute, the `id` column would be referencing object IDs that don't exist. You must first join it with the persons_id attribute, to get the correct Proximity IDs. Recall that the persons_id attribute has the following values:

```
printNST(DB.getObjectAttrs().getAttrDataNST("persons_id"))
```

```
[SHOWING head,id,value WHERE * LIMIT *]
[ 1@0,   200@0,   15    ]
[ 2@0,   201@0,   16    ]
[ 3@0,   202@0,   25    ]
[ 4@0,   205@0,   35    ]
```

You can do a simple join between the two tables, keeping the `id` of the persons_id attribute and the `value` column from the importedNST table. Because both tables have a `value` column, you must write `B.value` to indicate that you want to keep the one from the second (personIDs) table. Only the first (importedNST) table contains an `id` column, so no prefix is used.

```
personIDs = DB.getObjectAttrs().getAttrDataNST("persons_id")
classNST = importedNST.join(personIDs, "A.old_member_id = B.value", "id, A.value")
printNST(classNST)
```

```
[SHOWING head,id,A.value WHERE * LIMIT *]
[ 0@0,   200@0,   "+"    ]
[ 1@0,   202@0,   "+"    ]
[ 2@0,   205@0,   "-"    ]
```

This is exactly what you wanted. At this point, you can save the new NST as an attribute, using the techniques described in the recipe "Creating Attributes from Arbitrary Data".

```
DB.getObjectAttrs().defineAttributeWithData("class_label", "str", classNST)
```

## Discussion

This is a very simple recipe, but quite powerful. The key concept is that you can do incremental imports from an SQL database as long as you keep around Proximity attributes like persons_id, which map MySQL IDs to Proximity IDs. With those attributes in place, you can read in new table dumps and always be able to convert from old to new IDs.

This technique is not restricted to new attributes. For example, you can use the same technique to import new links.

1. Read in the dump for the new links.
2. Join the `o1_id` and `o2_id` endpoints with the attributes that allow you to convert from MySQL to Proximity OIDs.
3. Insert the new table into the Proximity links NST.

## Supporting Files

Download the following XML database file and new data file to create the database and attribute used above to demonstrate this recipe:

- Database: `Import_MismatchedIDs_DB.xml`
- New data file: `Import_MismatchedIDs_NewData.txt`

You may also want to examine the unit test file for this recipe at `$PROX_HOME/test/java/kdl/prox/cookbook/ImportAttributeTest.java`.

# Converting Databases to 64-Bit MonetDB

## Problem

MonetDB is giving you “Out of memory” errors, and you realize that your database no longer fits within the 2GB memory limit. What can do you?

## Solution

The obvious solution is to switch to the version of MonetDB that supports 64-bit pointers, which allows you to use much more memory than you will ever need (famous last words). The problem is that the database files are not compatible across the two versions—because one stores its data using longer words—so you will have to use a conversion utility before you can switch servers.

One alternative is to dump your 32-bit database into an XML file band then re-import it into a clean 64-bit version. Since XML is too verbose and too slow for large databases, we have created another pair of import and export utilities that use plain text files instead. These files are saved in a format that MonetDB can read directly, without much intervention from Proximity. These files don’t have the extra structure from XML files, take up less space, and don’t need to be parsed. Using this file format, the conversion from 32 to 64 bits can be done simply, and in a matter of minutes.

You can use these import and export utilities directly from the command line. First, create a temporary directory to hold the converted files:

```
$ mkdir temp-dir
$ cd temp-dir
```

Then start the 32-bit version of MonetDB on your original data, and export its contents into the current directory:

```
$ $PROX_HOME/bin/export-text.sh host:post `pwd`
```

where *host:port* will be `localhost:30000` if you are running this script from the same machine as the MonetDB server (which is required, see “Discussion”, below) and using the default `init-mserver.mil` MonetDB initialization script.

Quit the MonetDB server and start a 64-bit server on a new database. Initialize the database with Proximity’s schema, and import the data saved in the current directory:

```
$ $PROX_HOME/bin/db-util.sh host:port init-db
$ $PROX_HOME/bin/import-text.sh host:port `pwd`
```

Your 64-bit database is now ready to be used. The entire operation should be quick—dumping and reloading one database containing 5GB worth of data took less than 30 minutes on one of our servers.

## Discussion

These utilities are convenient because they create an intermediate representation of your data that is succinct and can be easily reloaded. You can use them any time that you need to transport or convert data. Example uses include creating a tarball of your data (where the text dump of your database will be noticeably smaller than its XML counterpart) and porting a database from a little-endian platform to a big-endian architecture (for example, to switch between Mac OS X and Linux/Windows).

Note that these utilities must be run on the same machine as your MonetDB servers—otherwise, MonetDB won’t be able to write or read into the *temp-dir* where the conversion files live.

## Supporting Files

There are no files for this recipe, but you can take a look at the unit tests for these two utilities: `$PROX_HOME/test/java/kdl/prox/app/ExportTextTest.java` and `$PROX_HOME/test/java/kdl/prox/app/ImportTextTest.java`.

# Chapter 4. Queries

# Efficient Parameter Queries

## Problem

You want to run *parameter queries*, where a set of individual QGraph queries is automatically derived by combining a *template query* with different conditions from a list. However, you find that the template query is too expensive, and the time it would take to run all the parametrized instances is prohibitive.

## Solution

Parameter queries are a neat solution to the problem of having to run several queries that only differ in small details. For example, suppose you want to run queries to find movies and their actors, but you want separate containers for movies from the 1970s, the 1980s, the 1990s, and so on. Writing several queries that are identical except for the value of the decade requires a lot of repetitive work. Following the DRY principle (Don't Repeat Yourself), you write a template query with a *Movie* vertex linked to an annotated *Actor* vertex, and compile a list of conditions in the form:

```
Movie.decade=70
Movie.decade=80
Movie.decade=90
```

You then iterate over the list of conditions and use the `ParameterQuery` class to add each one to the template query before executing it, saving the result of each run in its own container. But although the parameter query provides an elegant solution, it may not be practical in all cases.

If the template query has many vertices and edges to which the parametrized conditions do not apply (for example, if you also want to include a subquery that finds, for each actor, all the movies that actor appeared in), then you are asking QGraph to execute the same computations over and over again, unnecessarily. In other words, you are not repeating yourself, but QGraph is. This is not a problem for small queries, but it can become a serious problem for larger ones.

The solution comes by noticing that, at least in cases like our example, the subgraphs that match each parametrized query are a subset of the subgraphs that match the full template query. In other words, the contents of each subgraph is exactly the same as for the full template query, but the set of matching subgraphs is restricted for each parametrized condition. It is therefore possible, and much more efficient, to run the expensive template query just once, and then process each condition by copying the results into a new container and removing the subgraphs that don't match the extra requirement. The Jython commands below show you how to do it.

First, we run the template query and save the results in the container named "all-decades":

```
from java.io import File
from kdl.prox.qgraph2 import QueryXMLUtil
from kdl.prox.qgraph2 import QueryGraph2CompOp

f=File("parameter-query-template.qg2.xml")
q=QueryXMLUtil.graphQueryEleFromFile(f)
QueryGraph2CompOp.queryGraph(q,  None, "all-decades", 1)
```

The results will be stored in the objects and links tables of the all-decades container.

The results of executing the query on the sample database listed in the "Supporting Files" section of this recipe are shown below. (We sort the rows by `subg_id` to more easily see the objects in each of the subgraphs.)

```
printNST(DB.getContainer("all-decades").getObjectsNST().sort("subg_id", "*"))
```

```
[SHOWING head, item_id, subg_id, name WHERE * LIMIT *]
[ 0@0,   1@0,    0@0,   "Movie"   ]
[ 1@0,   10@0,   0@0,   "Actor"   ]
[ 2@0,   11@0,   0@0,   "Actor"   ]
[ 3@0,   12@0,   0@0,   "Actor"   ]
```

```
[ 4@0,   14@0,   1@0,   "Actor"   ]
[ 5@0,   2@0,    1@0,   "Movie"   ]
[ 6@0,   13@0,   1@0,   "Actor"   ]

[ 7@0,   3@0,    2@0,   "Movie"   ]
[ 8@0,   16@0,   2@0,   "Actor"   ]
[ 9@0,   15@0,   2@0,   "Actor"   ]
```

We see that the container has three subgraphs, with IDs 0, 1, and 2.

We can also take a look at the values of the decade attribute for those Movie objects:

```
printNST(objectAttrs.getAttrDataNST("decade"))
```

```
[SHOWING head, id, value WHERE * LIMIT *]
[ 1@0,   1@0,   70   ]
[ 2@0,   2@0,   80   ]
[ 3@0,   3@0,   80   ]
```

From the combination of the two tables we can see that subgraph 0 has a Movie object (object 1) that was made in the 1970s; subgraph 1 has a Movie object (object 2) that was made in the 1980s, and subgraph 2 has a Movie object that was also made in the 1980s.

Now that we have the container with the results for the full query, let's create a derivative container for a given decade. First we filter the container's objects to return only those rows that correspond to the *Movie* vertex *and* that were made in the 1970s. We use the powerful `getObjects()` method on the container, which allows us to specify complex conditions on the vertices and their attributes:

```
just70s=DB.getContainer("all-decades").getObjects("name = 'Movie' AND decade = 70")
printNST(just70s)
```

```
[SHOWING head, item_id, subg_id, name, decade WHERE * LIMIT *]
[ 0@0,   1@0,   0@0,   "Movie",   70   ]
```

For the 1970s container, the only subgraph that we want to keep is the first one (subgraph 0), and this is exactly what we now have in the just70s NST.

Now we're ready to filter the original objects and links tables from the all-decades container, using the `intersect()` operation:

```
c = DB.getContainer("all-decades")
newObjects = c.getObjectsNST().intersect(just70s,"subg_id = subg_id")
newLinks = c.getLinksNST().intersect(just70s,"subg_id = subg_id")
```

The new tables have copies of the data from the original table, but only for subgraph 0:

```
printNST(newObjects)
```

```
[SHOWING head, item_id, subg_id, name WHERE * LIMIT *]
[ 1@0,   1@0,    0@0,   "Movie"   ]
[ 4@0,   10@0,   0@0,   "Actor"   ]
[ 5@0,   11@0,   0@0,   "Actor"   ]
[ 6@0,   12@0,   0@0,   "Actor"   ]
```

```
printNST(newLinks)
```

```
[SHOWING head, item_id, subg_id, name WHERE * LIMIT *]
[ 0@0,   1@0,   0@0,   "ActedIn"   ]
[ 1@0,   2@0,   0@0,   "ActedIn"   ]
[ 2@0,   3@0,   0@0,   "ActedIn"   ]
```

We now have two new NSTs storing the objects and links of subgraphs where the movie was made in the 1970s.

The final step is to save these objects and in a new container. There is a method (with an admittedly unfriendly name) that allows us to do just that:

```
DB.getRootContainer().createChildFromTempSGINSTs("70s", newObjects,newLinks)
```

We now have a container, named “70s,” of subgraphs for 1970s movies.

By repeating the same procedure, we can quickly create the containers for other decades without having to run the (presumably expensive) QGraph query multiple times. You remain DRY, and so does QGraph.

## Discussion

The technique above is quite powerful and efficient, and its application has allowed us to run parametrized queries on very large databases that would have taken too long to execute with the traditional method. Notice, however, that the technique as described above is only suitable to those cases where the application of parametrized conditions has the effect of removing entire subgraphs from the results for the full template query. This is true when the conditions apply to unannotated objects outside of a subquery. In such cases, if a particular item no longer matches a vertex or edge because of the added condition, then the entire subgraph is removed, as an unannotated object is always required to be present in the resulting subgraph.

The case of annotated vertices and vertices inside of a subquery is more delicate, and it requires two steps: In the first step, the items that don’t match the added condition are removed from the objects table, and in the second step the annotation is rechecked, verifying that every single remaining subgraph still contains the number of vertices required by the annotation (`[0..]` annotations are exempt from this extra check).

In addition to the specific example of running efficient parametrized queries, this recipe also shows that it is possible to create new containers directly from generic objects and links NSTs, just as it is possible to create new attributes from arbitrary NSTs (see “Creating Attributes from Arbitrary Data”.)

## Supporting Files

Download the following files to create the database and query used to demonstrate this recipe:

- Sample database: `parameter-query-db.xml`
- Template query: `parameter-query-template.qg2.xml`

You may also want to examine the unit test file for this recipe at `$PROX_HOME/test/java/kdl/prox/cookbook/OptimizedParameterQueryTest.java`. The database and query files are also available in the same unit test directory.

# Chapter 5. Proximity Development

# Configuring Log4j

## Problem

You are testing code that you've written, and you want to output detailed debugging information while the program runs. However, turning debugging on shows you information about *all* packages, making it difficult to trace the execution of your code.

## Solution

Proximity uses log4j, a powerful tool that allows programmatic control of the debugging information that your program displays. Using log4j, you can send your output to different *loggers* using different *levels* of verbosity, you can dynamically change those levels, and you can associate different *appenders* to the loggers to send the output to different destinations such as the console, a file, a TCP/IP socket, or a database. Regretfully, log4j can also be hard to configure. In this recipe, we explain how Proximity uses log4j and guide you through the process of configuring log4j with some example setups.

Proximity currently uses three different loggers:

- `rootLogger`, the main logger, which receives the messages from all the classes in the system
- `mil`, a logger for MIL statements
- `milMessages`, another logger for MIL statements

The two MIL loggers are used exclusively by the low-level database classes. By default, all messages from the main logger are routed to the console at a low level of verbosity (`INFO`). However, if you want to see more or less debugging output, you can change the verbosity level, or attach new appenders with a different verbosity and send their output to a different location (for example, a file where you save all the `DEBUG` statements). Alternately, it is also possible to use different appenders for individual packages or classes, thereby having a finer control of the output.

Changing verbosity levels, changing or adding output destinations, and much more, is done through the `prox.lcf` configuration file. Proximity looks for this file in the directory from where it's launched, and uses it to configure log4j before execution begins. The Proximity distribution includes examples of different configuration files under the `example/config` directory.

The most basic configuration file, `info.lcf`, is equivalent to the default setup:

```
log4j.rootLogger=INFO, A1

log4j.appender.A1=org.apache.log4j.ConsoleAppender
   log4j.appender.A1.layout=kdl.prox.util.TimeStampLayout
   log4j.appender.A1.layout.ConversionPattern=%-9r %-5p %c{2}: %m%n
```

The first line sets the verbosity level of the main logger (`rootLogger`) to `INFO`, and attaches an appender named `A1` to it. The other lines define the `A1` appender, send its output to the console, and specify a particular format for the messages.

You can save a copy of all the logging output to a file, as the configuration file `info-plus-file.lcf` shows:

```
log4j.rootLogger=INFO, A1, A2

log4j.appender.A1=org.apache.log4j.ConsoleAppender
   log4j.appender.A1.layout=kdl.prox.util.TimeStampLayout
   log4j.appender.A1.layout.ConversionPattern=%-9r %-5p %c{2}: %m%n

log4j.appender.A2=org.apache.log4j.FileAppender
log4j.appender.A2.File=prox.log
log4j.appender.A2.Append=false
   log4j.appender.A2.layout=kdl.prox.util.TimeStampLayout
   log4j.appender.A2.layout.ConversionPattern=%-9r %-5p %c{2}: %m%n
```

The first line attaches an additional appender (`A2`) to the main logger, which is configured to send the output to a file named `prox.log`, using the same format as the `A1` appender.

What if you want to get more detailed debugging information for a particular package or individual file? Simply increase the verbosity level for that class, as is shown in the `module-debug.lcf` configuration file:

```
log4j.rootLogger=INFO, A1

log4j.appender.A1=org.apache.log4j.ConsoleAppender
   log4j.appender.A1.layout=kdl.prox.util.TimeStampLayout
   log4j.appender.A1.layout.ConversionPattern=%-9r %-5p %c{2}: %m%n

log4j.logger.kdl.prox.dbmgr.NST=DEBUG
log4j.logger.kdl.prox.sample=DEBUG
```

In this particular case, the output will be the same as before except for the `NST` class and all the classes in the `sample` package, which will report their information at the highest level of verbosity (`DEBUG`). Notice that the output won't be sent to a `prox.log` file, as in the previous example. To do that, add a second appender and configure it as `A2` in the configuration file above.

If you want to get as much debugging information as possible without getting distracted by the exhaustive logging of the database layers, you can set the default level to `DEBUG` and selectively decrease the level for the database loggers, following the technique demonstrated above. An example of this configuration is shown in `debug-except-db.lcf`:

```
log4j.rootLogger=DEBUG, A1

log4j.appender.A1=org.apache.log4j.ConsoleAppender
   log4j.appender.A1.layout=kdl.prox.util.TimeStampLayout
   log4j.appender.A1.layout.ConversionPattern=%-9r %-5p %c{2}: %m%n

# reduce monet-related debug output, and output from the DB classes
log4j.logger.mil=WARN
log4j.logger.milMessages=WARN
log4j.logger.kdl.prox.db=WARN
log4j.logger.kdl.prox.dbmgr=WARN
log4j.logger.kdl.prox.monet.Connection=WARN
log4j.logger.kdl.prox.util.MonetUtil=WARN
```

Note that the level for the root logger is set to `DEBUG`, the most verbose level, while all the database-related classes are set to `WARN`, the least verbose of all levels.

As a final example, consider the extremely rare situation in which the lower database layer is failing. In such a case, you want to use the highest level of debugging possible and also save a log of the Monet statements (MIL commands). The file `full-debug-plus-mil-file.lcf` illustrates the corresponding configuration:

```
log4j.rootLogger=DEBUG, A1

log4j.appender.A1=org.apache.log4j.ConsoleAppender
   log4j.appender.A1.layout=kdl.prox.util.TimeStampLayout
   log4j.appender.A1.layout.ConversionPattern=%-9r %-5p %c{2}: %m%n

# Send MIL to the console
log4j.additivity.mil=true

# And to a file appender
log4j.logger.mil=DEBUG, A3
log4j.logger.milMessages=DEBUG, A3

# Create the prox.log.mil file with RAW instructions
log4j.appender.A3=org.apache.log4j.FileAppender
log4j.appender.A3.File=prox.log.mil
log4j.appender.A3.Append=false
log4j.appender.A3.layout=org.apache.log4j.PatternLayout
log4j.appender.A3.layout.ConversionPattern=%m%n
```

Note the use of the database-related loggers, `mil` and `milMessages`. They are configured such that their output is added to that of the main logger, and such that the MIL commands get saved to a `prox.log.mil` file in a raw format.

## Discussion

The examples above cover all the configuring options that we have ever had to use while writing and debugging Proximity: They use different loggers, create one or more appenders, and change the level of verbosity for individual classes and packages. You can combine these options to suit your needs, including writing the output of different packages to separate files, or setting different verbosity levels for groups of classes in the same package.

> Whatever your needs, remember that Proximity looks for configuration information in the file `prox.lcf`, which must be present in the directory from which Proximity is launched.

## Supporting Files

All the examples above are included in the Proximity distribution, under the `/example/config/` directory.

For more information and configuration options, consult the log4j documentation.

# Index

All index enties point to the beginning of the relevant recipe rather than the page containing the indexed term.

## Q



## R



## S



## T



## V



## W