Proximity 4.3 Tutorial

Proximity 4.3 Tutorial

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.

The example database used to support the exercises in this tutorial, ProxWebKB, was developed from the publicly available WebKB relational data set developed by the Text Learning Group at Carnegie-Mellon University. The version used for the Proximity tutorial has been modified from the original distribution to meet the needs of this tutorial. The original dataset is available from www-2.cs.cmu.edu/~WebKB/.

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

List of Exercises

Chapter 1. Introduction

Proximity is an environment for knowledge discovery in relational data. It helps human analysts discover new knowledge by analyzing complex data sets about people, places, things, and events. New developments in this area are vital because of the growing interest in analyzing the Web, social networks, telecommunications and computer networks, relational and object-oriented databases, multi-agent systems, and other sources of structured and semi-structured data.

Proximity consists of novel algorithms that help manage, explore, sample, model, and visualize data. Proximity implements methods for learning statistical models that describe the probabilistic dependencies in relational data and can estimate probability distributions over unseen data. Proximity is an open-source application developed in Java, and it makes substantial use of MonetDB [Boncz and Kersten, 1995], [Boncz, 2002], an open-source, vertical database system designed for high performance on semi-structured data.

Conventional Knowledge Discovery

First-generation tools for knowledge discovery are already widely deployed in business, science, and government. These tools help epidemiologists identify emerging diseases, help engineers improve industrial processes, and help credit-card companies spot fraud.

Unfortunately, much of the technical work in knowledge discovery, and its underlying statistical theory, assumes that data records are structurally homogeneous and statistically independent. For example, to analyze a set of patient records to determine useful diagnostic rules for a new disease, traditional techniques would assume that the records provide the same type of information about each patient and that knowing something about one patient tells you nothing about another. Good work in epidemiology, however, regularly considers records of many types (e.g., patients, workplaces, industrial chemicals) as well as relationships among those records (genetic and social relationships among patients, occupational exposure of patients to chemicals, etc.).

Ignoring this relational information vastly oversimplifies many problems and can make their deep structure all but undiscoverable. Indeed, the importance of such relational information is precisely what led computer scientists to create relational databases and knowledge representations based on first-order logic. To date, however, most technologies for knowledge discovery have lagged behind these decades-old innovations, only addressing the data contained in a single database table and only expressing concepts in representations roughly equivalent to propositional logic.

Addressing fully relational tasks has raised a remarkable array of new problems in statistical inference, required the development of new technologies for knowledge discovery, and raised new questions about the assessment and management of these technologies. The need to investigate these interconnected questions has driven the work of the Knowledge Discovery Laboratory (KDL) at the University of Massachusetts Amherst, and the desire to disseminate our findings led us to create Proximity.

Relational Knowledge Discovery

Traditional machine learning and knowledge discovery techniques identify probabilistic dependencies among the attributes of a single record only. Proximity's modeling algorithms extend this to include attributes of related entities and characteristics of the surrounding relational structure of the data.

To enable efficient model creation, Proximity employs unusual technologies for data storage and access. Its core database uses the decomposition storage model [Copeland and Khoshafian, 1985], a method of vertical fragmentation that allows for a highly flexible data schema. Knowledge discovery virtually requires such a schema, because substantial reinterpretations of the data are frequent and highly desirable. Proximity also uses QGraph, a visual query language that returns graph fragments with highly variable structure, rather than returning sets of individual records with homogeneous structure. Visualization tools in Proximity allow users to browse the data as a graph, examining both the attributes of individual records as well as the higher-level structure of relationships that interconnect records.

Algorithms for constructing statistical models that estimate conditional and joint probability distributions are implemented on top of Proximity's database infrastructure. These algorithms construct relational probability trees [Neville et al., 2003], relational dependency networks [Neville and Jensen, 2003], [Neville and Jensen, 2004], and relational Bayesian classifiers [Neville, Jensen and Gallagher, 2003]. Each of these models is constructed by analyzing a data sample created using a QGraph query and is implemented as a set of operations run on the underlying database.

Proximity Advantages

Proximity's data representation and modeling techniques provide several advantages over traditional methods:

- **Relational models.** Conventional tools cannot exploit the relational structure of data sets. Analysts have to encode the relational structure as propositional features, rather than having the algorithm automatically search over all such features. In addition, such propositional encoding makes it impossible to adjust for relational characteristics of data such as autocorrelation and degree disparity.
- **Graph query language.** Conventional query languages such as SQL make it difficult to retrieve arbitrary subgraphs. Instead, users are limited to retrieving individual records or constructing new records that summarize relational structure. QGraph makes it easy to retrieve and examine arbitrary portions of the graph and thus eases the process of relational knowledge discovery.
- **Flexible data representation.** In a conventional relational database, transforming the schema of a database is a difficult and time-consuming process. A Proximity database does not have a fixed schema. Instead, QGraph queries are used to define the schema for a particular analysis. This can substantially improve the ability to discover knowledge in relational data [Jensen and Neville, *KDD*, 2002].
- **Efficient scaling.** In a traditional database system, increasing the number of attributes on an object decreases the number of records that can be paged into memory at once. In Proximity, each attribute is stored in its own table. While most operations require a join, MonetDB makes such operations very efficient. As a result, an analyst can create hundreds or even thousands of attributes with little or no impact on query speed.

Chapter 2. Getting Started with Proximity

Overview

Proximity provides a suite of applications to support knowledge discovery in relational databases. These applications let you manage and browse databases, create and execute queries, and learn and apply models. This chapter describes the general steps required to use any of these applications. Each specific application is described in more detail in later chapters.

Objectives

The text and exercises in this chapter demonstrate how to

- use this tutorial effectively
- run the MonetDB database server
- run Proximity applications

Using the Tutorial

The tutorial is designed to be read sequentially. Later chapters and sections assume that you are familiar with the preceding material, and earlier exercises create files and database entities that are used by later exercises. If you plan to work through the tutorial exercises, make sure that you create the ProxWebKB database (see Exercise 3.1) used in most of the remaining tutorial exercises. We suggest that you work through the chapters in order. To get the most benefit from the tutorial, complete the exercises using your local installation of Proximity.

The examples in this chapter demonstrate how to use Proximity for both Linux/Mac OS X and Windows systems. The rest of the *Tutorial* provides only the Linux/Mac OS X commands. In most cases, the only differences are using a `.bat` file instead of `.sh` file for Proximity applications, substituting appropriate paths to files, and using the appropriate syntax for the operating system. Windows users should refer back to this chapter if they have questions about specific Proximity applications and commands.

Tutorial conventions

This tutorial uses the following typographic conventions:

`Constant width, bold`	Text you type on the command line or in the Proximity Database Browser
`Constant width, italics`	Text you replace with the appropriate value
`Constant width`	Output from the application or code fragments

Code fragments, application output, and text you type on the command line are usually shown with a gray background. In some cases, the tutorial may include additional line breaks not present in actual application output so that the output fits within a standard page width.

The tutorial uses UNIX-style paths as a generic path syntax. You may need to make appropriate syntax substitutions if you are running Proximity on a Windows platform. Windows-specific examples are included only when users must enter different information or perform different actions to use Proximity on Windows platforms. Long command lines use continuation characters (****) to indicate that the following line is part of the same command. Enter such text on a single line, without the continuation character.

Setting up your environment

You must have Proximity installed locally to complete the exercises in the tutorial. If you have not already installed the Proximity software, see the instructions in Appendix B, *Installation* for complete instructions on obtaining and installing Proximity.

The tutorial uses the `PROX_HOME` environment variable to refer to the location of your local Proximity installation. Windows users must set this variable to use Proximity. Linux and Mac users may want to define this environment variable for their convenience in working through the tutorial. The exercises and examples in this tutorial show `PROX_HOME` set to `/proximity`; however, you can install Proximity in any directory.

The exercises in this tutorial use the ProxWebKB database, which is included in the Proximity distribution. Exercise 3.1 (p. 12) describes how to import this database into Proximity.

Using Proximity

Most Proximity applications can be run either through the Proximity Database Browser or via shell scripts (Linux/Mac OS X) or batch files (Windows) that call the applications.

Proximity applications are supported by a Java API. You can use the API as you would any other Java API in your own Java applications. Proximity also provides a scripting interface that lets you access the full API functionality through Python scripts. Proximity's scripting interface is an important mechanism for working in Proximity and is described in Chapter 6, *Using Scripts*.

The following sections describe how to use the MonetDB server and run Proximity applications. See Appendix A, *Proximity Quick Reference* for a convenient summary of this information.

Running the MonetDB database server

Proximity uses a MonetDB database to store its data. Unlike traditional database systems that start the server separately from connecting to a database, MonetDB attaches a server to a database. To connect to multiple databases, you must start multiple copies of the server.

When you start the MonetDB server, you specify the database to serve. MonetDB stores database files in different locations, depending on your platform:

- **Linux and Mac OS X:** `/usr/local/Monet-mars/var/MonetDB4/dbfarm/`
- **Windows:** `C:\\Documents and Settings\`*`username`*`\Application Data\MonetDB4\dbfarm\` where *`username`* is the login name for the user who installed MonetDB

Database names correspond to the directories immediately under `dbfarm`. When you create the ProxWebKB database in Exercise 3.1, MonetDB creates a `ProxWebKB` directory under `dbfarm`.

Proximity provides an MIL (Monet Interpreter Language) script, `init-mserver.mil`, that loads required modules and makes the server start listening for connections on a given port (30000 by default).

> Proximity 4.3 can be used with either the newer MonetDB 4.20 (Linux/Mac OS X) or MonetDB 4.18.2 (Windows), collectively known as the “Mars” versions of MonetDB, or existing users can opt to continue using MonetDB 4.6.2 with Proximity 4.3. Proximity uses the port number to select the appropriate protocol for the version of MonetDB being used. You must use a port number ≤ 40000 for the Mars versions and a port number > 40000 for MonetDB 4.6.2.

Full MonetDB documentation is available from CWI's website at monetdb.cwi.nl/projects/monetdb/MonetDB/Version4/Documentation/index.html.

Starting the MonetDB server under Linux/Mac OS X

Start the MonetDB server by executing the **Mserver** command, located in `/usr/local/Monet-mars/bin`. You can add this directory to `PATH` as the examples in the *Tutorial* assume, or specify the full path on the command line.

The command line below starts the MonetDB server the default port (30000) for a specified database, using the Proximity initialization script.

```
> Mserver --dbname name $PROX_HOME/resources/init-mserver.mil
```

where *name* is the name of the database to be served.

To use a different port, add the port information to the command line as shown below:

```
> Mserver --dbname name $PROX_HOME/resources/init-mserver.mil --set port=nnnnn
```

where *nnnnn* is the new port number. (Remember to use a port number > 40000 if you are using MonetDB 4.6.2.)

MonetDB responds with a startup message:

```
# MonetDB Server v4.20.0
# based on GDK   v1.20.0
# Copyright (c) 1993-2007, CWI. All rights reserved.
# Compiled for powerpc-apple-darwin8.10.0/32bit with 32bit OIDs; dynamically linked.
# Visit http://monetdb.cwi.nl/ for further information.
Listening on port 30000
MonetDB>
```

The startup message may be slightly different depending on your operating system, the version of MonetDB you are using, and whether you elected to use the default port or specify a different port number on the command line.

If the database does not exist, the MonetDB server starts on the specified port to let you work interactively or to create new databases, but prints the following warning message:

```
!WARNING: GDKlockHome: created directory
    /usr/local/Monet-mars/var/MonetDB4/dbfarm/name/
!WARNING: GDKlockHome: ignoring empty or invalid .gdk_lock.
!WARNING: BBPdir: initializing BBP.
```

where *name* is the name of the database.

Starting the MonetDB server under Windows

The MonetDB server is run from a command (DOS) window. The **Mserver.bat** batch file is located in your local MonetDB installation directory (`C:\Program Files\CWI\MonetDB4` if you installed MonetDB in the default location). You can add this directory to `PATH` as the examples in the *Tutorial* assume, or specify the full path on the command line.

The command line below starts the MonetDB server on the default port (30000) for a specified database, using the Proximity initialization script.

```
> Mserver.bat --dbname name %PROX_HOME%\resources\init-mserver.mil
```

where *name* is the name of the database to be served.

To use a different port, add the port information to the command line as shown below:

```
> Mserver.bat --dbname name %PROX_HOME%\resources\init-mserver.mil \
  --set port=nnnnn
```

where *nnnnn* is the new port number. (Remember to use a port number > 40000 if you are using

MonetDB 4.6.2.)

MonetDB responds with a startup message:

```
# MonetDB Server v4.18.2
# Copyright (c) 1993-2007, CWI. All rights reserved.
# Compiled for i686-pc-win32/32bit; dynamically linked.
# Visit http://monetdb.cwi.nl/ for further information.
Listening on port 30000
MonetDB>
```

The startup message may be slightly different depending on your operating system, the version of MonetDB you are using, and whether you elected to use the default port or specify a different port number on the command line.

If the database does not exist, the MonetDB server starts on the specified port to let you work interactively or to create new databases, but prints the following warning message:

```
!WARNING: GDKlockHome: created directory
    C:\Documents and Settings\All Users\Application Data\MonetDB4\dbfarm\name\
!WARNING: GDKlockHome: ignoring empty or invalid .gdk_lock.
!WARNING: BBPdir: initializing BBP.
```

where *`name`* is the name of the database.

Stopping the MonetDB server

To close the MonetDB server, enter

```
MonetDB> quit();
```

where `MonetDB>` is the MonetDB server prompt. The parentheses and semi-colon are required.

Running Proximity applications

Proximity supports several mechanisms for executing its component applications. You can run applications from the Proximity Database Browser or from the command line using shell scripts or batch files. You can also access complete Proximity API functionality via Python scripts and Java programs. Python scripting is considered to be a Proximity application and is supported through both the Proximity Database Browser and running scripts via the **script.sh** shell script or **script.bat** batch file. The examples in this tutorial provide instructions for running the Proximity applications both from the Proximity Database Browser and from the command line. Executing Proximity applications and code from within Java programs is beyond the scope of this tutorial. Users writing Proximity Java programs should refer to the Javadoc documentation available at `$PROX_HOME/javadoc/index.html`.

Proximity provides the shell scripts and batch files listed in the following table. These files are located in `$PROX_HOME/bin` and are described in more detail in the referenced chapters.

Application	**Linux/Mac OS X shell script**	**Windows batch file**	**Parameters**	**Reference**
Proximity Database Browser	gui.sh	gui.bat	*`host:port`*	Ch. 4

Query runner	query.sh	query.bat	*`host:port`*
			`query-file`
			`output-container`
			`input-container` (opt.)
				Ch. 5

Python script runner	script.sh	script.bat	*`host:port`* *`script-file`*	Ch. 6
XML data import	import-xml.sh	import-xml.bat	*`host:port`* *`input-filename`*	Ch. 3

Application	Linux/Mac OS X shell script	Windows batch file	Parameters	Reference
XML data export	export-xml.sh	export-xml.bat	*host:port* *output-filename* *exportType* (opt.) *exportSpec* (opt.)	Ch. 3
plain text data import	import-text.sh	import-text.bat	*host:port* *input-filename*	Ch. 3
plain text data export	export-text.sh	export-text.bat	*host:port* *output-filename* *exportType* (opt.) *exportSpec* (opt.)	Ch. 3
Database management utilities	db-util.sh	db-util.bat	*host:port* *command*	Ch. 3

For **query.sh**, specifying the input container is optional; the input container defaults to the root container (the entire database) if it is omitted. See “Querying Containers” in Chapter 5for additional information on limiting query scope to the objects and links within an existing container.

For **export-xml.sh**, specifying the exportType and exportSpec is optional; the export defaults to the full database if they are omitted.

For **db-util**, permitted commands are

- **`clear-db`** - clear the database
- **`init-db`** - initialize the database
- **`test-db`** - test the database configuration and print Proximity version information
- **`view-schema`** - print the schema log
- **`view-stats`** - print summary statistics for the database

All the Proximity shell scripts and batch files require a *`host:port`* specification as the first parameter. The parameter specifies the host machine and port number of the database server in the form

`hostname`:*`port_number`*

If you run MonetDB Mars on your local machine and use the default `init-mserver.mil` script when you start the MonetDB server, this correct value for this parameter is

`localhost:30000`

If you are serving the database on `localhost` (your local machine), you can omit that portion of the parameter and use

`:30000`

for the *`hostname`*:*`port_number`* parameter. (Remember that you must specify a port number > 40000 if you are using MonetDB 4.6.2.)

Proximity shell scripts and batch files set the classpath so you do not need to set `CLASSPATH` to use either the Proximity Database Browser or the shell scripts. The Proximity shell scripts and batch files are located in `$PROX_HOME/bin`. You can either add this directory to `PATH`, or specify the path when executing the scripts.

> Proximity shell scripts and batch files, including the command to run the Proximity Database Browser, include `proximity.jar` in the classpath. If you edit Proximity source files or add new classes, you must recreate `proximity.jar` to have the changes available when you use these scripts. If you use the Ant build tool, you can use the “jar” target to create `proximity.jar`.

Running Proximity applications from the command line

In addition to running applications from the Proximity Database Browser, Proximity applications can be executed from the command line.

- **Linux and Mac OS X:** Execute Proximity applications by executing the corresponding shell script from a terminal window.
- **Windows:** Execute Proximity applications by executing the corresponding batch file from a DOS window.

All Proximity commands interact with a database and thus require that you be serving the database using the MonetDB server. Proximity shell scripts and batch files require the MonetDB server's host and port as the first argument and may require additional arguments.

The examples in the tutorial assume that you are running the MonetDB server on your local machine and using the default port for Proximity. Substitute the appropriate host and port information if you are running the MonetDB server on a different machine or are using a different port.

The general form of a Proximity command, when called from the `$PROX_HOME` directory, is:

Linux/Mac OS X:

```
> bin/shell_script hostname:port_number arguments
```

Windows:

```
> bin\batch_file hostname:port_number arguments
```

(Remember that you must specify a port number > 40000 if you are using MonetDB 4.6.2.)

DTD files

Proximity uses XML to represent queries in files, to store model data, and as an off-line representation for database data. Associated document type definitions (DTDs) define the required syntax for these files:

- `graph-query.dtd` defines the syntax for XML query files
- `prox3db.dtd` defines the syntax for XML import data files
- `rbc2.dtd` defines the syntax for relational Bayesian classifier data
- `rpt2.dtd` defines the syntax for relational probability tree data
- `rdn.dtd` defines the syntax for relational dependency network wrapper files

> Users of earlier versions of Proximity should note that the DTDs for the relational Bayesian classifier and the relational probability tree have been updated to newer versions, `rbc2.dtd` and `rpt2.dtd`. Make sure that you use the newer versions of these DTDs.

Proximity requires that the DTD be in the same directory as the corresponding query, data, or model file, or in the directory from which you launch the Proximity application:

- Before editing or executing a saved query, copy `graph-query.dtd` to the directory containing the query file.
- Before importing data from an XML file, copy `prox3db.dtd` to the directory containing the XML data file.
- Before viewing the originating query from the resulting container page, copy `graph-query.dtd` to the directory from which you launch the Proximity Database Browser.
- Before displaying a relational probability tree or relational dependency network, copy the corresponding DTD to the directory containing the model file.
- Before loading a saved model file (e.g., in a script or Java program), copy the corresponding DTD to

the directory containing the model file.

DTD files are located in `$PROX_HOME/resources`.

File and directory shortcuts

Proximity lets you create shortcuts to commonly used files and directories. Shortcuts appear in one of the panes of the Open dialog that is displayed when you open or save Proximity files. Shortcuts are stored in the file `prox.accessory.dirs`, which is stored in your home directory.

Creating a file or directory shortcut

The following steps require that you be running the Proximity Database Browser. See Exercise 4.1 for information on starting and using the Proximity Database Browser.

1. Perform an action that causes the Open dialog to be displayed. For example, choose **Edit Query** from the **Query** menu.
2. Navigate to the directory containing the file or directory to which you want to create a shortcut.
3. In the file list pane of the Open dialog, choose the desired file or directory. Click **Add**. The selected file or directory is added to the Shortcuts list in the Shortcuts pane.

To further simplify file access, you can define aliases for commonly used files and directories. These aliases are displayed in the shortcut pane instead of the path to the file or directory name.

4. [Optional] Choose the file or directory name in the shortcuts pane.
5. [Optional] Enter the name you want to use for this file or directory in the Alias text box. Click **Set**. Proximity replaces the path with the designated alias.

Logging options

Proximity uses the log4j package to provide logging messages during program execution. Log4j is part of the Apache Logging Services project, an open-source library of tools for cross-language logging services. Designed to minimize the cost of logging, log4j permits output of log messages at arbitrary granularity.

Log4j uses configuration files to determine the level of detail included in logging messages. To set the logging level in Proximity, you can

- use the default settings (do not specify a configuration file)
- use one of the configuration files included in the Proximity distribution
- write your own configuration file

Proximity log4j configuration files are included in the `$PROX_HOME/example/config` directory. See the *Proximity Cookbook* for additional information on creating log4j configuration files and for details on the example configurations files included in the Proximity distribution. Follow the instructions below to use one of the log4j configuration files in Proximity.

Using the Proximity log4j configuration files

1. Copy the configuration file to the directory from which you launch Proximity applications. (Tutorial exercises launch Proximity applications from the `$PROX_HOME` directory.)
2. Rename the configuration file to `prox.lcf`.

   Proximity automatically uses `prox.lcf` as the log4j configuration file.

For additional information on log4j, including how to create new configuration files, see the project web pages at `logging.apache.org/log4j/docs/`.

Contact information

We welcome your comments and suggestions. Please use the following addresses to contact us about your experiences with Proximity:

- *proximity-bugs@kdl.cs.umass.edu* - Proximity bug reports and documentation errors
- *proximity-support@kdl.cs.umass.edu* - requests for general assistance with Proximity software
- *proximity@kdl.cs.umass.edu* - general comments, suggestions, and criticism

You may also contact us at

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

The Knowledge Discovery Laboratory maintains two mailing lists for Proximity news and information.

- *Proximity-announce* is a low-volume list carrying only announcements of significant project milestones, such as new releases. To subscribe, send an email message to *majordom@cs.umass.edu* with the text **`subscribe proximity-announce`** in the body of the message.
- *Proximity-list* is a general forum for discussing Proximity issues. To subscribe, send an email message to *majordom@cs.umass.edu* with the text **`subscribe proximity-list`** in the body of the message.

Tips and Reminders

- Windows users must set the environment variable `$PROX_HOME` to the root of the local Proximity installation; Linux and Mac users may wish to set this environment variable for convenience.
- Proximity shell scripts and batch files are located in `$PROX_HOME/bin`.
- Run Proximity applications from the Proximity Database Browser or from the command line using shell scripts or batch files.
- All Proximity shell scripts and batch files require a *`host:port`* parameter.
- The `init-mserver.mil` script sets the port to `30000`. You can change this value using `--set port=`*`nnnnn`* when starting Mserver.
- Proximity 4.3 uses the port number to select the appropriate protocol for the version of MonetDB being used. You must use a port number ≤ 40000 for the Mars versions (4.18.2 and 4.20) and a port number > 40000 for MonetDB 4.6.2.
- Proximity DTD files are located in `$PROX_HOME/resources`.
- Copy the appropriate DTD file to directories containing query files, XML import data files, and model files.
- If you edit or add source code, you must recreate `proximity.jar` to have those changes available when you use the Proximity shell scripts or batch files.
- Define shortcuts to frequently used files and directories in the Open dialog.
- Choose the appropriate log4j configuration file to set the level of detail included in logging messages.
- Keep up to date with future Proximity news and releases by subscribing to the *proximity-announce* mailing list.

Additional information

- See Appendix A, *Proximity Quick Reference* for a summary of important Proximity commands.
- See Chapter 3, *Importing and Exporting Proximity Data* for information on using the database management utilities and importing and exporting XML or plain text data.
- See Chapter 5, *Querying the Database* for information on using the query runner.
- See Chapter 6, *Using Scripts* for information on using the Python script runner.

Chapter 3. Importing and Exporting Proximity Data

Overview

Proximity provides two text formats for importing and exporting relational data to and from a Proximity database:

- an XML format
- a plain text format

In general, because the associated import and export utilities provide better error checking, we recommend using the XML format when feasible. The use of plain text may be appropriate when the task does not require error checking or the database is too large for satisfactory use of XML.

These formats let you define objects, links, and arbitrary attributes on those objects and links as well as specifying containers and subgraphs for databases that include these structures. (Containers and subgraphs are created as a result of executing queries and are described in more detail in Chapter 5, *Querying the Database*.) Proximity also provides utilities for importing and exporting data using these formats.

This chapter describes how to import and export both XML and plain text data. You can import and export individual containers and attributes as well as complete databases. Proximity also supports the export of selected data to tab-delimited text files from the Proximity Database Browser; these specialized export operations are described in “Specialized Data Export” later in this chapter. The XML data format is described in Appendix C, *Proximity XML Format*. The DTD for this XML format is located in `$PROX_HOME/resources/prox3db.dtd`. The plain text data format is described in Appendix D, *Proximity Text Data Format*.

> For all import and export operations, the data files must reside on the same machine as that serving the database.

Objectives

The text and exercises in this chapter demonstrate how to

- import XML data into Proximity
- import plain text data into Proximity
- create the sample database used for the tutorial exercises
- convert standard, tabular data to Proximity’s XML import format
- export Proximity data to XML
- export Proximity data to plain text
- export attribute values to a tab-delimited text file
- export NST data to a tab-delimited text file (see “Exporting NST data” (p. 24) for a description of NSTs)

Importing XML Data

This section describes how to import XML data into Proximity using the provided ProxWebKB database as an example. Proximity lets you import a complete database, including any subgraphs and containers, or you can import individual attributes or containers.

By default, Proximity restricts the structure of the XML data file to ensure that you cannot accidentally create identity conflicts. For example, Proximity prohibits adding additional objects to a database once the initial set of objects has been defined; therefore, the XML data file can contain only a single objects section and you cannot add more objects with a subsequent import. Similarly, you cannot add new links or add more values to an attribute once you have completed importing the initial set of links or attribute values. You can override these restrictions by using the `noChecks` option to the import script, described in “Importing XML data using `noChecks`” (p. 15).

⚠ You are responsible for ensuring the integrity of the data in an XML file. Proximity makes no checks to ensure that attributes are assigned to items (objects, links, subgraphs, or containers) that are actually present in the database. Assigning attribute values to non-existent items does not trigger an exception or warning.

> Proximity automatically converts all attribute names to lower case when importing XML data; attribute values retain their original case. (Proximity ignores case when matching attribute names but obeys case when matching attribute values.)
>
> The characters `<`, `>`, and `&` must be represented by the corresponding XML entities, `&lt;`, `&gt;`, and `&amp;`. For compatibility with MonetDB, single quotes, double quotes, the pipe symbol (`|`), and newline characters in the XML data are automatically changed to underscores during import.

The sections below walk through the process of first importing a database and then importing values for a new attribute on the existing database objects.

Importing databases using XML

The exercise below walks through the process of importing the XML version of the ProxWebKB database. ProxWebKB was developed from the WebKB relational data set [Craven et al., 1999] available from www-2.cs.cmu.edu/~WebKB/. The version used for the Proximity tutorial has been modified from the public distribution to meet the needs of this tutorial. Modifications include some data clean up and the the creation of additional object attributes based on the data in the distributed version.

> Most of the remaining tutorial exercises require that you use the ProxWebKB database. Make sure you complete Exercise 3.1 to create the ProxWebKB database.

Exercise 3.1. Importing the ProxWebKB data into Proximity:

1. Uncompress the compressed ProxWebKB XML data file.

```
> cd $PROX_HOME/doc/user/tutorial/examples
> gunzip proxwebkb.xml.gz
```

2. Copy the file `prox3db.dtd` from `$PROX_HOME/resources` to the directory containing the XML data file, `$PROX_HOME/doc/user/tutorial/examples`.

> Proximity requires that the DTD file, `prox3db.dtd`, be present in the directory containing the XML data file. The DTD file is included in `$PROX_HOME/resources`. Make sure you copy the DTD to any other directories containing XML data files you want to import into Proximity.

```
> cp $PROX_HOME/resources/prox3db.dtd $PROX_HOME/doc/user/tutorial/examples/
```

3. Start the MonetDB server. Data files must be on the same machine as that serving the database.

   ```
   > Mserver --dbname ProxWebKB $PROX_HOME/resources/init-mserver.mil
   ```

   The `init-mserver.mil` script sets the port for the server to 30000. To use a different port, add **--set port=*nnnnn*** (where *nnnnn* is the new port number) to the command line. For example:

   ```
   > Mserver --dbname ProxWebKB $PROX_HOME/resources/init-mserver.mil \
     --set port=45678
   ```

   Remember to use a port number > 40000 if you are using MonetDB 4.6.2. See “Running the MonetDB database server” (p. 4) for more information on starting and using the MonetDB server.

   Because the database does not exist, MonetDB prints warning statements along with its usual startup message:

   ```
   !WARNING: GDKlockHome: created directory
       /usr/local/Monet-mars/var/MonetDB4/dbfarm/ProxWebKB/
   !WARNING: GDKlockHome: ignoring empty or invalid .gdk_lock.
   !WARNING: BBPdir: initializing BBP.
   # MonetDB Server v4.20.0
   # based on GDK   v1.20.0
   # Copyright (c) 1993-2007, CWI. All rights reserved.
   # Compiled for powerpc-apple-darwin8.10.0/32bit with 32bit OIDs; dynamically linked.
   # Visit http://monetdb.cwi.nl/ for further information.
   Listening on port 30000
   MonetDB>
   ```

   The startup message may be slightly different depending on your operating system and the version of MonetDB you are using.

   MonetDB also creates a `ProxWebKB` directory in its `dbfarm` directory to hold the new database.

   Leave the MonetDB server running for the remainder of the import process. You must be serving the database for any Proximity action that interacts with database data.

4. Initialize the new Proximity database. (Substitute the appropriate port number if you are using a different port.)

   ```
   > cd $PROX_HOME
   > bin/db-util.sh localhost:30000 init-db
   ```

   Proximity outputs the following trace (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

   ```
   INFO  app.DBUtil: * connecting to db
   INFO  app.DBUtil: * database opened; initializing Prox tables
   INFO  db.DB: * initializing Proximity database
   INFO  app.DBUtil: * tables initialized
   INFO  app.DBUtil: * disconnecting from db
   INFO  app.DBUtil: * done
   ```

5. Import the XML data file into the new Proximity database. (Substitute the appropriate port number if you are using a different port.)

   ```
   > bin/import-xml.sh localhost:30000 \
     $PROX_HOME/doc/user/tutorial/examples/proxwebkb.xml
   ```

   When the import process is finished, Proximity reports on the number of database entities created.

   ```
   INFO  app.ImportXMLApp: * done importing; counts:
     4135 objects, 10934 links, 13 attributes, 222052 attribute values,
     0 containers, 0 subgraph items
   ```

Importing database elements using XML

In addition to importing a complete database, Proximity lets you import individual containers and attributes. This feature lets Proximity users share data and results and store data off line. To import additional data into an existing database, use the **import-xml.sh** script (**import-xml.bat** for Windows) to import a Proximity XML data file containing the new data.

By default, you cannot import objects or links into an existing database if it already contains any items of that type. That is, you cannot import any additional objects if you have previously created at least one object in the database and you cannot import any additional links if you have previously created at least one link in the database. Similarly, you can only import new attributes. Once an attribute has been defined for a database, you cannot add additional values for that attribute. This behavior can be overridden through use of the `noChecks` flag, described in “Importing XML data using `noChecks`” (p. 15).

Imported containers are always created at the top level (directly under the root container), regardless of where the container lived in the source database. Any nested containers within the imported container retain their relative nesting, however. For example, if you exported the `/1d-clusters/samples` container, which includes nested containers `/1d-clusters/samples/0` and `/1d-clusters/samples/1`, and later imported that container into another database, the destination database ends up with the containers `/samples/0` and `/samples/1` without the parent 1d-clusters container, regardless of whether the destination database already includes a 1d-clusters container. (The container hierarchy notation used in this paragraph is explained in “Exporting Data to XML”.)

When importing attributes and containers, you are responsible for ensuring that object, link, subgraph, and container identifiers match those in the existing database. Proximity makes no checks to ensure that attributes are assigned to items that are actually present in the database. Errors in identifiers may result in inaccurate data being stored in the database.

All containers must have a unique identifier. You cannot import a container if its identifier matches an existing container identifier in the database.

The following exercise walks through the process of importing a new attribute, url_hierarchy3, and its values into the existing ProxWebKB database. This attribute stores the third directory in the path after the domain name, extracted from the object’s URL. We can import this attribute into an existing database because the ProxWebKB database created in Exercise 3.1 does not include any values for the url_hierarchy3 attribute.

Exercise 3.2. Importing attribute values using XML:

Before beginning, make sure that you are serving the ProxWebKB database (created in Exercise 3.1) using Mserver. You must have completed Exercise 3.1 before running the current exercise. Data files must be on the same machine as that serving the database.

1. Uncompress the file containing the url_hierarchy3 attribute values.

```
> cd $PROX_HOME/doc/user/tutorial/examples
> gunzip url_hierarchy3_attr.xml.gz
```

2. Change to the `$PROX_HOME` directory.
3. Import the url_hierarchy3 attribute data. (Substitute the appropriate port number if you are using a different port.)

```
> bin/import-xml.sh localhost:30000 \
  $PROX_HOME/doc/user/tutorial/examples/url_hierarchy3_attr.xml
```

When the import process is finished, Proximity reports on the number of database entities created.

```
INFO  app.ImportXMLApp: * done importing; counts:
   0 objects, 0 links, 1 attributes, 868 attribute values,
   0 containers, 0 subgraph items
```

Because many URLs do not include three levels of directories after the domain name, only 868 out of 4135 objects have a value for this attribute.

Importing XML data using noChecks

To ensure that any new data does not conflict with existing data, Proximity's import process restricts when and how you can add additional elements to an existing database. In many instance, however, careful data management can eliminate such conflicts. In such cases, users can override the default import restrictions by setting the `noChecks` import option to true.

When `noChecks` is true, Proximity makes no checks to see if the imported data conflicts with the existing database. If imported objects or links already exist in the database, Proximity creates duplicate entries for these items. When `noChecks` is false or not specified, adding new objects or links to a database that already contains such items results in an error. You can also use the `noChecks` option to import additional values for previously defined attributes (an error without this option). The only check in this case is ensuring that the "new" attribute is of the same type as the previously defined attribute.

When `noChecks` is true, Proximity automatically recodes container identifiers when imported containers conflict with containers already present in the database. For example, if the database already has a container with an ID of 2, and the XML data file to be imported also includes a container with an ID of 2, the import process assigns a new, non-conflicting identifier to the imported container. Any associated data (container attributes and nesting relationships) are similarly recoded.

The following exercise adds a small number of new link_tag attribute values to the database. The new values are added solely for illustrative purposes; they are not used in later *Tutorial* exercises.

Exercise 3.3. Importing additional link_tag attribute values:

Before beginning, make sure that you are serving the ProxWebKB database (created in Exercise 3.1) using Mserver. You must have completed Exercise 3.1 before running the current exercise. Data files must be on the same machine as that serving the database.

1. Try to import the new link_tag attribute values without using the `noChecks` option. (Substitute the appropriate port number if you are using a different port.)

   ```
   > bin/import-xml.sh localhost:30000 \
     $PROX_HOME/doc/user/tutorial/examples/more-linktag-values.xml
   ```

   Proximity attempts to import the new attribute values, but reports an error because the link_tag attribute already exists.

   ```
   ERROR app.ImportXMLApp: java.lang.IllegalArgumentException:
   Attribute already exists in the database: link_tag. You may use the
   noChecks flag if you want to ignore this check.
   ```

2. Import the additional link_tag attribute values, specifying that `noChecks` is true.

   ```
   > bin/import-xml.sh localhost:30000 \
     $PROX_HOME/doc/user/tutorial/examples/more-linktag-values.xml true
   ```

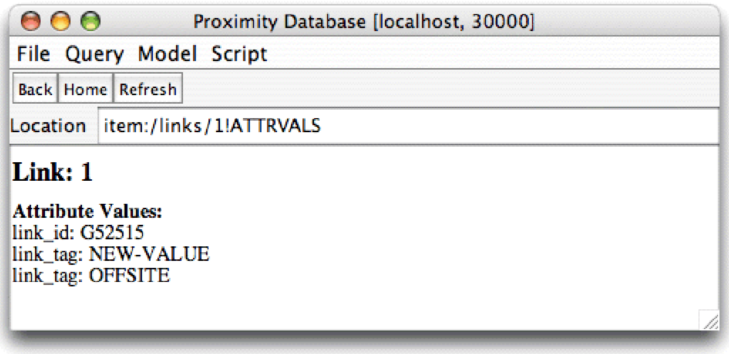
   When the import process is finished, Proximity reports on the number of database entities created (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

```
INFO  app.ImportXMLApp: * importing database from
   /proximity/doc/user/tutorial/examples/more-linktag-values.xml
   NOT CHECKING FOR DUPLICATES
INFO  app.ImportXMLApp: * converting xml to bulk import text files;
   dir: /tmp/prox-xml-import62294
INFO  app.ImportXMLApp:   attributes
INFO  app.ImportXMLApp:   attribute: link_tag, L, str
INFO  app.ImportTextApp: * importing database from /tmp/prox-xml-import62294
INFO  app.ImportTextApp:   Loading object table
INFO  app.ImportTextApp:   Loading link table
INFO  app.ImportTextApp:   Loading attributes
INFO  app.ImportTextApp:   Loading attribute: av_0
INFO  app.ImportTextApp:   Loading containers
INFO  app.ImportTextApp: * done importing
INFO  app.ImportXMLApp: * done importing; counts:
   0 objects, 0 links, 1 attributes, 5 attribute values,
   0 containers, 0 subgraph items
```

Note that the start of the trace includes a warning that the import process is not checking for duplicates.

3. If you want, examine link 1 in the Proximity Database Browser to see its attribute values. (A summary of steps for examining link 1’s attribute values is included, below. See Chapter 4, *Exploring Data* for complete information on using the Proximity Database Browser.)

   a. From the Proximity Database Browser start page, click **Links**.
   b. Click **1**.
   c. Click **attrs**.


Proximity Database [localhost, 30000]
File Query Model Script
Back Home Refresh
Location item:/links/1!ATTRVALS

Link: 1

Attribute Values:
link_id: G52515
link_tag: NEW-VALUE
link_tag: OFFSITE

Because link 1 already had a value of “OFFSITE” for the link_tag attribute, the new data is added as an additional value for this attribute.

Transforming Tabular Data to XML

Proximity allows you to import data from other databases using a three-step process:

1. Export the data to the specified tabular format.
2. Convert the data to the Proximity XML import format.
3. Import the XML data into Proximity.

See `$PROX_HOME/doc/user/text2xml/text2xml.txt` for a specification of the required file format and instructions for using the **text2xml.pl** script to transform tabular data to the Proximity XML import format.

To import XML data into Proximity, follow the procedure described in Exercise 3.1 to create and initialize the MonetDB database and import the data.

Exporting Data to XML

Proximity provides the **export-xml.sh** shell script (**export-xml.bat** for Windows) for exporting data in Proximity databases to an XML format. You can export complete databases as well as selected database elements (attributes and containers).

During export, Proximity converts the characters `<`, `>`, and `&` to the corresponding entities: `&lt;`, `&gt;`, and `&amp;`. Newline characters are converted to underscores.

The general form of a call to the **export-xml.sh** script is

```
export-xml.sh  host:port  filename  exportType  exportSpec
```

where

- *`host:port`* is the MonetDB server's host and port
- *`filename`* is the name of the file that will contain the XML output
- *`exportType`* is an optional argument that indicates the type of data to be exported
- *`exportSpec`* is an optional argument that indicates the specific instance of the data type specified by *`exportType`*

The *`exportType`* and *`exportSpec`* arguments are omitted when exporting the complete database.

Legal values for *`exportType`* are

- `object-attribute`
- `link-attribute`
- `container-attribute`
- `container`

Note that the values of *`exportType`* use hyphens as shown above only when used as arguments for the **export-xml.sh** shell script or **export-xml.bat** batch file. Use uppercase text and substitute underscores instead of hyphens (e.g., `OBJECT_ATTRIBUTE`) when using these keywords within a script or in Java code.

When exporting an attribute, the value of *`exportSpec`* is the name of the attribute to be exported.

When exporting a container, the value of *`exportSpec`* is the container name and a UNIX-like path that indicates where the target container resides in the container hierarchy. The root container is specified by a single slash (`/`) with child containers appended to the path. For example, to export the samples container, which is a child of the 1d-clusters container, use `/1d-clusters/samples` as the value of *`exportSpec`*. Do not include a trailing `/` in the path. The root container is a virtual container provided as a convenience for accessing other containers—you cannot export the root container.

Exporting databases to XML

Exercise 3.4 walks through the process of exporting the complete ProxWebKB database.

Exercise 3.4. Exporting a database to XML:

The export process overwrites any existing data in the output file without warning. Make sure that output file is empty or can be safely overwritten.

1. Change to the $PROX_HOME directory.
2. Export the database. (Substitute the appropriate port number if you are using a different port.)

```
> bin/export-xml.sh localhost:30000 \
  $PROX_HOME/doc/user/tutorial/examples/my_proxwebkb.xml
```

Data files can only be exported to the same machine as that serving the database.

Proximity outputs the following trace (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

```
INFO  app.ExportXMLApp: * exporting database to:
   /proximity/doc/user/tutorial/examples/my_proxwebkb.xml
INFO  app.ExportXMLApp: * saving objects
INFO  app.ExportXMLApp: * saving links
INFO  app.ExportXMLApp: * saving O attributes
INFO  app.ExportXMLApp: * saving attribute school
INFO  app.ExportXMLApp: * saving attribute url_server_info
INFO  app.ExportXMLApp: * saving attribute url_protocol
INFO  app.ExportXMLApp: * saving attribute url_hierarchy1
INFO  app.ExportXMLApp: * saving attribute url_hierarchy2
INFO  app.ExportXMLApp: * saving attribute url
INFO  app.ExportXMLApp: * saving attribute page_num_outlinks
INFO  app.ExportXMLApp: * saving attribute page_num_inlinks
INFO  app.ExportXMLApp: * saving attribute url_hierarchy1b
INFO  app.ExportXMLApp: * saving attribute page_words_top100
INFO  app.ExportXMLApp: * saving attribute pagetype
INFO  app.ExportXMLApp: * saving attribute url_hierarchy3
INFO  app.ExportXMLApp: * saving L attributes
INFO  app.ExportXMLApp: * saving attribute link_id
INFO  app.ExportXMLApp: * saving attribute link_tag
INFO  app.ExportXMLApp: * saving C attributes
INFO  app.ExportXMLApp: * done exporting
```

Exporting database elements to XML

Proximity lets you export individual attributes and containers for off-line storage or to share with other Proximity users. Care must be taken when importing this data back into Proximity. You must ensure that all relevant identifiers (for objects, links, containers, and subgraphs) in the data file correctly match the corresponding identifiers in the existing database. You cannot import attribute data if that attribute name is already used in the destination database. Similarly, you cannot import a container if that container identifier is already used by the destination database.[1] Importing containers can be problematic if different sets of queries have been run on the source and destination databases.

Exporting a container exports all the container's subgraphs, any attributes on those subgraphs, and any containers nested within the specified container. Exporting a container does not export the container's attributes; you must export any container attributes as a separate step using the `container-attribute` export type.

Exercise 3.5 walks through the process of exporting the url_server_info object attribute from the ProxWebKB database.

Exercise 3.5. Exporting an attribute to XML:

This exercise walks through the process of exporting an attribute to the Proximity XML data format. Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Data files can only be exported to the same machine as that serving the database.

The export process overwrites any existing data in the output file without warning. Make sure that output file is empty or can be safely overwritten.

[1] These import restrictions can be overridden by using the `noChecks` option. See "Importing XML data using `noChecks`"for information on using this import option.

1. Change to the `$PROX_HOME` directory.
2. Export the url_server_info object attribute. (Substitute the appropriate port number if you are using a different port.)

```
> bin/export-xml.sh localhost:30000 \
  $PROX_HOME/doc/user/tutorial/examples/url_server_info_attr.xml \
  object-attribute url_server_info
```

Data files can only be exported to the same machine as that serving the database.

Proximity outputs the following trace (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

```
INFO  app.ExportXMLApp: * exporting database to:
   /proximity/doc/user/tutorial/examples/url_server_info_attr.xml
INFO  app.ExportXMLApp: * saving attribute url_server_info
INFO  app.ExportXMLApp: * done exporting
```

Importing Plain Text Data

This section describes how to import plain text data into Proximity. Proximity lets you import a complete database, including any subgraphs and containers, or you can import individual attributes or containers.

The utilities that use the Proximity plain text data format perform no error checking. Proximity makes no checks to ensure that attributes are assigned to items (objects, links, subgraphs, or containers) that are actually present in the database. Assigning attribute values to non-existent items does not trigger an exception or warning. No checks are made to ensure that existing data is not being incorrectly overwritten. Although this format requires less disc space than the XML format and its use can improve import and export speed, you are solely responsible for maintaining data integrity and consistency when using this format.

Unlike when using the XML data format, Proximity does not prohibit adding additional elements to a database once those elements have been defined when importing data using the plain text data format. Therefore, there is no need for a equivalent to the `noChecks` option available for importing XML data. When using the plain text format, you must take care to ensure the integrity and consistency of your data; Proximity will not necessarily alert you to data errors when using this format.

As with importing XML data, Proximity automatically converts all attribute names to lower case when importing plain text data; attribute values retain their original case. (Proximity ignores case when matching attribute names but obeys case when matching attribute values.)

For compatibility with MonetDB, we recommend that all single quotes, double quotes, and newline characters be converted to underscores. This substitution is performed automatically when importing XML data but must be performed as a pre-processing step when importing text data. Note that values such as “`&quot;`” are treated as strings and not XML entities.

Importing databases using plain text

The exercise below walks through the process of importing a sample database using the plain text data format. This database stores selected data for a small set of movies, actors, and directors. The set of files for this exercise illustrate how the plain text data format represents all types of Proximity database entities.

Exercise 3.6. Importing a database using plain text data:

This exercise and exercise Exercise 3.7 use a different database than that used for most *Tutorial* exercises. Before beginning these exercises, make sure that you are no longer serving the ProxWebKB database. Data files must be on the same machine as that serving the database.

1. Uncompress the plain text data files.

   ```
   > cd $PROX_HOME/doc/user/tutorial/examples
   > gunzip movie_db.tar.gz
   ```

   Uncompressing this tar file creates a `MovieDB` directory under the `$PROX_HOME/doc/user/tutorial/examples` directory. All plain text data files required for the current import operation must be located in the same directory.

2. Start the MonetDB server. Data files must be on the same machine as that serving the database.

   ```
   > Mserver --dbname MovieDB $PROX_HOME/resources/init-mserver.mil
   ```

   The `init-mserver.mil` script sets the port for the server to 30000. Remember to use a port number > 40000 if you are using MonetDB 4.6.2. See “Running the MonetDB database server” (p. 4) for more information.

   Because the database does not exist, MonetDB prints warning statements along with its usual startup message:

   ```
   !WARNING: GDKlockHome: created directory
       /usr/local/Monet-mars/var/MonetDB/dbfarm/MovieDB/
   !WARNING: GDKlockHome: ignoring empty or invalid .gdk_lock.
   !WARNING: BBPdir: initializing BBP.
   # MonetDB Server v4.20.0
   # based on GDK   v1.20.0
   # Copyright (c) 1993-2007, CWI. All rights reserved.
   # Compiled for powerpc-apple-darwin8.10.0/32bit with 32bit OIDs; dynamically linked.
   # Visit http://monetdb.cwi.nl/ for further information.
   Listening on port 30000
   MonetDB>
   ```

   The startup message may be slightly different depending on your operating system and the version of MonetDB you are using.

   MonetDB also creates a `MovieDB` directory in its `dbfarm` directory to hold the new database.

   Leave the MonetDB server running for the remainder of the import process. You must be serving the database for any Proximity action that interacts with database data.

3. Initialize the new Proximity database. (Substitute the appropriate port number if you are using a different port.)

   ```
   > cd $PROX_HOME
   > bin/db-util.sh localhost:30000 init-db
   ```

   Proximity outputs the following trace (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

   ```
   INFO  app.DBUtil: * connecting to db
   INFO  app.DBUtil: * database opened; initializing Prox tables
   INFO  db.DB: * initializing Proximity database
   INFO  app.DBUtil: * tables initialized
   INFO  app.DBUtil: * disconnecting from db
   INFO  app.DBUtil: * done
   ```

4. Import the plain text data file into the new Proximity database. (Substitute the appropriate port number if you are using a different port.)

```
> bin/import-text.sh localhost:30000 \
  $PROX_HOME/doc/user/tutorial/examples/MovieDB
```

The plain text data files must be on the same machine as that serving the (still empty) database. You must provide an absolute path to the data files; relative paths cannot be used.

During import, Proximity reports on the entities being defined (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

```
INFO  app.ImportTextApp: * importing database from
   /proximity/doc/user/tutorial/examples/MovieDB
INFO  app.ImportTextApp:   Loading object table
INFO  app.ImportTextApp:   Loading link table
INFO  app.ImportTextApp:   Loading attributes
INFO  app.ImportTextApp:   Loading attribute: O_attr_objtype.data
INFO  app.ImportTextApp:   Loading attribute: O_attr_title.data
INFO  app.ImportTextApp:   Loading attribute: O_attr_name.data
INFO  app.ImportTextApp:   Loading attribute: L_attr_linktype.data
INFO  app.ImportTextApp:   Loading attribute: C_attr_qgraph_query.data
INFO  app.ImportTextApp:   Loading containers
INFO  app.ImportTextApp:   Loading container: si_0
INFO  app.ImportTextApp:   Loading container attribute: si_0_attr_samplenumber.data
INFO  app.ImportTextApp:   Loading container: si_1
INFO  app.ImportTextApp:   Loading container: si_2
INFO  app.ImportTextApp:   Loading container: si_3
INFO  app.ImportTextApp: * done importing
```

Importing database elements using plain text

In addition to importing a complete database, Proximity lets you import individual containers and attributes using the plain text data format. To import additional data into an existing database, use the **import-text.sh** (**import-text.bat** for Windows) script to import a plain text data file containing the new data.

To import data into an existing database, all relevant files must be present in the same directory. To ensure that no unwanted data is imported, we recommend using a different directory for storing the required files for each import operation.

Recall that Proximity performs no error checking when importing plain text data. You are entirely responsible for ensuring the consistency and integrity of data imported using this format.

The following exercise walks through the process of importing a new attribute, birthyear, and its values into the existing MovieDB database.

Exercise 3.7. Importing an attribute using plain text data:

Before beginning, make sure that you are serving the MovieDB database (created in Exercise 3.6) using Mserver. You must have completed Exercise 3.6 before running the current exercise. Data files must be on the same machine as that serving the database.

1. Uncompress the plain text data files.

```
> cd $PROX_HOME/doc/user/tutorial/examples
> gunzip movie_attr.tar.gz
```

Uncompressing this tar file creates a `MovieAttr` directory under the `$PROX_HOME/doc/user/tutorial/examples` directory. All plain text data files required for the current import operation must be located in the same directory.

2. Examine the files in the `$PROX_HOME/doc/user/tutorial/examples/MovieAttr` directory.

   Note that this directory contains two files:

   - `attributes.data`
   - `O_attr_birthyear.data`

   The `attributes.data` file defines the birthyearattribute that we want to import and the `O_attr_birthyear.data` file provides the values for this attribute. When importing selected database elements, you do not need to provide files for other types of data such as objects or links.

3. Import the birthyear attribute data. (Substitute the appropriate port number if you are using a different port.)

   ```
   > bin/import-text.sh localhost:30000 $PROX_HOME/doc/user/tutorial/examples/MovieDB
   ```

   Data files must be on the same machine as that serving the database.

   During import, Proximity reports on the entities being defined (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

   ```
   INFO kdl.prox.app.ImportTextApp - * importing database from
      /proximity/doc/user/tutorial/examples/MovieAttr
   INFO kdl.prox.app.ImportTextApp -   Loading object table
   INFO kdl.prox.app.ImportTextApp -   Loading link table
   INFO kdl.prox.app.ImportTextApp -   Loading attributes
   INFO kdl.prox.app.ImportTextApp -   Loading attribute: O_attr_birthyear.data
   INFO kdl.prox.app.ImportTextApp -   Loading containers
   INFO kdl.prox.app.ImportTextApp - * done importing
   ```

Exporting Plain Text Data

Proximity provides the **export-text.sh** shell script (**export-text.bat** for Windows) for exporting data in Proximity databases to an XML format. You can only export complete databases, but you can prune and edit the resulting files if you want to preserve only a portion of the exported data, such as the data for a single attribute or container.

Unlike XML data import, Proximity performs no character conversion during text data export. If attribute values contain symbols such as <, >, or &, they are exported as is to the resulting text files and are not converted to the corresponding XML entities.

The general form of a call to the **export-text.sh** script is

```
export-text.sh  host:port  directory
```

where

- *`host:port`* is the MonetDB server’s host and port
- *`directory`* is the absolute path to the directory that will contain the resulting data files; this directory must be on the same machine as that serving the database

Exporting databases using plain text

Exercise 3.8 walks through the process of exporting the complete database created in Exercise 3.6.

Exercise 3.8. Exporting a database to plain text:

Before beginning, make sure that you are serving the MovieDB database using Mserver. Data files can only be exported to the same machine as that serving the database.

The export process overwrites existing data files in the output directory without warning. Make sure that output directory is empty or can be safely overwritten.

1. Create a `$PROX_HOME/doc/user/tutorial/examples/TextExport` directory.
2. Export the database. (Substitute the appropriate port number if you are using a different port.) You can only export text to an existing directory on the machine serving the database.

   ```
   > bin/export-text.sh localhost:30000 \
     $PROX_HOME/doc/user/tutorial/examples/TextExport
   ```

   Proximity outputs the following trace (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

   ```
   INFO kdl.prox.app.ExportTextApp - * exporting database to
     /proximity/doc/user/tutorial/examples/TextExport
   INFO kdl.prox.app.ExportTextApp -   Exporting object table
   INFO kdl.prox.app.ExportTextApp -   Exporting link table
   INFO kdl.prox.app.ExportTextApp -   Exporting attributes
   INFO kdl.prox.app.ExportTextApp - objtype
   INFO kdl.prox.app.ExportTextApp - title
   INFO kdl.prox.app.ExportTextApp - name
   INFO kdl.prox.app.ExportTextApp - birthyear
   INFO kdl.prox.app.ExportTextApp - linktype
   INFO kdl.prox.app.ExportTextApp - qgraph_query
   INFO kdl.prox.app.ExportTextApp -   Exporting containers
   INFO kdl.prox.app.ExportTextApp -      si_0
   INFO kdl.prox.app.ExportTextApp -      si_1
   INFO kdl.prox.app.ExportTextApp -      si_2
   INFO kdl.prox.app.ExportTextApp -      si_3
   INFO kdl.prox.app.ExportTextApp - * done exporting
   ```

Specialized Data Export

Proximity can also export selected subsets of data using the Proximity Database Browser. You can export the set of values for a given attribute, or the data in a selected nested synchronized table (NST) to tab-delimited text files.

Exporting attribute values

It is occasionally useful to be able to use or analyze a set of attribute values in another application. To facilitate this, Proximity lets you easily export the set of values for any attribute to a tab-delimited text file.

Exporting attribute values to tab-delimited text files

The following steps require that you be serving a database using the MonetDB server and running the Proximity Database Browser. See Exercise 4.1 for information on starting and using the Proximity Database Browser.

1. Display the histogram for the target attribute in the Proximity Database Browser. (See “Exploring Attributes” (p. 32) for information on how to display the attribute’s histogram.)
2. Click **export**. Proximity displays the Open dialog.
3. Navigate to the target output directory and enter a file name for the exported data in the Save As box. You can only export data to a file on the same machine as that serving the database.
4. Click **Save**. Proximity saves the exported data in the specified file.

Exporting NST data

Nested synchronized tables (NSTs) are internal Proximity data structures that provide a more convenient and intuitive way to view the vertically fragmented data required by MonetDB's storage model. You can export the data in any NST to a tab-delimited text file for use in other applications. (See "Working with Proximity Tables" (p. 86) for additional information about NSTs and their use in Proximity.)

Exporting NST data to tab-delimited text files

The following steps require that you be serving a database using the MonetDB server and running the Proximity Database Browser. See Exercise 4.1 for information on starting and using the Proximity Database Browser.

1. From the Proximity Database Browser start page, click **browse tables**. Proximity opens the NST browser.
2. Select and display the target NST, drilling down to the appropriate data NST as needed.
3. From the **File** menu, choose **Export to File.** Proximity displays the Open dialog.
4. Navigate to the target output directory and enter a file name for the exported data in the Save As box. You can only export data to a file on the same machine as that serving the database.
5. Click **Save**. Proximity saves the exported data in the specified file.

Deleting Proximity Databases

Proximity uses a MonetDB database to store its data. To delete a MonetDB database, make sure that you are not serving that database and delete the database files. Database files are located in

Linux/Mac OS X:

`/usr/local/Monet-mars/var/MonetDB4/dbfarm/`*`name`*

where *`name`* is the name of the database.

Windows:

`C:\\Documents and Settings\`*`username`*`\Application Data\MonetDB4\dbfarm\`*`name`*

where *`username`* is the login name for the current user and *`name`* is the name of the database.

This permanently deletes the database.

Tips and Reminders

General

- All imported and exported data files must reside on the same machine as that serving the database.
- You must initialize a database using **db-util** before importing data.
- Proximity automatically converts all attribute names to lower case when importing data; attribute values retain their original case.
- Use specialized export functionality available through the Proximity Database Browser to export attribute or NST data to tab-delimited text files.
- Delete a database by deleting its files in the `dbfarm` directory in your local MonetDB installation.

XML import and export

- The DTD, for the XML data format `prox3db.dtd`, must reside in the same directory as the import data file. Copy the DTD file from `$PROX_HOME/resources/` to the directory containing the data file before importing data.
- For compatibility with MonetDB, single quotes, double quotes, and newline characters in the XML data are automatically changed to underscores during import.
- The XML data format and associated utilities provide limited error checking against some data

corruption problems by restricting the types of data that can be imported into an exiting database.

- To override the default restrictions on importing XML data into existing databases, set the import script `noChecks` parameter to true.
- Exporting a container to XML exports all the container's subgraphs, any attributes on those subgraphs, and any containers nested within the specified container, but not the container's attributes.

Plain text import and export

- The plain text data format and associated utilities provide no error checking; you are completely responsible for the consistency and integrity of your imported data.
- For compatibility with MonetDB, convert single quotes, double quotes, and newline characters to underscores in a pre-processing step before importing data using the plain text format.

Additional Information

- See Appendix C, *Proximity XML Format* (p. 133) for a description of the XML data format.
- See Appendix D, *Proximity Text Data Format* (p. 143) for a description of the plain text data format.

Chapter 4. Exploring Data

Overview

The Proximity Database Browser is a browser-based interface that lets you explore database entities, create and execute queries, examine models, and run Python scripts that operate on the database.

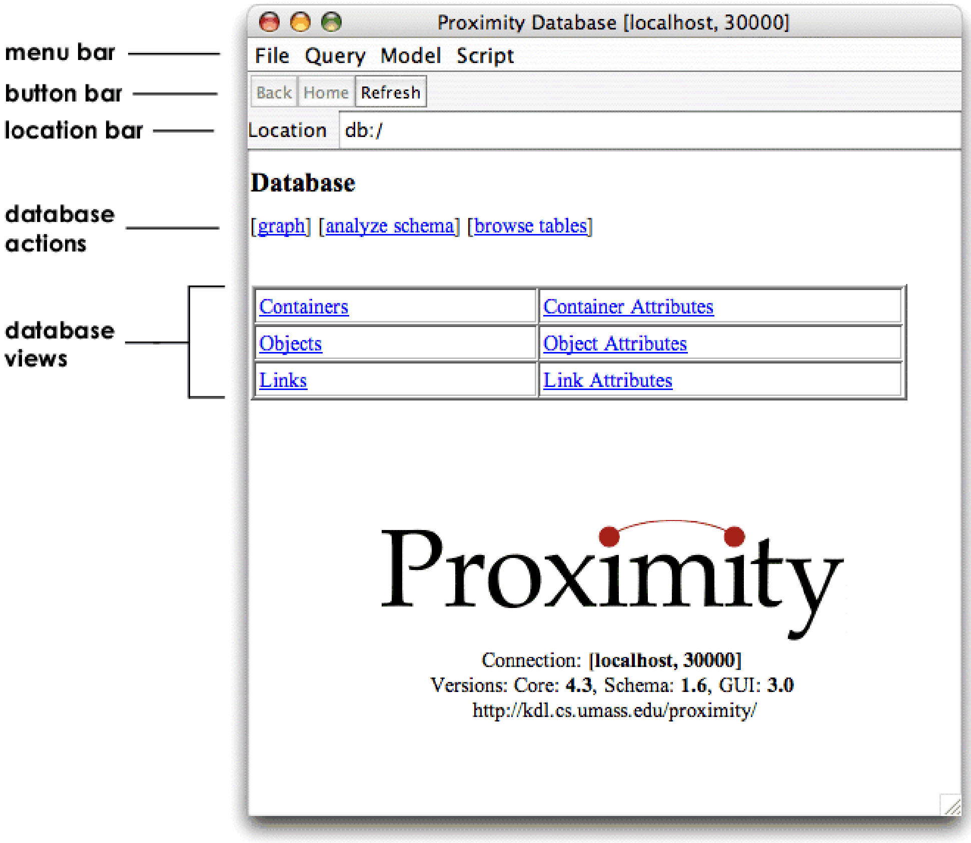
Objectives

The text and exercises in this chapter demonstrate how to

- start and use the Proximity Database Browser
- explore objects, links, and attributes in a Proximity database
- visualize portions of the database as a graph
- set display preferences for the Proximity Database Browser
- create a schema analysis report

The Proximity User Interface

The Proximity Database Browser operates much like a web browser. Most of its functionality is accessed through standard linked (blue and underlined) text items.



Location db:/

Database

[graph] [analyze schema] [browse tables]

Containers	Container Attributes
Objects	Object Attributes
Links	Link Attributes

Connection: **[localhost, 30000]**
Versions: Core: **4.3**, Schema: **1.6**, GUI: **3.0**
http://kdl.cs.umass.edu/proximity/

The home page includes a button bar, menu bar, a list of database actions, and set of specific views into the database.

The menu bar provides access to browser settings and Proximity's query processing, script handling functionality, and model display functionality:

- The **File** menu controls browser settings and lets you view the contents of text files, such as queries or scripts.
- The **Query** menu lets you create, edit, and execute queries (see Chapter 5, *Querying the Database*).
- The **Model** menu lets you display graphical representations of Proximity models (see Chapter 7, *Learning Models*).
- The **Script** menu lets you execute Python commands interactively and in scripts (see Chapter 6, *Using Scripts*).

The button bar provides access to navigation shortcuts and general browser functionality:

- The **Back** button returns the browser to the previous page.
- The **Home** button returns the browser to the Proximity Database Browser start page.
- The **Refresh** button updates the current display to include any new information.

The location bar provides direct access to any Proximity database entity using a URL-like syntax described in "Using the Location Bar" (p. 35).

The database actions list provides access to database-level functionality:

- The **graph** action opens the graphical data browser, which lets you interactively explore the graph structure of the database. See "Visualizing Data" (p. 36).
- The **analyze schema** action opens the schema analysis dialog to begin the process of analyzing the distribution of object and link types in the database. See "Analyzing the Database Schema" (p. 41).
- The **browse tables** action opens the Proximity NST browser, a tool for examining Proximity's internal data structures. NSTs are discussed in Chapter 6, *Using Scripts*. See "Working with Proximity Tables" (p. 86).

The database views section lets you explore the entities in your database. Objects and links are primary database entities. Exploring these primary database entities is the focus if this chapter. Containers are created as a result of executing queries. Containers and subgraphs, and how to examine them in the Proximity Database Browser, are described in more detail in Chapter 5, *Querying the Database*.

Exploring Objects and Links

This exercise describes how to start the Proximity Database Browser and walks through an exploration of several objects and links in the ProxWebKB database.

Exercise 4.1. Exploring objects and links:

1. If it is not already running, start the MonetDB server on the ProxWebKB database.

   ```
   > Mserver --dbname ProxWebKB $PROX_HOME/resources/init-mserver.mil
   ```

   The `init-mserver.mil` script sets the port for the server to 30000. To use a different port, add **`--set port=`*****`nnnnn`*** (where *nnnnn* is the new port number) to the command line, e.g.,

   ```
   > Mserver --dbname ProxWebKB $PROX_HOME/resources/init-mserver.mil \
     --set port=45678
   ```

   Remember to use a port number > 40000 if you are using MonetDB 4.6.2.

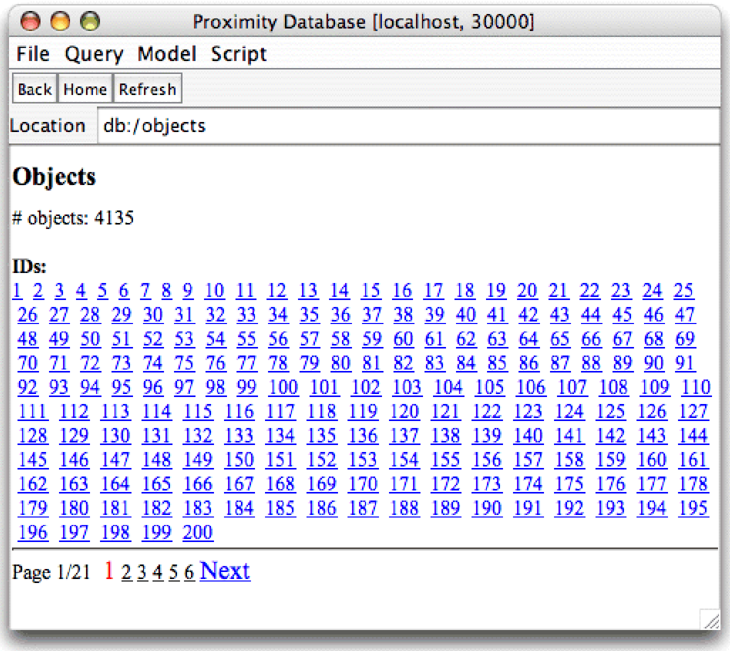
2. Start the Proximity Database Browser. Substitute the appropriate host and port information if you are running the MonetDB server on a different machine or are using a different port.

   ```
   > cd $PROX_HOME
   > bin/gui.sh localhost:30000
   ```

   Proximity starts the Proximity Database Browser. The Proximity interactive Python interpreter opens in a separate window. See “Using the Proximity Python Interpreter” (p. 75) for information on using the interpreter to interactively work with the data. You can close the interpreter window when you are not using it.

To exit the Proximity Database Browser, choose **Quit** from the **File menu**.

3. In the Proximity Database Browser home page, click **Objects**. Proximity displays the total number of objects in the database and lists the IDs for the first 200 objects. The ProxWebKB database includes 4135 objects.



Location db:/objects

Objects

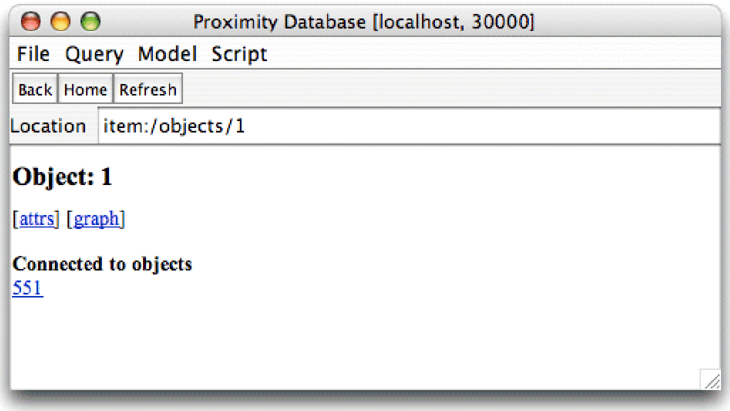
objects: 4135

IDs:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161
162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178
179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195
196 197 198 199 200

Page 1/21 1 2 3 4 5 6 Next

Displaying long lists can take a considerable amount of time; therefore, Proximity displays 200 items at a time. To display more objects, click **Next** to see the next 200 objects, or click the corresponding page number to go to that portion of the object list. For example, clicking **4** displays objects 601 through 800.

4. Click **1** to see the details for object 1. Proximity displays information about object 1.



Location item:/objects/1

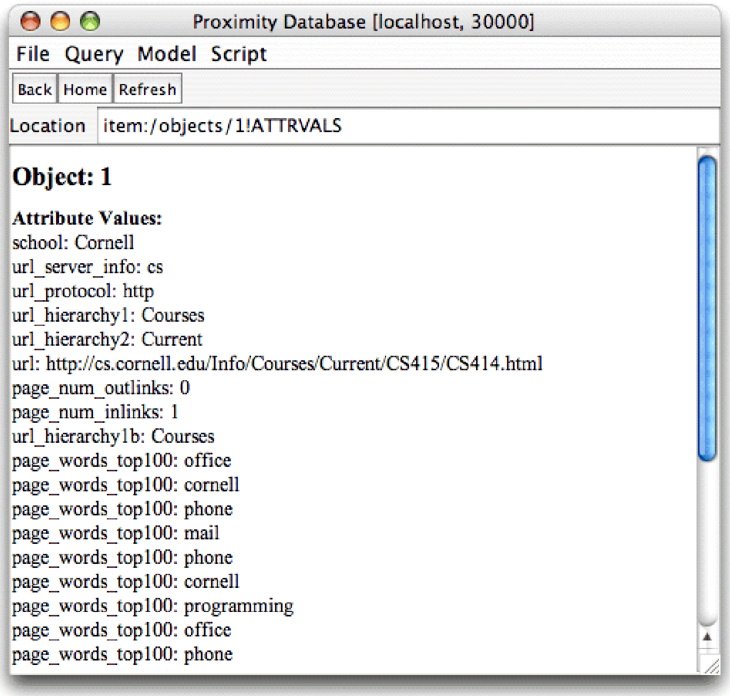
Object: 1

[attrs] [graph]

Connected to objects
551

Because MonetDB optimizes column-centric operations at the expense of row-centric operations, it can be slow to collect the attributes and values for a specific object or link in databases containing a large number of attributes. Proximity therefore separates the display of attribute information from other information about objects and links.

5. Click **attrs**. Proximity displays a list of the attributes and their values for object 1. (The order in which attributes are listed may differ from that shown below.)

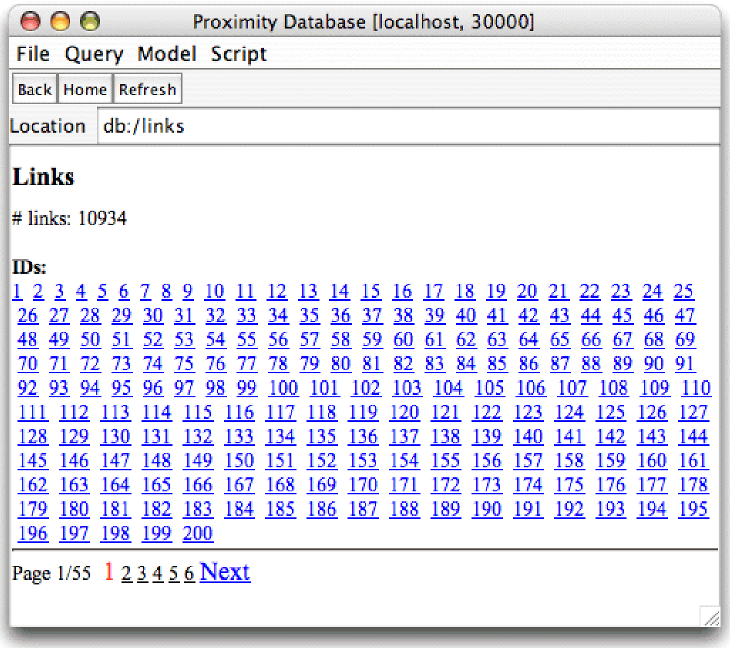

Proximity Database [localhost, 30000]
File Query Model Script
Back Home Refresh
Location item:/objects/1!ATTRVALS

Object: 1

Attribute Values:
school: Cornell
url_server_info: cs
url_protocol: http
url_hierarchy1: Courses
url_hierarchy2: Current
url: http://cs.cornell.edu/Info/Courses/Current/CS415/CS414.html
page_num_outlinks: 0
page_num_inlinks: 1
url_hierarchy1b: Courses
page_words_top100: office
page_words_top100: cornell
page_words_top100: phone
page_words_top100: mail
page_words_top100: phone
page_words_top100: cornell
page_words_top100: programming
page_words_top100: office
page_words_top100: phone

Proximity supports set-based attribute values allowing objects and links to have multiple values, including duplicated values, for an attribute. Object 1 has multiple values for the page_words_top100 attribute. This attribute represents occurrences of the top 100 words found in the web pages used to create the ProxWebKB database. Individual words can occur more than once in this list if they occur more than once in the source web page.

6. Click **Home** to return to the Proximity Database Browser start page.
7. Click **Links**. Proximity displays the total number of links in the database and lists the IDs of the first 200 links. The ProxWebKB database contains 10,934 links.



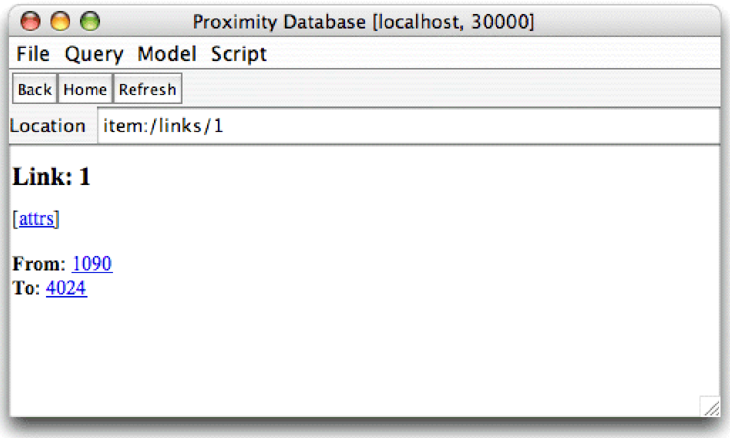
Links

links: 10934

IDs:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161
162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178
179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195
196 197 198 199 200

Page 1/55 1 2 3 4 5 6 Next

8. Click **1** to see the details for link 1. Proximity displays summary information about link 1 including the IDs of the objects it connects.

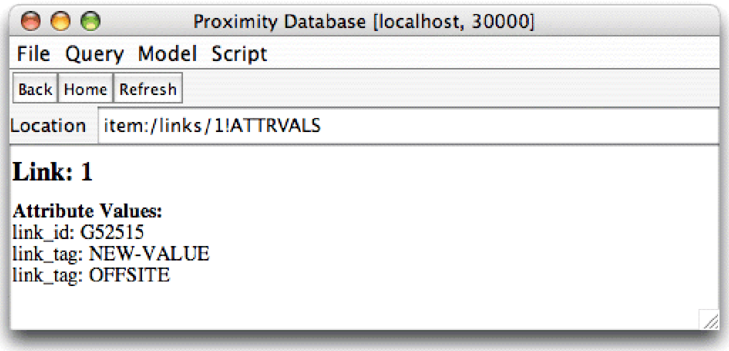


Link: 1

[attrs]

From: 1090
To: 4024

9. Click **attrs**. Proximity displays a list of attributes and their values for link 1.


Proximity Database [localhost, 30000]
File Query Model Script
Back Home Refresh
Location item:/links/1!ATTRVALS

Link: 1

Attribute Values:
link_id: G52515
link_tag: NEW-VALUE
link_tag: OFFSITE

In Proximity, links are database entities on a par with objects. Like objects, links have IDs and set-based attribute values. Because we imported additional values for the link_tag attribute in Exercise 3.3, some links, including link 1, have more than one value for this attribute.

10. Continue exploring the ProxWebKB database. The remaining exercises in this tutorial use the ProxWebKB database, so understanding the data it represents can be helpful as you work through the remaining exercises. When you are finished, continue to the next section.

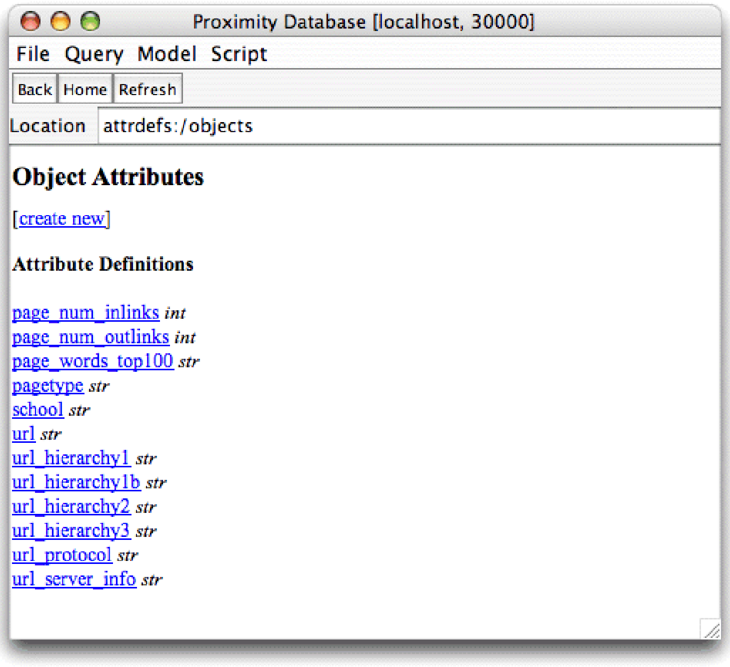
Exploring Attributes

In addition to looking at at database in terms of its objects and links, Proximity allows you to approach it in terms of its attributes. Proximity separates object attributes from link attributes, but otherwise both support the same operations.

Exercise 4.2. Exploring attributes:

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. Click **Home** to go to the Proximity Database Browser start page.
2. Click **Object Attributes**. Proximity displays a list of the object attributes in the ProxWebKB database.



attrdefs:/objects

Object Attributes

[create new]

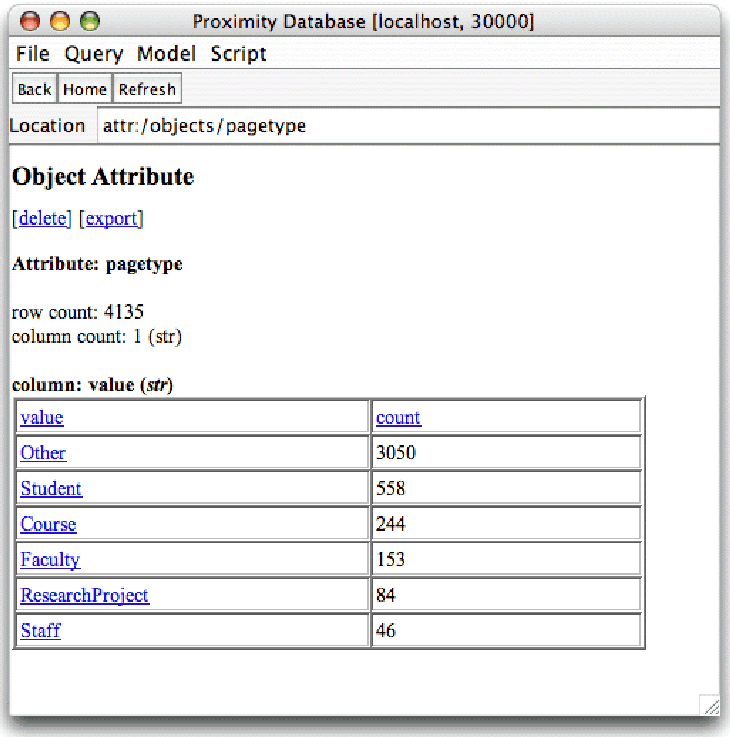
Attribute Definitions

page_num_inlinks *int*
page_num_outlinks *int*
page_words_top100 *str*
pagetype *str*
school *str*
url *str*
url_hierarchy1 *str*
url_hierarchy1b *str*
url_hierarchy2 *str*
url_hierarchy3 *str*
url_protocol *str*
url_server_info *str*

This list shows all the attributes that an object may have. Each object need not have a value for every attribute.

The Object Attributes page includes a **create new** link for adding a new object attribute to the database. Clicking **create new** opens the Create New Attribute dialog, allowing you to specify the new attribute name and a function for assigning values. See “Adding a New Attribute” (p. 81) for an example of a function that assigns values to the new attribute; the syntax for this function is further described in the Javadoc documentation for the `AddAttribute` class.

3. Click **pagetype**. Proximity displays a histogram (table) showing the values and counts for the pagetype attribute. The table is initially sorted by count, with the highest count shown first. Click **value** to sort the table by attribute value.


Proximity Database [localhost, 30000]
File Query Model Script
Back Home Refresh
Location attr:/objects/pagetype

Object Attribute

[delete] [export]

Attribute: pagetype

row count: 4135
column count: 1 (str)

column: value (*str*)

value	count
Other	3050
Student	558
Course	244
Faculty	153
ResearchProject	84
Staff	46

Row count indicates how many instances of this attribute occur in the database. Because each object in ProxWebKB has a pagetype attribute, there are 4135 instances of the pagetype attribute in the database. Because Proximity supports set-valued attributes, some attributes such as page_words_top100 may have a higher row count, indicating that some objects have more than one value for that attribute.

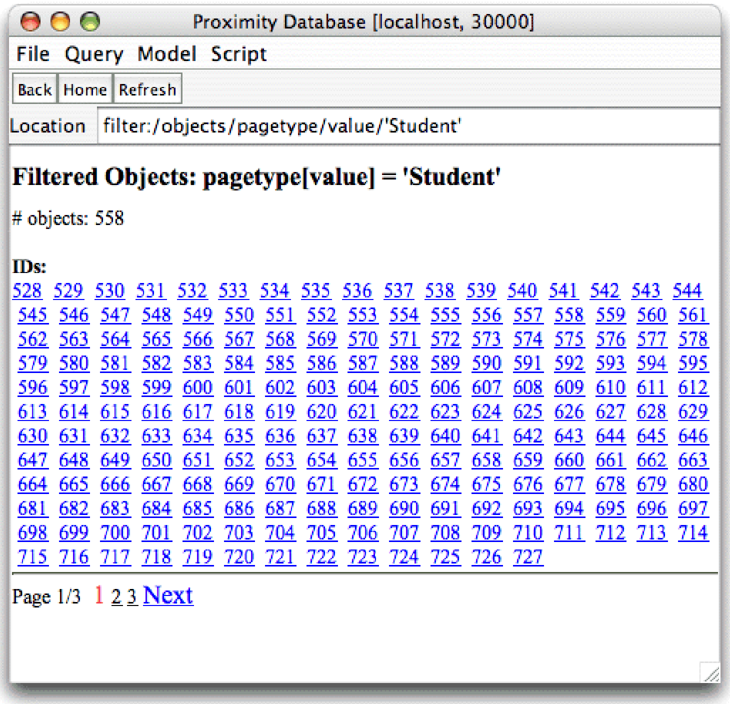
Column count indicates how many dimensions a single value contains. In the ProxWebKB database, all original attributes have just a single dimension, although attributes that store model predictions (added by completing the exercises in Chapter 7) employ two dimensions. Other databases might have multi-dimensional attributes, such as an attribute containing an (x,y) pair representing spatial coordinates. Proximity can store multi-dimensional data, but it cannot currently use multi-dimensional attribute values in queries.

Object attribute pages also include **delete** and **export** links:

- Clicking **delete** removes this attribute and all its values from the database. (If you want to experiment with removing attributes, take care not to remove the pagetype attribute as it is needed in later exercises. You can safely remove the url_hierarchy2 or url_hierarchy3 attributes without affecting later *Tutorial* exercises.)
- Clicking **export** writes the object or link IDs and attribute values to a tab-delimited text file suitable for importing into another application.

The attribute list is initially sorted by count, with the most numerous value shown first. Click **value** to sort the list alphabetically.

4. Click **Student**. Proximity displays a list of objects whose pagetype attribute has the value Student. Said differently, Proximity *filters* the list of objects by the selected attribute value.



filter:/objects/pagetype/value/'Student'

Filtered Objects: pagetype[value] = 'Student'

objects: 558

IDs:

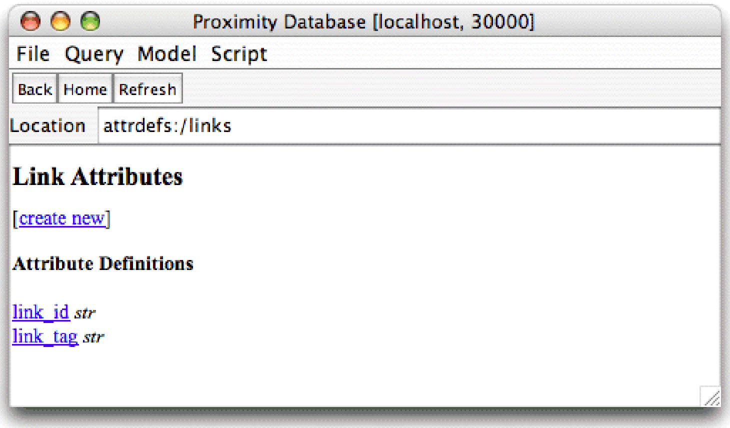
528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544
545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561
562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578
579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595
596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612
613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629
630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646
647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663
664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680
681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697
698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714
715 716 717 718 719 720 721 722 723 724 725 726 727

Page 1/3 1 2 3 Next

5. Click **Back** to return to the Object Attributes page.
6. Click **page_words_top100**. Proximity displays summary information for this attribute.

   Because ProxWebKB objects have multiple values for this attribute, its row count is much higher than the number of objects in the database.

7. Click **Home** to return to the Proximity Database Browser start page.
8. Click **Link Attributes**. Proximity displays a list of the link attributes in the ProxWebKB database. Links in ProxWebKB have two attributes, link_id and link_tag.



attrdefs:/links

Link Attributes

[create new]

Attribute Definitions

link_id *str*
link_tag *str*

The Link Attributes page also includes a **create new** link for adding a new link attribute to the database. Clicking **create new** opens the Create New Attribute dialog, allowing you to specify the new attribute name and a function for assigning values. See "Adding a New Attribute" (p. 81) for an example of a function that assigns values to the new attribute; the syntax for this function is further described in the Javadoc documentation for the `AddAttribute` class.

9. Explore the link attributes. Every link in ProxWebKB has a value for each of these attributes: a

link_tag that indicates the relationship of one web page to another in a file hierarchy, and a link_id that provides a unique identifier for each link. As with other long pages, the link_id page only displays the first 200 entries.

Like object attribute pages, link attribute pages include a **delete** link removes that attribute and all its values from the database, and an **export** link that exports the attribute values to a tab-delimited text file suitable for importing into another application.

10. Continue exploring the ProxWebKB database. When you are finished, continue to the next section.

Using the Location Bar

The Proximity Database Browser location bar uses a URL-like path to provide direct access to any Proximity database entity. The following exercise illustrates how to use the location bar to access the different types of Proximity database entities (objects, links, and attributes). Using the location bar to access containers and subgraphs is described in “Exploring Containers and Subgraphs” (p. 50). See Appendix A, *Proximity Quick Reference* for a summary of how to use the location bar.

Exercise 4.3. Using the location bar:

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. Click **Home** to go to the Proximity Database Browser start page.
2. In the location bar at the top of the Proximity Database Browser, type **`db:/objects`** and press Return.

Location db:/objects

Proximity reports the total number of objects in the database and lists the IDs for the first 200 objects.

The leftmost portion of the location bar path (to the left of the colon) can be loosely viewed as a protocol describing the type of information to be displayed. Use `db:/` as the path protocol to access the list of objects or the list of links. See “Location Bar Path Syntax” (p. 125) for a complete list of location bar path protocols.

3. Type **`db:/objects#2`** in the location bar and press Return. Proximity displays the second page of object IDs.

When Proximity breaks up a list over multiple pages, follow the location bar path with the pound symbol (#) and a page number to display the corresponding page of items.

4. Enter **`item:/objects/1`** in the location bar. Proximity displays information about object 1.

Note that the entire path has changed so that it no longer begins with the `db:` protocol. Use the `item:` protocol to access individual objects or links.

5. Enter **`item:/objects/1!ATTRVALS`** in the location bar. Proximity displays the attributes and their values for object 1.

Add `!ATTRVALS` to the end of the location bar path to display the attribute list for any database entity that has attributes (objects, links, containers, and subgraphs).

6. Enter **`db:/`** in the location bar. Proximity returns to the Proximity Database Browser start page.

With the exception of `db:/` to indicate the Proximity Database Browser start page, location bar paths do not use a trailing slash (`/`).

7. Enter **`db:/links`** in the location bar. Proximity displays the list of link IDs.
8. Enter **`item:/links/1`** in the location bar. Proximity displays information about link 1. Again, the path protocol changes from `db:/` to `item:/`.

9. Enter **`item:/links/1!ATTRVALS`** in the location bar. Proximity displays the attributes and their values for link 1.

   As we saw before, adding `!ATTRVALS` to the end of the location bar path displays the attribute values for the corresponding database item.

10. Enter **`attrdefs:/objects`** in the location bar. Proximity displays the list of object attributes.

    Use the `attrdefs:` protocol to display either the list of object attributes or the list of link attributes.

11. Enter **`attr:/objects/url`** in the location bar. Proximity displays the table of values and counts for the url object attribute.

    As we saw with the change from viewing a list of objects to viewing a single object, changing from displaying a list of attributes to displaying a single attribute requires changing the path protocol. Use the `attr:` protocol to display information on a specific attribute.

    Because the url attribute has over 200 distinct values in the ProxWebKB database, Proximity breaks up the list of values across several pages. Add # and a page number to the end of the location bar path to display the corresponding page of attribute values. For example, to see the second page of values for the url attribute, enter **`attr:/objects/url#2`** in the location bar.

12. Enter **`filter:/objects/pagetype/value/'Student'`** in the location bar. Proximity displays the list of objects filtered by the specified attribute value. The text entered in the location bar states that we only want to see those objects whose pagetype attribute has a value of Student.

    The `filter:` protocol filters objects and links by the value of an attribute. The syntax of the `filter:` protocol is:

    `filter:`*`/item-type/attribute-name/column-name/attribute-value`*

    where

    - *`item-type`* is either `objects` or `links`
    - *`attribute-name`* is the name of the attribute to use
    - *`column-name`* is the name of the data column for that attribute. Most Proximity attribute data, including the data in ProxWebKB, is one-dimensional and thus requires only a single column. Unless you have specified a different column name when importing data, this column will have the default name `value`. Column names are shown above the histogram (table of values and counts) for an attribute.
    - *`attribute-value`* is the specific value to filter on. Only database items having this value will be displayed. If the attribute is of type `str` (string), the value must be quoted.

Visualizing Data

To complement the exploration capabilities available in the Proximity Database Browser, Proximity also provides a way to visualize data graphically. The graphical data browser lets you interactively explore and visualize the local neighborhood of a selected object.

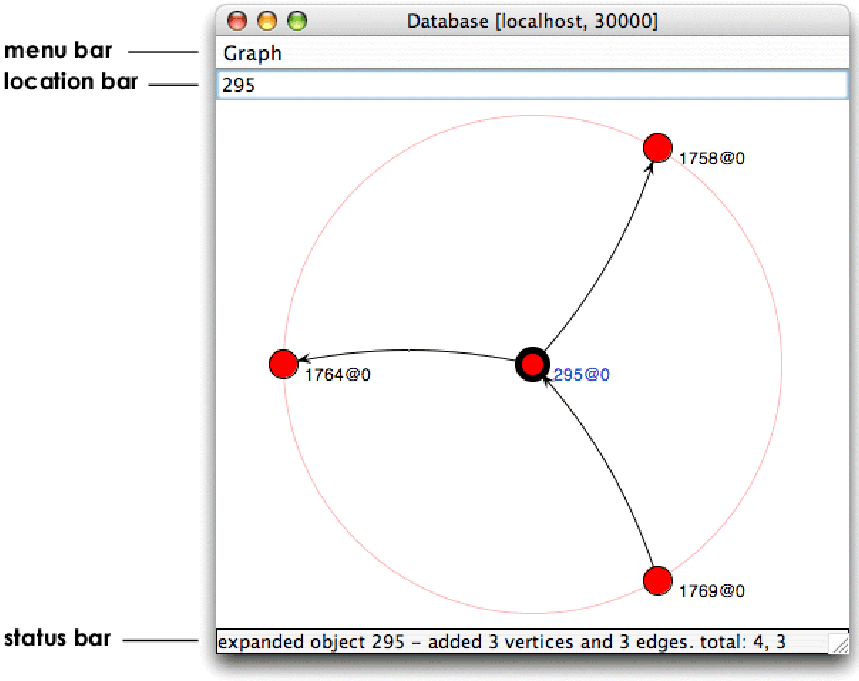
Exercise 4.4. Exploring data with the graphical data browser:

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. Display the details for object 295.

   From the Proximity Database Browser start page, either click **Objects** then click **Next** and choose object **295**, or enter **`item:/objects/295`** in the location bar.

2. Click **graph**. Proximity opens a new window displaying a graph of object 295 and its immediate neighbors.



expanded object 295 – added 3 vertices and 3 edges. total: 4, 3

The initial graph display shows the *expansion* of the start object to show the object and its immediate neighbors, labeled with their object IDs. In this example, the start object, object 295, is linked to three neighbors: objects 1758, 1764, and 1769.

The graph visualization window includes its own location bar at the top of the window and a status bar at the bottom of the window. Enter an object's ID to visualize the immediate neighborhood around that object. The location bar displays the ID of the most recently expanded object—in this example it indicates that we are displaying the local neighborhood of object 295.

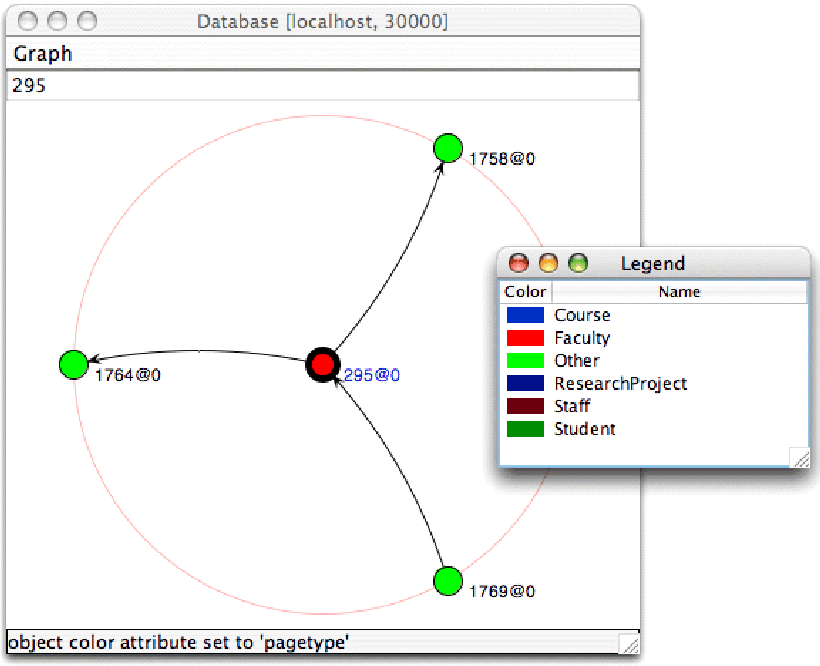
The status bar reports the results of the most recent expansion—that expanding object 295 adds three new objects and three new links for a total of four objects and three links.

3. Set the display preferences for the graph. Display options determine

   - labels for nodes in the graph
   - labels for links in the graph
   - the color of nodes in the graph
   - whether to use of animation when updating the graph (enabled by default)

   Labels and colors are based on the attribute values and OIDs of the displayed items:

   a. From the **Graph** menu, choose **Set Object Colors**. Proximity opens the Choose Attribute dialog.
   b. Choose **pagetype** from the list of attributes and click **Set**. Proximity updates the node colors in the graph.
   c. From the **Graph** menu, choose **Show Color Legend**. Proximity displays a key that illustrates how node color maps to pagetype.
   d. To change the graph's object labels, choose **Set Object Labels** from the **Graph** menu. You can label the objects with the object's OID, label objects with the value of a selected attribute, or choose to use no object labels.



295

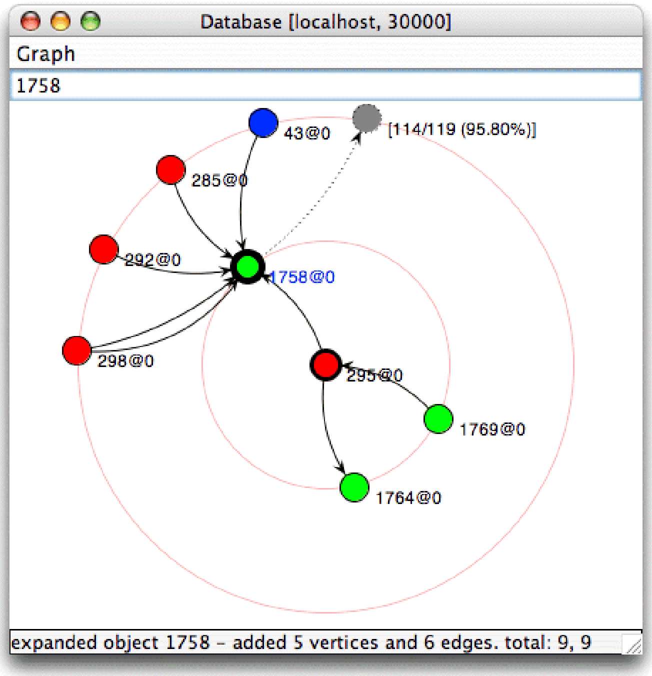
Color	Name
🟦	Course
🟥	Faculty
🟩	Other
🟦	ResearchProject
🟫	Staff
🟩	Student

object color attribute set to 'pagetype'

Although we do not do so as part of this exercise, you can also add and remove labels for the links in the graph. Like node labels, link labels can use either the link's ID number or the value of a selected attribute.

4. Click the node for object 1758.

   Proximity expands this node to show you some of the objects linked to object 1758.



1758

expanded object 1758 - added 5 vertices and 6 edges. total: 9, 9

Notice that the location bar has been updated to show you the most recently expanded object.

Nodes that have previously been expanded are marked by a medium-weight black outline. The most recently expanded node is marked by a thick black outline and a blue label.

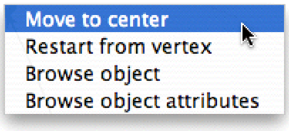
One of the newly added nodes is a *proxy* node. Proxy nodes are colored gray and labeled with a percentage that indicates how many of the neighboring nodes it represents. In this case, object 1758 is linked to 119 neighboring objects, of which 5 are currently shown in the graphing window. (The 5 linked objects include the 4 objects on the outside circle and object 295 in the center of the graph.) An additional 114 objects linked to object 1758 are not shown, so the proxy node reports that 95.80% (or 114 out of 119) of object 1758's neighbors are not displayed.

5. To show more objects linked to object 1758, click the gray proxy node on the outside circle.

   Proximity displays six additional objects linked to object 1758. Because Proximity redraws the graph after each expansion, nodes and links may change position after you expand an object or proxy node.

   Expanding either an object or proxy node normally adds up to six additional nodes to the graph. New nodes are selected from the remaining (undisplayed) objects linked to the expanded node. Expanding a proxy node that represents more than six objects produces a display containing another proxy node representing the remaining unexpanded objects. You can continue clicking each new proxy node to display additional neighboring objects.

6. Right-click (Ctrl-click for Mac OS X) the node for object 1758. Proximity displays a context menu for the selected node. Choose **Move to center**.

   Proximity redraws the graph, placing the selected node in the center. Re-centering a graph can make it easier to understand the immediate neighborhood around an object. In this example, by placing object 1758 in the center, we can more easily see that object 295 is part of object 1758's immediate neighborhood, and that it has the same pagetype as object 1758's other neighbors.

   After a couple of expansions to the proxy node representing object 1758's unshown neighbors, the graph shows that the web page represented by object 1758 is linked to a large number of faculty and research project pages. We might therefore suspect that object 1758 represents some type of main page or directory listing. We test this suspicion in the next step.

   Tip: If the graph display becomes crowded, hover the mouse over any node or link to display a tool tip containing that item's OID.

7. Right-click the node for object 1758 and choose **Browse object attributes** from the context menu.

   Proximity opens a new Proximity Database Browser window that shows the attribute values for object 1758. The URL for this page contains only the domain, suggesting that object 1758 indeed represents a home page for a computer science department.

   The context menu also provides a **Browse object** menu option. Choosing **Browse object** opens a new Proximity Database Browser window that displays the main object page (i.e., the list of linked objects) for the selected object.

8. Continue exploring the data with the graphical data browser. To explore a new object's neighborhood, enter the object's ID number in the graphical data browser's location bar. Alternately, if the new object is currently visible in the graphical data browser, you can right-click the new object and choose **Restart from vertex**. When you are finished, continue to the next section.

You can also start the graphical data browser from the Proximity Database Browser start page:

1. Click **Home** to return to the Proximity Database Browser start page.
2. Click **graph**. Proximity displays the graphical data browser, but without graphing any object’s neighborhood.
3. Type the number of the object to explore in the graphical data browser’s location bar and press Return. Proximity displays the object and its immediate neighborhood.

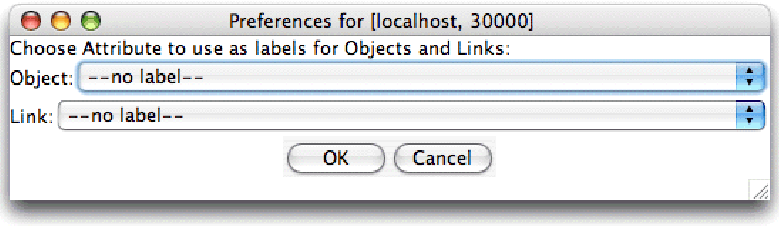
Setting Display Preferences

Proximity lets you customize the labels used to display objects and links in the Proximity Database Browser. You can use the value of any object or link attribute as a label for the corresponding type of database element. Labels are set globally and apply to all objects or links.

Exercise 4.5. Customizing object and link labels:

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. From the **File** menu, choose **Preferences**. Proximity displays the Preferences dialog.



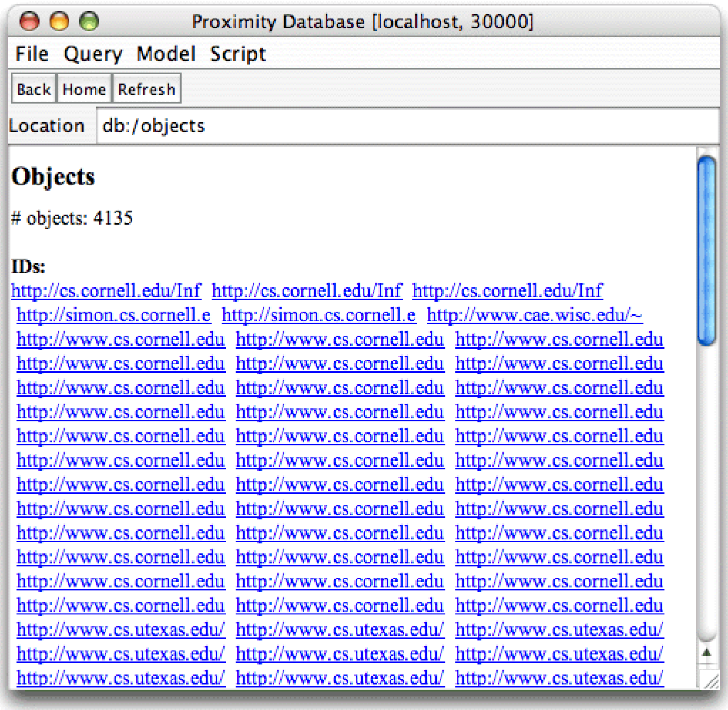
Preferences for [localhost, 30000]

Choose Attribute to use as labels for Objects and Links:

Object: --no label--

Link: --no label--

2. In the Object list, choose **url**. In general, you should choose an attribute that has a unique value for each object.
3. In the Link list, choose **link_id**. Again, you should generally choose an attribute that has a unique value for each link.
4. Click **OK**. Proximity closes the Preferences dialog.
5. Click **Objects**. Proximity displays the first 200 objects. Because you set the display preferences to use the value of the url attribute as the object label, Proximity shows you the URLs for these objects instead of their ID numbers.



Location db:/objects

Objects

objects: 4135

IDs:

http://cs.cornell.edu/Inf http://cs.cornell.edu/Inf http://cs.cornell.edu/Inf
http://simon.cs.cornell.e http://simon.cs.cornell.e http://www.cae.wisc.edu/~
http://www.cs.cornell.edu http://www.cs.cornell.edu http://www.cs.cornell.edu
http://www.cs.cornell.edu http://www.cs.cornell.edu http://www.cs.cornell.edu
http://www.cs.cornell.edu http://www.cs.cornell.edu http://www.cs.cornell.edu
http://www.cs.cornell.edu http://www.cs.cornell.edu http://www.cs.cornell.edu
http://www.cs.cornell.edu http://www.cs.cornell.edu http://www.cs.cornell.edu
http://www.cs.cornell.edu http://www.cs.cornell.edu http://www.cs.cornell.edu
http://www.cs.cornell.edu http://www.cs.cornell.edu http://www.cs.cornell.edu
http://www.cs.cornell.edu http://www.cs.cornell.edu http://www.cs.cornell.edu
http://www.cs.cornell.edu http://www.cs.cornell.edu http://www.cs.cornell.edu
http://www.cs.cornell.edu http://www.cs.cornell.edu http://www.cs.cornell.edu
http://www.cs.cornell.edu http://www.cs.cornell.edu http://www.cs.cornell.edu
http://www.cs.cornell.edu http://www.cs.cornell.edu http://www.cs.cornell.edu
http://www.cs.utexas.edu/ http://www.cs.utexas.edu/ http://www.cs.utexas.edu/
http://www.cs.utexas.edu/ http://www.cs.utexas.edu/ http://www.cs.utexas.edu/
http://www.cs.utexas.edu/ http://www.cs.utexas.edu/ http://www.cs.utexas.edu/

Proximity limits the display of object and link labels to 25 characters, making this feature of limited usefulness for this specific database. Future Proximity development may allow users to change this character limit.

6. Click **Home** to return to the Proximity Database Browser start page.
7. Click **Links**. Proximity displays the first 200 links. Because you set the display preferences to use the value of the link_id attribute as the link label, Proximity shows you these values instead of their Proximity ID numbers.
8. Because these new labels do not provide any additional useful information, change the display preferences back to `no label` for both objects and links. The remaining tutorial examples continue to use Proximity IDs to label objects and links in the Proximity Database Browser.

Analyzing the Database Schema

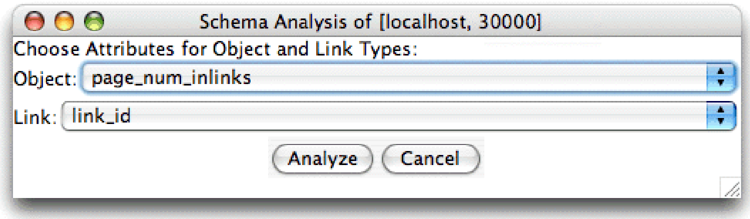
Schema analysis provides summary information about the objects and links in a database. To run a schema analysis, you need to tell Proximity which attributes identify the *type* of an object or link. Proximity then analyzes the database to determine the relationships among each type of object or link (as identified by these attributes) and other database entities.

Because MonetDB maximizes performance for joins at the expense of row-centric operations, schema analysis can be slow. Therefore Proximity performs schema analysis in a separate thread.

Exercise 4.6. Exploring the database schema:

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. In the database actions list, click **analyze schema**. Proximity displays the Schema Analysis dialog.



Schema Analysis of [localhost, 30000]

Choose Attributes for Object and Link Types:

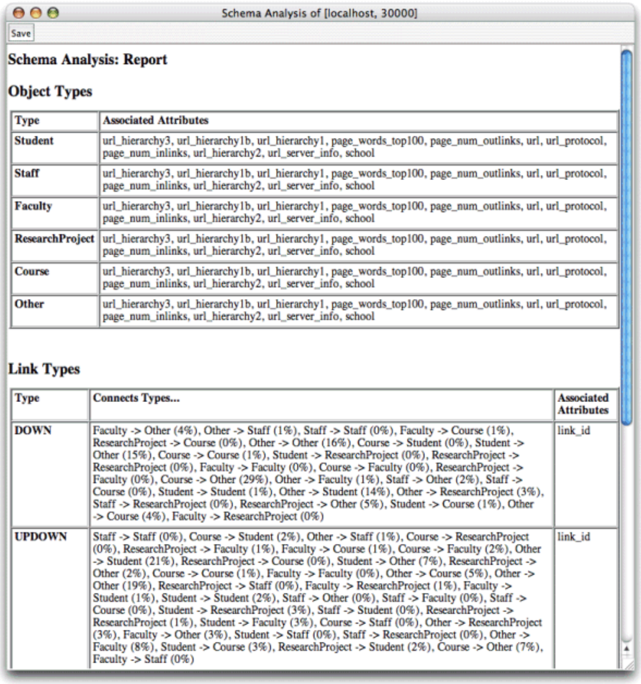
Object: page_num_inlinks

Link: link_id

2. In the Object list, choose **pagetype**.
3. In the Link list, choose **link_tag**.
4. Click **Analyze**. Proximity performs the schema analysis.

   A new window displays the trace of Proximity’s schema analysis processing. Because schema analysis runs in a separate thread, you can continue to use the Proximity Database Browser for other tasks while this process runs.

   When the schema analysis is complete, Proximity displays a schema analysis report. The report shows the other object attributes associated with each type of object (as identified by the object “type” attribute you selected). For links, the report shows both the kinds of objects connected to each link type as well as the other link attributes associated with each type of link.



Schema Analysis of [localhost, 30000]

Schema Analysis: Report

Object Types

Type	Associated Attributes
Student	url_hierarchy3, url_hierarchy1b, url_hierarchy1, page_words_top100, page_num_outlinks, url, url_protocol, page_num_inlinks, url_hierarchy2, url_server_info, school
Staff	url_hierarchy3, url_hierarchy1b, url_hierarchy1, page_words_top100, page_num_outlinks, url, url_protocol, page_num_inlinks, url_hierarchy2, url_server_info, school
Faculty	url_hierarchy3, url_hierarchy1b, url_hierarchy1, page_words_top100, page_num_outlinks, url, url_protocol, page_num_inlinks, url_hierarchy2, url_server_info, school
ResearchProject	url_hierarchy3, url_hierarchy1b, url_hierarchy1, page_words_top100, page_num_outlinks, url, url_protocol, page_num_inlinks, url_hierarchy2, url_server_info, school
Course	url_hierarchy3, url_hierarchy1b, url_hierarchy1, page_words_top100, page_num_outlinks, url, url_protocol, page_num_inlinks, url_hierarchy2, url_server_info, school
Other	url_hierarchy3, url_hierarchy1b, url_hierarchy1, page_words_top100, page_num_outlinks, url, url_protocol, page_num_inlinks, url_hierarchy2, url_server_info, school

Link Types

Type	Connects Types...	Associated Attributes
DOWN	Faculty -> Other (4%), Other -> Staff (1%), Staff -> Staff (0%), Faculty -> Course (1%), ResearchProject -> Course (0%), Other -> Other (16%), Course -> Student (0%), Student -> Other (15%), Course -> Course (1%), Student -> ResearchProject (0%), ResearchProject -> ResearchProject (0%), Faculty -> Faculty (0%), Course -> Faculty (0%), ResearchProject -> Faculty (0%), Course -> Other (29%), Other -> Faculty (1%), Staff -> Other (2%), Staff -> Course (0%), Student -> Student (1%), Other -> Student (14%), Other -> ResearchProject (3%), Staff -> ResearchProject (0%), ResearchProject -> Other (5%), Student -> Course (1%), Other -> Course (4%), Faculty -> ResearchProject (0%)	link_id
UPDOWN	Staff -> Staff (0%), Course -> Student (2%), Other -> Staff (1%), Course -> ResearchProject (0%), ResearchProject -> Faculty (1%), Faculty -> Course (1%), Course -> Faculty (2%), Other -> Student (21%), ResearchProject -> Course (0%), Student -> Other (7%), ResearchProject -> Other (2%), Course -> Course (1%), Faculty -> Faculty (0%), Other -> Course (5%), Other -> Other (19%), ResearchProject -> Staff (0%), Faculty -> ResearchProject (1%), Faculty -> Student (1%), Student -> Student (2%), Staff -> Other (0%), Staff -> Faculty (0%), Staff -> Course (0%), Student -> ResearchProject (3%), Staff -> Student (0%), ResearchProject -> ResearchProject (1%), Student -> Faculty (3%), Course -> Staff (0%), Other -> ResearchProject (3%), Faculty -> Other (3%), Student -> Staff (0%), Staff -> ResearchProject (0%), Other -> Faculty (8%), Student -> Course (3%), ResearchProject -> Student (2%), Course -> Other (7%), Faculty -> Staff (0%)	link_id

5. To save a copy of the schema analysis report, click **Save**. Proximity displays the Open dialog.

   In the Save As box, enter a name for the file. Navigate to the directory in which you want to save the file and click **Save**. Proximity saves the report as an HTML file. To view the report, open the file in a web browser.

6. When you have finished examining the schema analysis report, click the window close button to close the report window.

If you want, you can perform additional analyses using different object and link attributes. Some analyses may be quite slow. In general, the more distinct values an attribute has, the slower the corresponding schema analysis will be.

Tips and Reminders

- Use the location bar for direct access to any Proximity database entity.
- Use the graphical data browser to explore the local neighborhood of an object.
- Create a new attribute interactively by clicking **create new** in the Object Attributes or Link Attribute page.
- Delete an existing attribute interactively by clicking **delete** in the corresponding attribute page.
- Export an attribute’s values to a tab-delimited text file by clicking **export** in the corresponding attribute page.
- Use display preferences (“Setting Display Preferences” (p. 40)) to change the labels used for objects and links in the Proximity Database Browser.
- Proximity attributes are set-valued; an object or link can have multiple values for an attribute.
- Proximity can import and store multi-dimensional attribute values, but it cannot use multi-dimensional attributes in a query.

Additional Information

- See “Exploring Containers and Subgraphs” (p. 50) for information on exploring containers and subgraphs in the Proximity Database Browser.
- See “Location Bar Path Syntax” (p. 125) for a summary of how to use the location bar path to directly access Proximity database entities.
- See “Adding a New Attribute” (p. 81) for additional information on creating new attributes.

Chapter 5. Querying the Database

Overview

Proximity uses QGraph [Blau, Immerman, and Jensen, 2002], a visual query language, for defining queries. This chapter provides information on QGraph's features and walks through the process of creating several example queries. See the *Proximity QGraph Guide* for a full description of the QGraph language as implemented in Proximity.

A QGraph query is a labeled graph of vertices and edges. The query vertices correspond to objects in the database and the query edges correspond to links in the database. QGraph lets you easily describe a specific configuration of objects and links, conditions (required attribute values), and global constraints (restrictions across objects or links). To match a query, a database subgraph must have the correct structure and satisfy all the conditions and constraints.

The result of executing a QGraph query is a collection of matching subgraphs called a *container*. When you execute a query, the container holding the matching subgraphs is added as a persistent item in the Proximity database. Proximity creates an empty container when the query returns no matching subgraphs.

QGraph also provides extensive data update functionality; however, only a portion of this functionality has been implemented in Proximity to date. Proximity implements the ability to add links to the database through the use of QGraph queries.

Proximity represents queries in an XML format. Although you can create queries by writing this XML representation in a text editor, Proximity provides a Query Editor that lets you create queries interactively using a natural graphical representation. The exercises in this chapter use the Query Editor to create queries of increasing complexity, illustrating the range of QGraph's functionality as implemented in Proximity. Details on the query XML file format are included in the *Proximity QGraph Guide* .

Proximity provides many ways to execute queries. This tutorial describes how to execute queries

- from the Query Editor
- from the Proximity Database Browser
- from the command line using Proximity shell scripts

You can also call methods that execute queries from within Proximity Python scripts or Java programs.

All of the queries used in this chapter are available in `$PROX_HOME/doc/user/tutorial/examples`. You can follow the exercises in this chapter to create the queries interactively or execute the example queries provided with the Proximity distribution.

> Exercise 5.7 creates a container used by exercises in later chapters. Make sure that you complete Exercise 5.7 if you plan to complete the exercises in the following chapters.

Objectives

The exercises in this chapter demonstrate how to

- create queries using the Query Editor
- add links to the data by executing queries
- execute queries from the Query Editor, Proximity Database Browser, and command line
- execute a query against the contents of a container
- examine query results (containers and subgraphs)
- create the database entities used in later tutorial exercises

Each exercise in this chapter introduces new QGraph features. A complete explanation of the QGraph query language is available in the *Proximity QGraph Guide* . The “Tips and Reminders” section at the end of this chapter provides a summary of rules for well-formed QGraph queries.

A First Proximity Query

The ProxWebKB database contains objects and links corresponding to web pages and the links between them. Each object has a pagetype attribute that identifies whether the page belongs to or describes a student, faculty member, staff member, research project, course, or something else (other).

The following exercise steps through the process of creating a simple query using the Query Editor. The query finds all the research project pages in the database along with the pages directly linked to that research project page. Each successful match of the query identifies a portion of the database that matches this structure. Because a Proximity database can be represented as a large graph, a match is represented as a subgraph.

Exercise 5.1. Creating a first Proximity query:

The query created in this exercise is also available in the Proximity 4.3 distribution in `$PROX_HOME/doc/user/tutorial/examples/research-clusters1.qg2.xml`.

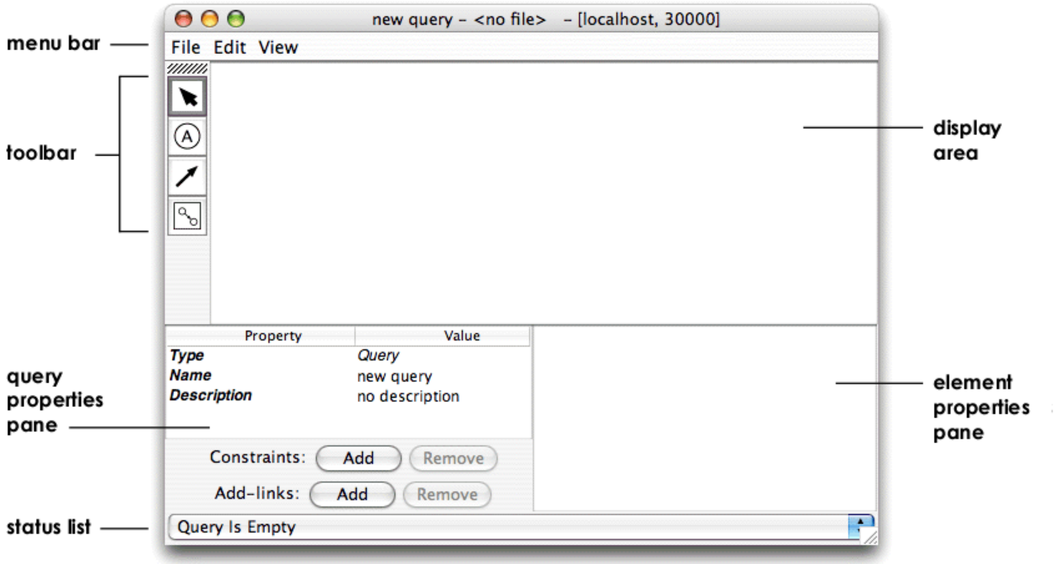
Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. If you plan to execute any of the saved queries in `$PROX_HOME/doc/user/tutorial/examples`, copy `graph-query.dtd` from `$PROX_HOME/resources` to the directory containing the example queries.

```
> cp $PROX_HOME/resources/graph-query.dtd $PROX_HOME/doc/user/tutorial/examples/
```

Proximity queries are represented using an XML format. The DTD that describes this format must be in the directory containing the query file to execute the query.

2. From the **Query** menu, choose **New Query**. Proximity starts the Query Editor.



Property	Value
Type	*Query*
Name	new query
Description	no description

Query Is Empty

The Query Editor includes two properties panes at the bottom of the window. The query properties pane displays properties of the query as a whole. The element properties pane displays the properties of the selected query element. The element properties pane is blank when no query element is selected.

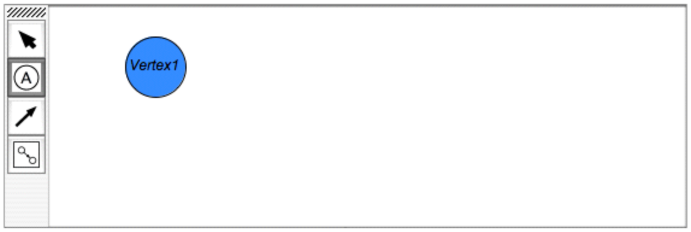
3. Click Ⓐ or press Ctrl-2 to choose the vertex tool. Tool selection is persistent; the vertex tool remains selected until you choose another tool.

Tip: Use keyboard shortcuts to change the Query Editor selection mode. You can open, close, save, and run queries, select tools, and select query elements using keyboard shortcuts. Keyboard shortcuts are shown next to the corresponding command in the Query Editor's menus; a summary of the tool selection shortcuts is shown below:

Ctrl-1	Choose the selection tool
Ctrl-2	Choose the vertex tool
Ctrl-3	Choose the edge tool
Ctrl-4	Choose the subquery tool

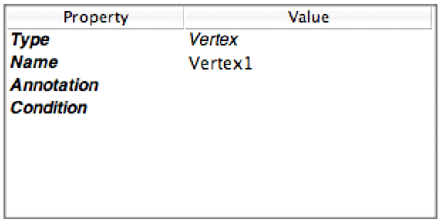
A complete list of keyboard shortcuts for the Query Editor is included in Appendix A, *Proximity Quick Reference*.

Click in the Query Editor display area. The Query Editor creates a new vertex.



Tip: To delete an element from a query in the Query Editor, click ↖ to choose the selection tool, select the element, then press Delete.

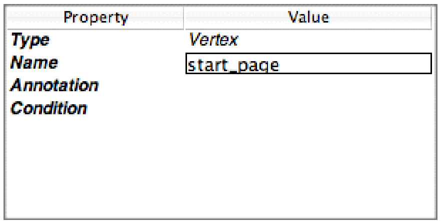
4. Click ↖ or press Ctrl-1 to choose the selection tool. Click the vertex you just created to select it. The Query Editor selects the vertex and displays the vertex's properties in the element properties pane.



Property	Value
Type	*Vertex*
Name	Vertex1
Annotation	
Condition	

Tip: To rearrange a query's layout, use the selection tool to drag vertices inside the display area.

5. In the element properties pane, double-click the vertex name (currently "Vertex1") to edit its value. Enter **`start_page`** and press Tab.



Property	Value
Type	*Vertex*
Name	start_page
Annotation	
Condition	

Vertex and edge names are used for your convenience in creating the query, in understanding and using query results, and for identifying subgraph elements in learning and applying models. Although we may label query elements with names that remind us of specific attribute values, these names have no effect on how query elements match different database elements.

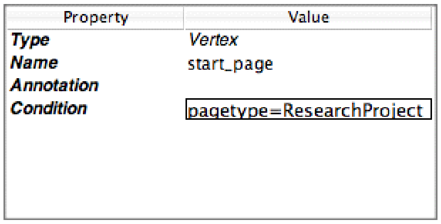
Tip: Double-click a vertex or edge in the Query Editor display area to edit the corresponding label.

Other vertex properties—annotations and conditions—are optional mechanisms for restricting matches in the database. Conditions restrict matches to those entities (objects or links) that match specified attribute values. In this exercise, you specify a condition for one of the vertices. Numeric annotations are covered in the next example.

6. Double-click in the (currently blank) value column of the Condition property to edit its value. Enter

```
pagetype = ResearchProject
```

and press Tab. Attribute names and values containing spaces must be surrounded by single quotes. Spaces surrounding the "=" sign are not significant.



Property	Value
Type	*Vertex*
Name	start_page
Annotation	
Condition	pagetype=ResearchProject

Proximity ignores case when matching attribute *names* in a query's condition or constraint to those in the database. (All attribute names in the database are automatically converted to lower case during import.) Proximity obeys case, however, when matching attribute *values* specified in a query with those in the database.

Proximity supports two types of conditions:

- Attribute value conditions compare the value of an object's or link's attribute with a specified value.
- Existence conditions merely check to see whether an object or link has *any* value for the specified attribute.

Both of these types of conditions are described more formally, below.

The general form of an attribute value condition is

```
attribute operator value
```

where

- `attribute` is the name of an attribute
- `value` is a legal value for `attribute`

- *`operator`* is one of =, <, <=, >, >=, and <>.

A vertex or edge may have at most one condition statement; however, that condition may be complex, that is, it may include boolean combinations of simple conditions. Proximity requires the use of disjunctive normal form in prefix notation for combining simple conditions. For example, to match all research project pages at either Cornell or Wisconsin, enter

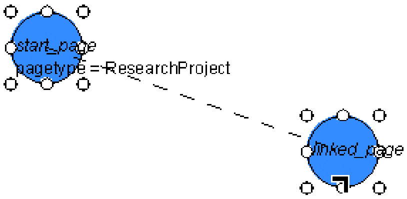
```
AND(pagetype=ResearchProject, OR(school=Cornell,
school=Wisconsin)).
```

Proximity also supports *existence conditions* that require only that an an attribute value be defined for the target object or edge, rather than requiring a specific value or value range. The syntax for an existence condition is

```
exists(attribute)
```

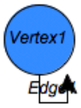
See the *Proximity QGraph Guide* for more information on and examples of attribute and existence conditions.

7. Create a second vertex labeled **`linked_page`**. Because we don't want to restrict the kinds of pages linked to research project pages, do not enter a condition for this vertex.
8. Click ↗ or press Ctrl-3 to choose the edge tool. Drag the mouse from *start_page* to *linked_page*.



The Query Editor creates a new, directed edge connecting the first vertex to the second vertex.

Tip: To create a "loop" edge (an edge connecting a vertex back to itself), choose the edge tool and click the target vertex without dragging.



9. Choose the selection tool and click the new edge to edit its properties. The Query Editor selects the edge and displays the edge's properties in the element properties pane. Enter **`linked_to`** for the edge's label.
10. Double-click in the value column of the edge's Is Directed property to change this value. Enter **`false`** and press Tab. For this example, we don't care about the direction of matching links.

Proximity queries can use either directed or undirected edges. Although Proximity's data model uses only directed links, QGraph lets you use undirected edges in a query when you do not know or do not care about the directionality of a link.

11. In the query properties pane, double-click in the value column of the query's Name property to edit its value (currently "new query"). Enter **`research-clusters1`** and press Tab.
12. Enter a description for the query and press Tab. This query finds pages connected to research project pages.

13. Check the status list at the bottom of the Query Editor window to make sure your completed query is valid.

    Proximity checks to make sure that the query obeys the syntactic requirements of the DTD. No semantic checking is performed. Specifically, validation does not check whether attribute names correspond to actual database entities. If a query is invalid, the status list shows the number of errors and provides a list of the errors in the query.

14. From the **File** menu, choose **Save As** to save your query. Save the query as `rc1.qg2.xml` in the `$PROX_HOME/doc/user/tutorial/examples` directory.

    The Proximity distribution includes this query in the file `$PROX_HOME/doc/user/tutorial/examples/research-clusters1.qg2.xml`. Be careful not to overwrite this file.

    The Query Editor also saves basic layout information for the query so that it will be displayed as shown when you next open the query in the Query Editor. If you prefer a different arrangement, you can rearrange the query by dragging vertices and saving the query again. If the query was invalid when saved, the Query Editor attempts to read and display the query; however, some errors may result in the query being partially or incorrectly displayed.

15. From the **File** menu, choose **Run** or press Ctrl-R to execute your query.

    Proximity prompts you for a name for the results container. Enter **`research-clusters1`** and click **OK**.

Container names may not include `/`, `?`, `!`, `<`, `>`, or `#`.

If the database already includes a container with this name, Proximity asks whether you want to delete the existing container. Answering yes lets Proximity overwrite the contents of this container. This also deletes any containers inside the existing container.

Proximity opens a window to show you a trace of the query execution. The last lines should be similar to the following excerpt (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

```
INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> found 1294 subgraphs
INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> query results saved in
   container: research-clusters1
INFO kdl.prox.qgraph2.QueryGraph2CompOp - * query: done
Status: finished running query
```

Close the trace window after the query finishes.

The next section describes how to view the results of your query.

Exploring Containers and Subgraphs

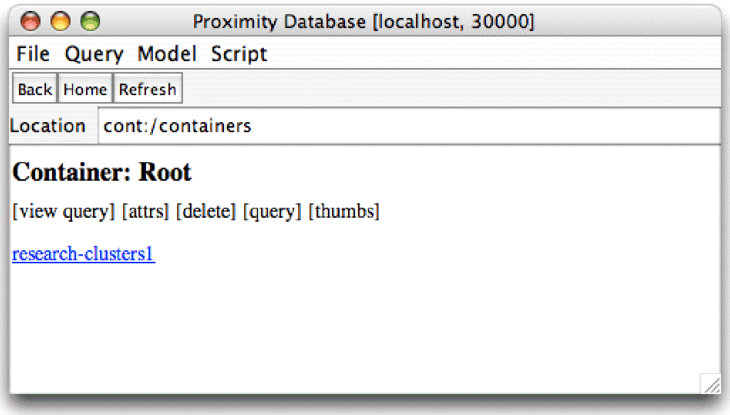
You can use the Proximity Database Browser to examine the results of the queries you execute. The Proximity Database Browser supports examining the subgraphs and containers created as a result of executing queries via both browser-style access (clicking identifiers to see that item’s details) and direct access via the location bar. The following exercise illustrates how to use both of these methods to examine the contents of the container created in the previous exercise. See Appendix A, *Proximity Quick Reference* for a summary of how to access Proximity database elements using the location bar.

Exercise 5.2. Exploring containers and subgraphs:

This exercise requires the container created in Exercise 5.1. You must have completed Exercise 5.1 before beginning the current exercise.

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. Click **Home** to go to the Proximity Database Browser start page.
2. Click **Containers** or enter **`cont:/containers`** in the location bar. Use the `cont:` protocol to access containers using the location bar. Proximity displays the list of top-level containers in the ProxWebKB database.



cont:/containers

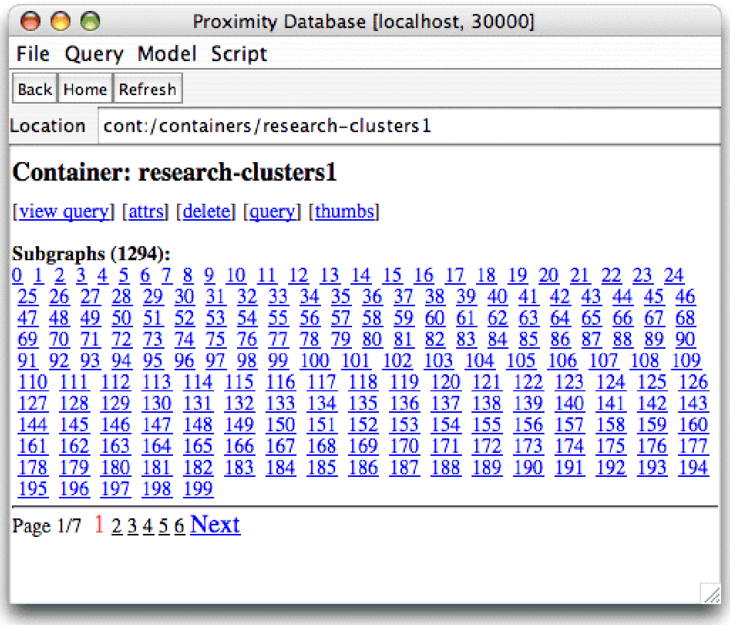
Container: Root

[view query] [attrs] [delete] [query] [thumbs]

research-clusters1

The initial container page shows that you are exploring the contents of the root container. The root container is provided as a convenience for accessing a database's other containers; it does not explicitly exist in the database. The location bar path `cont:/containers` corresponds to the root container.

3. Click **research-clusters1** or enter **`cont:/containers/research-clusters1`** in the location bar. Proximity displays a list of subgraphs in the research-clusters1 container.



cont:/containers/research-clusters1

Container: research-clusters1

[view query] [attrs] [delete] [query] [thumbs]

Subgraphs (1294):
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109
110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160
161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177
178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194
195 196 197 198 199

Page 1/7 1 2 3 4 5 6 Next

The research-clusters1 page lists the ID numbers of the first 200 subgraphs in the container. Click a page number at the bottom of the page or add the page number modifier to the end of the location bar path (e.g., `cont:/containers/research-clusters1#2`) to display more subgraph IDs. Depending on the query, subgraph identifiers may or may not correspond to object identifiers in the database. In general, you should not assume that the subgraph labels correspond to other database entities.

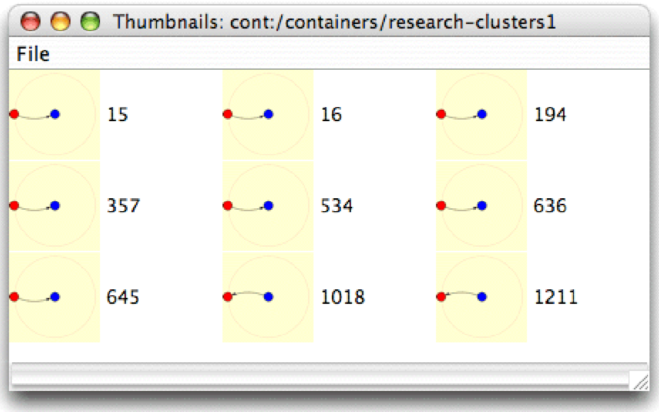
To access any container using the location bar, use the `cont:` protocol and a UNIX-like path to the target container. In this example, the research-clusters1 container is a child of the root container, so the full path to this container is `cont:/containers/research-clusters1`.

Each container page includes a list of container actions at the top of the page:

- The **view query** action displays the query used to create this container. Proximity displays the query in the Query Editor. You must have a copy of `graph-query.dtd` in the directory from which you launched the Proximity Database Browser (i.e., `$PROX_HOME`) to view the query. Note that containers created by means other than querying (e.g., via scripts) and containers created in Proximity 4.2 or earlier versions do not store the originating query with the resulting container. The **view query** option is disabled for these containers.
- The **delete** action deletes the current container. You cannot delete the root container.
- The **attrs** action displays any attributes for the current container. Although you can create container attributes for your own purposes, Proximity does not currently use container attributes other than to store the XML version of the query used to create the container. (The Proximity Database Browser may not display the value of the qgraph_query attribute correctly; use the **view query** action to display the query in the Query Editor instead.)
- The **query** action executes a query against the contents of the current container. See “Querying Containers” (p. 69) for additional information on querying containers.
- The **thumbs** action displays thumbnail images of a random selection of up to nine subgraphs from this collection.

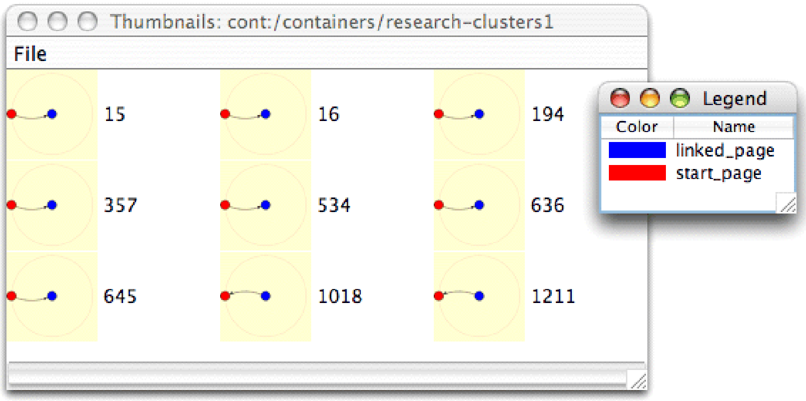
These actions are disabled for the root container.

4. Click **thumbs**. Proximity displays a set of thumbnail images of up to nine randomly selected subgraphs from this container. Because the thumbnails are selected at random, you may see a different set of subgraphs than those shown below.



The thumbnails show that each subgraph in the research-clusters1 container has the same structure: two objects connected by a single link.

5. From the thumbnail window **File** menu, choose **Show Color Legend**. Proximity displays a key that shows how vertex colors in the thumbnails map to vertex labels from the query.

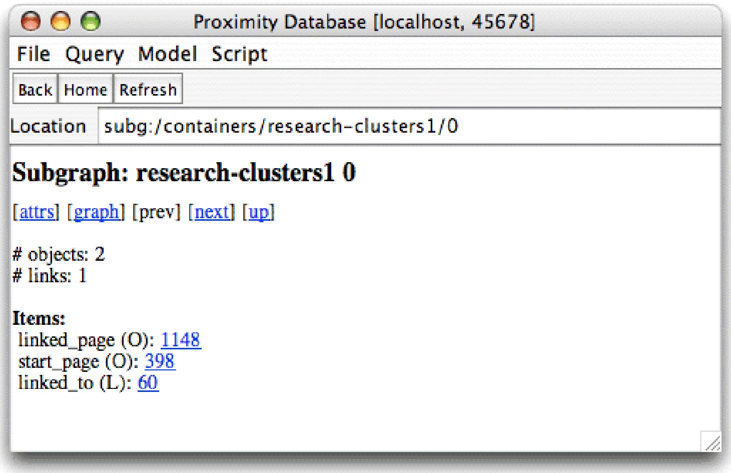


6. Click to select a thumbnail. From the thumbnail window **File** menu, choose **Open Selected Subgraph**. Proximity displays a full-size graph of the selected subgraph.

Tip: You can also display a full-size graph of a subgraph in the thumbnail window by double-clicking the corresponding thumbnail image.

More information on using the full-size subgraph display is included later in this exercise.

7. From the Proximity Database Browser, click **0** or enter **`subg:/containers/research-clusters1/0`** in the location bar to see the contents of subgraph 0. Proximity displays information about this subgraph.


Proximity Database [localhost, 45678]
File
Query
Model
Script
Back
Home
Refresh
Location
subg:/containers/research-clusters1/0

Subgraph: research-clusters1 0

[attrs] [graph] [prev] [next] [up]

objects: 2
links: 1

Items:
linked_page (O): 1148
start_page (O): 398
linked_to (L): 60

The subgraph page includes a list of subgraph actions at the top of the page. These actions provide additional information about the subgraph and enable navigation within the container.

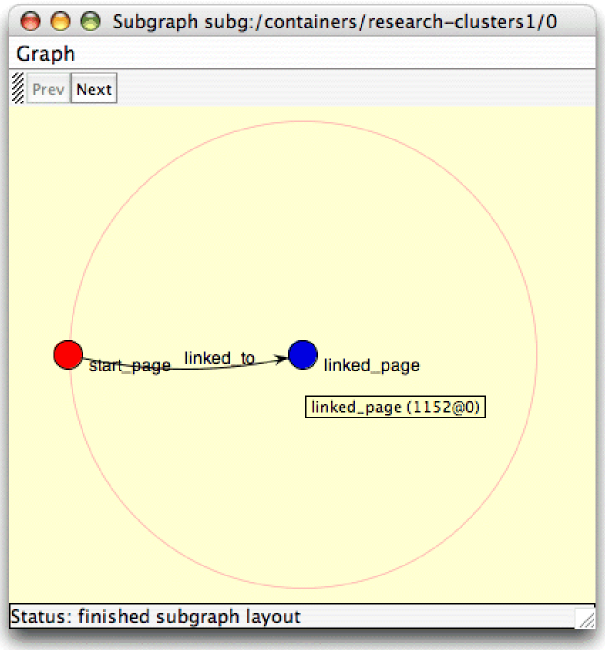
- The **attrs** action displays any attributes for the current subgraph.
- The **graph** action displays a graph of the subgraph.
- The **prev** action displays the previous subgraph in the container. This link is disabled if you are viewing the first subgraph in the container.
- The **next** action displays the next subgraph in the container. This link is disabled if you are viewing the last subgraph in the container.
- The **up** action returns to the parent container page.

Subgraph information includes a list of the subgraph’s member objects and links, identified by “(O)” or “(L)” respectively. The names of the objects and links in a subgraph correspond to the vertex and edge labels from the query that produced this subgraph. The listed objects and links point to the actual database entities, not copies.

As we saw in “Using the Location Bar” (p. 35), the location bar protocol changes when you access individual entities instead of lists of entities. In this case, to access an individual subgraph, we use the `subg:` protocol. The container path remains the same, but we add the target subgraph ID to the end of the path.

Tip: If your database includes an attribute whose values provide semantically meaningful labels for objects, such as a `title` attribute for movies, you can set the Proximity Database Browser preferences to display this attribute value instead of object IDs. See “Setting Display Preferences” (p. 40) for information on using this feature. (The ProxWebKB database does not provide a suitable attribute for this purpose, therefore we continue to display object IDs in the Proximity Database Browser for the remaining exercises.)

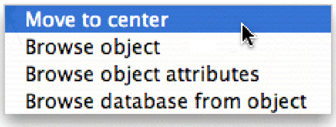
8. Click **graph** to display a graph of this subgraph. Proximity opens a new window displaying a graph of the objects and links in the subgraph. Hover the mouse over an object or link to see its OID.



Status: finished subgraph layout

- **Prev** and **Next** buttons in the graph window change the display to show the previous or next subgraph, respectively, in the container. You can also use the **Next** and **Prev** commands from the **Graph** menu to change to another subgraph.
- To display a key that shows how vertex colors in the graph map to vertex labels from the query, choose **Show Color Legend** from the **Graph** menu.
- To change how items in the subgraph are labeled, choose either **Set Object Labels** or **Set Link Labels** from the subgraph **Graph** menu. You can label items with their OID, their names from the query, or the value of a selected attribute.
- To display the details for this subgraph in a new Proximity Database Browser window, choose **Browse Subgraph** from the subgraph window **Graph** menu.

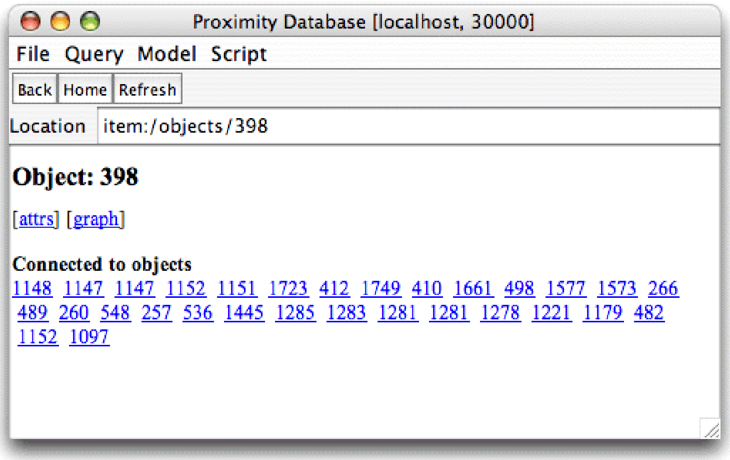
9. Right-click (Ctrl-click for Mac OS X) any object in the graph to display a context menu for that object. The context menu lists actions applicable to the selected object:



- Choose **Move to center** to make the selected object the new center of the display.
- Choose **Browse object** to display basic information about the selected object in a new Proximity Database Browser window.
- Choose **Browse object attributes** to display attribute details for the selected object in a new Proximity Database Browser window.
- Choose **Browse database from object** to graph the selected object and its immediate neighbors in the graphical data browser.

Close the legend and graph windows when you are through examining the graph.

10. Click the object and link IDs in the Items list to explore the individual items in this subgraph. For example, click **start_page (O): 398** or enter **`item:/objects/398`** in the location bar to see the details of the object in this subgraph that matched the *start_page* vertex from the QGraph query. Proximity displays the information for object 398.



item:/objects/398

Object: 398

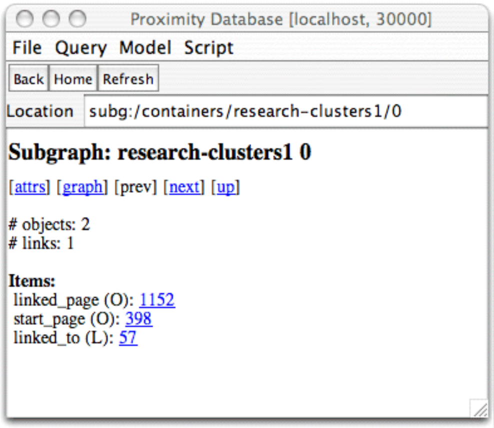
[attrs] [graph]

Connected to objects
1148 1147 1147 1152 1151 1723 412 1749 410 1661 498 1577 1573 266 489 260 548 257 536 1445 1285 1283 1281 1281 1278 1221 1179 482 1152 1097

11. Click **attrs** or enter **`item:/objects/398!ATTRVALS`** to display the attributes and values for object 398. Because object-based attribute lookup is computationally expensive in MonetDB, the main object page does not show the object's attribute values. If necessary, scroll down to see the value of the pagetype attribute to confirm that this is a research project page.

12. Return to the research-clusters1 container page. From the **File** menu, choose **New Window** to open a new Proximity Database Browser window. Arrange the windows so you can see the contents of both windows.

13. Display the contents of subgraph 0 in one of the windows. Display the contents of subgraph 2 in the other window. Notice that subgraphs 0 and 2 have the same *start_page* object.



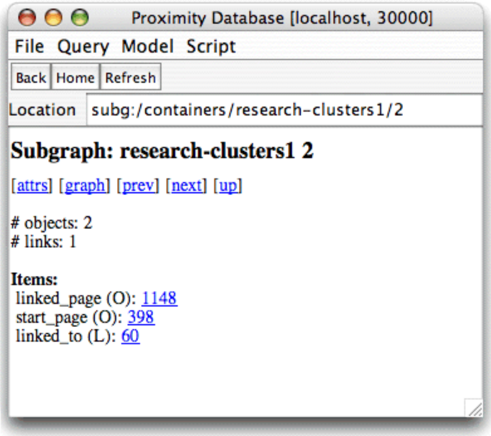
subg:/containers/research-clusters1/0

Subgraph: research-clusters1 0

[attrs] [graph] [prev] [next] [up]

objects: 2
links: 1

Items:
linked_page (O): 1152
start_page (O): 398
linked_to (L): 57



subg:/containers/research-clusters1/2

Subgraph: research-clusters1 2

[attrs] [graph] [prev] [next] [up]

objects: 2
links: 1

Items:
linked_page (O): 1148
start_page (O): 398
linked_to (L): 60

14. Close the window for subgraph 2. In the other window, click **Back** to return to the page for subgraph research-clusters1.
15. Continue to explore the results of your query. When you are finished, continue to the next section.

The query created above matches individual objects and links in the database instead of groups of objects and links. To obtain results that group all the linked pages together for each research project page, we need to add numeric annotations to the query. The next section describes how to use numeric annotations in a Proximity query.

Grouping Elements in a Query

The previous query matched individual elements in the database. To match groups of elements we use *numeric annotations*. A numeric annotation groups together repeated isomorphic substructures that would otherwise create multiple matches for the query. Annotations can also be used to place limits on the number of substructures that can occur in matching portions of the database.

Numeric annotations take one of three forms:

- An *unbounded range* [`i`..] on a vertex or edge means that at least `i` instances of the corresponding database element must be present in the database to match the query.
- A *bounded range* [`i`..`j`] means that at least `i` and no more than `j` instances are required for a match.
- An *exact* annotation [`i`] means that exactly `i` instances are required for a match.

The lower bound, `i`, can be any integer greater than or equal to zero. The upper bound, `j`, must be an integer greater than `i`.

You can specify that the annotated substructure is optional for the match by using a lower bound of zero (e.g., `[0..]`). An exact annotation of `[0]` means that the substructure must *not* be present to match the query. Proximity currently permits the use of optional and negated vertices but prohibits the use of optional or negated edges.

The following exercise finds, for each research project object (web page) in the database, all objects directly linked to that page. It uses a numeric annotation to *group* matching database entities into a single matching subgraph.

This query illustrates one of the most common query forms, the one-dimensional star or *1d-star* query. A 1d-star is the cluster of objects directly linked to a central, core object. Star queries can use QGraph features such as conditions and constraints to restrict matches to objects with specific attribute values.

Executing the research-clusters2 query creates a container used by Exercise 5.9. Make sure that you complete this exercise before beginning Exercise 5.9.

Exercise 5.3. Creating a query with numeric annotations:

The query created in this exercise is also available in the Proximity 4.3 distribution in `$PROX_HOME/doc/user/tutorial/examples/research-clusters2.qg2.xml`.

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. Open the research-clusters1 query that you created in Exercise 5.1:

   a. From the **Query** menu, choose **Edit Query**.

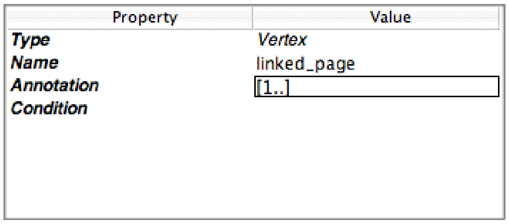
   b. Navigate to the location where you saved the query created in Exercise 5.1. (If you saved the query using the suggested directory and file name, this is `$PROX_HOME/doc/user/tutorial/examples/rc1.qg2.xml`.)

c. Click **OK**.

So that you do not overwrite the existing query, save this new query as `rc2.qg2.xml`.

The Proximity distribution includes the new query in the file `$PROX_HOME/doc/user/tutorial/examples/research-clusters2.qg2.xml`. Be careful not to overwrite this file.

2. Select the *linked_page* vertex. Double-click in the Value column of the Annotation property. Enter **`[1..]`** and press Tab.



Property	Value
Type	*Vertex*
Name	linked_page
Annotation	[1..]
Condition	

> Only one of any two adjacent vertices may be annotated. Annotating adjacent vertices is prohibited as the resulting query is ambiguous. See the *Proximity QGraph Guide* for additional information on and reasons for QGraph annotation rules..

3. Check the status list at the bottom of the Query Editor. Notice that the query is now invalid. Click the arrows at the right edge of the status list to see the specific errors.

   An edge next to an annotated vertex must itself be annotated. To fix the error, select the *linked_to* edge, enter **`[1..]`** for the edge's annotation, and press Tab.

> An edge next to an annotated vertex must also be annotated. The vertex annotation takes precedence over the edge annotation.

You can optionally set the Query Editor to automatically add a `[1..]` annotation to every new edge. The `[1..]` edge annotation groups multiple links connecting the same two objects, which is typically the intended behavior for most Proximity queries. To automatically add a `[1..]` annotation to new edges, choose **Add [1..] To New Edges** from the Query Editor's **Edit** menu. You can edit an individual edge annotation later if you determine that a different annotation is more appropriate.

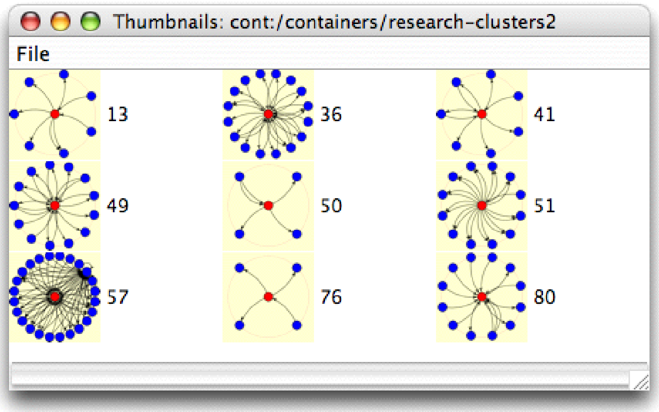
4. Update the name and description of the query. Instead of finding individual linked pages, the new query finds clusters of pages connected to research project pages.
5. Make sure the query is valid by checking the status list at the bottom of the Query Editor window. If the query is not valid, examine the errors in the list and fix any problems before saving.
6. [Optional] Update the name and description for the query.
7. From the **File** menu, choose **Save** or press Ctrl-S to save the changes to your query.
8. From the **File** menu, choose **Run** or press Ctrl-R to execute your query. Proximity prompts you for a name for the results container. Enter **`research-clusters2`** and click **OK**.

   Proximity opens a window to show you a trace of the query execution. The last lines should be similar to the following excerpt (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

```
INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> found 83 subgraphs
INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> query results saved in
   container: research-clusters2
INFO kdl.prox.qgraph2.QueryGraph2CompOp - * query: done
Status: finished running query
```

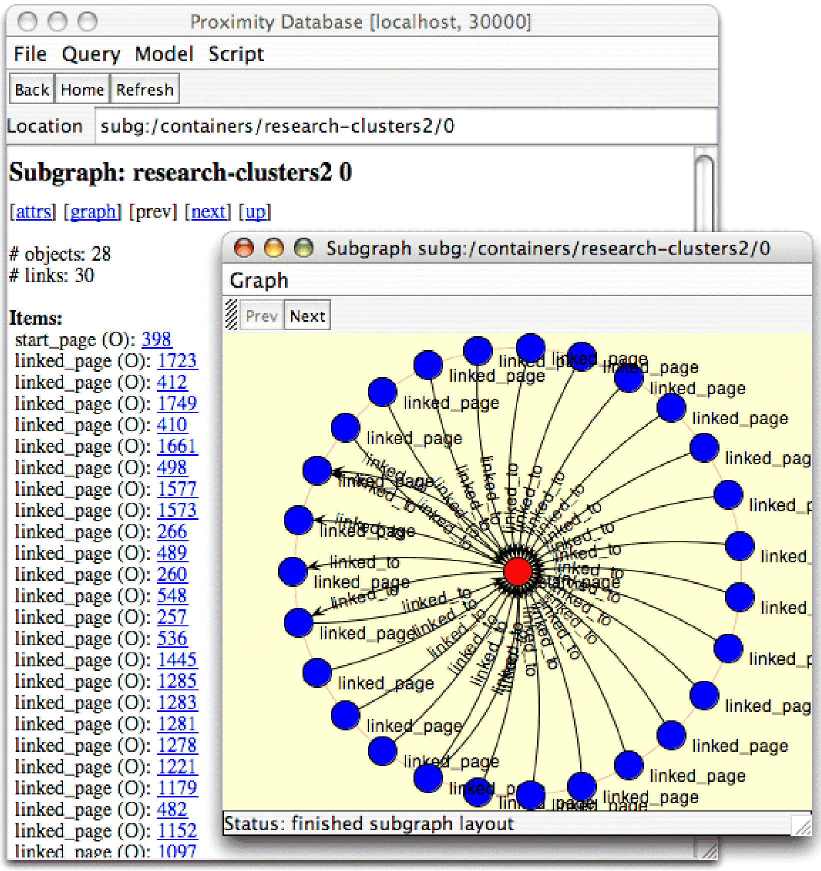
Close this window after the query finishes.

9. If needed, move or close the Query Editor window so that you can see the Proximity Database Browser. In the Proximity Database Browser start page, click **Containers** to display the list of containers in the database.
10. Click **research-clusters2** to display the list of subgraphs in this container. You can see that there are far fewer subgraphs in this container than there were in the research-clusters1 container.
11. Click **thumbs** to display thumbnail images for a set of randomly selected subgraphs. Because the thumbnails are selected at random, you may see a different set of subgraphs than those shown below.



Unlike the subgraphs in the research-clusters1 container, these subgraphs vary in the number of *linked_page* objects linked to each *start_page* objects. And in some of the subgraphs, some of the *linked_page* objects are connected to the *start_page* object by two or more links. The `[1..]` annotation on the *linked_page* vertex groups the matching objects and the `[1..]` annotation on the *linked_to* edge groups the corresponding links.

12. Click **0** to see the contents of subgraph 0. Notice that we still have a single start page, object 398 (you may need to scroll down to see the *start_page* object), but the subgraph now includes many linked pages. Click **graph** to display the graph of this subgraph.



Location subg:/containers/research-clusters2/0

Subgraph: research-clusters2 0

[attrs] [graph] [prev] [next] [up]

objects: 28
links: 30

Items:

start_page (O): 398
linked_page (O): 1723
linked_page (O): 412
linked_page (O): 1749
linked_page (O): 410
linked_page (O): 1661
linked_page (O): 498
linked_page (O): 1577
linked_page (O): 1573
linked_page (O): 266
linked_page (O): 489
linked_page (O): 260
linked_page (O): 548
linked_page (O): 257
linked_page (O): 536
linked_page (O): 1445
linked_page (O): 1285
linked_page (O): 1283
linked_page (O): 1281
linked_page (O): 1278
linked_page (O): 1221
linked_page (O): 1179
linked_page (O): 482
linked_page (O): 1152
linked_page (O): 1097

Status: finished subgraph layout

Using numeric annotations successfully collapses the multiple subgraphs for each research page into a single subgraph that includes all the linked pages.

Comparing Items in a Query

Conditions let you specify restrictions on individual items in a query. To place restrictions across different items we use *constraints*. Constraints compare one vertex or edge in the query to another vertex or edge.

QGraph permits two types of constraints:

- *Identity constraints* compare the identities of two database objects or links.
- *Attribute constraints* compare the attribute values of two database objects or links.

Identity constraints are typically used to ensure that the same database entity does not match two different query elements. Exercise 5.5 illustrates a typical use of an identity constraint.

Attribute constraints allow you to compare different attributes as long as their data types are comparable. In addition to comparing attribute values with the same data type, Proximity also lets you to compare `DBL` with `FLT` and `INT`, and `FLT` with `INT`. This exercise uses an attribute constraint to compare values for the school attribute across objects.

> You cannot mix vertices and edges within a constraint. Vertices can only be compared to other vertices and edges can only be compared to other edges.
>
> QGraph only lets one of the two constrained items be annotated; constraints between two

annotated items are not allowed. With one exception, Proximity also prohibits constraints between two edges when one of them is annotated. Proximity allows such constraints only when the vertex adjacent to the annotated edge isoptional (annotated with `[0..]` or `[0..j]`).

The following exercise creates a query that finds linked web pages from different schools. The example query uses an attribute constraint to compare the value of the school attribute for the two web page objects.

Exercise 5.4. Adding constraints to a query:

The query created in this exercise is also available in the Proximity 4.3 distribution in `$PROX_HOME/doc/user/tutorial/examples/different-schools.qg2.xml`.

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

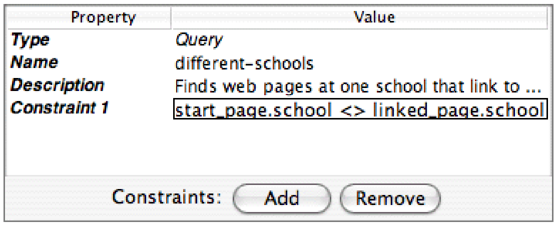
1. From the **Query** menu, choose **New Query**. Proximity starts the Query Editor.
2. Create a vertex labeled **`start_page`**.
3. Create a second vertex labeled **`linked_page`**.
4. Add the numeric annotation **`[1..]`** to the *linked_page* vertex.
5. Create a directed edge from *start_page* to *linked_page*. Label the edge **`linked_to`**.
6. If you do not have automatic edge annotation enabled, add the numeric annotation **`[1..]`** to the *linked_to* edge.
7. In the Constraints area at the bottom of the query properties pane, click **Add** to add a new constraint to the query.

   The Query Editor adds a temporary constraint, item1 > item2, as an example to be edited with the correct values. (Because this temporary constraint uses vertex labels not present in the query, the status list shows that the query is invalid.)

8. Replace the example constraint with

   ```
   start_page.school <> linked_page.school
   ```

   and press Tab.

Property	Value
Type	*Query*
Name	different-schools
Description	Finds web pages at one school that link to ...
Constraint 1	start_page.school <> linked_page.school

   The query should now be valid.

The general form of an attribute constraint is

```
element1.attribute1  operator  element2.attribute2
```

where

- `element1` and `element2` are the names of two vertices or two edges in the query
- `attribute1` is the name of an attribute for `element1`
- `attribute2` is the name of an attribute for `element2`
- `attribute1` and `attribute2` are of comparable types
- `operator` is one of =, <>, <, <=, >, and >=.

The general form of an identity constraint is

```
element1 operator element2
```

Surround vertex, edge, and attribute names and values containing spaces with single quotes.

You can compare attribute values for two vertices or two edges, but you cannot mix vertex and edge attributes in the same constraint. Proximity does not permit constraints between two annotated elements.

9. Make sure the query is valid by checking the status list at the bottom of the Query Editor window. If the query is not valid, examine the errors in the drop-down list and fix any problems before continuing.
10. [Optional] Add a name and description and save the query.

    The Proximity distribution includes the new query in the file `$PROX_HOME/doc/user/tutorial/examples/different-schools.qg2.xml`. Be careful not to overwrite this file.

11. From the **File** menu, choose **Run** or press the Ctrl-R to execute your query. Proximity prompts you for a name for the results container. Enter **`different-schools`** and click **OK**.

    Proximity opens a window to show you a trace of the query execution. The last lines should be similar to the following excerpt (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

    ```
    INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> found 0 subgraphs
    INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> query results saved in
       container: different-schools
    INFO kdl.prox.qgraph2.QueryGraph2CompOp - * query: done
    Status: finished running query
    ```

    Close this window after the query finishes.

    The resulting container has no subgraphs—there are no web pages in this database that link to a page at another school. Proximity creates a container even when there are no matches to the query.

Matching Complex Subgraphs with Subqueries

Subqueries allow you to group complex substructures with numeric annotations just as you grouped individual query elements in Exercise 5.3.

In this example we want to find, for every faculty member, all their research projects and all the individuals associated with each of those research projects. We interpret links from faculty pages to research project pages to mean that this faculty member supervises the associated project. We similarly interpret links connecting research project pages to other individuals' web pages as indicating involvement in or interest in that project, regardless of the direction of the link. Taken together, all these links identify individuals associated with a particular project. The subgraphs that match the query created in this exercise each identify a single faculty web page, all the research project pages it links to, and all the other faculty, student, and staff pages linked to or from those research project pages.

The structure of this query is a *2d-star*. Similar to the 1d-star query shown in Exercise 5.3, a 2d-star is the cluster of 1d-stars directly linked to a core object. We use a subquery to specify the 1d-star substructure within a 2d-star query.

In describing this query we made several assumptions about how objects and links in the database correspond to entities and relationships in the world. Although these assumptions may seem reasonable, it's important to remember that we might be wrong in some or all of these assumptions. If we go on to use the results of this query in a model, appropriate care must be taken in applying the models's predictions to real world situations.

Exercise 5.5. Using subqueries in a query:

The query created in this exercise is also available in the Proximity 4.3 distribution in `$PROX_HOME/doc/user/tutorial/examples/project-people.qg2.xml`.

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. From the **Query** menu, choose **New Query**. Proximity starts the Query Editor.
2. Create an unannotated vertex labeled **`project`**. Add the condition

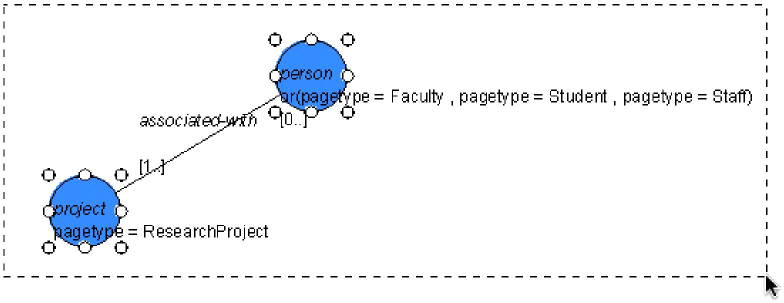
   ```
   pagetype = ResearchProject
   ```

   to this vertex.
3. Create another vertex labeled **`person`**. Add the condition

   ```
   OR(pagetype = Faculty, pagetype = Student,
   pagetype = Staff)
   ```

   to this vertex. Proximity requires the use of prefix notation and disjunctive normal form to express complex conditions.
4. Add the annotation **`[0..]`** to the *person* vertex. This annotation makes matching the *person* vertex optional. Any pages matching the *person* vertex will be included in the query results, but the query will match appropriate research project web pages regardless of whether they link to another faculty, student, or staff page.
5. Create an undirected edge labeled **`associated-with`** linking the *project* vertex to the *person* vertex. If you do not have automatic edge annotation enabled, add the numeric annotation **`[1..]`** to this edge.

   The correct annotation for an edge adjacent to an optional vertex is almost always **`[1..]`**. See the *Proximity QGraph Guide* for an explanation of why this is the appropriate annotation.
6. Click or press Ctrl-4 to choose the subquery tool. Drag the mouse to create a rectangle that contains both vertices and the connecting edge.

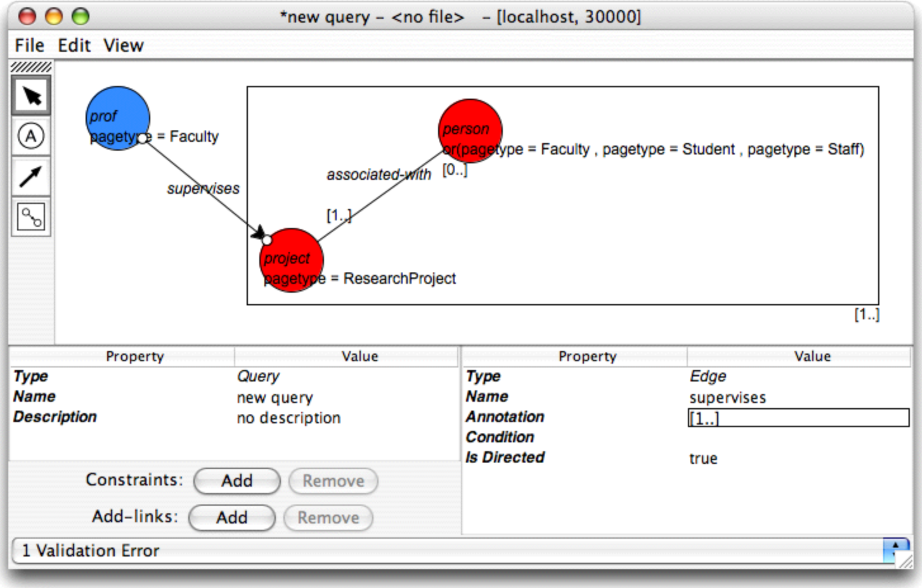
   
   person
   or(pagetype = Faculty , pagetype = Student , pagetype = Staff)
   [0..]
   associated-with
   [1..]
   project
   pagetype = ResearchProject

   Release the mouse button. The Query Editor changes the color of the vertices inside the rectangle to red to indicate that they are part of a subquery.

   All subqueries must be annotated. The Query Editor automatically adds a `[1..]` annotation to new subqueries. To change this annotation, click inside the subquery box and edit the subquery's properties in the element properties pane.
7. Create a vertex named **`prof`** outside the subquery area. Add the condition

   ```
   pagetype = Faculty
   ```

   to this vertex.
8. Create a directed edge named **`supervises`** from the *prof* vertex to the *project* vertex.
9. Add the numeric annotation **`[1..]`** to the *supervises* edge. The boundary edge of a subquery must be annotated.



Property	Value
Type	Query
Name	new query
Description	no description

Property	Value
Type	Edge
Name	supervises
Annotation	[1..]
Condition	
Is Directed	true

1 Validation Error

10. Add the identity constraint

    `prof <> person`

    to the query. This constraint ensures that the same object does not match both the *prof* and *person* vertices in the same subgraph.
11. Check the status list at the bottom of the Query Editor window to make sure the query is valid. If the query is not valid, examine the errors and fix any problems before continuing.
12. [Optional] Add a name and description and save the query.

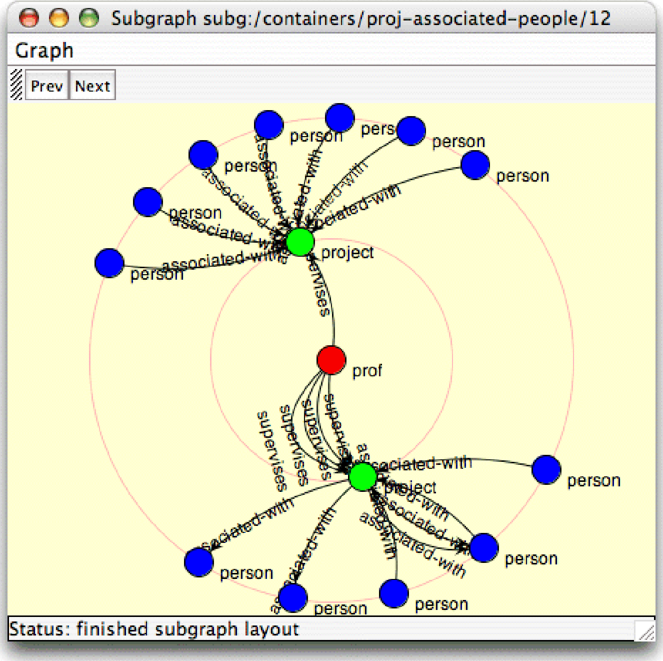
> The Proximity distribution includes the new query in the file `$PROX_HOME/doc/user/tutorial/examples/project-people.qg2.xml`. Be careful not to overwrite this file.

13. From the **File** menu, choose **Run** or press Ctrl-R to execute your query. Proximity prompts you for a name for the results container. Enter **`proj-associated-people`** and click **OK**.

    Proximity opens a window to show you a trace of the query execution. The last lines should be similar to the following excerpt (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

    ```
    INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> found 42 subgraphs
    INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> query results saved in
       container: proj-associated-people
    INFO kdl.prox.qgraph2.QueryGraph2CompOp - * query: done
    Status: finished running query
    ```

    Close this window after the query finishes.
14. Examine the query results in the proj-associated-people container. Click **Home** to go to the Proximity Database Browser start page. Click **Containers**, then click **proj-associated-people**. Proximity displays the list of subgraphs for this container.
15. Click **12** to see the details of this subgraph. This subgraph includes two research project objects.
16. Click **graph** to display the subgraph’s graph structure. Because graph layout is non-deterministic, your graph may look somewhat different than that shown below.


Subgraph subg:/containers/proj-associated-people/12
Graph
Prev
Next
person
project
prof
associated-with
supervises
Status: finished subgraph layout

17. Continue to explore the results of your query. When you are finished, continue to the next section.

Adding Links to Data with Queries

In addition to being able to extract data in the form of subgraphs, the QGraph language provides the ability to update data—add or delete objects, links, or attributes—by executing queries. Although much of this functionality remains to be implemented in Proximity, you can currently use QGraph queries to add new links to the data.

QGraph query processing separates the match phase from the update phase, thus any new links added to the database are *not* included in the query's results. Importantly, the match phase applies to the data as it exists at the start of query execution. Applying an update cannot add new matches to the current query's results.

The query created in the following exercise continues some of the assumptions we made in Exercise 5.5 about how the data in the ProxWebKB database correspond to entities and relationships in the world. Specifically, we interpret links from Faculty pages to ResearchProject pages to mean that this faculty member directs the project. Similarly, we interpret links from ResearchProject pages to Student pages to mean that that those students work on the project. Therefore, we infer that the faculty member supervises these students, allowing us to add links that make this relationship explicit. Although these assumptions may seem reasonable, it's important to remember that we might be wrong in some or all of these assumptions. If we go on to use the results of this query in a model, appropriate care must be taken in applying the models's predictions to real world situations.

Exercise 5.6. Adding links with a query:

The query created in this exercise is also available in the Proximity 4.3 distribution in `$PROX_HOME/doc/user/tutorial/examples/add-supervises-links.qg2.xml`.

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. From the **Query** menu, choose **New Query**. Proximity starts the Query Editor.

2. Create an unannotated vertex labeled **`project`**. Add the condition

   ```
   pagetype = ResearchProject
   ```

   to this vertex.
3. Create another vertex labeled **`professor`**. Add the condition

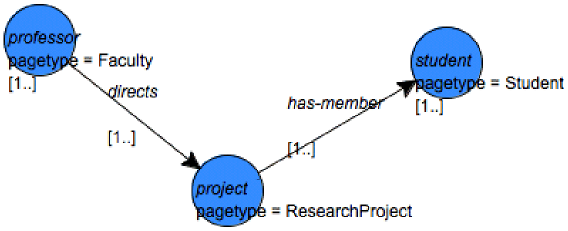
   ```
   pagetype = Faculty
   ```

   to this vertex. Add the annotation **`[1..]`** to this vertex.
4. Create a directed edge labeled **`directs`** from the *professor* vertex to the *project* vertex. If you do not have automatic edge annotation enabled, add the annotation **`[1..]`** to this edge.
5. Create a vertex labeled **`student`**. Add the condition

   ```
   pagetype = Student
   ```

   to this vertex. Add the annotation **`[1..]`** to this vertex.
6. Create a directed edge labeled **`has-member`** from the *project* vertex to the *student* vertex. If you do not have automatic edge annotation enabled, add the annotation **`[1..]`** to this edge.

   The query should look similar to that shown below:



7. In the Add-links area at the bottom of the query properties pane, click **Add** to specify the new links to be added.

   The Query Editor creates a temporary link specification, vertex1, vertex2, attrname, "attrval", as an example to be edited with the correct values. (Because this temporary link specification uses vertex labels not present in the query, the status indicator now shows that the query is invalid.)

The general form of a new link specification is

`starting-vertex, ending-vertex, attribute-name, attribute-value`

where

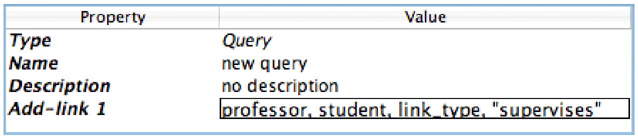
- *`starting-vertex`* is the name of the query vertex that corresponds to the link's starting object
- *`ending-vertex`* is the name of the query vertex that corresponds to the link's ending object
- *`attribute-name`* is the name of an attribute that will be placed on each new link. The specified attribute can be a new attribute or it can already exist as a link attribute in the database.
- *`attribute-value`* is the value of *`attribute-name`* to be assigned to each new link. The new attribute value must be the same for all links created by this specification.

The attribute name and value are required—all new links must be assigned a value for a new or existing link attribute.

8. Replace the example link specification with

   `professor, student, link_type, "supervises"`

   and press Tab.



Property	Value
Type	*Query*
Name	new query
Description	no description
Add-link 1	professor, student, link_type, "supervises"

   The link specification states that we want to add links from objects matching the *professor* vertex to objects matching the *student* vertex, assigning the value supervises to a new link_type attribute for that link.

   If the resulting container includes more than one subgraph that would create the same new link, only one instance of the new link is created. However, re-running the query creates a new container and thus a new set of identical links.

9. Make sure the query is valid by checking the status list at the bottom of the Query Editor window. If the query is not valid, examine the errors in the drop-down list and fix any problems before continuing.
10. [Optional] Add a name and description and save the query.

   The Proximity distribution includes the new query in the file `$PROX_HOME/doc/user/tutorial/examples/add-supervises-links.qg2.xml`. Be careful not to overwrite this file.

11. From the **File** menu, choose **Run** or press Ctrl-R to execute your query. Proximity prompts you for a name for the results container. Enter **`project-teams`** and click **OK**.

   Proximity opens a window to show you the a trace of the query execution. The last lines should be similar to the following excerpt (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

```
INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> found 20 subgraphs
INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> query results saved in
   container: project-teams
INFO kdl.prox.qgraph2.QueryGraph2CompOp - Adding links to database
INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> Adding add-link
   professor, student, link_type, "supervises":49 links created
INFO kdl.prox.qgraph2.QueryGraph2CompOp - * query: done
Status: finished running query
```

   The trace includes a count of the number of links created. In this example, we added 49 new links to the database.

   Close this window after the query finishes.

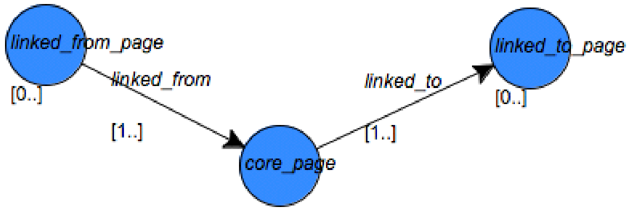
12. To explore the newly added links, browse the links having a value of supervises for the new link_type attribute by drilling down through the Link Attributes area in the Proximity Database Browser or typing

   `filter:/links/link_type/value/'supervises'`

   in the Proximity Database Browser location bar and pressing Return. Proximity displays a list of the new links.

Executing a Query from the Proximity Database Browser

In addition to executing queries from the Query Editor, you can execute previously saved queries directly from the Proximity Database Browser. The following exercise executes the query shown below to find, for every object (web page) in the database, the 1d-cluster of objects directly connected to that core object.



This is another example of the common 1d-star query. The query places no restrictions on the core object or on any related objects—that is, there are no conditions or constraints. Additionally, the use of the `[0..]` annotation on the *linked_from_page* and *linked_to_page* vertices makes these matches optional. By having no conditions or constraints and making the linked objects optional, we ensure that the query will return a subgraph for every object in the database.

Note that this query distinguishes links pointing to the core object from links leaving the core object, and thus defines two types of related objects: *linked_from_page*, those objects that link to the core object, and *linked_to_page*, those objects that are linked to by the core object. The matching subgraphs retain these vertex labels, allowing you to identify and use each type of linked object independently when learning and applying models. If this distinction were not important, we could have instead used a simpler query of two vertices connected by an undirected edge.

> Executing the 1d-clusters query creates a container used by later tutorial exercises. Make sure that you complete this exercise if you plan to complete the exercises in the following chapters.

Exercise 5.7. Executing a saved query from the Proximity Database Browser:

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. If you have not already done so, copy `graph-query.dtd` from `$PROX_HOME/resources` to the directory containing the example queries.

   ```
   > cp $PROX_HOME/resources/graph-query.dtd $PROX_HOME/doc/user/tutorial/examples/
   ```

> Proximity queries are represented using an XML format. The DTD that describes this format must be in the directory containing the query file in order to execute the query.

2. From the **Query** menu, choose **Run Query**. Proximity displays the Open dialog.
3. Navigate to the `$PROX_HOME/doc/user/tutorial/examples` directory and choose the query `1d-clusters.qg2.xml`.

   You can create shortcuts to commonly used directories for easier access to query files. Shortcuts appear in the Shortcuts pane of the Open dialog. See "Creating a file or directory shortcut" (p. 9) for information on creating shortcuts.
4. Click **Open**. Proximity prompts you for a name for the results container. Enter **`1d-clusters`** and click **OK**.

   If the database already includes a container with this name, Proximity asks whether you want to delete the existing container. Answering yes lets Proximity overwrite the contents of this container. This also deletes any containers inside the existing container.

Proximity opens a window to show you a trace of the query execution. The last lines should be similar to the following excerpt (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

```
INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> found 4135 subgraphs
INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> query results saved in
   container: 1d-clusters
INFO kdl.prox.qgraph2.QueryGraph2CompOp - * query: done
Status: finished running query
```

Close this window after the query finishes.

5. Explore the 1d-clusters container. Note that because we used the `[0..]` annotation on the linked objects making them optional for the match, some subgraphs contain only a single object.

Executing a Query from the Command Line

In addition to executing queries from the Proximity Database Browser and Query Editor, you can execute saved queries from the command line using the **query.sh** shell script (Linux/Mac OS X) or **query.bat** (Windows) batch file.

Exercise 5.8. Executing a query from the command line:

This exercise re-runs the research-clusters2 query. You can use either the `rc2.qg2.xml` query you created in Exercise 5.3 or the `research-clusters2.qg2.xml` query included with the distribution in `$PROX_HOME/doc/user/tutorial/examples`. The query results will overwrite the existing research-clusters2 container. Specify a different container name if you want to keep your original research-clusters2 container.

When executing queries from the Proximity Database Browser or Query Editor, Proximity prompts you to confirm overwriting an existing container. Proximity does not provide a similar prompt when executing queries from the command line and instead silently overwrites the existing container.

Before beginning, make sure that you are serving the ProxWebKB database using Mserver.

➤ Execute the **query.sh** shell script or **query.bat** batch file, specifying

`host:port` the Monet server's host and port (`localhost:30000` if you are running the Monet server on your local machine and did not explicitly set the port to a different value on the command line)

`queryXMLFile` the file containing the saved query; use either your `rc2.gq2.xml` query or the query listed below

`collectionName` the name of the results container

`inputContainer` [optional] the name of the input container— the container to search when finding matches for the query

The optional *`inputContainer`* defaults to the root container (the entire database) if it is omitted. To explicitly specify the input container, Proximity uses a path-like syntax that reflects the container hierarchy for the specified container. Container names are separated by forward slashes with the initial `/` representing the root container. For example, `/c1/c2` signifies container "c2", which is under container "c1", which in turn is under the root container. All container paths are absolute in this sense—they all must include an initial slash indicating the root container. Do not include a trailing “/”. See the “Querying Containers” section, below, for more information on querying containers.

In the command lines below, substitute the appropriate host and port information if you are running the Monet server on a different machine or are using a different port.

```
> cd $PROX_HOME
> bin/query.sh localhost:30000 \
  $PROX_HOME/doc/user/tutorial/examples/research-clusters2.qg2.xml \
  research-clusters2
```

Proximity writes the execution trace to the window in which you executed the above command. The last lines should be similar to the following excerpt (leading information showing elapsed time has been omitted from the trace for brevity):

```
INFO  qgraph2.QueryGraph2CompOp: -> found 83 subgraphs
INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> query results saved in
   container: research-clusters2
INFO  qgraph2.QueryGraph2CompOp: * query: done
INFO  app.Query: * done executing query
```

Querying Containers

In addition to executing queries against the entire database, Proximity lets you execute queries against existing containers. In such cases, Proximity ignores the container subgraph information and merely limits query matches to the objects and links in the container. Executing a query against a container creates a new container of matching subgraphs at the top level, i.e., directly under the root container. This functionality is available through both the Proximity Database Browser and the command line query interface. You cannot execute a query against a container from the Query Editor.

The following exercise executes a query against the research-clusters2 container created in Exercise 5.3. The research-clusters2 container holds the 1d-star clusters of objects linked to research project pages. The new query searches this data to find pages from Cornell and those pages connected to the Cornell page by “out” links (links pointing from the core page to the linked page).

To execute a query against a container instead of against the full database, you can either navigate to the input container and call the query from the Proximity Database Browser as shown in Exercise 5.9, below, or specify the input container name as an optional argument to the query shell script, as shown above in Exercise 5.8. All queries run from the Query Editor are executed against the complete database.

Exercise 5.9. Querying a container from the Proximity Database Browser:

This exercise requires the container created in Exercise 5.3. You must have completed Exercise 5.3 before running the current exercise.

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. Click **Home** to go to the Proximity Database Browser start page.
2. Click **Containers**. Proximity displays the current list of containers in the database.
3. Click **research-clusters2**. Proximity displays the research-clusters2 container page.
4. Click **query**. Proximity displays the Open dialog.
5. Navigate to the `$PROX_HOME/doc/user/tutorial/examples` directory and choose the query `cornell-out-clusters.qg2.xml`.
6. Click **Open**. Proximity prompts you for a name for the results container. Enter **`cornell-out-clusters`** and click **OK**.

If the database already includes a container with this name, Proximity asks whether you want to delete the existing container. Answering yes lets Proximity overwrite the contents of this container. This also deletes any containers inside the existing container.

Proximity opens a window to show you a trace of the query execution. The last lines should be similar to the following excerpt (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

```
INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> found 99 subgraphs
INFO kdl.prox.qgraph2.QueryGraph2CompOp - -> query results saved in
   container: cornell-out-clusters
INFO kdl.prox.qgraph2.QueryGraph2CompOp - * query: done
Status: finished running query
```

Close this window after the query finishes.

7. Explore the results of the cornell-out-clusters container.

Executing the query creates a new, top-level container of 99 subgraphs using data in the original container of 83 subgraphs. Remember that Proximity uses the input container simply to limit the database items considered in executing the query; it does not use the existing subgraph information as part of this process. Therefore, the subgraphs identified as a result of executing this query are not related to those found by the query that initially created the input container. The new query does not require that its core_page object be a research project page, only that the school attribute have Cornell as a value. Thus the resulting subgraphs may have little resemblance to those in the input container.

Tips and Reminders

General query requirements

- See the *Proximity QGraph Guide* for a complete description of the QGraph language as implemented in Proximity.
- Queries run from the Query Editor are always executed against the entire database.
- Queries run from the Proximity Database Browser or from the command line can be optionally executed against the contents of a container.
- The QGraph DTD, `graph-query.dtd`, must reside in the same directory as the query file. Make sure that you copy the DTD file from `$PROX_HOME/resources` to the directory containing the query file before executing the query.
- Every query must have at least one vertex.
- Every query edge must have vertices at both ends.
- Queries must be connected.
- Queries must remain connected after any optional or negated elements are removed.
- Each vertex and edge in a query must have a unique label (name).
- Proximity ignores case when matching attribute names to those in the database. Proximity obeys case when matching attribute values with those in the database.

Condition requirements

- Attribute value conditions restrict query matches to objects or links that match the specified attribute value.
- Existence conditions restrict query matches to objects or links that have any value for the specified attribute.
- Use disjunctive normal form in prefix notation to create complex conditions from simple conditions.
- Surround values (i.e., vertex and edge labels or attribute names and values) containing spaces with single quotes.

Annotation requirements

- An edge incident to an annotated vertex must be annotated.

- The boundary edge of an annotated subquery must be annotated.
- At most, only one of two adjacent elements can be annotated; no edges are allowed between two annotated elements.
- Optional and negated edges are not permitted.

Constraint requirements

- Identity constraints compare IDs with each other.
- Attribute constraints compare attribute values with each other.
- Mixing IDs and attribute values in a constraint is not permitted. (You may, however, compare values for different attributes as long as they are of comparable types.)
- Constraints are only allowed between two vertices or two edges; mixing vertices and edges is not permitted.
- No constraints are allowed between two annotated elements.
- If one of two items in a constraint is annotated, the other cannot be part of an annotated subquery.
- With one exception, Proximity prohibits constraints between two edges when one of them is annotated. Proximity allows such constraints when the vertex adjacent to the annotated edge is itself annotated with `[0..]` or `[0..j]`.
- Constraints can only be combined with AND; OR and NOT are not allowed.
- Constraints that cross subquery boundaries require that one of the items being compared be optional (have an annotation of `[0..]` or `[0..j]`).
- Surround values (i.e., vertex and edge labels or attribute names and values) containing spaces with single quotes.

Subquery requirements

- A subquery must be a well-formed query by itself.
- All subqueries must be annotated.
- The boundary edge of a subquery must be annotated.
- The boundary vertex of a subquery cannot be annotated.
- The vertex outside the subquery connected to the boundary edge cannot be annotated.

Update requirements

- The QGraph language provides full update functionality; however, Proximity currently implements only the ability to add links using queries.
- A query's match phase is distinct from its update phase; new database elements added by a query do not change a query's matches.
- New database elements added by a query are not included in the query's results (subgraphs).
- You must specify an attribute value for all new links. The attribute value must be constant—the same for all new links added by the query.

Additional Information

- See the *Proximity QGraph Guide* for additional details on the QGraph language and it's implementation in Proximity.
- See Chapter 7, *Learning Models* for more information on how query elements and labels are used by Proximity's models.
- See Appendix A, *Proximity Quick Reference* for a summary of how to use the location bar to directly access Proximity database elements including containers and subgraphs.

Chapter 6. Using Scripts

Overview

In addition to using the Proximity Database Browser and command shell scripts to work with Proximity data, you can access the complete Proximity API through the Proximity scripting interface. This interface supports Python scripts run via the Jython interpreter, permitting access to the full Proximity Java class library. Scripts provide the means for many Proximity tasks including learning and applying models. Proximity lets you run full Python programs from files as well as providing a full-featured Python interpreter within the Proximity interface for convenient access to Proximity's scripting interface.

The examples in this chapter are written in Jython, a Java implementation of Python that lets you interact with Java code. The classes and methods used in these examples are, of course, also available for use in Java code. Source code files for all the scripts discussed in this chapter are available in `$PROX_HOME/doc/user/tutorial/examples`.

For each exercise in this chapter, we first walk through the source code and identify how Proximity classes and methods are used to accomplish the desired task. This is followed by instructions for running the script in Proximity.

Objectives

The exercises in this chapter demonstrate how to

- run scripts from the Proximity Database Browser
- run scripts from the command line
- execute Python statements interactively using Proximity's Python interpreter
- find objects having specified attribute values
- sample containers to create training and test sets
- access database elements (containers, subgraphs, objects, links, and attributes) from a script
- add a new attribute to the database
- create synthetic data that exhibits specific autocorrelation and degree disparity effects
- use social networking algorithms to understand database structure

Working with Scripts

Processing overview

You can execute scripts from the Proximity Database Browser or from the command line using the Proximity shell scripts. Proximity also provides a Python interpreter that lets you execute Python statements interactively so that you can explore results and run short programs without needing to create script files. Both methods for running scripts from files are described in "Running Proximity Scripts" (p. 74). The interactive Proximity Python interpreter is described in "Using the Proximity Python Interpreter" (p. 75).

Importing Proximity packages

You only need to import a Proximity class or package when you refer to a class by name, such as when instantiating a class object, using a static method, or referring to one of its class variables. You do not need to import Proximity classes to use their non-static methods.

The *prox* object

When Proximity runs a script or executes commands in the Proximity Python interpreter, it makes an instance of the `Proximity` class available. This instance is stored in the variable *`prox`*. The `Proximity` class and *`prox`* object provide access to the database itself as well as much of Proximity's higher level functionality including querying the database, creating samples, and instantiating models.

The *`prox`* object provides shortcuts to many important Proximity data structures. In particular, you can directly access the data structures containing database objects, links, attributes, and containers through the *`prox`* object:

Data structure	**Shortcut**
objects	`prox.objectNST`
links	`prox.linkNST`
object attributes	`prox.objectAttrs`
link attributes	`prox.linkAttrs`
root container	`prox.rootContainer`
arbitrary container	`prox.getContainer("`*`container-name`*`")`
container attributes	`prox.containerAttrs`

See the example scripts in this chapter and in Chapter 7, *Learning Models* for examples of using these shortcuts in Proximity scripts.

Running Proximity Scripts

Proximity provides two methods for running scripts:

- from the Proximity Database Browser
- from the command line

The exercises in this section demonstrate both methods of running scripts in Proximity. The script used in these exercises prints a “Hello world” greeting along with the connection information (host and port) for the database. See “Using the Proximity Python Interpreter” (p. 75) for information on using Proximity's interactive Python interpreter.

Code example: helloworld.py

This section describes the script found in `$PROX_HOME/doc/user/tutorial/examples/helloworld.py`.

Proximity scripts use the *`prox`* object to access the `Proximity` class's methods. The method `getDbName()` returns a string containing the connection information for the current database.

```
dbname = prox.getDbName()
print "Hello world, you're using database ", dbname
```

Exercise 6.1. Running a script from the Proximity Database Browser:

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. From the **Script** menu, choose **Run Script**. Proximity displays the Open dialog.
2. Navigate to the `$PROX_HOME/doc/user/tutorial/examples` directory and choose `helloworld.py`. Click **Open**.

   Proximity opens a window to show you any output from the script along with a trace of the script execution. Your output should look similar to the following:

```
Status: starting running script:
    /proximity/doc/user/tutorial/examples/helloworld.py
Hello world, you're using database  [localhost, 30000]
Status: finished running script
```

You can close this window after the script finishes.

You can create shortcuts to commonly used directories for easier access to script files. Shortcuts appear in the Shortcuts pane of the Open dialog. See “Creating a file or directory shortcut” (p. 9) for information on creating shortcuts.

You can also execute Python scripts directly from the command line using the Proximity **script.sh** shell script or **query.bat** batch file.

Exercise 6.2. Running a script from the command line:

Before beginning, make sure that you are serving the ProxWebKB database using Mserver.

➤ Execute the **script.sh** shell script or **script.bat** batch file, specifying

`host:port`	the MonetDB server’s host and port
`scriptName`	the file containing the Python script

Substitute the appropriate host and port information if you are running the MonetDB server on a different machine or are using a different port.

```
> cd $PROX_HOME
> bin/script.sh localhost:30000 \
  $PROX_HOME/doc/user/tutorial/examples/helloworld.py
```

Proximity writes the execution trace to the window from which you executed the above command. The last lines should be similar to the following excerpt (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

```
INFO  app.PythonScript: * connecting to db
INFO  app.PythonScript: * executing script:
      /proximity/doc/user/tutorial/examples/helloworld.py
Hello world, you're using database  [localhost, 30000]
INFO  app.PythonScript: * done executing script
```

Using the Proximity Python Interpreter

Proximity also lets you execute Python statements interactively from a special Python interpreter. The interpreter provides the same access to Proximity methods and to the special *prox* object available in the other scripting interfaces.

The interpreter is a complete Jython interpreter and behaves in much the same way as the command line Python interpreter:

- Variables defined during an interpreter session remain available for use in later commands until you close the interpreter window. New interpreter windows have a separate namespace; variables defined in one interpreter window are not available in another.
- Nested blocks are indented as usual. You must indent lines with tabs. As in other Python environments, all statements in a single block must be indented consistently.

Method name completion

The Proximity Python interpreter provides method name completion for command lines that begin with a variable name. Press Ctrl-Space to display a menu of possible methods for the current context. For example:

- Enter `prox.` followed by Ctrl-Space to show all the methods in the `Proximity` class. (Note the trailing period.)
- Enter `prox.add` followed by Ctrl-Space to show all the methods in the `Proximity` class that begin with `"add"`.
- Enter `prox.addA` followed by Ctrl-Space to complete the command line with the only `Proximity` class method that begins with `"addA"` (`addAttribute()`).

Method name completion works for variables other than the *`prox`* object and can be invoked recursively. For example, typing `prox.getContainer().` followed by Ctrl-Space displays the methods in the `Container` class.

Parameter lists

Use Ctrl-P (after an opening parenthesis) to display a menu of parameter overloads for the current method. For example, typing `prox.addAttribute(` followed by Ctrl-P shows the two signatures for the `addAttribute()` method.

Method name completion and parameter list display are enabled only for command lines that begin with a variable name. They do not work for variables elsewhere on the command line.

Command line history and editing

The Proximity Python interpreter also includes selected functionality for working with the command-line history and editing the current command line. The key bindings for command-line editing in the Proximity Python interpreter follow those used for command-line editing in the bash and csh shells.

Command	Description	Command	Description
Up arrow	Previous command in history	Down arrow	Next command in history
Right arrow, Ctrl-B	Move backward one character	Left arrow, Ctrl-F	Move forward one character
Ctrl-A	Move to beginning of line	Ctrl-E	Move to end of line
Del	Delete one character backward	Ctrl-D	Delete one character forward
Ctrl-K	Kill to end of line	Ctrl-Y	Paste contents of clipboard
Ctrl-Alt-S[a]	Save command history	Ctrl-Alt-C	Clear command history
Ctrl-Alt-X	Execute Jython file		

[a] The Mac OS X equivalent to Ctrl-Alt is Ctrl-Option.

After you have developed a sequence of commands for accomplishing a task, you can save the command history to a file for future use.

- Use Ctrl-Alt-S to save the current command history to a file.
- Use Ctrl-Alt-C to clear the command history. (Ctrl-Alt-C does not clear the interpreter window or change any variable values.)
- Use Ctrl-Alt-X to execute a Jython file, such as a saved history file.

The Proximity Python interpreter runs in the same thread as the Proximity Database Browser. Statements that take a long time to execute will prevent interacting with the browser until execution finishes.

Code Example: Finding Selected Objects

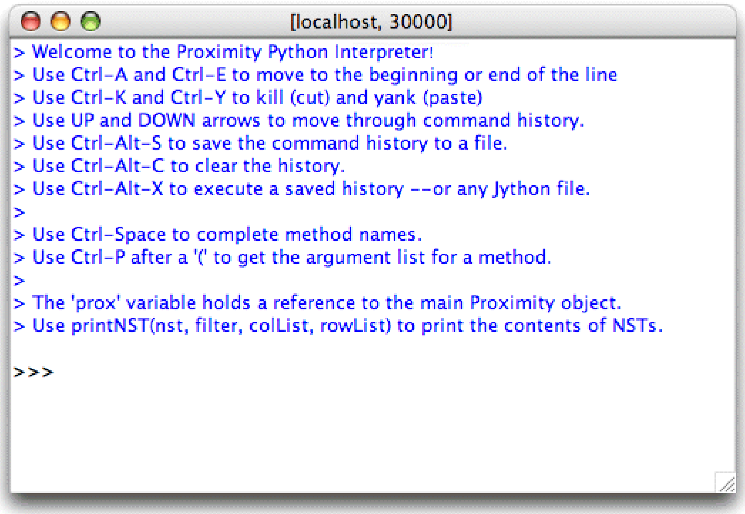
This exercise walks through a series of interactive Python commands that find objects with a specified attribute value. In this example, we find and explore objects with a value of ftp for the url_protocol attribute. The steps described below also show how to use the interpreter's history, command-line editing, and completion features.

This exercise illustrates two methods in the `Proximity` class useful for exploring the information in a database: `find()` and `browse()`. The `find()` method searches for objects whose attribute values match the specified pattern. This pattern accepts the wildcards _ (underscore, matches a single instance of any character), and `%` (matches zero or more characters). The `browse()` method opens a new browser window to display the specified object or container.

Exercise 6.3. Running a script interactively:

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. If the interactive Python interpreter window is not already open, choose **Open Interpreter** from the **Script** menu. Proximity opens a Proximity Python interpreter window.

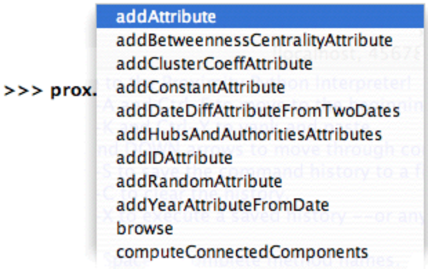


```
> Welcome to the Proximity Python Interpreter!
> Use Ctrl-A and Ctrl-E to move to the beginning or end of the line
> Use Ctrl-K and Ctrl-Y to kill (cut) and yank (paste)
> Use UP and DOWN arrows to move through command history.
> Use Ctrl-Alt-S to save the command history to a file.
> Use Ctrl-Alt-C to clear the history.
> Use Ctrl-Alt-X to execute a saved history --or any Jython file.
>
> Use Ctrl-Space to complete method names.
> Use Ctrl-P after a '(' to get the argument list for a method.
>
> The 'prox' variable holds a reference to the main Proximity object.
> Use printNST(nst, filter, colList, rowList) to print the contents of NSTs.

>>>
```

The initial Proximity Python interpreter window includes reminders of the keyboard commands for using the interpreter's history, command-line editing, and completion features. (The `printNST` method operates on NSTs, an internal Proximity data structure described in "Working with Proximity Tables".

2. Use the `find()` method to identify objects with an attribute value of ftp.

   a. In the Proximity Python interpreter, type **`prox.`** and press Ctrl-Space. (Note the trailing period.) Proximity displays a menu of applicable methods in the `Proximity` class.



```
>>> prox.
```

addAttribute
addBetweennessCentralityAttribute
addClusterCoeffAttribute
addConstantAttribute
addDateDiffAttributeFromTwoDates
addHubsAndAuthoritiesAttributes
addIDAttribute
addRandomAttribute
addYearAttributeFromDate
browse
computeConnectedComponents

   b. Choose `find` from the menu. Proximity adds the method name to the current command line.

c. Complete the method call by entering the pattern to match:

```
prox.find("%ftp%")
```

The `%` wildcards before and after `ftp` each match zero or more characters; this pattern matches all attribute values that contain the string `"ftp"` anywhere in the value.

d. Press Return.

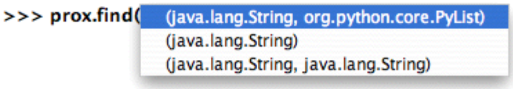
Proximity returns a map (Python dictionary) with attribute names as keys and a list of object IDs that match the specified pattern for that attribute as the value for each key.

```
{url=[1086, 1087, 1088, 1089, 1091, 1092, 1093, 1094, 1095, 1096, 1097, 1098,
1099, 1100, 1101, 1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111,
1112, 1113, 1114, 1115, 1116, 1117, 1118, 1119, 1120, 1121, 1122, 3448],
url_protocol=[1091, 1092, 1093, 1094, 1095, 1096, 1097, 1098, 1099, 1100, 1101,
1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113, 1114,
1115, 1116, 1117, 1118, 1119, 1120, 1121, 1122], url_server_info=[1086, 1087,
1088, 1089, 1091, 1092, 1093, 1094, 1095, 1096, 1097, 1098, 1099, 1100, 1101,
1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113, 1114,
1115, 1116, 1117, 1118, 1119, 1120, 1121]}
```

The returned value shows that three attributes (url, url_protocol, and url_server_info) have values that contain the string `"ftp"`.

3. Edit the previous command to find only those objects whose url_protocol attribute has a value of ftp. We use an overload of the `find()` method that accepts a second argument, the name of the attribute to search.

   a. Press the up arrow to display the previous command.
   b. Use the arrow keys to change the cursor position so that it is immediately after the opening parenthesis.
   c. Press Ctrl-K to delete all text from the cursor position to end of line. This text is placed on the clipboard and can be pasted with Ctrl-Y.
   d. Press Ctrl-P to display a menu of parameter options for the `find()` method.



```
>>> prox.find(
```

(java.lang.String, org.python.core.PyList)
(java.lang.String)
(java.lang.String, java.lang.String)

   This time we use the `find()` method that takes two strings as arguments. Press Return or Esc to clear the parameter list menu.

   e. Complete the command so that it now reads

```
prox.find("ftp","url_protocol")
```

   Note that this time we are searching for exact matches as the search pattern has no wildcards.

   f. Press Return.

Proximity returns a map with the target attribute as the key and the list of object IDs with the specified attribute value as the value for that key.

```
{url_protocol=[1091, 1092, 1093, 1094, 1095, 1096, 1097, 1098, 1099, 1100, 1101,
1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113, 1114,
1115, 1116, 1117, 1118, 1119, 1120, 1121, 1122]}
```

4. Create and execute a file containing the most recent `find` command.

   a. Press Ctrl-Alt-S to save the command history. Proximity displays the Open dialog. Save the history in the location of your choice.
   b. Edit the saved file to remove the first `find` command and assign the results of the remaining `find` command to the variable *`ftp_protocols`*. The file should now read

```
# Proximity Interpreter History
ftp_protocols=prox.find("ftp","url_protocol")
```

   Save the edited file.

c. In the Proximity Python interpreter, press Ctrl-Alt-C to clear the command history. Proximity displays a confirmation message in a pop-up window. Click **OK** to dismiss the confirmation message.

d. Press Ctrl-Alt-X to execute a saved Jython file. Proximity displays the Open dialog. Choose the file you just saved and edited. Proximity responds by printing and executing the `execfile` command that runs the saved script file:

```
>>> execfile("/proximity/saved-interpreter-history.py")
```

e. To verify that the script completed successfully, enter **`ftp_protocols`** in the Proximity Python interpreter window and press Return. Proximity returns the value of *`ftp_protocols`*.

5. Examine one of the listed objects.

   a. In the Proximity Python interpreter window, enter **`prox.b`** and press Ctrl-Space. Proximity completes the method name (`browse`).

   b. Enter **`(`** and press Ctrl-P to display the parameter options for the `browse()` method. This method takes either an `int` (object ID) or `String` (container name) as its argument. Press Return or Esc to dismiss the list of parameter options.

   c. Complete the command line to read

   ```
   prox.browse(1091)
   ```

   and press Return.

   Proximity displays object 1091 in a new browser window.

6. Click **attrs** to display the attribute values for object 1091 and confirm that the value of the url_protocol attribute is ftp.

Close the second browser window when you have finished exploring object 1091.

Sampling the Database

To use the Proximity models, you need to create sets of subgraphs. In the simplest case, one or more of these sets is used to learn (train) the model. You then test the results of that training by applying the model to the remaining set of subgraphs. You can create these sets of subgraphs by *sampling* a container.

Proximity provides a `sampleContainer()` method for creating random samples of existing containers.[1] You can either create a specified number of samples, or create samples suitable for cross validation. Because subgraphs in a container may contain objects and links that also appear in other subgraphs, the resulting samples may not satisfy strict independence requirements imposed by some experimental designs. Exercise 6.4 illustrates the simplest case of creating two samples—one for training and one for testing.

> Running the sample-1d-clusters script creates containers used by later tutorial exercises. Make sure that you complete this exercise if you plan to complete the exercises in the following chapters.

Code example: sample-1d-clusters.py

This section describes the script found in `$PROX_HOME/doc/user/tutorial/examples/sample-1d-clusters.py`.

To create the samples, we need to write a script that calls the `sampleContainer()` method on the *`prox`* object. For this example, we create two samples. Each will be placed in a new container and each will hold approximately one-half the subgraphs from the original container. The counts are approximate

[1] You cannot sample the root container.

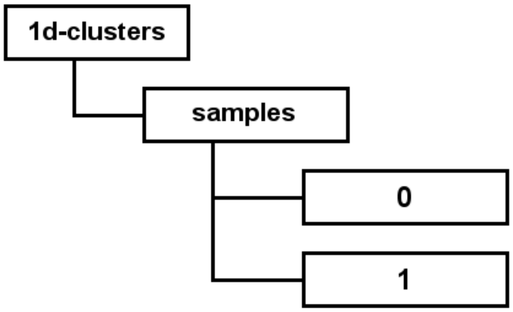
because the initial container may not contain an even number of subgraphs.

```
print "Sampling database..."
prox.sampleContainer("1d-clusters",2,"samples")
```

This use of the `sampleContainer()` method requires three arguments:

containerName	the container to be sampled
numFolds	the number of samples to create from this container
sampleName	the name of the container that holds the sample containers

The `sampleContainer()` method creates both a parent container, *sampleName*, to hold the samples as well as the sample containers themselves. The *sampleName* container is in turn, contained by the initial *containerName* container. For example, the script above creates the following container hierarchy:



where the 1d-clusters container existed in the database before running the scripts and the samples, 0, and 1 containers were created as a result of running the script.

The `sampleContainer()` method provides an overload for creating samples for cross validation. This overload omits the *sampleName* parameter, placing the resulting samples in either a training or test container. For example, when creating samples for 10-fold cross validation, the method partitions the data into 10 segments and creates 10 training sets, each containing nine of those segments, and 10 corresponding test sets containing the remaining tenth segment.

Exercise 6.4. Creating training and test sets:

This script requires the container created in Exercise 5.7. You must have completed Exercise 5.7 before running the current exercise.

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

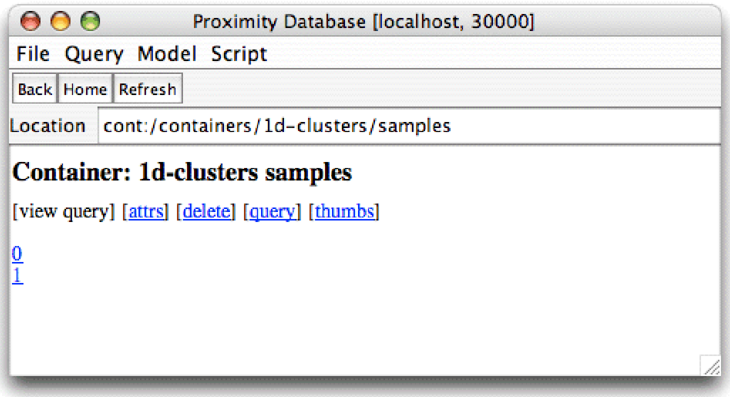
1. From the **Script** menu, choose **Run Script**. Proximity displays the Open dialog.
2. Navigate to the `$PROX_HOME/doc/user/tutorial/examples` directory and choose `sample-1d-clusters.py`. Click **Open**.

   Proximity opens a window to show you any output from the script along with a trace of the script execution. Proximity reports:

   ```
   Status: starting running script:
      /proximity/doc/user/tutorial/examples/sample-1d-clusters.py
   Sampling database...
   Status: finished running script
   ```

   when the script successfully completes. You can close this window after the script finishes.
3. To verify that the script created the sample containers, click **Containers** to display the list of containers in the database.

4. Click **1d-clusters**. Proximity shows you that 1d-clusters has a child container, samples.
5. Click **samples**. Proximity displays samples's child containers, numbered 0 and 1.



Location cont:/containers/1d-clusters/samples

Container: 1d-clusters samples

[view query] [attrs] [delete] [query] [thumbs]

0
1

The **view query** link is disabled because this container was created by a script rather than by querying.

Sampling adds an attribute to each of the sampled subgraphs *in the initial container* indicating which sample they belong to. In this case, subgraphs in 1d-clusters get the new attribute, but corresponding subgraphs in containers 0 and 1 do not. This attribute will be overwritten if the container is re-sampled.

Adding a New Attribute

Sometimes a database doesn't include the necessary representation required for a specific task. For example, one of the classification models used in Chapter 7, *Learning Models* uses a boolean attribute to identify members of a class. The ProxWebKB database doesn't include such an attribute, so we must create one based on other existing attribute values. Although you can create attributes directly in the Proximity Database Browser, it is sometimes more convenient to do so as part of a script. This section describes how to create a new attribute and use a function to assign values to that new attribute in a script. The syntax for the assignment function is identical, regardless of whether the attribute is created in a script or directly in the Proximity Database Browser.

In this example, we create an attribute needed for later *Tutorial* exercises: a binary isStudent object attribute that indicates whether or not a web page belongs to a student. We assign the value 1 to the isStudent attribute if the page's pagetype attribute is Student, and 0 otherwise.

Creating a useful attribute requires two steps: creating the appropriate database tables and populating its values. The `Proximity` class's `addAttribute()` method lets you accomplish both of these steps in one method call and provides great flexibility in how you assign values to the new attribute.

To create and populate a new attribute, you specify a function that describes how to assign values to the new attribute for each entity. New attribute values can be constants or can be based on existing attribute values. In this example, we use the value of the existing pagetype attribute to determine the value of the new isStudent attribute. This function is specified as a string passed into the `addAttribute()` method. You can create attributes for any class of database entity—objects, links, containers, or subgraphs. See the `AddAttribute` class for a full description of the syntax for specifying the creation function.

> You can also create a new attribute directly in the Proximity Database Browser. Both methods use the same syntax to define the function that assigns values to the new attribute. See Exercise 4.2 for information on creating attributes in the Proximity Database Browser.

Code example: create-isStudent-attr.py

This section describes the script found in `$PROX_HOME/doc/user/tutorial/examples/create-isStudent-attr.py`.

Get the current set of object attributes and define the new attribute name. We use the set of current attributes to determine whether the new attribute already exists in the database.

```
currentAttrs = prox.objectAttrs
newAttrName = "isStudent"
```

Define the function that determines the new attribute value. This example uses a switch function. It sets the value of the new attribute to `1` if the object's pagetype attribute is equal to Student and `0` otherwise. See the `AddAttribute` class for details on specifying function strings for defining new attributes.

```
newAttrFunction = "pagetype = \"Student\" ==> 1, default ==> 0"
```

See if this object attribute already exists in the database. If it does, ask the user if we should delete it so we can redefine it. Proximity provides two methods, `getYesNoFromUser()` and `getStringFromUser()`, both members of the `Proximity` class, that you can use to interact with your Python programs. Both methods display a dialog with the specified prompt. The `getYesNoFromUser()` method returns `true` if the user selects Yes and `false` if the user selects No. The `getStringFromUser()` method returns the string entered by the user.

```
prompt = "Attribute already exists. Delete and recreate?"

if currentAttrs.isAttributeDefined(newAttrName):
   deleteExisting = prox.getYesNoFromUser(prompt)
```

If yes, delete the existing attribute and create it again.

```
   if deleteExisting:
      currentAttrs.deleteAttribute(newAttrName)
      print "Creating new ", newAttrName, " attribute"
      prox.addAttribute(currentAttrs,newAttrName,newAttrFunction)
```

If the attribute does not already exist, go ahead and create it.

```
else:
   print "Creating new ", newAttrName, " attribute"
   prox.addAttribute(currentAttrs,newAttrName,newAttrFunction)
```

> Running the `create-isStudent-attr.py` script creates an attribute used by later tutorial exercises. Make sure that you complete this exercise if you plan to complete the exercises in Chapter 7, *Learning Models*.

Exercise 6.5. Adding a new attribute:

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. From the **Script** menu, choose **Run Script**. Proximity displays the Open dialog.
2. Navigate to the `$PROX_HOME/doc/user/tutorial/examples` directory and choose `create-isStudent-attr.py`. Click **Open**.

   Proximity opens a window to show you any output from the script along with a trace of the script execution. Proximity reports:

   ```
   Status: starting running script:
      /proximity/doc/user/tutorial/examples/create-isStudent-attr.py
   Creating new  isStudent  attribute
   Status: finished running script
   ```

You can close this window after the script finishes.

3. Click **Object attributes** to see the new isStudent attribute.

Social Networking Algorithms

Social networking algorithms are a useful part of the analyst's toolkit. The algorithms illustrated in this section help in understanding the structure of a database.

The clustering coefficient is a measure of the interrelatedness of an object's neighbors. It indicates the ratio of existing links connecting an object's neighbors to each other to the maximum possible number of such links. An object must have at least two neighbors to calculate the clustering coefficient. No value is calculated for objects having less than two neighbors.

The connected components algorithm identifies the distinct components of the input data. Two objects are in the same component if there is a path from one object to the other.

The hubs and authorities algorithm, originally developed with web search in mind, attempts to identify objects that are likely to be good sources of information (authorities) and objects that are likely to point to many such authorities (hubs) through the link structure of the data.

Proximity adds the results of running these algorithms to the database, either by creating new object attributes (clustering coefficient or hubs and authorities) or by creating a new container of subgraphs (connected components).

Code example: social-networking-algs.py

This section describes the script found in `$PROX_HOME/doc/user/tutorial/examples/social-networking-algs.py`.

Initialization

Define a prefix to use for attribute and container names. For example, when running these algorithms on the whole database, you might use the database name as a prefix (as shown below). You might use a container name when running the algorithms on the objects and links in a specified container.

```
attrPrefix = "proxwebkb"
```

Get the current set of object attributes. We use this to determine whether the new attributes already exist in the database.

```
currentAttrs = prox.objectAttrs
```

Clustering coefficient

For the purposes of calculating the clustering coefficient, Proximity ignores both link direction and the possibility of multiple links when calculating the number of actual and potential links among an object's neighbors. Proximity uses only the number of distinct object pairs to determine this number.

```
print "Computing clustering coefficients"
```

Proximity stores the value of the clustering coefficient in a new object attribute specified by the user. We start by specifying the name of the new attribute.

```
clusterAttrName = attrPrefix + "-cluster-coeff"
```

In this example, we assume that the user wants to overwrite existing values so we automatically delete any existing attribute with the same name. See Exercise 6.5 for an example of asking the user whether to delete an existing attribute.

```
if currentAttrs.isAttributeDefined(clusterAttrName):
   print "  Attribute " + clusterAttrName + " already exists. Deleting..."
   currentAttrs.deleteAttribute(clusterAttrName)
```

Create and calculate values for the new clustering coefficient attribute. Proximity lets you calculate the clustering coefficient for either an entire database or limit execution to the data in a specified container. When limited to a container, Proximity runs the algorithm on the objects and links in the container, ignoring subgraph boundaries. The code in this example calculates clustering coefficients for the entire database.

```
prox.addClusterCoeffAttribute(clusterAttrName)
```

To compute the clustering coefficient for objects in a container, use the overloaded `addClusterCoeffAttribute()` method with the input container as the second argument, e.g., `prox.addClusterCoeffAttribute(clusterAttrName, inputContainer)`.

Connected components

Proximity ignores link direction in determining whether a path exists between two objects, implementing the weak sense of connectivity. Proximity creates a subgraph for each component and places all component subgraphs in a new container.

```
print "Computing connected components"
```

Define the name of the new container. Each subgraph in this container is a connected component. The new container will be a child of the root container.

```
outputContainerName = attrPrefix + "-connected-components"
```

Delete this container if it already exists

```
rootContainer = prox.rootContainer
if rootContainer.hasChild(outputContainerName):
   print "  Container " + outputContainerName + " already exists. Deleting..."
   rootContainer.deleteChild(outputContainerName)
```

Find the connected components. Proximity lets you calculate the connected components for either an entire database or limit execution to the data in a specified container. When limited to a container, Proximity runs the algorithm on the objects and links in the container, ignoring subgraph boundaries. The code in this example identifies connected components for the entire database.

```
prox.computeConnectedComponents(outputContainerName)
```

To find connected components within a container, use the overloaded method `computeConnectedComponents()` method with the input container as the second argument, e.g., `prox.computeConnectedComponents(outputContainerName, inputContainer)`.

Hubs and authorities

Due to the circular relationship between hubs and authorities, we use an iterative algorithm for calculating these weights. Proximity’s implementation of the hubs and authorities algorithm is based on work by Kleinberg [Kleinberg, 1998], which suggests that 20 iterations is usually sufficient for most applications.

The `addHubsAndAuthoritiesAttributes()` method adds two attributes to the input objects, an authority weight and hub weight. Both weights are non-negative and normalized so that the values for each attribute sums to 1.0. Larger authority or hub weights indicate that the corresponding object is a better authority or hub, respectively.

```
print "Computing hubs and authorities"
```

Specify the name of the new attributes.

```
hubsAttrName = attrPrefix + "-hubs"
authoritiesAttrName = attrPrefix + "-authorities"
```

Delete these attributes if they already exist.

```
if currentAttrs.isAttributeDefined(hubsAttrName):
   print "  Attribute " + hubsAttrName + " already exists. Deleting..."
   currentAttrs.deleteAttribute(hubsAttrName)
if currentAttrs.isAttributeDefined(authoritiesAttrName):
   print "  Attribute " + authoritiesAttrName + " already exists. Deleting..."
   currentAttrs.deleteAttribute(authoritiesAttrName)
```

Specify the number of iterations.

```
numIterations = 20
```

Create and calculate values for the new hubs and authorities attributes. Proximity lets you calculate the hubs and authorities scores for either an entire database or limit execution to the data in a specified container. When limited to a container, Proximity runs the algorithm on the objects and links in the container, ignoring subgraph boundaries. The code in this example calculates hubs and authorities scores for the entire database.

```
prox.addHubsAndAuthoritiesAttributes(numIterations, hubsAttrName,
     authoritiesAttrName)
```

To compute hubs and authorities scores for objects in a container, use the overloaded method `addHubsAndAuthoritiesAttributes()` with the input container as the fourth argument, e.g., `prox.addHubsAndAuthoritiesAttributes(numIterations, hubsAttrName, authoritiesAttrName, input Container)`.

Exercise 6.6. Running social networking algorithms:

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. From the **Script** menu, choose **Run Script**. Proximity displays the Open dialog.
2. Navigate to the `$PROX_HOME/doc/user/tutorial/examples` directory and choose `social-networking-algs.py`. Click **Open**.

   Proximity opens a window to show you any output from the script along with a trace of the script execution. Note that this script may take several minutes to run. Proximity reports:

   ```
   Status: starting running script:
     /proximity/doc/user/tutorial/examples/social-networking-algs.py
   Computing clustering coefficients
   Computing connected components
   Computing hubs and authorities
   Status: finished running script
   ```

   You can close this window after the script finishes.
3. Click **Object Attributes** to see the new attributes created by the clustering coefficient and hubs and authorities calculations. If you wish, you can explore these attributes further.
   - The proxwebkb-cluster-coeff attribute records the clustering coefficient for each input object having two or more neighbors in the input data. The clustering coefficient is a ratio, thus all values lie between 0 and 1, inclusive.
   - The proxwebkb-authorities attribute records the authority weight calculated for each input object.
   - The proxwebkb-hubs attribute records the hub weight calculated for each input object.
4. Return to the Proximity Database Browser start page and click **Containers** to see the new proxwebkb-connected-components container created by the connected components method.
5. Click **proxwebkb-connected-components** to see the list of subgraphs in this container. Proximity found 17 distinct components in the ProxWebKB data. If you wish, you can explore these subgraphs further.

Working with Proximity Tables

The database engine underlying Proximity, MonetDB, uses a *vertically fragmented* design. This design stores all data in two-column tables called BATs (binary association tables). All BATs include a *head* column that contains an identifier (OID) and a *tail* column that contains data. The contents of the tail can also point to other tables. Thus, rather than the object-centric view provided by familiar relational databases where an entity's attributes are stored in the columns of a table, Proximity uses separate tables for each attribute. Other tables store the Proximity ID for each object or link in the database. Each attribute value is associated with the appropriate entity through the corresponding ID.

Proximity's vertically fragmented design makes some database operations more or less efficient than they would be in a traditional relational database. Operations required by common knowledge discovery tasks, such as the joins required for finding subgraphs matching a query, are significantly faster than they would be in a common relational database. Object- and link-centric operations, such as finding all the attribute values for a specified object, are less efficient.

To enable a more intuitive view of the data, Proximity provides an intermediate representation called *nested synchronized tables* (NSTs) that combines the data from several related BATs into a single, multi-column table. Proximity includes an NST browser for direct examination of these data structures. For many tasks, using NSTs provides the best mechanism for working with data in Proximity databases.

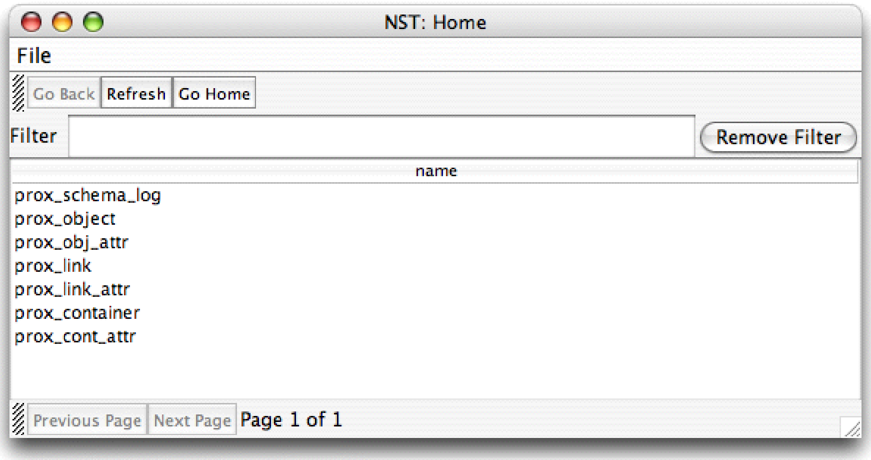
The exercise below illustrates how to use NSTs and filtering to identify objects and links based on attribute values—in this case, to find all links from Wisconsin staff web pages to Wisconsin faculty pages. More specifically, we first identify all the objects that correspond to Wisconsin staff pages (all objects where the value of the school attribute equals Wisconsin and the value of the pagetype attribute equals Staff). We then similarly identify the objects that correspond to Wisconsin faculty pages. Finally, we find all links that originate from an object in the first set and terminate at an object in the second set.

To better understand the NSTs used in this example, the following exercise uses the Proximity interactive Python interpreter rather than a Python script, allowing you to pause to examine the relevant NSTs as you proceed. The exercise begins by exploring the standard Proximity NSTs and illustrating how to use the NST browser features. This is followed by the steps needed to create the NSTs that identify the restricted sets of objects and links listed above.

Exercise 6.7. Finding specific links:

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. From the Proximity Database Browser start page, click **browse tables** to browse the built-in Proximity NSTs. Proximity opens the NST browser.



name
prox_schema_log
prox_object
prox_obj_attr
prox_link
prox_link_attr
prox_container
prox_cont_attr

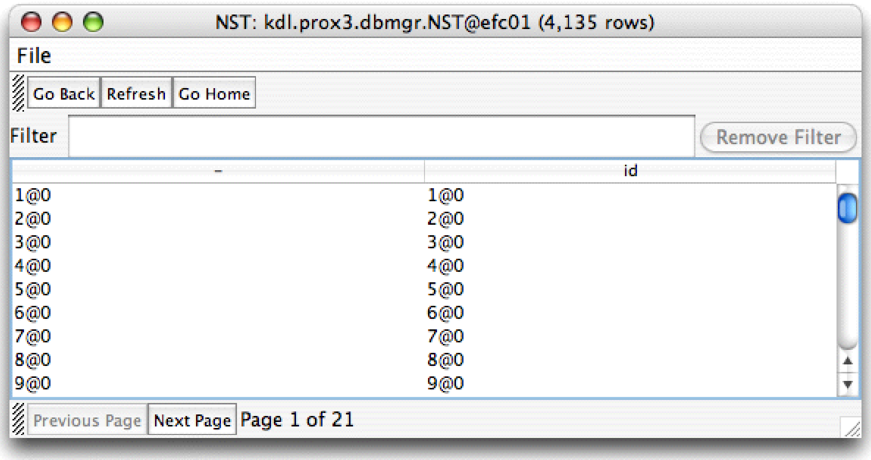
The Proximity NST browser lets you directly examine Proximity data as represented by NSTs. Every Proximity database uses several standard NSTs for storing objects, links, containers, and attributes for these entities. A Proximity database can also have an arbitrary number of user-created NSTs that typically store subsets or joins of existing NSTs. The Proximity NST browser includes a menu bar, top button bar for navigation, filter text box, bottom button bar for paging through long lists, and a large area used to display the data in the selected NST.

The **File** menu's single command, **Export to File**, writes the data for the current NST to a tab-delimited text file for use in other applications. Exported NST data does not include the head column (OIDs), and the trailing `@0` is omitted from ID data.

The initial NST browser display shows the standard NSTs included in all Proximity databases:

- The prox_object NST stores the IDs of objects in the database.
- The prox_obj_attr NST stores the name, data type, and pointers to tables that contain the values for object attributes.
- The prox_link NST stores the IDs of the links and the IDs of the objects they connect to and from (recall that all links in Proximity are directed).
- The prox_link_attr NST stores the name, data type, and pointers to tables that contain the values for link attributes.
- The prox_container NST stores the ID, name, and parent container for all the containers in the database, plus pointers to tables that contain the data (objects, links, and attributes) for those subgraphs.
- The prox_cont_attr NST stores the name and data type for all the container attributes in the database, plus pointers to tables that contain the values for those attributes

2. Double-click **prox_object**. Proximity displays the NST that stores object IDs.



NST: kdl.prox3.dbmgr.NST@efc01 (4,135 rows)

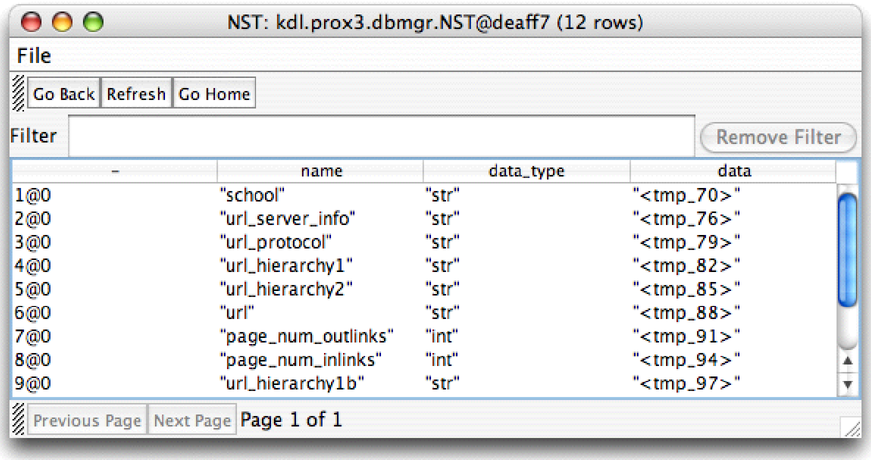
-	id
1@0	1@0
2@0	2@0
3@0	3@0
4@0	4@0
5@0	5@0
6@0	6@0
7@0	7@0
8@0	8@0
9@0	9@0

Page 1 of 21

The `prox_object` NST includes two columns: the head column (labeled "`-`"), which stores the OIDs for each object, and an `id` column, which stores the object IDs for each object.

The OID is an internal value used by Proximity; the ID is a visible value that you can use to refer to specific objects in the database. Although they are identical in this case, the OID and ID values need not match.

3. Click **Go Back** and then double-click **prox_obj_attr**. Proximity displays the NST that stores object attribute data.

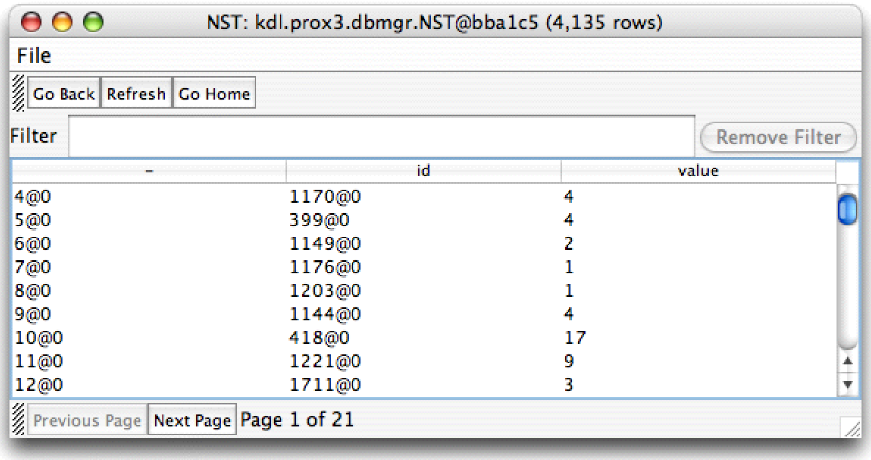


NST: kdl.prox3.dbmgr.NST@deaff7 (12 rows)

-	name	data_type	data
1@0	"school"	"str"	"<tmp_70>"
2@0	"url_server_info"	"str"	"<tmp_76>"
3@0	"url_protocol"	"str"	"<tmp_79>"
4@0	"url_hierarchy1"	"str"	"<tmp_82>"
5@0	"url_hierarchy2"	"str"	"<tmp_85>"
6@0	"url"	"str"	"<tmp_88>"
7@0	"page_num_outlinks"	"int"	"<tmp_91>"
8@0	"page_num_inlinks"	"int"	"<tmp_94>"
9@0	"url_hierarchy1b"	"str"	"<tmp_97>"

The prox_obj_attr NST includes four columns: the head column, plus columns storing the attribute name, data type, and a `data` column that contains a pointer to the table containing the values for this attribute. Each of these attribute value tables is another NST. To see the values for an attribute, double-click the name of the corresponding attribute value table.

4. Double-click the name of the attribute value table for the page_num_inlinks attribute (the name may vary but will start with `tmp`, e.g., <tmp-94>). Proximity displays the table of values for the page_num_inlinks attribute.

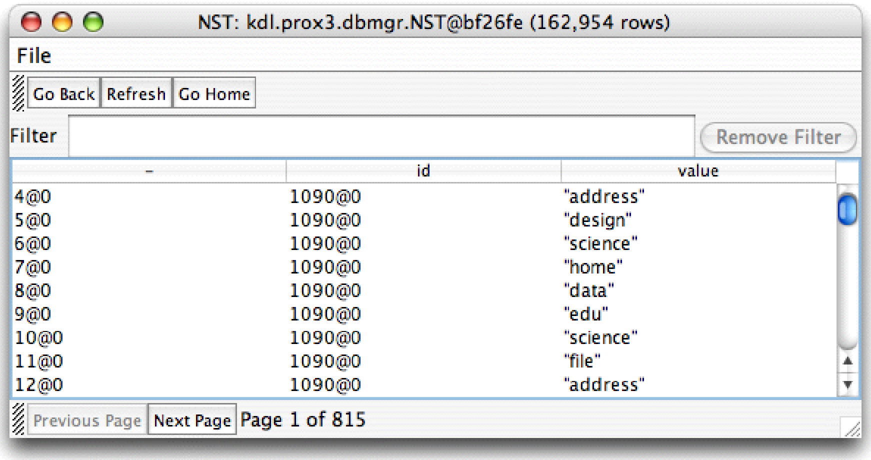


NST: kdl.prox3.dbmgr.NST@bba1c5 (4,135 rows)

-	id	value
4@0	1170@0	4
5@0	399@0	4
6@0	1149@0	2
7@0	1176@0	1
8@0	1203@0	1
9@0	1144@0	4
10@0	418@0	17
11@0	1221@0	9
12@0	1711@0	3

Attribute value tables have three columns: the head column, an `id` column that stores the ID of a database entity (object, link, container, or subgraph) and a `value` column that stores the value of the attribute for that entity. Note that in this case the values in the `id` column do not match the OIDs in the head column.

5. Click **Go Back** and then double-click **<tmp-100>**. Proximity displays the table of values for the page_words_top100 attribute.



-	id	value
4@0	1090@0	"address"
5@0	1090@0	"design"
6@0	1090@0	"science"
7@0	1090@0	"home"
8@0	1090@0	"data"
9@0	1090@0	"edu"
10@0	1090@0	"science"
11@0	1090@0	"file"
12@0	1090@0	"address"

Page 1 of 815

In contrast to the previous attribute NST, the table of values for page_words_top100 contains multiple rows with the same `id` value. This indicates that page_words_top100 is a set-valued attribute—objects in the database may have multiple values for this attribute. And as we can see in the table shown above, this set can include the same value (e.g., "`science`") more than once. Proximity entities may have zero, one, or many values for a specified attribute.

6. Click **Go Back** and then double-click **<tmp-70>**. Proximity displays the table of values for the school attribute.

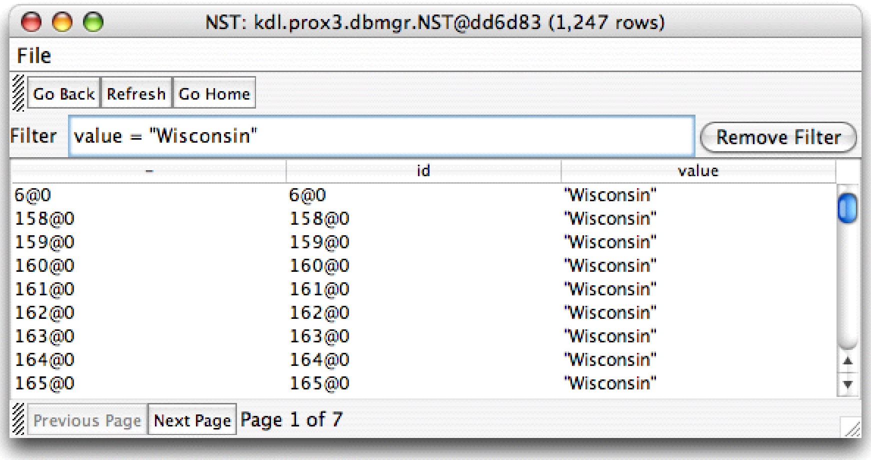
In the Filter box, enter

```
value = "Wisconsin"
```

and press Return.

Filters are a set of conditions that rows in the NST must match. For example, a common use for filters is identifying all objects having a specified attribute value. Filters can be used in Python scripts or entered directly in the NST browser's Filter text box.

This filter limits the displayed rows to those that have Wisconsin in the value column. The NST browser now shows only the objects that have this attribute value.



Filter: value = "Wisconsin"

-	id	value
6@0	6@0	"Wisconsin"
158@0	158@0	"Wisconsin"
159@0	159@0	"Wisconsin"
160@0	160@0	"Wisconsin"
161@0	161@0	"Wisconsin"
162@0	162@0	"Wisconsin"
163@0	163@0	"Wisconsin"
164@0	164@0	"Wisconsin"
165@0	165@0	"Wisconsin"

Page 1 of 7

Filters specify a set of conditions that rows in the NST must match. You can filter an NST based on values in any of the non-head columns in that NST.

The general form of a filter is

```
column-name operator value
```

where

- *column-name* is the name of a column in the NST. Although we typically filter on attribute values, we can create a filter on any non-head column in the NST.
- *operator* is one of the operators listed in the `FilterFactory` class. In addition to the familiar comparison operators (=, !=, <, <=, >, and >=), Proximity provides common database filter operators such as `DISTINCT ROWS` and `LIKE`. See the Javadoc for this class for a complete list of available operators.
- *value* is a specific value for this attribute. Note that string values must be quoted. Some operators, such as `DISTINCT ROWS`, do not require a *value* parameter.

7. If it is not already available, open the interactive Python interpreter by choosing **Open Interpreter** from the **Script** menu.
8. In the interactive Python interpreter, enter

   ```
   wiscPages = DB.getObjects("school = 'Wisconsin'")
   ```

   The `getObjects()` method requires a single string argument that specifies the desired filter. Note that because the filter requires a string *value*, `'Wisconsin'`, it requires single quotes in addition to the double quotes surrounding the complete filter string.

   Proximity creates an NST named `wiscPages` containing the same set of objects as those listed in the filtered NST from Step 6.

   ⚠ Although filters resemble conditions, they have different semantics. Conditions operate on objects or links whereas filters operate on NST rows. A database entity satisfies a condition only if it has a value for the relevant attribute that satisfies the condition; entities having no value for the attribute cannot satisfy the condition. Filters pass all rows that satisfy the filter.

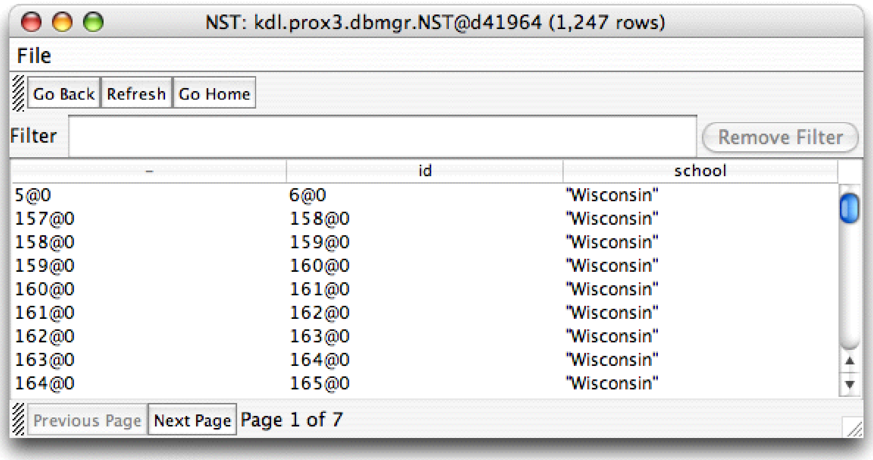
   For example, if some but not all objects have a value for the school attribute, only those objects having a value can satisfy the inequality condition school < > UTexas. In contrast, a similar-looking filter, `school != UTexas`, admits any NST row that does not have UTexas as a value. When applied to the full objects NST, both objects having a different value for the school attribute as well as objects having no value pass through the filter.

   Add *`attribute`* `!= nil` to a filter to omit entities having no value for the specified attribute.

9. In the interactive Python interpreter, enter

   ```
   prox.browse(wiscPages)
   ```

   Proximity opens another NST browser window that shows you the contents of the `wiscPages` NST you just created.



-	id	school
5@0	6@0	"Wisconsin"
157@0	158@0	"Wisconsin"
158@0	159@0	"Wisconsin"
159@0	160@0	"Wisconsin"
160@0	161@0	"Wisconsin"
161@0	162@0	"Wisconsin"
162@0	163@0	"Wisconsin"
163@0	164@0	"Wisconsin"
164@0	165@0	"Wisconsin"

Although the `wiscPages` NST contains the same information as the filtered school NST, these NSTs are not identical. Specifically, the values in the head column are different across the two NST browser windows. Remember to use the value in the `id` column, not the OID in the head column, when you need the ID of a database entity.

You can also use the `browse()` method to examine the standard Proximity NSTs. Use the "shortcuts" to these tables described in "The *`prox`* object" (p. 74) as the argument to `browse()`. For example, to display the objects NST in a new NST browser window, enter

```
prox.browse(prox.objectNST)
```

Note that the shortcut `prox.objectAttrs` returns an `Attributes` object; therefore, to browse one of the standard attribute NSTs use the form

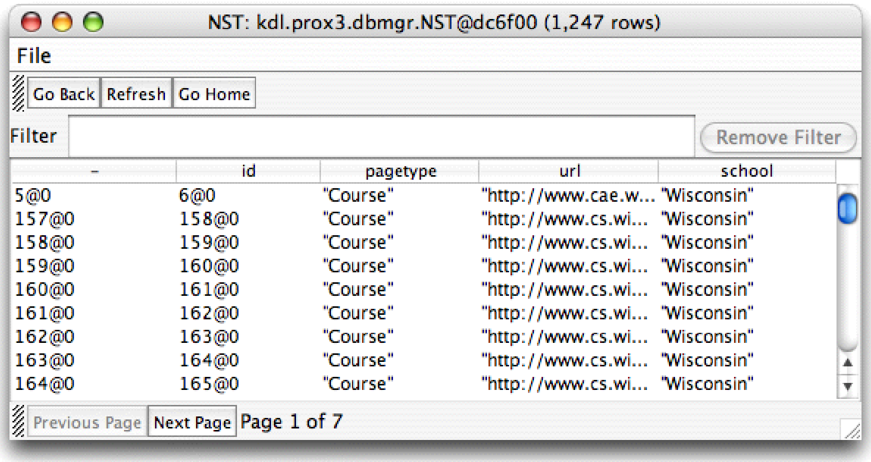
```
prox.browse(prox.objectAttrs.getAttrNST())
```

to get the NST for this data.

10. In the interactive Python interpreter, enter

```
wiscPages2 = DB.getObjects("school = 'Wisconsin'","pagetype, url")
prox.browse(wiscPages2)
```

Proximity creates and displays an NST named wiscPages2 containing columns for the pagetype and url attributes in addition to the head, `id`, and `value` columns we saw before.



-	id	pagetype	url	school
5@0	6@0	"Course"	"http://www.cae.w...	"Wisconsin"
157@0	158@0	"Course"	"http://www.cs.wi...	"Wisconsin"
158@0	159@0	"Course"	"http://www.cs.wi...	"Wisconsin"
159@0	160@0	"Course"	"http://www.cs.wi...	"Wisconsin"
160@0	161@0	"Course"	"http://www.cs.wi...	"Wisconsin"
161@0	162@0	"Course"	"http://www.cs.wi...	"Wisconsin"
162@0	163@0	"Course"	"http://www.cs.wi...	"Wisconsin"
163@0	164@0	"Course"	"http://www.cs.wi...	"Wisconsin"
164@0	165@0	"Course"	"http://www.cs.wi...	"Wisconsin"

In both of the previous methods of filtering the school attribute, the resulting NST included only the head, `id`, and `value` columns. We can include additional data in the NST by using an overload of the `getObjects()` method that lets us specify which additional columns to include in the resulting NST. This overload requires two string arguments: a filter, and a list of additional columns to be included in the resulting NST. Proximity provides similar overloads for many methods that work with NSTs.

11. In the interactive Python in interpreter, enter

```
wiscPages.describe()
```

Proximity reports

```
1,247 rows, 2 columns [id,school:oid,str] -- in memory
```

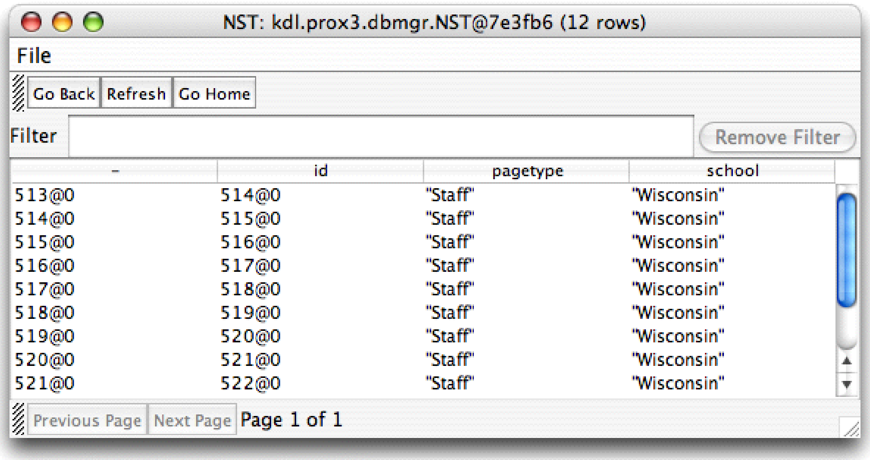
The `describe()` method provides summary information about an NST. As was the case for the `browse()` method, use the “shortcut” names to pass one of the standard tables as an argument to `describe()`. Remember that the attribute shortcuts return an `Attributes` object, not an NST, so you must explicitly get the corresponding NST, for example,

```
prox.objectAttrs.getAttrNST().describe()
```

12. In the interactive interpreter, enter

```
wiscStaff = DB.getObjects("school = 'Wisconsin' AND pagetype = 'Staff'")
prox.browse(wiscStaff)
```

Proximity creates and displays an NST named wiscStaff.



-	id	pagetype	school
513@0	514@0	"Staff"	"Wisconsin"
514@0	515@0	"Staff"	"Wisconsin"
515@0	516@0	"Staff"	"Wisconsin"
516@0	517@0	"Staff"	"Wisconsin"
517@0	518@0	"Staff"	"Wisconsin"
518@0	519@0	"Staff"	"Wisconsin"
519@0	520@0	"Staff"	"Wisconsin"
520@0	521@0	"Staff"	"Wisconsin"
521@0	522@0	"Staff"	"Wisconsin"

To find the objects that correspond to Wisconsin staff web pages, we need to create a filter that requires matching *two* conditions: the value of the school attribute must be Wisconsin and the value of the pagetype attribute must be Staff. The filter combines these two conditions with the logical operator `AND`.

Filter conditions can be combined with `AND` and `OR`, using infix notation. The filter is passed in as a single string parameter to the `getObjects()` method. String values must be quoted; note the single quotes surrounding the required attribute values. For more information about using logical operators in filters, see the `FilterFactory` class.

Because the filter used values from both the `school` and `pagetype` columns, these columns are automatically included in the resulting NST. You can use the overload of the `getObjects()` method, described above, to include data from additional columns in the resulting NST.

13. In the interactive Python interpreter, enter

```
wiscFaculty = DB.getObjects("school = 'Wisconsin' AND pagetype = 'Faculty'")
```

This filter combines two conditions to find all the Wisconsin faculty pages. The results are stored in a new NST named wiscFaculty.

Next, we need to find the links from objects in the set of Wisconsin staff pages (wiscStaff) to objects in the set of Wisconsin Faculty pages (wiscFaulty). We begin by getting the standard Proximity links NST.

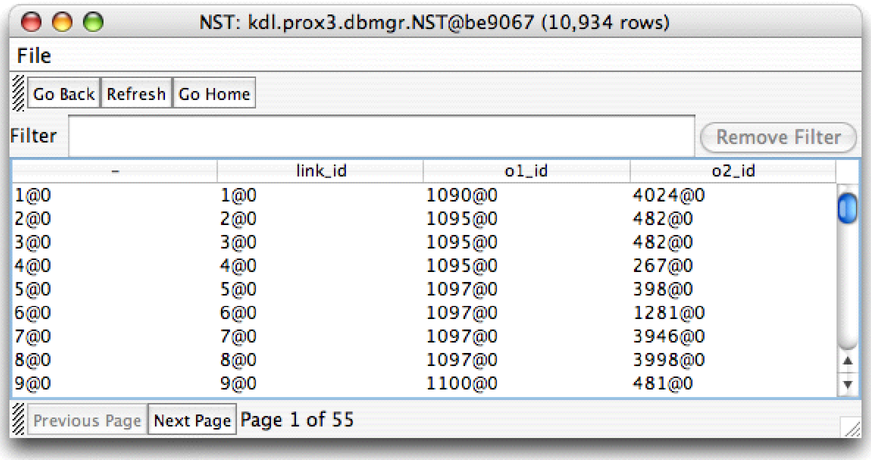
14. In the interactive Python interpreter, enter

```
linksNST = prox.linkNST
```

Examine this NST in the browser by either entering

```
prox.browse(linksNST)
```

in the interactive Python interpreter or by double-clicking **prox_link** in the NST browser's home page.



-	link_id	o1_id	o2_id
1@0	1@0	1090@0	4024@0
2@0	2@0	1095@0	482@0
3@0	3@0	1095@0	482@0
4@0	4@0	1095@0	267@0
5@0	5@0	1097@0	398@0
6@0	6@0	1097@0	1281@0
7@0	7@0	1097@0	3946@0
8@0	8@0	1097@0	3998@0
9@0	9@0	1100@0	481@0

The links NST has four columns: the head column, the ink IDs (`link_id`), and the starting (`o1_id`) and ending (`o2_id`) objects for each link.

Now we need to identify those links where the starting object is a Wisconsin staff page (the link's `o1_id` value must be in the list of IDs from the wiscStaff NST) and the ending object is a Wisconsin faculty page (the link's `o2_id` value must be in the list of IDs from the `wiscFaculty` NST.

15. In the interactive Python interpreter, enter

```
facOutLinks = linksNST.intersect(wiscStaff, "o1_id = id")
```

This identifies those links where the starting object (`o1_id`) is a Wisconsin staff page. That is, the link's `o1_id` value must equal the value of `id` for an object in the wiscStaff NST.

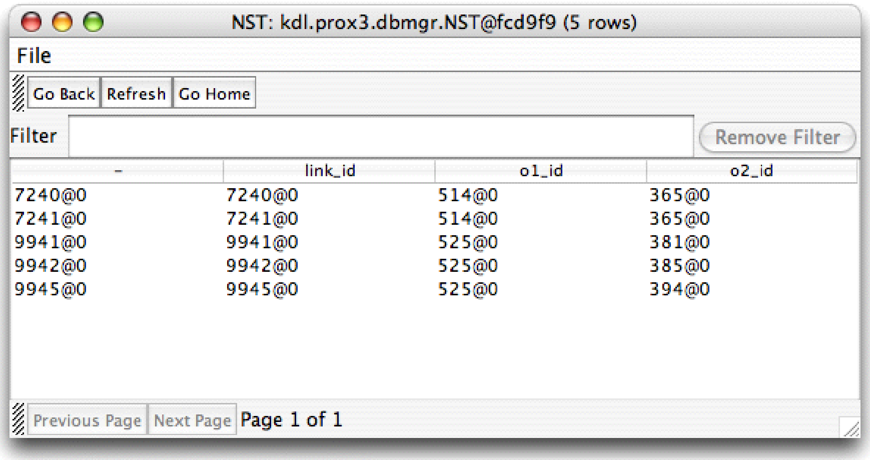
Once again, we use a filter to identify the relevant rows in these NSTs. But because we need to match values across two different NSTs, the filter is used as an argument to the `intersect()` method. The filter for this instance, `o1_id = id`, compares the values in linksNST's `o1_id` column with the values in wiscStaff's `id` column.

Because these NSTs have unique column names, Proximity knows which NST each column comes from. When both NSTs share a column name that you want to use in a filter, add a prefix indicating which NST it comes from. For example, if both NSTs had an `id` column, write `A.id` to indicate that this refers to the `id` column from NST that the method operates on (linksNST) and `B.id` to indicate that this refers to the `id` column from the NST passed in as a parameter to the `intersect()` method (`wiscStaff`).

16. In the interactive Python interpreter, enter

```
wiscStaffFacLinks = facOutLinks.intersect(wiscFaculty, "o2_id = id")
prox.browse(wiscStaffFacLinks)
```

Proximity creates and displays an NST names wiscStaffFacLinks.



-	link_id	o1_id	o2_id
7240@0	7240@0	514@0	365@0
7241@0	7241@0	514@0	365@0
9941@0	9941@0	525@0	381@0
9942@0	9942@0	525@0	385@0
9945@0	9945@0	525@0	394@0

This identifies the links in facOutLinks where the ending object is a Wisconsin faculty page. That is, the link's `o2_id` value must equal the value of `id` for an object in the wiscFaculty NST. Because facOutLinks includes only the links beginning at a Wisconsin staff page, the wiscStaffFacLinks NST contains the links we want—links from Wisconsin staff pages to Wisconsin faculty pages. Remember, the `link_id` column, not the head column, lists the IDs for the final set of links.

Synthetic Data Generation

Researchers frequently need to explore the effects of certain data characteristics on their models. To help construct datasets exhibiting specific properties, such as autocorrelation or degree disparity, Proximity can generate synthetic data having one of several types of graph structure:

- random graphs
- independent and identically distributed (i.i.d.) connected components
- lattice graphs having a ring structure
- lattice graphs having a grid structure
- forest fire graphs
- cluster graphs with nodes arranged in separate clusters (cliques)

In all cases, the data generation process follows the same process:

1. Generate the empty graph structure.
2. Generate attribute values based on user-supplied prior probabilities.

Because the attribute values of one object may depend on the attribute values of related objects, the attribute generation process assigns values collectively.

Details for each of these steps are presented in the context of a script that generates i.i.d. data, shown below.

Generating i.i.d. data

The i.i.d. graph generation process creates a set of independent connected components containing two object types: *S* and *T*. Each component contains a single *S* object that is linked to a variable number (≥ 0) of *T* objects. The number of linked *T* objects (the degree of *S*) follows a normal distribution with user-specified mean and standard deviation. You can specify multiple normal distributions to create *S* objects having different degree distributions.

We typically consider the *S* objects to be the target objects to be classified, with the *T* objects used as peripheral objects during classification. Each *S* object is assigned a single, discrete attribute (s_class_label in this example) that can be used as a class label. In order to generate datasets with degree disparity, the assignment of class labels is conditioned on the degree of the object in the graph. The generation process can also add additional discrete attributes to the *S* and *T* objects, respectively.

The following script creates four connected components using two different degree distributions. Both *S* and *T* objects are given one additional attribute each (s_attr0_label and t_attr1_label, respectively). These attribute values are conditioned by the models in `s-attr-rpt.xml` and `t-attr-rpt.xml`.

Code example: generate-iid-data.py

This section describes the script found in `$PROX_HOME/doc/user/tutorial/examples/generate-iid-data.py`.

Import needed classes.

```
from kdl.prox.datagen2.structure import SyntheticGraphIID
from kdl.prox.datagen2.attributes import AttributeGenerator
from kdl.prox.model2.rpt import RPT
from kdl.prox.util.stat import NormalDistribution
from java.io import File
```

Define a helper function that loads an RPT from a file.

```
def loadRPT(modelFile):
    return RPT().load(modelFile)
def loadFile(fileName):
    return File(fileName)
```

Data generation requires an empty database. Check to see if the database already contains data so we can warn users accordingly. The method `isProxTablesEmpty()` returns true if the top-level tables are defined and empty.

```
isEmpty = DB.isProxTablesEmpty()
```

If the database is not empty, ask the user whether to overwrite the existing data.

```
genData = 1
if (not isEmpty):
   prompt = "Database is not empty. Overwrite existing data?"
   genData = prox.getYesNoFromUser(prompt)
```

Do nothing if the database is not empty and the user says not to overwrite the existing data.

```
if (not genData):
   print 'No data generated'
```

Continue with data generation if the database is empty or if the user says to overwrite existing data.

Clear and initialize the database.

```
else:
   print 'Clearing database'
   DB.clearDB()
   print 'Initializing database'
   DB.initEmptyDB()
```

To define the i.i.d structure, we specify a probability distribution over degree distributions for the *S* objects. This script creates connected components with two different degree distributions for the *S* objects. One half of the *S* objects have normally distributed degrees with a mean of 2 and standard deviation approaching zero. The second half have normally distributed degrees with a mean of 5 and standard deviation of 1. The exact number of components having each of the specified degree distributions is determined probabilistically so you may not end up with exactly two *S* objects for each degree distribution.

```
   degreeDistribs = [[0.5, NormalDistribution(2.0, 0.0000001)],
                     [0.5, NormalDistribution(5.0, 1.0)]]
```

Generate the i.i.d. graph structure by calling a constructor for the `SyntheticGraphIID` class. The `SyntheticGraphIID` generator creates objects and gives them a type (saved in an objecttype attribute): *S* for the core objects and *T* for the peripheral objects (the objects linked to the core *S* objects). Generating i.i.d. data is unique in this regard—other structure generators do not attach any attributes to the objects or links that they create. The generated data will include four *S* objects.

```
   print 'creating graph structure'
   SyntheticGraphIID(4, degreeDistribs)
```

After creating the structure, we generate the attribute values. Note that the attribute generator can only generate attributes on objects and that all attributes are string valued.

We use a relational dependency network (RDN) to generate the values for the attributes. The attribute names and their distributions are specified by the constituent models of this RDN in the form of relational probability trees (RPTs). The data generator requires one RPT and one query for each attribute in the generated data. In this example, we provide an RPT for the *S* class labels and one each for the other attributes on *S* and *T* objects. For each each attribute in the generated data, the attribute generator executes a query that produces subgraphs having the structure required by the corresponding RPT, saving the results in a temporary container. For example, the `iid-coreS-query` query creates the 1d-star subgraphs required by the RPT for the s_attr0_label attribute (`s-attr-rpt.xml`).

The data generation process iterates the specified number of times, applying the RPTs one by one to the subgraphs in the corresponding container, in a random order. Applying the RPT predicts values for the associated attribute, which are recorded in the database. The values predicted by one application of the RPT are used as input to succeeding applications. In this way, the attribute values change over time (are *conditioned*) from their initial random values to values consistent with the provided model. See Chapter 7, *Learning Models* for more information on RPTs and RDNs.

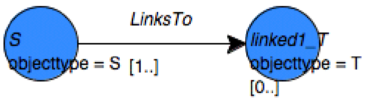
The attribute generator appends the string “`_label`” and prepends “`s_`” or “`t_`” as appropriate to the attribute names specified in the RPTs. For example, if you specify an attribute named “class” in `s-class-rpt.xml`, the *S* objects will be given an attribute named s_class_label. Similarly, specifying an attribute named “`attr1`” in `t-attr-rpt.xml` results in an attribute named t_attr1_label in the generated data. Although Proximity modifies the final attribute names as outlined above, all attribute names must be unique within the RPT files.

Proximity requires that the RPT and query files be in the current working directory (`$PROX_HOME` if you are following the tutorial).

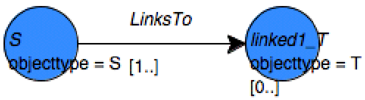
```
   print 'generating i.i.d. attributes'

   sClassQuery = loadFile("iid-class-query.xml")
   sClassRPT   = loadRPT("s-class-rpt.xml")

   sAttrQuery  = loadFile("iid-coreS-query.xml")
   sAttrRPT    = loadRPT("s-attr-rpt.xml")

   tAttrQuery  = loadFile("iid-coreT-query.xml")
   tAttrRPT    = loadRPT("t-attr-rpt.xml")
```

The first two queries get the 1d-stars for each *S* object—all the *T* objects connected to their core *S* objects. The last query, `iid-coreT-query`, gets the 2d-stars for each *T* object—for each *T* object it finds any connected *S* objects and any *T* objects connected to those *S* objects.

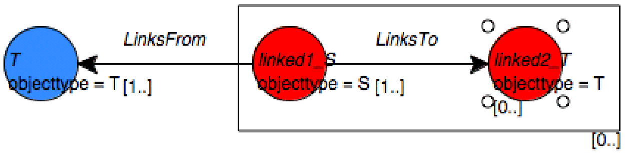
iid-class-query



iid-coreS-query

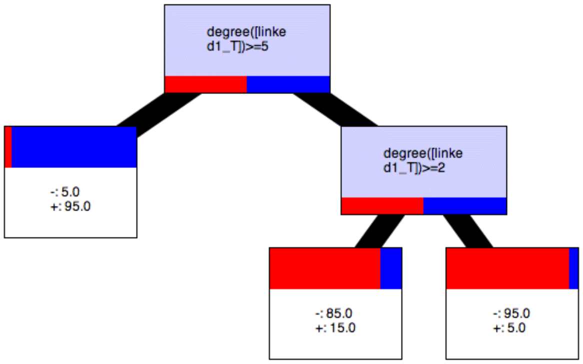


iid-coreT-query

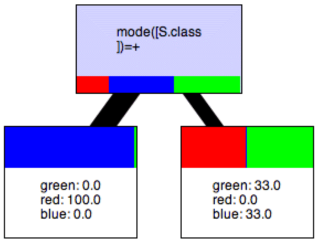


The *S* class RPT (`s-class-rpt.xml`) infers the value of s_class_label based on the degree of the *S* objects. The other two RPTs infer the value of their respective attributes based on the value of s_class_label for the current object (for `s-attr-rpt`) or linked *S* objects (for `t-attr-rpt`).

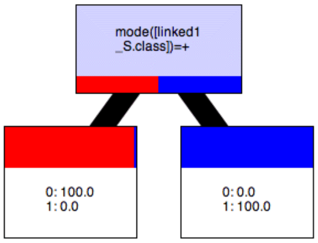
s-class-rpt



s-attr-rpt



t-attr-rpt



Store the queries and RPTs in a Python dictionary that pairs queries with the RPTs that specify the corresponding attributes.

```
queriesAndModels = {sClassQuery: sClassRPT, \
    sAttrQuery: sAttrRPT, tAttrQuery: tAttrRPT}
```

Specify the number of Gibbs sampling iterations to use in conditioning the data.

```
iters = 3
```

To generate the attribute values, we call a constructor for the `AttributeGenerator` class.

```
AttributeGenerator(queriesAndModels, iters)
```

Exercise 6.8. Generating synthetic i.i.d. data:

The i.i.d. data generation script clears and overwrites the current database. Make sure that you are serving an empty or test database.

1. All RPT files required for this exercise must be available in the current working directory (`$PROX_HOME` if you are following the tutorial). Copy the required files to the correct directory.

   ```
   > cd $PROX_HOME
   > cp $PROX_HOME/doc/user/tutorial/examples/s-class-rpt.xml .
   > cp $PROX_HOME/doc/user/tutorial/examples/s-attr-rpt.xml .
   > cp $PROX_HOME/doc/user/tutorial/examples/t-attr-rpt.xml .
   > cp $PROX_HOME/doc/user/tutorial/examples/iid-class-query.xml .
   > cp $PROX_HOME/doc/user/tutorial/examples/iid-coreS-query.xml .
   > cp $PROX_HOME/doc/user/tutorial/examples/iid-coreT-query.xml .
   ```

   The DTD describing the RPT format must be present in the same directory as the RPT files. If you have not already done so, copy this file to the current working directory.

   ```
   > cp $PROX_HOME/resources/rpt2.dtd .
   ```

2. Serve a new (empty) database.

   ```
   > Mserver --dbname DataGen_IID $PROX_HOME/resources/init-mserver.mil
   ```

   Remember to use a port number > 40000 if you are using MonetDB 4.6.2.

3. Initialize the new Proximity database. Substitute the appropriate host and port information if you are running the MonetDB server on a different machine or are using a different port.

   ```
   > cd $PROX_HOME
   > bin/db-util.sh localhost:30000 init-db
   ```

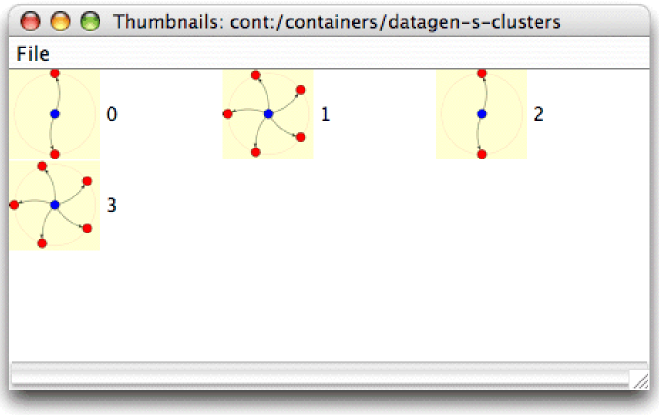
4. Start the Proximity Database Browser.
5. From the **Script** menu, choose **Run Script**. Proximity displays the Open dialog.
6. Navigate to the `$PROX_HOME/doc/user/tutorial/examples` directory and choose `generate-iid-data.py`. Click **Open**.

   Proximity opens a window to show you any output from the script along with a trace of the script execution. The final lines of your output should look similar to the following (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

   ```
   INFO kdl.prox.qgraph2.QueryGraph2CompOp  - * done executing query
   INFO kdl.prox.model2.rdn.RDN  - RDN Iteration: 0
   Status: finished running script
   ```

   You can close this window after the script finishes.

   Examine the generated objects and attributes in the Proximity Database Browser. You may want to execute a 1d-star query that finds all *T* objects directly connected to each *S* object (for example, `iid-coreS-query.xml`). The subgraphs created by such a query are shown below. The results of each data generation run will be different and your results will not necessarily look exactly like that shown here.



Thumbnails: cont:/containers/datagen-s-clusters

Tips and Reminders

- The `Proximity` class provides methods that support many Proximity operations including accessing databases, executing queries, adding attributes, sampling containers, and instantiating models.
- Use the *`prox`* object to access methods in the `Proximity` class.
- Use the shortcuts provided by the *`prox`* object to refer to important Proximity data structures.
- You only need to import a Proximity class or package when you refer to a class by name.
- Use the Proximity Python interpreter to execute individual Python statements interactively when you do not need to create a script.
- Indent lines and blocks using tabs rather than spaces in the Proximity Python interpreter.
- Each Proximity Python interpreter window has its own namespace.
- Use the NST browser to examine Proximity internal data structures when you need to work directly with Proximity entities.
- Filters and conditions have different semantics. Conditions operate on objects or links whereas filters operate on NST rows.

Additional Information

- See the Proximity Cookbook and Chapter 7, *Learning Models* in this tutorial for additional examples of Proximity scripts.
- See Appendix A, *Proximity Quick Reference* for a summary of command line editing options in the interactive Proximity Python interpreter.
- See “Creating a file or directory shortcut” (p. 9) for information on creating shortcuts to commonly used files and directories.

Chapter 7. Learning Models

Overview

The exercises in this chapter walk through the process of training, testing, and evaluating several probabilistic models designed for relational knowledge discovery. For each exercise in this chapter, we first walk through the source code and identify how Proximity classes and methods are used to accomplish the desired task. This is followed by instructions for executing the script in Proximity.

> Probabilistic relational models can consider huge amounts of data and take a corresponding amount of time to learn and apply. The exercises in this chapter were designed to run in a reasonable amount of time, and they therefore limit the number of input attributes to be considered. These examples may therefore not necessarily represent realistic or rigorous experimental design.

The examples in this chapter are written in Jython, a Java implementation of Python that lets you interact with Java code. The classes and methods used in these examples are, of course, also available for use in Java code. Source code files for all the scripts discussed in this chapter are available in `$PROX_HOME/doc/user/tutorial/examples`.

Objectives

The exercises in this chapter demonstrate how to

- learn, apply, and evaluate the relational Bayesian classifier model
- learn, apply, and evaluate the relational probability tree model
- learn, apply, and evaluate the relational dependency network model
- use temporal attributes to restrict the set of applicable attribute values in a relational probability tree
- display and interpret graphical representations of relational probability trees
- display and interpret graphical representations of relational dependency networks

The Modeling Process in Proximity

The process of learning, applying, and evaluating a model is substantially the same for each of Proximity's models. In each case we will

1. *Define the problem.* We identify the attribute that we want to predict (the *class label*) and the pieces of information that can be used in making that prediction (*sources*). Sources can include attributes on objects or links in the subgraph, attributes on the subgraph itself, or structural features of objects in the subgraph.
2. *Instantiate the model.* A simple constructor call instantiates a default version of the model. For some models, you can add modules to change the behavior as desired.
3. *Learn the model.* Proximity uses labeled data to determine how to predict class labels. The process for predicting class labels varies depending on the type of model being learned.
4. *Save the model.* [Optional] Proximity lets you save the model to an XML file. You can then import the saved model at a later time, avoiding the need to regenerate the model.
5. *Apply the model.* Use the model to make predictions about the class label.
6. *Write the model's predictions to the database.* [Optional] You can save the predicted values in the database, typically as an attribute on the corresponding subgraph.
7. *Evaluate the model.* Calculating evaluation metrics lets you compare and evaluate model performance.

Relational Bayesian Classifier

The relational Bayesian classifier (RBC) is a simple Bayesian classification algorithm adapted to the context of relational data [Neville, Jensen and Gallagher, 2003]. Like many classification systems, the RBC is trained on one set of instances and then applied to (or tested on) another. Because the RBC operates on relational data, the training and test sets are sets of subgraphs, defined by a container in Proximity. Each subgraph in the container is assumed to contain one target item (the core item) to be classified. Attributes and structural characteristics of the target item, of the other objects and links in the subgraph, and of the subgraph itself can all be used by the model in classifying the target item. Because these instances preserve the relational structure of the data, the model can exploit the connections among objects to improve classification accuracy.

In the example below, we want to predict whether or not a web page is a student page, that is, whether the value of its isStudent attribute is “1”. Each input subgraph consists of a target web page and all objects (also web pages) connected to it, including both links to and links from the target page. The following script learns an RBC on one of the samples we created in Exercise 6.4, then applies the RBC to the other sample to evaluate the its predictions.

Code example: run-1d-clusters-rbc.py

This section describes the script found in `$PROX_HOME/doc/user/tutorial/examples/run-1d-clusters-rbc.py`.

Import the necessary class definitions.

```
from kdl.prox.model2.common.sources import *
from kdl.prox.model2.rbc import RBC
```

We use the samples created in Exercise 6.4 as our training and test sets. Make sample “0” the training set and sample “1” the test set.

```
trainContainer = prox.getContainer("1d-clusters/samples/0")
testContainer = prox.getContainer("1d-clusters/samples/1")
```

First we identify the *core item*, the database entity whose label (attribute value) we want to predict. In this case, we want to predict the label for the central object in our 1d-clusters subgraphs. This object was named “core_page” in the query that generated these subgraphs, and thus also in the resulting subgraphs.

```
coreItemName = 'core_page'
```

Next we specify the name of the attribute we want to predict (the class label). Our model predicts whether the object in question is a student web page. We use the boolean attribute isStudent to identify whether or not a page belongs to a student. The RBC model requires that the predicted attribute be discrete.

```
attrToPredict = 'isStudent'
```

We create an `AttributeSource` instance to represent the information about the class label. This attribute source specifies the item in the subgraph that we want to label (core_page) and the name of the attribute that we want to predict (isStudent).

```
classLabel = AttributeSource(coreItemName, attrToPredict)
```

Next, we specify the set of sources to use in learning the model. Recall that sources can be attributes on any of the objects or links in the input subgraph including attributes on the core item, attributes of links connecting to the core item, or attributes of objects linked to the core item. In principle, we can extend this to use attributes on objects and links multiple “hops” away from the core object, provided that such items are included in the subgraphs that the model will use. In this example, because the subgraphs only extend a single hop from the core object, we are limited to attributes of objects of the core object and of objects and links directly connected to that core object. Sources can also include structural characteristics of the subgraph or attributes on the subgraph itself, although neither of these types of sources are used in this example. See Exercise 7.2 for an example of using the degree of an object (a structural characteristic) as a source.

Each source is specified as an instance of `AttributeSource`. Each `AttributeSource` provides the name of an item (object or link) from the input subgraphs and the attribute whose values will be evaluated for their predictive value to the model. We provide the set of sources as a Python list.

```
inputSources = [ \
   AttributeSource('core_page', 'url_server_info'), \
   AttributeSource('core_page', 'url_hierarchy1b'), \
   AttributeSource('linked_from_page', 'url_server_info'), \
   AttributeSource('linked_from_page', 'url_hierarchy1b'), \
   AttributeSource('linked_from_page', 'page_num_outlinks'), \
   AttributeSource('linked_from_page', 'page_num_inlinks'), \
   AttributeSource('linked_to_page', 'url_server_info'), \
   AttributeSource('linked_to_page', 'url_hierarchy1b'), \
   AttributeSource('linked_to_page', 'page_num_outlinks'), \
   AttributeSource('linked_to_page', 'page_num_inlinks'), \
   AttributeSource('linked_to', 'link_tag'), \
   AttributeSource('linked_from', 'link_tag')]
```

Begin the modeling portion of the script by instantiating the model. Instantiation is a simple `RBC` constructor call.

```
print "Beginning modeling section"
print "Instantiating model..."
rbc = RBC()
```

Train (learn) the model on the training set.

```
print "Learning model..."
rbc.learn(trainContainer, classLabel, inputSources)
```

Write the trained model to an XML file. The file is written to the current working directory, which is `$PROX_HOME` if you are following the tutorial.

```
xmlFileName = 'ProxWebKB_RBC.xml'
rbc.save(xmlFileName)
print "RBC written to ", xmlFileName
```

You can read in the XML file at a later time to apply the model to other data sets having the same structure and features without having to retrain the model.

Apply the model to the test set.

```
print "Applying model..."
predictions = rbc.apply(testContainer)
```

Applying the learned model returns a `Predictions` instance that stores the predicted class labels. To save these predictions, we store the predicted values as attributes on the subgraphs. The `savePredictions()` method silently overwrites any existing values for this subgraph attribute.

```
print "Writing predictions..."
rbcAttrName = "rbc_isstudent_prediction"
predictions.savePredictions(testContainer.getSubgraphAttrs(), rbcAttrName)
```

Finally, we evaluate the model's results. The `Predictions` class provides several methods for calculating the error of the predictions, but to use them, we have to first tell the predictions instance where to find the true values for the class labels. The true values are stored in the isStudent attribute on the core items in the test container's subgraphs.

```
predictions.setTrueLabels(testContainer, classLabel)
```

Proximity provides several built-in evaluation functions. Accuracy measures the percentage of test instances correctly classified by the model. Because accuracy requires knowing the correct prior distribution of instances in the population and does not consider that false positive and false negative errors may entail widely different costs, other evaluation measures, such as area under the ROC curve, are often more useful. Area under the ROC (receiver operating characteristic) curve [Provost, Fawcett and Kohavi, 1998] provides a method of evaluating the accuracy of a ranking of all the test instances according to their estimated probability, letting you compare one model’s performance against another’s, independent of error cost and without knowing the prior class distribution. Accuracy and area under the ROC curve approach 1.0 as the results improve; a value of 0.5 for area under the ROC curve indicates random performance. Conditional log likelihood is less intuitive and useful only for relative comparison of comparable entities, with higher values signifying better performance.

```
print "Computing accuracy (ACC)..."
acc = (1 - predictions.getZeroOneLoss())
```

Computing area under the ROC curve requires a binary classification problem. We identify the positive instances and group all other class values into the negative instances. In this example, student pages (pages with isStudent = 1) are positive instances.

```
print "Computing area under ROC curve (AUC)..."
auc = predictions.getAUC("1")
print "Computing conditional likelihood (CLL)..."
cll = predictions.getConditionalLogLikelihood()
```

Finally, print a summary of the evaluation results.

```
print "RBC results:"
print "  ACC: ", str(acc)
print "  AUC: ", str(auc)
print "  CLL: ", str(cll)
```

Exercise 7.1. Learning and applying the relational Bayesian classifier model:

This script requires entities created in Exercise 5.7, Exercise 6.4 and Exercise 6.5. You must have completed these exercises before running the script in the current exercise.

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. From the **Script** menu, choose **Run Script**. Proximity displays the Open dialog.
2. Navigate to the `$PROX_HOME/doc/user/tutorial/examples` directory and choose `run-1d-clusters-rbc.py`. Click **Open**.

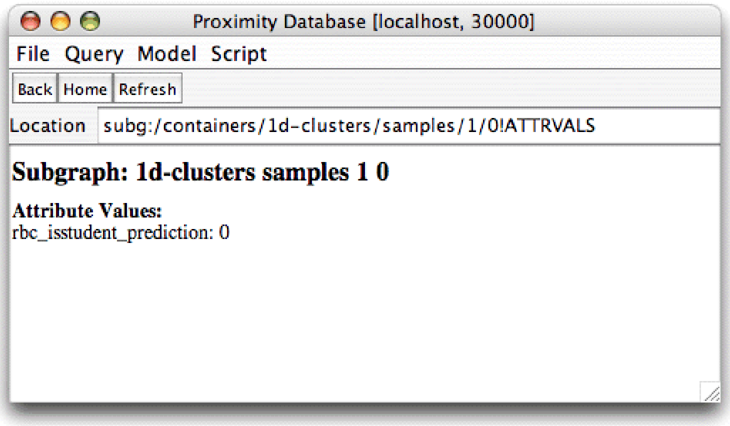
   Proximity opens a window to show you any output from the script along with a trace of the script execution. Your output should look similar to the following (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

   ```
   Status: starting running script:
      /proximity/doc/user/tutorial/examples/run-1d-clusters-rbc.py
   Beginning modeling section
   Instantiating model...
   Inducing model...
   INFO kdl.prox.model2.rbc.RBC - starting to induce model...
   INFO kdl.prox.model2.rbc.RBC - updating model with [core_page.url_server_info]...
   INFO kdl.prox.model2.rbc.RBC - updating model with [core_page.url_hierarchy1b]...
   INFO kdl.prox.model2.rbc.RBC - updating model with [linked_from_page.url_server_info]...
   INFO kdl.prox.model2.rbc.RBC - updating model with [linked_from_page.url_hierarchy1b]...
   INFO kdl.prox.model2.rbc.RBC - updating model with [linked_from_page.page_num_outlinks]...
   INFO kdl.prox.model2.rbc.RBC - updating model with [linked_from_page.page_num_inlinks]...
   INFO kdl.prox.model2.rbc.RBC - updating model with [linked_to_page.url_server_info]...
   INFO kdl.prox.model2.rbc.RBC - updating model with [linked_to_page.url_hierarchy1b]...
   INFO kdl.prox.model2.rbc.RBC - updating model with [linked_to_page.page_num_outlinks]...
   INFO kdl.prox.model2.rbc.RBC - updating model with [linked_to_page.page_num_inlinks]...
   INFO kdl.prox.model2.rbc.RBC - updating model with [linked_to.link_tag]...
   INFO kdl.prox.model2.rbc.RBC - updating model with [linked_from.link_tag]...
   INFO kdl.prox.model2.rbc.RBC - induce model done...
   RBC written to  ProxWebKB_RBC.xml
   Applying model...
   INFO kdl.prox.model2.rbc.RBC - starting to apply model...
   INFO kdl.prox.model2.rbc.RBC - apply model done...
   Writing predictions...
   Computing accuracy (ACC)...
   ```

```
Computing area under ROC curve (AUC)...
Computing conditional likelihood (CLL)...
RPT results:
  ACC:  0.946298984034833
  AUC:  0.9747484409075087
  CLL:  -495.61207422760805
Status: finished running script
```

Due to the probabilistic nature of the RBC algorithm, your results may differ slightly from that shown above. You can close this window after the script finishes.

3. Examine the values predicted by the RBC. The script stored the predicted values in the rbc_isstudent_prediction subgraph attribute. Recall that our testing instances are in the /1d-clusters/samples/1 container. To see these values, drill down through the container hierarchy in the Proximity Database Browser to display the list of subgraphs for this container. Click a subgraph ID, then click **attrs** to display the attributes for that subgraph. The example below shows the value for subgraph 0, which indicates that the model predicts that the core page for this subgraph (object 1) is not a Student page. You can compare this to the actual value of the isStudent attribute by examining the attribute values for object 1.


Proximity Database [localhost, 30000]
File Query Model Script
Back Home Refresh
Location subg:/containers/1d-clusters/samples/1/0!ATTRVALS

Subgraph: 1d-clusters samples 1 0

Attribute Values:
rbc_isstudent_prediction: 0

Recall that the RBC only makes predictions for core objects in the test container. Therefore, after learning the model, only the subgraphs in the 1d-clusters/samples/1 container have a value for the rbc_isstudent_prediction attribute.

Relational Probability Trees

A relational probability tree (RPT) [Neville et al., 2003] is a form of classification tree for relational data that considers attributes of related objects and links, and automatically constructs complex relational aggregates of these attributes to build a probabilistic model. These relational aggregations include mode, average, count, and proportion. Each of these aggregations can operate on related objects or links and each dynamically determines the best threshold. In addition to the aggregation functions, RPTs also consider structural features of the data. Specifically, RPTs can include degree features (counts of named objects or links in a subgraph) in the resulting models. Advantages of this model include ease of model understanding (the model is a series of hierarchical rules) and the ability to dynamically select predictive features and thresholds.

Proximity’s RPT code has been modularized to permit easier maintenance of and additions to the code. Model creation has been split into learning and pruning modules, with each of these having their own set of components used to specify particular performance parameters.

The learning module includes several component modules that control the learning phase:

- The *splitting module* evaluates the possible splits at each branch point in the tree, selecting the best split according to scores calculated by the scoring module.

- The *scoring module* computes a score for each possible split in the tree. The results of the scoring module are used to select the best split by the splitting module.
- The *significance module* determines whether the split chosen by the splitting module reaches the specified level of statistical significance. If not, that point in the tree becomes a leaf node.
- The *stopping module* determines when to stop looking for additional splits in the tree. The current release only includes a single implementation, `DefaultStoppingModule`, which stops splitting a tree after it reaches the specified depth (with a default depth of three).

The pruning module provides the ability to apply different pruning strategies to the completed tree. Implementing pruning strategies is planned for future releases; the current release includes only the `DefaultPruningModule`, which does no pruning.

The example below performs the same classification task as the previous RBC example—we want to infer whether or not a web page is a student page. But this time we use the value of the pagetype attribute to identify known student pages in the training phase. This attribute can take one of six (string) values: Student, Course, Faculty, Staff, ResearchProject, and Other. We use the same training and test sets as in the RBC example.

Code example: run-1d-clusters-rpt.py

This section describes the script found in `$PROX_HOME/doc/user/tutorial/examples/run-1d-clusters-rpt.py`.

Import the necessary class definitions.

```
from kdl.prox.model2.common.sources import *
from kdl.prox.model2.rpt import RPT
```

As before, sample 0 serves as the training set and sample 1 is the test set.

```
trainContainer = prox.getContainer("1d-clusters/samples/0")
testContainer = prox.getContainer("1d-clusters/samples/1")
```

Our classification task remains the same—we want to predict the value of one of the attributes for the central object in our 1d-clusters subgraphs. This object is named *core_page* in the subgraphs in both our test and training containers.

```
coreItemName = 'core_page'
```

This time, we’re predicting the value of the pagetype attribute.

```
attrToPredict = 'pagetype'
```

Create an `AttributeSource` instance that stores both the core item name and the class label (attribute) we want to predict for that item.

```
classLabel = AttributeSource(coreItemName, attrToPredict)
```

As before, we specify the set of sources to be considered in determining the model’s features. See “Code example: run-1d-clusters-rbc.py” for more information on specifying attribute sources.

In addition to attribute sources, models can also consider structural features such as the degree of an object. To tell Proximity to also consider structural attributes, we provide an `ItemSource` instance for the appropriate subgraph item. In this example, we want to consider structural features for the related objects (linked_to_page and linked_from_page), but not for the core object. Although the `('linked_from_page', 'page_num_inlinks')` attribute source provides similar information, these sources are not the same. The value of page_num_inlinks for related linked_from_page objects does not consider the full degree of the object (it ignores out links and only counts in links) and the value is stored as an attribute on the object. It must therefore be defined as an attribute source. The full degree of the linked objects is not available as an attribute value but must be calculated for each object. It must therefore be specified as an item (structural) source.

```
inputSources = [ \
   AttributeSource('core_page', 'url_server_info'), \
   AttributeSource ('core_page', 'url_hierarchy1b'), \
   AttributeSource('linked_from_page', 'page_num_outlinks'), \
   AttributeSource('linked_to_page', 'page_num_inlinks'), \
   ItemSource("linked_from_page"), \
   ItemSource("linked_to_page")]
```

Proximity uses these attributes to construct the specific features used by the RPT model. Relational features typically identify an attribute and a test for the values of that attribute. For example, an RPT feature might test to see if an attribute is equal to a designated value. Relational features can also examine and aggregate the set of values found in linked objects. For example, the RPT might use the number of *linked_to_page* objects that have more than 10 outlinks as a feature in the trained model.

We instantiate the model by calling the appropriate constructor.

```
print "Beginning modeling section"
print "Instantiating model..."
rpt = RPT()
```

This instantiates the model using default values for all parameters. You can override these defaults by specifying particular modules to be used with the model. For this example, we want a maximum tree depth of three, so we add that specification to the stopping module.

```
rpt.learningModule.stoppingModule.setMaxDepth(3)
```

Train (learn) the model on the training set.

```
print "Learning model..."
rpt.learn(trainContainer, classLabel, inputSources)
```

Write the trained model to an XML file. The file is written to the current working directory, which is `$PROX_HOME` if you are following the tutorial.

```
xmlFileName = 'ProxWebKB_RPT.xml'
rpt.save(xmlFileName)
print "RPT written to ", xmlFileName
```

Unlike the RBC, the learned RPT model is designed to be human interpretable. The XML file for an RPT describes a probability estimation tree, which can be viewed in the Proximity Database Browser (see Exercise 7.3, below). The tree represents a series of questions to ask about a web page and the pages in its relational neighborhood. The leaf nodes contain the predicted class label for pages that correspond to the matching path through the tree.

Apply the model to the test set.

```
print "Applying model..."
predictions = rpt.apply(testContainer)
```

Tell the `Predictions` instance where to find the true values for the class labels. The true values are required for evaluating the model’s predictions.

```
predictions.setTrueLabels(testContainer, classLabel)
```

To save the predicted values, save them as attributes on the subgraphs. The `savePredictions()` method silently overwrites any existing values for this subgraph attribute.

```
print "Writing predictions..."
rptAttrName = "rpt_pagetype_prediction"
predictions.savePredictions(testContainer.getSubgraphAttrs(), rptAttrName)
```

Evaluate the model. Accuracy and area under the ROC curve approach 1.0 as the results improve. Conditional log likelihood is useful only for relative comparison of comparable entities with higher values signifying better performance.

```
print "Computing accuracy (ACC)..."
acc = (1 - predictions.getZeroOneLoss())
```

Computing area under the ROC curve requires a binary classification problem. We identify the positive instances and group all other class values into the negative instances. In this example, student pages (pages with a value of Student for the pagetype attribute) are positive instances.

```
print "Computing area under ROC curve (AUC)..."
auc = predictions.getAUC("Student")
print "Computing conditional likelihood (CLL)..."
cll = predictions.getConditionalLogLikelihood()
```

Print a summary of the evaluation results.

```
print "RPT results:"
print "  ACC: ", str(acc)
print "  AUC: ", str(auc)
print "  CLL: ", str(cll)
```

Exercise 7.2. Learning and applying the relational probability tree model:

This script requires entities created in Exercise 5.7 and Exercise 6.4. You must have completed these exercises before running the script in the current exercise.

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

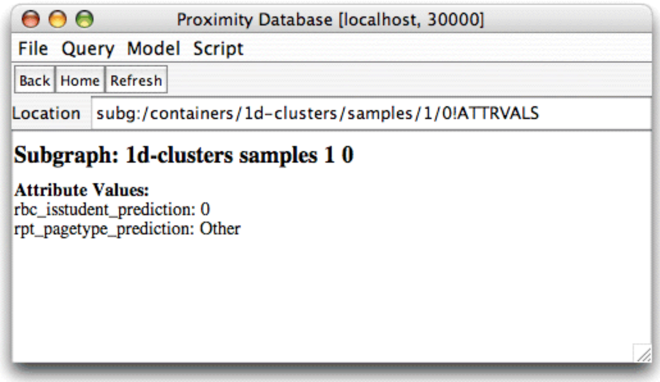
1. From the **Script** menu, choose **Run Script**. Proximity displays the Open dialog.
2. Navigate to the `$PROX_HOME/doc/user/tutorial/examples` directory and choose `run-1d-clusters-rpt.py`. Click **Open**.

   Proximity opens a window to show you the output from the script along with a trace of the script execution. The `run-1d-clusters-rpt.py` script may take several minutes or longer to run. Your output should look similar to the following (leading information showing elapsed time and execution thread has been omitted from the trace for brevity):

   ```
   Status: starting running script:
       /proximity/doc/user/tutorial/examples/run-1d-clusters-rpt.py
   Beginning modeling section
   Instantiating model...
   Inducing model...
   INFO kdl.prox.model2.rpt.RPT - Creating feature tables
   INFO kdl.prox.model2.rpt.RPT - Done creating feature tables: 792 features.
   INFO kdl.prox.model2.rpt.modules.learning.DefaultLearningModule
       - Choosing split for 2068 subgs

       portion of trace deleted


   RPT written to  ProxWebKB_RPT.xml
   Applying model...
   Writing predictions...
   Computing accuracy (ACC)...
   Computing area under ROC curve (AUC)...
   Computing conditional likelihood (CLL)...
   RPT results:
     ACC:  0.8355104015481374
     AUC:  0.8423346214179187
     CLL:  -1312.0900625717052
   Status: finished running script
   ```

   Note that some parts of the RPT model, such as choosing between two equivalent features, are non-deterministic, so your results may differ slightly from that shown above. You can close this window after the script finishes.

3. Examine the values predicted by the RPT. Drill down through the container hierarchy in the Proximity Database Browser to display the list of subgraphs for the /1d-clusters/samples/1 container. Click a subgraph ID, then click **attrs** to display the attributes for that subgraph. The example below shows the value for subgraph 0, which shows that the model predicts that the core page for this subgraph (object 1) has a pagetype of Other You can compare this to the actual value of pagetype by examining the attribute values for object 1.



subg:/containers/1d-clusters/samples/1/0!ATTRVALS

Subgraph: 1d-clusters samples 1 0

Attribute Values:
rbc_isstudent_prediction: 0
rpt_pagetype_prediction: Other

Recall that the RPT only makes predictions for core objects in the test container. Therefore, after learning the model, only the subgraphs in the 1d-clusters/samples/1 container have a value for rpt_pagetype_prediction.

Understanding and viewing relational probability trees

To help you understand and visualize the resulting RPT model, Proximity provides an RPT viewer. This section describes how to display a graphic representation of the RPT and how interpret the labels on the tree’s nodes.

Exercise 7.3. Viewing relational probability trees:

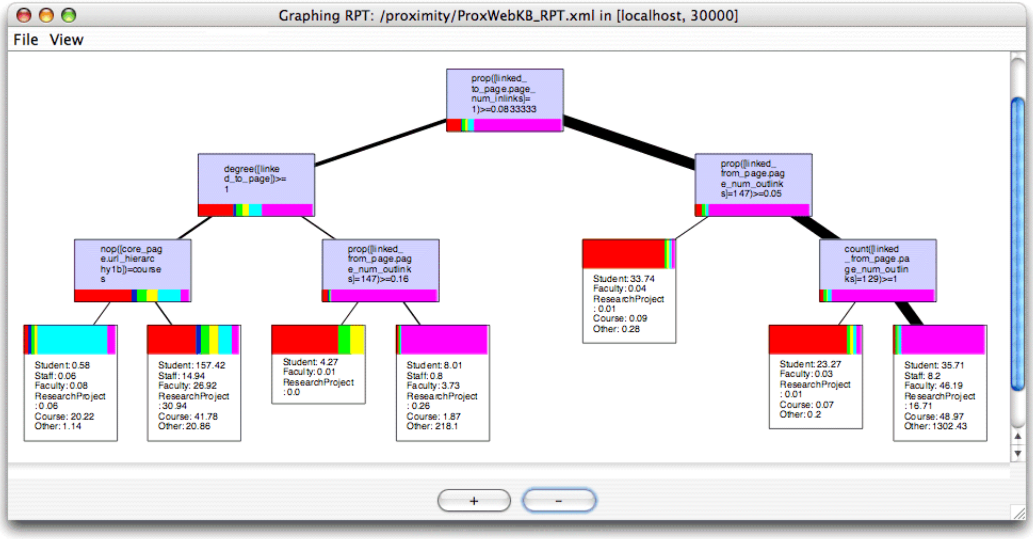
This script requires the RPT created in Exercise 7.2. You must have completed Exercise 7.2 before running the current exercise. Start the Proximity Database Browser if it is not already running.

1. If you have not already done so, copy `rpt2.dtd` to the directory containing the saved RPT XML file, `ProxWebKB_RPT.xml`.

   ```
   > cp $PROX_HOME/resources/rpt2.dtd $PROX_HOME
   ```

   Proximity requires that the DTD `rpt2.dtd` be in the same directory as the RPT XML file to be displayed, (`$PROX_HOME` if you are following the tutorial).
2. From the **Model** menu, choose **Graph RPT**. Proximity displays the Open dialog.
3. Navigate to the current working directory (`$PROX_HOME` if you are following the tutorial) and choose `ProxWebKB_RPT.xml`. Click **Open**.

   Proximity opens a window showing a tree view of the RPT learned for this data.



4. If the graph does not fit the window properly, you can change the magnification level to adjust the display.
   - To make the graph fit the current window, choose **Zoom to Fit** from the **View** menu. Proximity adjusts the size of the graph to fit the window. Proximity cannot shrink the graph beyond a certain size, so large graphs may still extend beyond small window borders.
   - To display the graph at the default magnification, choose **Center on Root** from the **View** menu.
   - To zoom in, click the **+** button at the bottom of the graph window.
   - To zoom out, click the **-** button at the bottom of the graph window.

   You can drag the graph to a different position in the window, or use the bottom and side scroll bars to reposition the graph in the window.
5. You can optionally save the tree as a JPEG file.

   a. In the RPT display window, choose **Save as JPEG Image** from the **File** menu. Proximity displays the Save dialog.

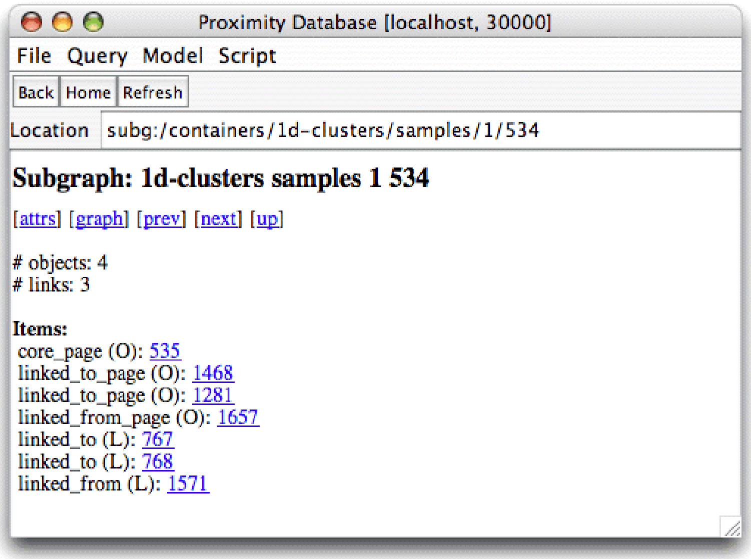
   b. Navigate to the target directory and enter a name for the JPEG file. Click **Save**.

The following section discusses how to interpret the displayed RPT.

Understanding relational probability trees

An RPT represents a series of questions to ask about an item to be classified and its relational neighborhood. At each node in the tree, the RPT can ask questions about any of the attributes used to train and test the model. The answers to those question determine the path through the tree. The leaf nodes in the tree tell us the class label predicted by the model. In this example, the target item is a web page, and the model predicts the value for the page's pagetype attribute.

To see how the model makes predictions, we examine an example web page, object 535, and see how the RPT classifies this object. As we can see from the RPT, to classify this page we need to know the attribute values not just for the target object, but also for the objects that link to and are linked from this object. In the 1d-clusters subgraphs, these related objects are identified as *linked_to_page* and *linked_from_page* objects, respectively. The subgraph whose *core_page* object is 535 is shown below.



subg:/containers/1d-clusters/samples/1/534

Subgraph: 1d-clusters samples 1 534

[attrs] [graph] [prev] [next] [up]

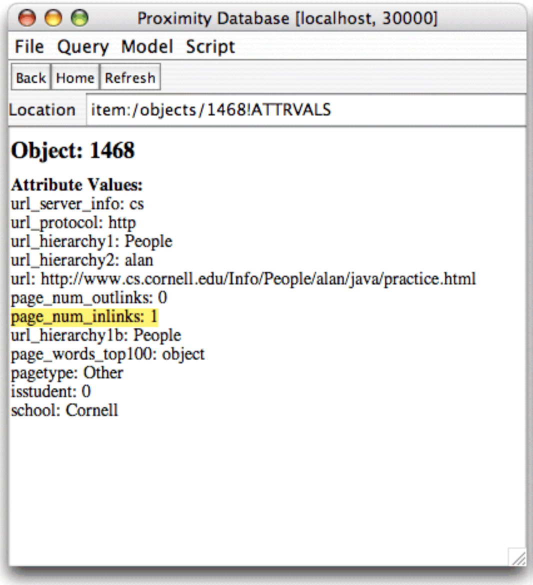
objects: 4
links: 3

Items:
core_page (O): 535
linked_to_page (O): 1468
linked_to_page (O): 1281
linked_from_page (O): 1657
linked_to (L): 767
linked_to (L): 768
linked_from (L): 1571

The root node in the RPT starts by examining the value of the page_num_inlinks attribute for the *linked_to_page* objects in the current subgraph (denoted by linked_to_page.page_num_inlinks in the RPT). If the proportion of all linked_to_page objects having page_num_inlinks = 1 is greater than or equal to 0.083, we traverse the left edge in the RPT.

> For all nodes in the RPT, if the subgraph matches the feature specified in a node, we traverse the left or "yes" edge; if the subgraph does not match the node's feature, we traverse the right or "no" edge. Hover the mouse over an edge in the tree to see its label.

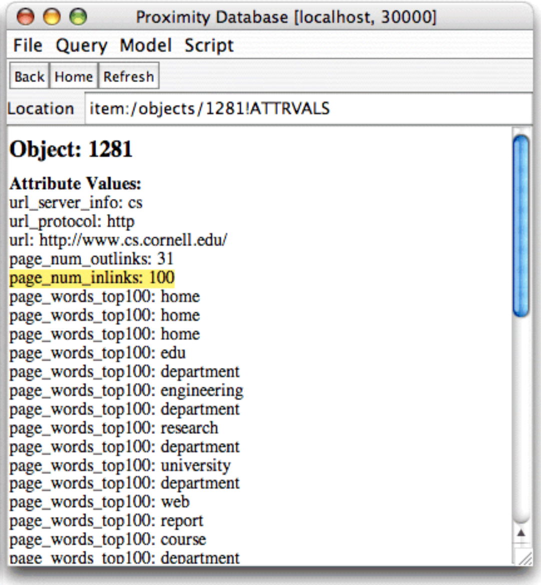
Subgraph 534 has two *linked_to_pages*, so we check the value of the page_num_inlinks attribute for these pages.



item:/objects/1468!ATTRVALS

Object: 1468

Attribute Values:
url_server_info: cs
url_protocol: http
url_hierarchy1: People
url_hierarchy2: alan
url: http://www.cs.cornell.edu/Info/People/alan/java/practice.html
page_num_outlinks: 0
page_num_inlinks: 1
url_hierarchy1b: People
page_words_top100: object
pagetype: Other
isstudent: 0
school: Cornell



item:/objects/1281!ATTRVALS

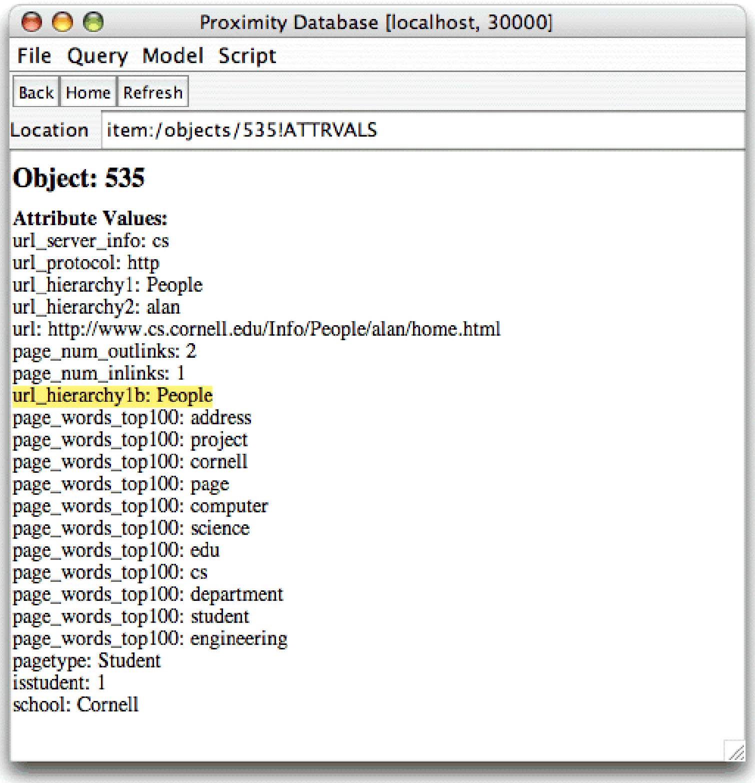
Object: 1281

Attribute Values:
url_server_info: cs
url_protocol: http
url: http://www.cs.cornell.edu/
page_num_outlinks: 31
page_num_inlinks: 100
page_words_top100: home
page_words_top100: home
page_words_top100: home
page_words_top100: edu
page_words_top100: department
page_words_top100: engineering
page_words_top100: department
page_words_top100: research
page_words_top100: department
page_words_top100: university
page_words_top100: department
page_words_top100: web
page_words_top100: report
page_words_top100: course
page_words_top100: department

Object 1468 has a value of 1 for its page_num_inlinks attribute, and object 1281 has a value of 100 for its page_num_inlinks attribute. The proportion of linked_to_pages with a value of 1 is thus 0.50, which is greater than 0.083. Therefore, subgraph 534 satisfies the first test, and we traverse the left edge in the RPT to the next node in the tree.

The next test looks at how many *linked_to_page* objects are linked to the core object in the subgraph, represented by degree(linked_to_page) in the tree. Specifically, we test whether the subgraph contains at least one *linked_to_page* object. Subgraph 534 contains two *linked_to_page* objects, so we pass this test and again traverse the left edge to the next node in the tree.

The third node examines the value of the url_hierarchy1b attribute for the subgraph's *core_page* object. Specifically, the RPT asks whether the value of this attribute is courses. Because this node examines a single value for a single object, it does not require an aggregation function like proportion or degree. This is indicated by the nop (no operator) aggregation function as nop([core_page.url_hierarchy1b])=courses.



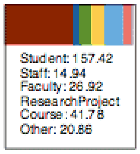
item:/objects/535!ATTRVALS

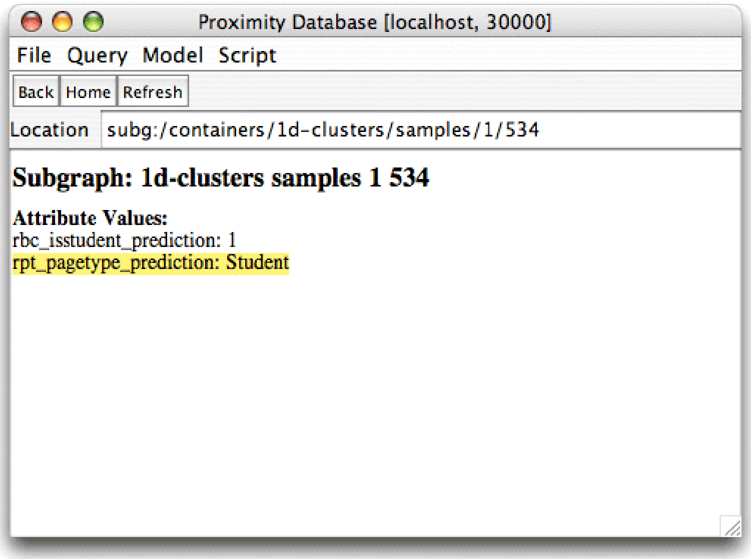
Object: 535

Attribute Values:
url_server_info: cs
url_protocol: http
url_hierarchy1: People
url_hierarchy2: alan
url: http://www.cs.cornell.edu/Info/People/alan/home.html
page_num_outlinks: 2
page_num_inlinks: 1
url_hierarchy1b: People
page_words_top100: address
page_words_top100: project
page_words_top100: cornell
page_words_top100: page
page_words_top100: computer
page_words_top100: science
page_words_top100: edu
page_words_top100: cs
page_words_top100: department
page_words_top100: student
page_words_top100: engineering
pagetype: Student
isstudent: 1
school: Cornell

The value of url_hierarchy1b for object 535 is People, so this time we fail the test and traverse the right edge to the leaf node.

Leaf nodes in the RPT tell us the probabilistic counts (out of all objects from the training set that reach this leaf node) for each potential classification of this object. Because some training instances may not have the information necessary to determine which path to take at a split (e.g., it may not have a value for the relevant attribute), the model assigns an appropriate percentage of the instance to each path, resulting in non-integer values at the leaf nodes. We can see that objects that reach this leaf node are much more likely to be student pages than any other page type. Therefore, the model predicts that object 535 has a pagetype of Student, which our script stored in the rpt_pagetype_prediction attribute for the subgraph.



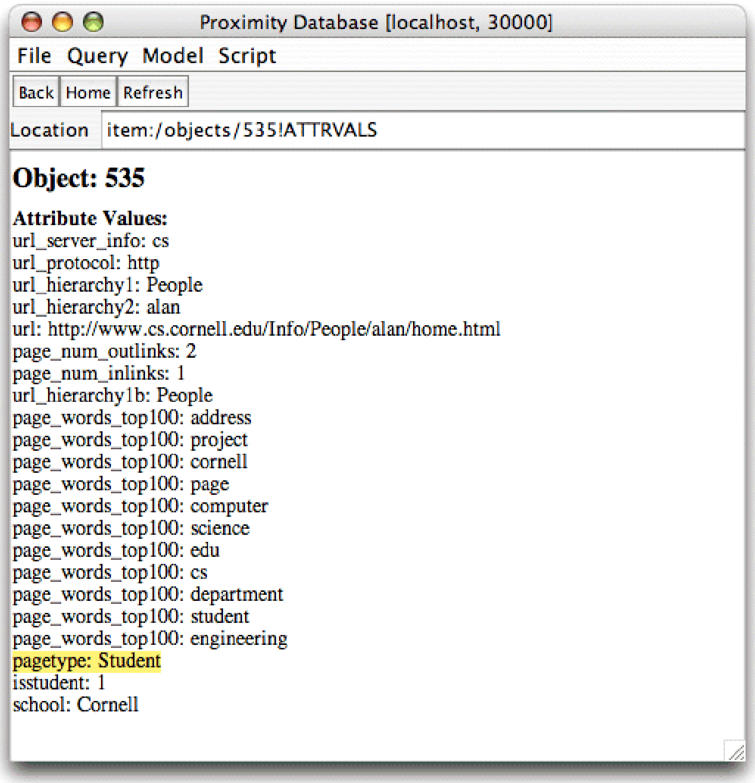


subg:/containers/1d-clusters/samples/1/534

Subgraph: 1d-clusters samples 1 534

Attribute Values:
rbc_isstudent_prediction: 1
rpt_pagetype_prediction: Student

Examining the attributes of object 535, we see that its pagetype attribute is indeed Student and that the model made the correct prediction for this subgraph.



item:/objects/535!ATTRVALS

Object: 535

Attribute Values:
url_server_info: cs
url_protocol: http
url_hierarchy1: People
url_hierarchy2: alan
url: http://www.cs.cornell.edu/Info/People/alan/home.html
page_num_outlinks: 2
page_num_inlinks: 1
url_hierarchy1b: People
page_words_top100: address
page_words_top100: project
page_words_top100: cornell
page_words_top100: page
page_words_top100: computer
page_words_top100: science
page_words_top100: edu
page_words_top100: cs
page_words_top100: department
page_words_top100: student
page_words_top100: engineering
pagetype: Student
isstudent: 1
school: Cornell

Using temporal attributes in a relational probability tree

In some datasets, objects or links may have an attribute that specifies a timestamp associated with that object. For example, a movie object might have a date attribute whose value is the movie's opening date. Proximity allows you to use such temporal attributes to restrict the set of related items considered by an

RPT. (More specifically, it uses the temporal attribute to restrict which values for another attribute can be aggregated by the RPT.) To use temporal attributes, we provide a `TemporalAttributeSource` instead of an `AttributeSource` to specify the related items and attributes that the RPT is to consider.

To illustrate how to use temporal attributes in an RPT, consider a database of actors linked by co-starring relationships (an actor is linked to another actor if both appeared in the same movie). The task for our RPT is to estimate the likelihood that an actor has won an award. Each actor has attributes for the actor's birth year and for any awards won (birth_year and num_awards_won, respectively).

As input to the RPT, we construct 1-dimensional "stars" around each actor. That is, each subgraph contains a *core-actor* object and zero to many linked *co-starring-actor* objects. We consider the number of awards won by an actor's costars in estimating whether the core actor has won any awards, but we only want to consider awards won by an actor's peers in terms of age. In this example, we consider awards won by costars born within ten years (five years before to five years after) of the core actor, but not those won by actors born earlier or later. That is, we use the temporal attribute birth_year to restrict which num_awards_won values are aggregated. Only those values for actors born within the specified range are aggregated; the rest are ignored.

The following text describes how to set up an RPT to use the temporal birth_year attribute in this manner.

Before setting up the input source, we must import the necessary class definitions:

```
from kdl.prox.model2.common.sources import AttributeSource
from kdl.prox.model2.common.sources import TemporalAttributeSource
```

To specify the temporal range relative for the birth_year attribute, we first create an `AttributeSource` that indicates which attribute on the core item the range is relative to.

```
coreTemporalAttr = AttributeSource('core-actor', 'birth_year')
```

Here `core-actor` is the name of the core item, and `birth_year` is the name of the attribute on the core item that the related item's attribute will be relative to.

We also must create an `AttributeSource` that specifies the related item and the temporal attribute on that item that is considered relative to the core item's attribute.

```
relatedTemporalAttr = AttributeSource('co-starring-actor', 'birth_year')
```

In this case, values of the attribute on the related item are considered relative to the value of the same attribute (birth_year) on the core item, but this need not always be the case. We might consider actors with birth years less than 30 years before a movie's opening date, for example. The only requirement is that both attributes must be of comparable types.

The following code specifies the range around the core item's value for birth_year that we allow for the related item's birth_year values. In this case, we consider related actors born within ten years (five years before to five years after) of the core actor.

```
interval = "-5.0:5.0"
```

We then create an `AttributeSource` that specifies the item whose values are to be aggregated (co-starring-actor) and the attribute whose values we want to aggregate (num_awards_won).

```
attrOfInterest = AttributeSource('co-starring-actor','num_awards_won')
```

The above sources are used to specify the `TemporalAttributeSource` for this model. As we did for the non-temporal RPT example, above, the `TemporalAttributeSource` is placed in a list of sources.

```
inputSources = [ \
   TemporalAttributeSource(attrOfInterest, coreTemporalAttr, \
      relatedTemporalAttr, interval) \
   ]
```

As we saw in the previous RPT example, the list of input sources can specify multiple sources, including additional temporal or non-temporal sources.

You can also use a temporal attribute to determine how structural features are aggregated. For example, you could use the birth_year of co-starring-actor objects to determine which objects are counted when determining the degree of the core item. In such cases, use a `TemporalItemSource` (the temporal equivalent of an `ItemSource`) to define the input source.

You can specify multiple temporal attributes and mix temporal attributes and non-temporal attributes in the list of `inputSources`. You can also specify multiple ranges for a given temporal attribute. You must provide a separate `TemporalAttributeSource` or `TemporalItemSource` for each range.

Proximity imposes two important restrictions on the use of temporal attributes in RPTs:

1. Temporal attributes must be single valued. Multiple values and ranges are not permitted.
2. The related temporal attribute must be on the same item as the attribute to be aggregated. In our example, the birth-year attribute is on the same object as the aggregated num_awards_won attribute.

The attribute used for the `coreTemporalAttr` need not necessarily be located on the core object of the subgraph. The only requirement is that the subgraph contain only a single instance of the “core” temporal attribute. (Other items in the subgraph can also have this attribute, but there must be only one object or link that matches the definition of `coreTemporalAttr`.) Because subgraphs can only contain a single core object, we typically use an attribute on the core object for the `coreTemporalAttr`.

Relational Dependency Networks

A relational dependency network (RDN) [Neville and Jensen, 2003], [Neville and Jensen, 2004] is a graphical model that extends the concept of a dependency network [Heckerman, et al., 2000] for relational domains. RDNs approximate a joint probability distribution over the attributes of objects in a network with a set of conditional probability distributions. The RDN learning algorithm is based on pseudolikelihood techniques, which estimate a set of conditional probability distributions independently. This approach avoids the complexities of estimating a full joint distribution and can incorporate existing techniques for learning conditional probability distributions of relational data (e.g., RPTs). Gibbs sampling inference techniques are used to recover a full joint distribution and to estimate probabilities of interest.

The example below continues the task of classifying web pages. The web pages in the ProxWebKB database use the pagetype attribute to indicate a page’s type. We train a new RPT to use as the conditional probability distribution for the pagetype attribute in the RDN. The RDN uses this conditional probability distribution (i.e., this RPT) to collectively infer the value of the pagetype attribute for all of the core objects in the test set.

Code example: run-1d-clusters-rdn.py

This section describes the script found in `$PROX_HOME/doc/user/tutorial/examples/run-1d-clusters-rdn.py`.

Import the necessary class definitions.

```
from kdl.prox.model2.common.sources import *
from kdl.prox.model2.rpt import RPT
from kdl.prox.model2.rdn RDN
from kdl.prox.model2.rdn.modules.listeners import LoggingListener
```

Get the training and test sets. We use the same containers for the training and test sets as we did for the previous RBC and RPT examples.

```
trainContainer = prox.getContainer("1d-clusters/samples/0")
testContainer = prox.getContainer("1d-clusters/samples/1")
```

Train an RPT that predicts the value of the pagetype attribute. For this RPT, we also consider the value of pagetype for related objects in predicting its value for the core object. Because we may not know the value of pagetype for the related objects during inference, the RDN uses the conditional probability distribution represented by the RPT in a Gibbs sampling procedure to collectively infer the value of pagetype for all core objects simultaneously.

See Exercise 7.2 for a more detailed description of the data structures used in the RPT portion of this script.

```
coreItemName = 'core_page'
attrToPredict = 'pagetype'
classLabel = AttributeSource(coreItemName, attrToPredict)
```

Define the set of sources to be used in learning the RPT.

```
inputSources = [ \
   AttributeSource('core_page', 'url_server_info'), \
   AttributeSource('core_page', 'url_hierarchy1b'), \
   AttributeSource('linked_from_page', 'page_num_outlinks'), \
   AttributeSource('linked_from_page', 'pagetype'), \
   AttributeSource('linked_to_page', 'page_num_inlinks'), \
   AttributeSource('linked_to_page', 'pagetype'), \
   ItemSource('linked_from_page'), \
   ItemSource('linked_to_page') ]
```

Begin the modeling portion of the script by instantiating the component RPT. Set the maximum tree depth to three.

```
print "Beginning modeling section"
print "Instantiating component RPT..."
rpt = RPT()
rpt.learningModule.stoppingModule.setMaxDepth(3)
```

Train (learn) the tree.

```
print "Learning component RPT..."
rpt.learn(trainContainer, classLabel, inputSources)
```

Write the RPT to an XML file. The file is written to the current working directory, which is `$PROX_HOME` if you are following the tutorial.

```
xmlFileName = 'ProxWebKB_RPTforRDN.xml'
rpt.save(xmlFileName)
print "RPT written to ", xmlFileName
```

Begin the RDN portion of the script by instantiating the RDN using the default constructor.

```
print "Instantiating RDN..."
rdn = RDN()
```

Like the RPT code, Proximity’s RDN code has been modularized to permit easier maintenance and additions to the code. Use these modules to override the default values for the model’s parameters.

RDNs use Gibbs sampling for inference. Use the statistics module to specify the parameters for the Gibbs sampling. For this example, we skip the first 100 trials (*burnIn*) before beginning sampling and record every third trial. A value of 2 means that we skip two trials between recordings.

```
rdn.statisticModule.setBurnInSteps = 100
rdn.statisticModule.setSkipSteps = 2
```

The example script stops after 200 iterations to limit execution time for the purposes of the this tutorial. Determining the appropriate number of Gibbs sampling iterations can require judgment and experience with this technique. Many more iterations will likely be needed in practice.

```
numIterations = 200
```

Finally, to help us trace script execution, we print a logging statement every 10 iterations.

```
rdn.addListener(LoggingListener(10))
```

Because the RPT has already been trained, there is no separate training step in this script and we can apply the RDN to the test container. Each component RPT makes predictions about the subgraphs in the test container. Applying the RDN returns a map of RPTs to `Predictions` objects. In this example, we have a single component RPT.

```
print "Applying RDN..."
predictionMap = rdn.apply({rpt: testContainer}, numIterations)
rptPredictions = predictionMap.get(rpt)
```

As we saw in Exercise 7.2, we have to tell the RPT where to find the true values for the class labels.

```
rptPredictions.setTrueLabels(testContainer, classLabel)
```

Write the predictions to the database as attributes on the subgraphs in the training container. The RDN uses Gibbs sampling to jointly estimate the marginal probabilities for each of its component models (the single RPT in this case). The RDN then sets the predictions in each component model. Therefore, we write out the predictions from this component RPT rather than the RDN.

```
print "Writing predictions..."
rdnAttrName = "rdn_pagetype_prediction"
rptPredictions.savePredictions(testContainer.getSubgraphAttrs(), rdnAttrName)
```

Evaluate the RDN.

```
print "Computing accuracy..."
acc = (1 - rptPredictions.getZeroOneLoss())
```

To compute area under the ROC curve we need to know which pagetype value is considered to be a positive instance. A student page (a positive instance) has a value of “Student” for the pagetype attribute.

```
print "Computing area under ROC curve..."
auc = rptPredictions.getAUC('Student')
```

Print a summary of evaluation results.

```
print "RDN results:"
print "  Accuracy:                        ", str(acc)
print "  Area under ROC curve (Student):  ", str(auc)
```

Exercise 7.4. Learning and applying the relational dependency network model:

This script requires entities created in Exercise 6.4 and Exercise 7.2. You must have completed these exercises before running the script in the current exercise.

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. If you have not already done so, copy `rpt2.dtd` to the same directory as that containing the saved RPT XML file, `ProxWebKB_RPT.xml`.

   ```
   > cp $PROX_HOME/resources/rpt2.dtd $PROX_HOME
   ```

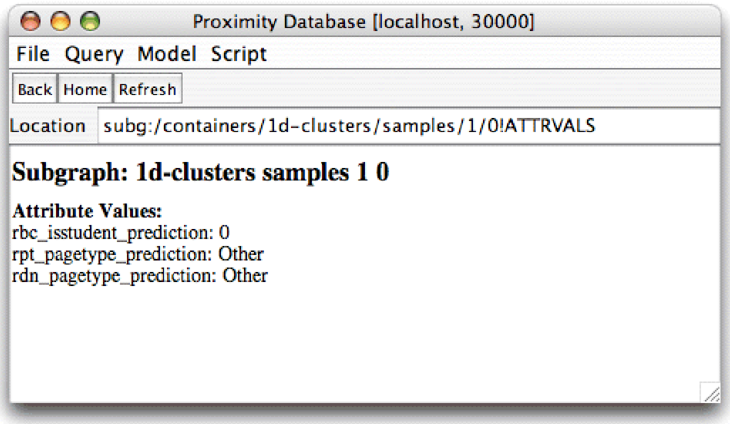
   Proximity requires that the DTD file `rpt2.dtd` be in the same directory as the RPT file to be read.
2. From the **Script** menu, choose **Run Script**. Proximity displays the Open dialog.
3. Navigate to the `$PROX_HOME/doc/user/tutorial/examples` directory and choose `run-1d-clusters-rdn.py`. Click **Open**.

   Proximity opens a window to show you the output from the script along with a trace of the script execution. The `run-1d-clusters-rdn.py` script may take many minutes to run. Your output should look similar to the following trace (a portion of the trace as well as leading information showing elapsed time and execution thread have been omitted from the trace for brevity):

```
Status: starting running script:
   /proximity/doc/user/tutorial/examples/run-1d-clusters-rdn.py
Beginning modeling section
Instantiating model...
Inducing model...
INFO kdl.prox.model2.rpt.RPT - Creating feature tables
INFO kdl.prox.model2.rpt.RPT - Done creating feature tables: 908 features.
INFO kdl.prox.model2.rpt.modules.learning.DefaultLearningModule -
   Choosing split for 2068 subgs

    portion of trace deleted

RPT written to  ProxWebKB_RPTforRDN.xml
Instantiating RDN...
Applying RDN...
INFO kdl.prox.model2.rdn.RDN - RDN Iteration: 0
INFO kdl.prox.model2.rdn.RDN - RDN Iteration: 10
INFO kdl.prox.model2.rdn.RDN - RDN Iteration: 20

    portion of trace deleted

INFO kdl.prox.model2.rdn.RDN - RDN Iteration: 190
INFO kdl.prox.model2.rdn.RDN - RDN Iteration: 200
Writing predictions...
Computing accuracy...
Computing area under ROC curve...
RDN results:
  Accuracy:                          0.8180938558297048
  Area under ROC curve (Student):    0.8578509392814735
Status: finished running script
```

Note that some parts of the RPT model used in creating the RDN are non-deterministic, so your results may differ slightly from that shown above. You can close this window after the script finishes.

4. Examine the values predicted by the RDN. Drill down through the container hierarchy in the Proximity Database Browser to display the list of subgraphs for the /1d-clusters/samples/1 container. Click a subgraph ID, then click **attrs** to display the attributes for that subgraph. The example below shows the value for subgraph 0, which shows that the model predicts that the core page for this subgraph (object 1) has a pagetype of Other You can compare this to the actual value of pagetype by examining the attribute values for object 1.



subg:/containers/1d-clusters/samples/1/0!ATTRVALS

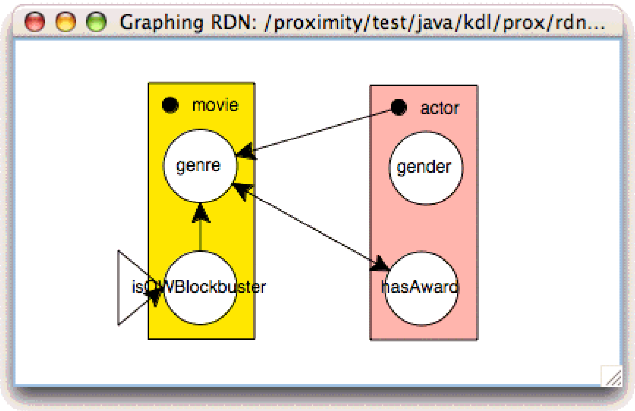
Subgraph: 1d-clusters samples 1 0

Attribute Values:
rbc_isstudent_prediction: 0
rpt_pagetype_prediction: Other
rdn_pagetype_prediction: Other

Recall that the RDN only makes predictions for core objects in the test container. Therefore, after learning the model, only the subgraphs in the 1d-clusters/samples/1 container have a value for the rdn_pagetype_prediction attribute.

Viewing relational dependency networks

Proximity provides an RDN viewer to graphically illustrate the dependencies in the RDN. The figure below shows an artificial RDN constructed to illustrate its components.



This RDN describes the probabilistic dependencies between attributes of movies and actors. Each plate corresponds to an object type, circles represent attributes on those types, and arrows indicate probabilistic dependencies among these attributes.

In the example above, movie genre depends on movie success (represented by the isOWBlockbuster attribute, which indicates whether the movie is an opening weekend blockbuster). That is, if we know whether a movie grossed over two million dollars in its opening weekend, we are better able to predict its genre. (Strictly speaking, the conditional probability distribution for genre includes movie success.) Similarly, predicting genre also depends on whether the movie's actors have won any awards. In this case, the arrow points in both directions indicating that predicting whether a movie's actors have won any awards depends on genre, as well.

The loop leaving and re-entering the movie plate connecting isOWBlockbuster to itself indicates autocorrelation. The success of a movie can be predicted more accurately if we know the success of other movies that link directly or indirectly to the target object. In this case, we link indirectly through studio objects—if other movies made by the same studio are successful, we are more likely to predict that the current movie will have a big opening weekend. Because RPTs do not represent these intermediate objects, the RDN viewer does not show these connecting studio objects.

The arrow from the large dot on the actor plate to movie genre indicates that genre also depends on the degree of related actor objects (how many actors are linked to this movie in the database). Linking to the dot indicates that this dependency involves a graph feature (degree) rather than one based on attribute values.

To view an RDN, you must construct an RDN "wrapper" file that lists the RPTs that were used by the RDN and identifies different labels (from these RPTs) that refer to the same object class. The example below shows such a wrapper file for the RDN created in Exercise 7.4.

```
<!DOCTYPE rdn SYSTEM "rdn.dtd">
<rdn>
  <rpt-files>
    <file>ProxWebKB_RPTforRDN.xml</file>
  </rpt-files>
  <item-maps>
    <map>
      <from>core_page</from>
      <to>linked_to_page</to>
    </map>
    <map>
      <from>core_page</from>
      <to>linked_from_page</to>
    </map>
  </item-maps>
</rdn>
```

The `<rpt-files>` element lists the component RPTs, each enclosed by `<file>` tags. Our RDN uses only a single RPT, `ProxWebKB_RPTforRDN.xml`.

The `<item-map>` section contains a sequence of `<map>` elements. Each `<map>` element contains a pair of `<item>` names from the RPT, where both names refer to the same type of object in the database. In this example, all database objects are web pages, so we map all the object labels to `core_object`. The name in the `<from>` element is used to label the corresponding plate in the RDN viewer.

Exercise 7.5. Viewing relational dependency network graphs:

This script requires entities created in Exercise 7.4. You must have completed this exercise before running the script in the current exercise.

Before beginning, make sure that you are serving the ProxWebKB database using Mserver. Start the Proximity Database Browser if it is not already running.

1. All files required for viewing the RDN must be in the same directory. If you have not already done so, copy `rpt2.dtd` and `rdn.dtd` to the directory containing the saved RPT XML file, `ProxWebKB_RPTforRDN.xml`.

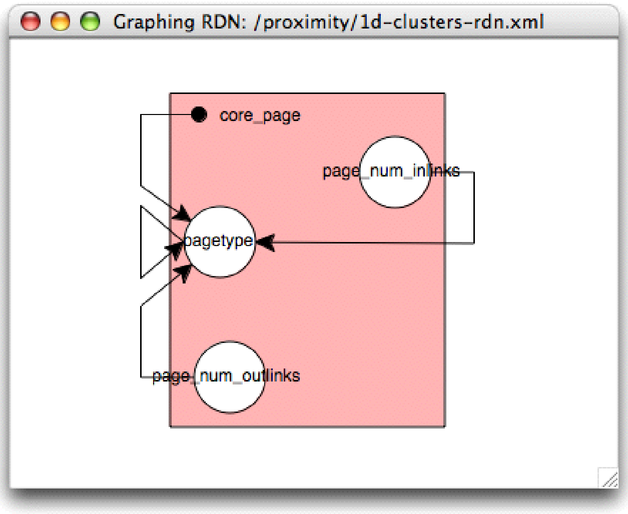
   ```
   > cp $PROX_HOME/resources/rpt2.dtd $PROX_HOME
   > cp $PROX_HOME/resources/rdn.dtd $PROX_HOME
   ```

   Proximity requires that `rpt2.dtd` be in the same directory as the component RPT XML files and that `rdn.dtd` be in the same directory as the RDN wrapper file.
2. Copy the RDN wrapper file to the same directory.

   ```
   > cp $PROX_HOME/doc/user/tutorial/examples/1d-clusters-rdn.xml $PROX_HOME
   ```

3. From the **Model** menu, choose **Graph RDN**. Proximity displays the Open dialog.
4. Navigate to the directory containing the RDN files (`$PROX_HOME` if you are following the tutorial) and choose `1d-clusters-rdn.xml`. Click **Open**.

   Proximity opens a window showing a graph of the RDN learned for this data.



   You may need to drag some of the RDN elements to see the overlapping lines more clearly.

   The RDN for the ProxWebKB data shows the probabilistic dependencies used to predict the value of a page’s pagetype attribute. Specifically, the value of a page’s pagetype attribute depends on the

the known values of page_num_inlinks and page_num_outlinks for related pages and the inferred value of pagetype for related pages. The link from the core_page dot in the upper left corner to the pagetype attribute indicates that pagetype is also dependent on structural features (i.e., the degree) of neighboring pages. All of the links in the RDN graph leave and re-enter the core_page plate, indicating that we use the attributes of related pages to predict the value of pagetype, rather than the attributes of the target page itself.

Tips and Reminders

- Save models to XML files for later use without having to rerun the model code.
- Save predictions to the database to preserve the predicted values.
- Instantiate models using the corresponding constructor, then modify model parameters by adding the appropriate modules to the model instance.
- Model input sources can include attribute values (`AttributeSource`) and structural characteristics (`ItemSource`) for the target item as well as for entities in its relational neighborhood.
- Specify temporal input sources (`TemporalAttributeSource` and `TemporalItemSource`) to use temporal attributes to restrict how other attributes values are aggregated.

Appendix A. Proximity Quick Reference

MonetDB Server

The **Mserver** command is in `/usr/local/Monet-mars/bin`. You can add this directory to `PATH` or specify the full path on the command line.

To start the the MonetDB server on the default port (30000):

```
> Mserver --dbname name $PROX_HOME/resources/init-mserver.mil
```

where *name* is the name of the database to be served. The database name is the name of the directory under `/usr/local/Monet-mars/var/MonetDB4/dbfarm/` in which the database files are stored.

Proximity uses the port number to select the appropriate protocol for the version of MonetDB being used. You must use a port number ≤ 40000 for the Mars versions (4.18.2 and 4.20) and a port number > 40000 for MonetDB 4.6.2.

To start the MonetDB server on a different port:

```
> Mserver --dbname name $PROX_HOME/resources/init-mserver.mil --set port=nnnnn
```

where

- *name* is the name of the database to be served
- *nnnnn* is the port number

To stop the MonetDB server:

```
MonetDB> quit();
```

Proximity Shell Scripts and Batch Files

The following table lists the shell scripts and batch files for launching Proximity applications. You can also execute queries and run scripts from the Proximity Database Browser. See the text following this table for additional details on selected parameters.

Application	Linux/Mac OS X shell script	Windows batch file	Parameters	Reference
Proximity Database Browser	gui.sh	gui.bat	*host:port*	Ch. 4
Query runner	query.sh	query.bat	*host:port* *query-file* *output-container* *input-container* (opt.)	Ch. 5
Python script runner	script.sh	script.bat	*host:port* *script-file*	Ch. 6

Application	Linux/Mac OS X shell script	Windows batch file	Parameters	Reference
XML data import	import-xml.sh	import-xml.bat	*host:port* *input-filename*	Ch. 3
XML data export	export-xml.sh	export-xml.bat	*host:port* *output-filename* *exportType* (opt.) *exportSpec* (opt.)	Ch. 3
plain text data import	import-text.sh	import-text.bat	*host:port* *input-filename*	Ch. 3
plain text data export	export-text.sh	export-text.bat	*host:port* *output-filename* *exportType* (opt.) *exportSpec* (opt.)	Ch. 3
Database management utilities	db-util.sh	db-util.bat	*host:port* *command*	Ch. 3

For **query.sh**, specifying the input container is optional; the input container defaults to the root container (the entire database) if it is omitted. To explicitly specify the input container, Proximity uses a path-like syntax that reflects the container hierarchy for the specified container. Container names are separated by forward slashes with the initial `/` representing the root container, e.g., “`/1d-clusters/samples`”.

For **export-xml.sh**, specifying the exportType and exportSpec is optional; the export defaults to the full database if they are omitted.

For **db-util.sh**, permitted commands are

- **`clear-db`** - clear the database (deletes all database content)
- **`init-db`** - initialize the database
- **`test-db`** - test the database connection and print Proximity version information
- **`view-schema`** - print the schema log (a list of the database’s schema versions)
- **`view-stats`** - print summary statistics for the database

All Proximity shell scripts and batch files include `proximity.jar.` You must rebuild `proximity.jar` to have source code changes available when you use the Proximity shell scripts and batch files.

Query Editor Keyboard Shortcuts

The following tables lists the keyboard shortcuts for working with the Proximity Query Editor.

Command	Keyboard shortcut	Command	Keyboard shortcut
New query	Ctrl-N	Selection tool	Ctrl-1
Open query file	Ctrl-O	Vertex tool	Ctrl-2
Close query	Ctrl-W	Edge tool	Ctrl-3
Save query	Ctrl-S	Subquery tool	Ctrl-4
Run query	Ctrl-R	Flip edge direction	Ctrl-F
Select next item	Ctrl-→	Select previous item	Ctrl-←
Zoom in	Ctrl-=	Zoom to fit	Ctrl-0 (zero)
Zoom out	Ctrl-Minus	Reset zoom	Ctrl-Backspace

Proximity Python Interpreter Commands

The following table lists the keyboard shortcuts for working with the Proximity interactive Python interpreter. Method name completion and parameter list display are enabled only for command lines that begin with a variable name.

Command	**Description**	**Command**	**Description**
Ctrl-Space	List method completions	Ctrl-P	List parameter options
Up arrow	Previous command in history	Down arrow	Next command in history
Right arrow, Ctrl-B	Move backward one character	Left arrow, Ctrl-F	Move forward one character
Ctrl-A	Move to beginning of line	Ctrl-E	Move to end of line
Del	Delete one character backward	Ctrl-D	Delete one character forward
Ctrl-K	Kill to end of line	Ctrl-Y	Paste contents of clipboard
Ctrl-Alt-S	Save command history	Ctrl-Alt-C	Clear command history
Ctrl-Alt-X	Execute Jython file		

Location Bar Path Syntax

The following table lists the location bar paths for displaying sets of database entities. These correspond to the choices available from the database view links on the Proximity Database Browser start page:

Sets of entities	
Database entity	*Location bar path*
objects	`db:/objects`
links	`db:/links`
containers	`cont:/containers`
object attributes	`attrdefs:/objects`
link attributes	`attrdefs:/links`
container attributes	`attrdefs:/containers`

The following table lists the location bar paths for displaying information for individual objects, links, containers, subgraphs, and their attributes. In each of these paths you must substitute the appropriate name or ID number as shown.

Individual entities	
Database entity	*Location bar path*
specific object	`item:/objects/`*`object-id`*
specific link	`item:/links/`*`link-id`*
specific container	`cont:/containers/`*`container-path`*
specific subgraph	`subg:/`*`container-path/subg-id`*

Container names are separated by forward slashes with the initial `/` representing the root container, e.g., “`/1d-clusters/samples`”.

Proximity permits the use of the following location bar path modifiers at the end of the path:

`!ATTRVALS`	Display attributes and values for the current item
`#pagenum`	Display page *`pagenum`* of values for the current item

If both are used, the page number modifier must follow the attribute display modifier, e.g.,

```
item:/objects/23!ATTRVALS#2
```

DTD Files

Proximity uses document type definitions (DTDs) to define the required syntax for files containing queries, models, and data to be imported. Proximity requires that the appropriate DTD be in the same directory as the corresponding query or XML data file. Copy the appropriate DTD from `$PROX_HOME/resources/` to any directories containing these files.

DTD	**Required for**
`graph-query.dtd`	query file format
`prox3db.dtd`	import/export XML file format
`rbc2.dtd`	RBC file format
`rpt2.dtd`	RPT file format
`rdn.dtd`	RDN wrapper file format

Technical Support and Documentation

Please use the following addresses to report any problems or questions you have about Proximity:

- *proximity-bugs@kdl.cs.umass.edu* - Proximity bug reports and documentation errors
- *proximity-support@kdl.cs.umass.edu* - requests for general assistance with Proximity software
- *proximity@kdl.cs.umass.edu* - general comments, suggestions, and criticism

The Knowledge Discovery Laboratory maintains two mailing lists for Proximity news and information.

- *Proximity-announce* is a low-volume list carrying only announcements of significant project milestones, such as new releases. To subscribe, send an email message to *majordom@cs.umass.edu* with the text **`subscribe proximity-announce`** in the body of the message.
- *Proximity-list* is a general forum for discussing Proximity issues. To subscribe, send an email message to *majordom@cs.umass.edu* with the text **`subscribe proximity-list`** in the body of the message.

The Proximity 4.3 distribution includes the *Proximity Tutorial* (this document) and the *Proximity QGraph Guide* (a detailed description of the QGraph query language). Both documents are available in the directory `$PROX_HOME/doc/user/` in the following files and formats.

Tutorial:

- `tutorial/Tutorial.pdf` (PDF)
- `tutorial/HTML/index.html` (HTML)

QGraph Guide

- `qgraph/QGraphGuide.pdf` (PDF)
- `qgraph/HTML/index.html` (HTML)

Appendix B. Installation

Obtaining Proximity

The Proximity distribution includes the Proximity client and the required Monet database system (MonetDB). The Proximity client is implemented in Java and can be run on any platform that supports Java 2 Platform, Standard Edition v1.5 (J2SE) or later. The distribution includes both binaries and source files for the Proximity client. MonetDB binaries are available for Mac OS X (10.2 or higher), Linux i86 (glib 2.3), and Windows (Windows 2000 Professional or later). Proximity 4.3 operates with MonetDB 4.6.2 and selected later versions.

We recommend that new users download and install MonetDB 4.20 (Linux and Mac OS X) or MonetDB 4.18.2 (Windows), collectively known as the “Mars” versions. Proximity 4.3 is *not* yet fully compatible with the latest release of MonetDB 5.

Current Proximity users upgrading to Proximity 4.3 can continue to use MonetDB 4.6.2 (provided with the current release) or update to MonetDB Mars and recreate their databases. Proximity databases can be easily recreated for use with later versions of MonetDB by exporting existing databases as plain text or XML, then importing the data into the new database. See “Updating MonetDB Databases” later in this appendix for instructions on converting MonetDB 4.6.2 databases for use with MonetDB Mars.

Users updating from Proximity 3.1 or earlier must update both the Proximity client and MonetDB installation, including updating the database schema (a step not covered in this appendix). Please contact technical support at *proximity-support@kdl.cs.mass.edu* for assistance if you are updating from Proximity 3.1 or earlier.

Obtaining the Proximity distribution files

1. In a web browser, go to kdl.cs.umass.edu/proximity/downloads.html.
2. Download the MonetDB binaries for your operating system:

Mac OS X on PPC:	`monet-mars-distr-mac-ppc.tgz`
Mac OS X on PPC-64:	`monet-mars-distr-mac-ppc-64.tgz`
Mac OS X on Intel:	`monet-mars-distr-mac-intel.tgz`
Linux on Intel:	`monet-mars-distr-mac-linux.tgz`
Linux on Intel-64:	`monet-mars-distr-mac-linux-64.tgz`
Windows:	`monet-mars-distr-win.msi`

3. All supported platforms: Download `proximity-4.3.tgz`.

Installing MonetDB

Users updating from Proximity 4.0 or later versions may choose to stay with their current MonetDB installation (MonetDB 4.6.2) or update to MonetDB Mars (4.20 for Linux/Mac OS X, 4.18.2 for Windows). The later MonetDB versions provide some performance improvements, but as the underlying database formats are not compatible, using these versions also requires converting existing databases to the newer format. See “Updating MonetDB Databases” later in this appendix for instructions on converting MonetDB 4.6.2 databases for use with MonetDB Mars.

See “Running the MonetDB database server” in Chapter 2 for information on using the MonetDB server with Proximity. Full MonetDB documentation is available from CWI’s website at monetdb.cwi.nl/projects/monetdb/MonetDB/Version4/Documentation/index.html.

Installing MonetDB for Linux/Mac OS X platforms

You must have administrator privileges (either by using `sudo` or being logged in as “root”) to install MonetDB.

1. Change directories to `/usr/local`.

   ```
   > cd /usr/local
   ```

   The Linux/Mac OS X binary distribution of MonetDB is not relocatable.
2. Extract the distribution. You must have a version of tar (e.g., GNU tar) that supports long file names.

   ```
   > tar xvfz path-to-tarfile/monet-mars-distr-platform.tgz
   ```

   where *`platform`* reflects your operating system.
3. Change permissions for the `Monet-mars/var` directory to permit write access.

   ```
   > cd /usr/local/Monet-mars
   > chmod -R a+w var
   ```

4. [Optional] Add `/usr/local/Monet-mars/bin` to `PATH`.

Installing MonetDB for Windows platforms

The Windows version of the MonetDB sever requires Windows 2000 Professional or later. You must be logged in as a user with administrator privileges to install MonetDB.

1. If you have an older version of MonetDB already installed on your system, remove the following DLL files from the `C:\WINDOWS\System32\` directory:
   - `libstream.dll`
   - `libmutils.dll`
   - `libMonetODBCs.dll`
   - `libMonetODBS.dll`
   - `libMapi.dll`
2. Extract the installation files using WinZip or an equivalent application.
3. Run the Microsoft installer for MonetDB, `monet-mars-distr-win.msi`, by double-clicking the file name to start the installation wizard. Follow the instructions to install the MonetDB server in the desired location. The default installation directory is `C:\Program Files\CWI\MonetDB`.

   Proximity does not require the MonetDB ODBC driver. You can unselect this option on the third page of the installation wizard if you plan to only use MonetDB to serve Proximity databases. You can ignore warnings that “installation will fail if this component is not installed” if you are not installing the MonetDB ODBC driver; this warning refers to installing the MonetDB ODBC driver without having other required software already installed.

   Your choice of installing MonetDB for “Just me” versus “All users” affects only whether MonetDB is added to the Start Menu for the current user or for all users. In all cases MonetDB stores its databases in

   ```
   C:\\Documents and Settings\username\Application Data\MonetDB4\dbfarm\
   ```

   where *`username`* is the login name for the current user.

4. Run **Mserver.bat** (located in the installation directory).

   ```
   > C:\\Documents and Settings\username\Application Data\MonetDB4\dbfarm\Mserver.bat
   ```

   where *`username`* is the login name of the current user. This creates the required directory hierarchy for MonetDB data files. Substitute the appropriate path if you installed the Monet server someplace other than the default location.

5. Enter **`quit();`** at the Monet server prompt to quit the server (note the parentheses and semi-colon).
6. [Optional] Add the MonetDB installation directory to `PATH`.

Installing Proximity

Installing Proximity for Linux/Mac OS X platforms

1. Extract the distribution to the location of your choice.

   ```
   > cd directory
   > tar xvfz path-to-tarfile/proximity-4.3.tgz
   ```

   where *`directory`* is the directory in which you want to install Proximity and *`path-to-tarfile`* is the relative path from the current directory to the directory containing the tar file. You must use a version of tar (e.g., GNU tar) that supports long filenames.
2. Change permissions for the shell scripts in `proximity/bin` to make them executable.

   ```
   > cd proximity
   > chmod -R a+x bin
   ```

3. [Optional] If you plan to work through the Proximity tutorial, set the environment variable `PROX_HOME` to your Proximity installation directory.
4. [Optional] Add `$PROX_HOME/bin` to `PATH`.

Installing Proximity for Windows platforms

1. Extract the distribution to the location of your choice using WinZip or an equivalent application.
2. Create a new environment variable, `PROX_HOME`, and set it to the location where you installed Proximity. For example, if you installed Proximity in `C:\Proximity`, then you would set `PROX_HOME` = `C:\Proximity\`. Setting `PROX_HOME` is required for using the Proximity client on Windows. You may need to restart your command (DOS) window to make the new environment variable available.
3. [Optional] Add `%PROX_HOME%\bin` to `PATH`.

Updating MonetDB Databases

Proximity 4.3 is fully compatible with MonetDB 4.6.2, the version of MonetDB distributed with Proximity 4.0 through 4.2. Users updating from Proximity 4.0 or later versions can continue to use their existing MonetDB installation and databases and safely ignore this section. However, if you choose to update your MonetDB installation to one of the “Mars” versions, you must also update your databases to use the new database format. Users updating from Proximity 3.1 or earlier should contact technical support at *proximity-support@kdl.cs.umass.edu* for assistance.

Updating databases from MonetDB 4.6.2 to Monet Mars (4.18.2 or 4.20) requires exporting the current database to the Proximity plain text or XML data format and then importing the data into the new version of MonetDB. For most import or export operations, we recommend using the more robust Proximity XML format in order to take advantage of its error checking capabilities. In this case, because you are importing unchanged data exported from Proximity, the use of the plain text format is recommended for performance reasons. See Chapter 3, *Importing and Exporting Proximity Data*, for additional information on exporting and importing data using Proximity’s plain text and XML data formats.

Databases converted to MonetDB Mars may not be usable in older versions of MonetDB. We recommend that you backup any database that you might want to use in an older version of MonetDB before converting it to MonetDB Mars.

> Do not delete your current MonetDB 4.6.2 installation until you have completed updating all databases.

Updating databases using plain text

1. If you have not already done so, install the new versions of Proximity and MonetDB. Do not delete MonetDB 4.6.2.
2. Serve the database using MonetDB 4.6.2.

   ```
   >/usr/local/Monet-46/bin/Mserver  --dbname db-name \
   $PROX_HOME/resources/init-mserver.mil --set port=45678
   ```

   where *db-name* is the database name and $PROX_HOME is the location of your local Proximity installation. Using a port number > 4000 tells Proximity to use the MonetDB 4.6.2 protocol.

   Windows users should substitute the appropriate path to their MonetDB 4.6.2 installation's Mserver command.

3. Export the data to plain text files. You can only export data to the same machine as that serving the database.

   ```
   >$PROX_HOME/bin/export-text.sh localhost:45678 data-dir
   ```

   where *data-dir* is the absolute path to the directory that will hold the exported plain text data files.

   Windows users should use **export-text.bat** instead of **export-text.sh**.

4. Quit the MonetDB server.

   ```
   MonetDB> quit();
   ```

5. Serve the new (empty) database. Make sure that you use the correct path to your new Mserver command either by changing PATH or by explicitly including the path on the command line.

   ```
   > /usr/local/Monet-mars/bin/Mserver --dbname db-name \
   $PROX_HOME/resources/init-mserver.mil.
   ```

   where *db-name* is the database name. Note that Proximity 4.3 changes the default port number to 30000. Proximity 4.3 requires a port number ≤ 40000 for MonetDB Mars.

   Windows users should substitute the appropriate path to their MonetDB 4.18.2 installation's Mserver command.

6. Initialize the new database.

   ```
   > $PROX_HOME/bin/db-util.sh localhost:45678 init-db
   ```

   Windows users should use **db-util.bat** instead of **db-util.sh**.
7. Import the plain text data. You can only import data residing on the same machine as that serving the database.

   ```
   > $PROX_HOME/bin/import-text.sh localhost:30000 data-dir
   ```

   where *data-dir* is the absolute path to the directory containing the plain text data files created in step 3.

   Windows users should use **import-text.bat** instead of **import-text.sh**.

Location of database files

MonetDB Mars stores database files in a different directory that that used by earlier versions of MonetDB.

- Database files for Linux and Mac OS X installations are stored in `/usr/local/Monet-mars/var/MonetDB4/dbfarm`.
- Database files for Windows installations are stored in `C:\\Documents and Settings\`*`username`*`\Application Data\MonetDB4\dbfarm\` where *`username`* is the login name of the user who installed MonetDB.

Appendix C. Proximity XML Format

This appendix describes the XML format for importing data into and exporting data out of Proximity. In addition to handling complete databases, Proximity can import and export individual attributes and containers. Users are solely responsible for ensuring that imported attribute and container data correctly matches the identifiers and data types in the existing database.

The DTD for the Proximity XML format is located in `$PROX_HOME/resources/prox3db.dtd`.

> The characters `<`, `>`, and `&`, are represented by the corresponding XML entities, `&lt;`, `&gt;`, and `&amp;`. Proximity makes the conversion when exporting data to the XML format; users must use the appropriate entities in XML data to be imported into Proximity.
>
> For compatibility with MonetDB, single quotes, double quotes, and newline characters in the XML data are automatically changed to underscores during import.

The examples in this appendix are designed to illustrate the relevant XML syntax and are not intended to represent, in whole or in part, a valid or semantically meaningful database.

Declarations

XML declaration

The XML data file must start with the following XML declaration:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Document type declaration

The document type declaration immediately follows the XML declaration. The XML data file must include the following document type declaration:

```
<!DOCTYPE PROX3DB SYSTEM "prox3db.dtd">
```

Note that the document type declaration uses a system identifier for the DTD. Proximity expects to find the `prox3db.dtd` file in the same directory as the XML data file.

The `PROX3DB` root element

The root element for the XML data file is `PROX3DB`. The `PROX3DB` tag must appear immediately after the document type declaration. The XML data file ends with the closing `</PROX3DB>` tag.

`<PROX3DB>`

Attributes None.

Children

`OBJECTS`	zero or one	The objects in the database
`LINKS`	zero or one	The links in the database
`ATTRIBUTES`	zero or one	Attributes for objects, links, and containers
`CONTAINERS`	zero or one	Containers and their subgraphs, including subgraph attributes

Content Model

```
(OBJECTS?, LINKS?, ATTRIBUTES?, CONTAINERS?)
```

Example

```
<PROX3DB>
    <OBJECTS>...</OBJECTS>
    <LINKS>...</LINKS>
    <ATTRIBUTES>...</ATTRIBUTES>
    <CONTAINERS>...</CONTAINERS>
</PROX3DB>
```

Objects

Database objects are specified by `OBJECT` elements, all of which must be included in a single `OBJECTS` element. The `OBJECT` element is empty and includes one required attribute, `ID`, which specifies the value for the object’s identifier. `ID` values must be unique non-negative integers and are typically sequential numbers.

<OBJECTS>

Attributes None.

Children

`OBJECT`	zero or more	The data file includes one `OBJECT` element for each object in the database.

Content Model

```
(OBJECT*)
```

Example See `OBJECT`.

<OBJECT>

Attributes

`ID`	required	ID values must be unique non-negative integers and are typically sequential numbers.

Children None.

Content Model

```
EMPTY
```

Example

```
<OBJECTS>
   <OBJECT ID="1"/>
   <OBJECT ID="2"/>
   <OBJECT ID="3"/>
   <OBJECT ID="4"/>
   <OBJECT ID="5"/>
</OBJECTS>
```

Links

All Proximity links are binary (they connect exactly two objects) and directional. Links are specified by LINK elements, all of which must be included in a single LINKS element. The LINK element is empty; its data is included in three required attributes, described below. Object IDs must correspond to ID numbers specified in the OBJECT elements.

<LINKS>

Attributes None.

Children

LINK	zero or more	The data file includes one LINK element for each link in the database.

Content Model

```
(LINK*)
```

Example See LINK.

<LINK>

Attributes

ID	required	ID values must be unique non-negative integers and are typically sequential numbers.
O1-ID	required	The ID of the start object for this link
O2-ID	required	The ID of the end object for this link

Children None.

Content Model

```
EMPTY
```

Example

```
<LINKS>
   <LINK ID="1" O1-ID="1" O2-ID="2"/>
   <LINK ID="2" O1-ID="2" O2-ID="1"/>
   <LINK ID="3" O1-ID="2" O2-ID="3"/>
   <LINK ID="4" O1-ID="1" O2-ID="4"/>
   <LINK ID="5" O1-ID="2" O2-ID="4"/>
   <LINK ID="6" O1-ID="3" O2-ID="5"/>
</LINKS>
```

Attributes

Attributes on the objects, links, and containers in the database are specified by ATTRIBUTE elements, all of which must be included in a single ATTRIBUTES element. The ATTRIBUTE element has three required attributes, described below. Each ATTRIBUTE element contains a list of ATTR-VALUE elements, one for each instance of that attribute in the database. The ATTR-VALUE element specifies the value of the attribute. Because Proximity supports multi-dimensional attributes, values are enclosed in a COL-VALUE, with one COL-VALUE element for each dimension. The ATTR-VALUE element has one required attribute, ITEM-ID, which specifies the ID of the object or link to which this attribute applies. Include multiple ATTR-VALUE elements when an object or link has multiple values for an attribute.

Although Proximity supports the use of multi-dimensional attributes for data representation, import, and display in the Proximity Database Browser, Proximity does not yet allow the use of multi-dimensional attributes in queries. Furthermore, attribute type definitions (the value of a `DATA-TYPE` element) must be defined using the default format of omitting the column name, or must use the column name “value”, if you want to use that attribute in query conditions or constraints.

Subgraph attributes are specified by `SUBG-ATTRIBUTE` elements.

<ATTRIBUTES>

Attributes None.

Children

`ATTRIBUTE`	zero or more	The data file includes one `ATTRIBUTE` element for each object, link, or container attribute in the database.

Content Model

```
(ATTRIBUTE*)
```

Example See `COL-VALUE`.

<ATTRIBUTE>

Attributes

`NAME`	required	The name of the object or link attribute
`ITEM-TYPE`	required	Specifies whether this is an object (O), link (L), or container (C) attribute
`DATA-TYPE`	required	The data type of the attribute. For multi-column (multi-dimensional) data, use the format “columnName0:dataType0, columnName1:dataType1, ...” Single-column attributes can omit the column name and simply indicate the data type. See `Attributes.defineAttribute()` for more information on this specification and `DataTypeEnum` for a list of valid Proximity data types.

Children

`ATTR-VALUE`	zero or more	The data file includes one `ATTR-VALUE` element for each instance of this attribute in the database, that is, one for each object, link, or container that has a value for this attribute. Each `ATTR-VALUE` specifies a single, possibly multi-dimensional, value. The file includes multiple `ATTR-VALUE` elements when an object, link, or container has multiple values for a specified attribute.

Content Model

```
(ATTR-VALUE*)
```

Example See COL-VALUE.

<ATTR-VALUE>

Attributes

ITEM-ID	required	The ID of the object, link, or container to which this attribute applies

Children

COL-VALUE	one or more	The value of one dimension of the attribute. For multi-dimensional attributes, include one COL-VALUE element for each dimension’s value.

Content Model

```
(COL-VALUE+)
```

Example See COL-VALUE.

<COL-VALUE>

Attributes None.

Children None.

Content Model

```
(#PCDATA)
```

Example

```
<ATTRIBUTES>
   <ATTRIBUTE NAME="obj-type" ITEM-TYPE="O" DATA-TYPE="STR">
       <ATTR-VALUE ITEM-ID="1">
          <COL-VALUE>person</COL-VALUE>
       </ATTR-VALUE>
       <ATTR-VALUE ITEM-ID="6">
          <COL-VALUE>box</COL-VALUE>
       </ATTR-VALUE>
   </ATTRIBUTE>
   <ATTRIBUTE NAME="pets" ITEM-TYPE="O" DATA-TYPE="STR">
       <ATTR-VALUE ITEM-ID="1">
          <COL-VALUE>Spot</COL-VALUE>
       </ATTR-VALUE>
       <ATTR-VALUE ITEM-ID="1">
          <COL-VALUE>Snowball</COL-VALUE>
       </ATTR-VALUE>
   </ATTRIBUTE>
   <ATTRIBUTE NAME="xy-dimensions" ITEM-TYPE="O" DATA-TYPE="X:DBL, Y:DBL">
       <ATTR-VALUE ITEM-ID="6">
          <COL-VALUE>3.5</COL-VALUE>
          <COL-VALUE>11.25</COL-VALUE>
       </ATTR-VALUE>
   </ATTRIBUTE>
   <ATTRIBUTE NAME="link-type" ITEM-TYPE="L" DATA-TYPE="STR">
       <ATTR-VALUE ITEM-ID="1">
          <COL-VALUE>has-friend</COL-VALUE>
       </ATTR-VALUE>
       <ATTR-VALUE ITEM-ID="2">
          <COL-VALUE>owns</COL-VALUE>
       </ATTR-VALUE>
   </ATTRIBUTE>
   <ATTRIBUTE NAME="uses-directed-links" ITEM-TYPE="C" DATA-TYPE="STR">
       <ATTR-VALUE ITEM-ID="1">
```

```
          <COL-VALUE>true</COL-VALUE>
       </ATTR-VALUE>
       <ATTR-VALUE ITEM-ID="2">
          <COL-VALUE>false</COL-VALUE>
       </ATTR-VALUE>
   </ATTRIBUTE>
</ATTRIBUTES>
```

Containers

Containers are groups of subgraphs sharing some common characteristic. Containers are usually created as a result of running a query on a Proximity database.

Containers in the database are represented by `CONTAINER` elements, all of which must be included in a single `CONTAINERS` element. The `CONTAINER` element includes a list of subgraph items, subgraph attributes, and any nested containers within the parent container.

<CONTAINERS>

Attributes None.

Children

`CONTAINER`	zero or more	The data file includes one `CONTAINER` element for each container in the data. Nested containers are represented by nested `CONTAINER` elements in the data file.

Content Model

```
(CONTAINER*)
```

Example See `CONTAINER`.

<CONTAINER>

Attributes

`NAME`	required	The name of the container
`ID`	required	The ID of the container. ID values must be unique non-negative integers.

Children

`SUBG-ITEMS`	zero or one	A list of database objects and links within this container and the subgraphs to which they belong. Individual database entities can appear multiple times in this list if they appear in multiple subgraphs or if they appear in multiple locations within a single subgraph.
`SUBG-ATTRIBUTES`	zero or one	A list of attributes for subgraphs in this container
`CONTAINER`	zero or more	A nested container within this container

Content Model

```
(SUBG-ITEMS?, SUBG-ATTRIBUTES?, CONTAINER*)
```

Example

```
<CONTAINERS>
   <CONTAINER>
      <SUBG-ITEMS>...</SUBG-ITEMS>
      <SUBG-ATTRIBUTES>...</SUBG-ATTRIBUTES>
      <CONTAINER>...</CONTAINER>
   </CONTAINER>
</CONTAINERS>
```

Subgraphs

Containers are collections of subgraphs. Each subgraph represents a match of the query against the database.

All objects and links included in the container are represented by `ITEM` elements, all of which are included in a single `SUBG-ITEMS` element. Each `ITEM` element identifies the subgraph to which that database entity belongs. If an object appears in multiple subgraphs, or if it appears more than once in a single subgraph, it will also appear in multiple `ITEM` elements.

<SUBG-ITEMS>

Attributes None.

Children

`ITEM`	zero or more	Each `ITEM` element identifies the enclosing subgraph and specifies a member entity (object or link).

Content Model

```
(ITEM*)
```

Example See `ITEM`.

<SUBG-ATTRIBUTES>

Attributes None.

Children

`SUBG-ATTRIBUTE`	zero or more	Each `SUBG-ATTRIBUTE` element describes an attribute on subgraphs in the parent container and lists the values for specific subgraphs.

Content Model

```
(SUBG-ATTRIBUTE*)
```

Example See `SUBG-ATTRIBUTE`.

<SUBG-ATTRIBUTE>

Attributes

`NAME`	required	The name of the subgraph attribute
`DATA-TYPE`	required	The data type of the attribute. See `DataTypeEnum` for a list of valid Proximity data types.

Children

ATTR-VALUE	zero or more	The data file includes one ATTR-VALUE element for each instance of this attribute in the parent container, that is, one for each subgraph that has a value for this attribute. Each ATTR-VALUE specifies a single, possibly multi-dimensional, value. The file includes multiple ATTR-VALUE elements when an subgraph has multiple values for a specified attribute.

Content Model

```
(ATTR-VALUE*)
```

Example

```
<SUBG-ATTRIBUTES>
   <SUBG-ATTRIBUTE NAME="sample-membership" DATA-TYPE="INT">
      <ATTR-VALUE ITEM-ID="1">
         <COL-VALUE>0</COL-VALUE>
      </ATTR-VALUE>
      <ATTR-VALUE ITEM-ID="2">
         <COL-VALUE>1</COL-VALUE>
      </ATTR-VALUE>
   </SUBG-ATTRIBUTE>
   <SUBG-ATTRIBUTE NAME="originating-query" DATA-TYPE="STR">
      <ATTR-VALUE ITEM-ID="1">
         <COL-VALUE>research-clusters1</COL-VALUE>
      </ATTR-VALUE>
      <ATTR-VALUE ITEM-ID="2">
         <COL-VALUE>research-clusters1</COL-VALUE>
      </ATTR-VALUE>
   </SUBG-ATTRIBUTE>
</SUBG-ATTRIBUTES>
```

<ITEM>

Attributes

SUBG-ID	required	The ID of the subgraph to which this object or link belongs. Objects and links can belong to multiple subgraphs and thus appear in multiple ITEM elements.
ITEM-ID	required	The ID of the object or link
ITEM-TYPE	required	Specifies whether the referenced entity is an object (O) or link (L)
NAME	required	The name of this item within the subgraph. Subgraphs are usually created as the result of running queries. Proximity queries assign labels to their vertices and edges (corresponding to the objects and links in the database); these labels are included in the resulting subgraphs.

Children None.

Content Model

```
EMPTY
```

Example

```
<SUBG-ITEMS>
   <ITEM SUBG-ID="1" ITEM-ID="2" ITEM-TYPE="O" NAME="core_page"/>
   <ITEM SUBG-ID="1" ITEM-ID="7" ITEM-TYPE="O" NAME="linked_page"/>
   <ITEM SUBG-ID="1" ITEM-ID="4" ITEM-TYPE="L" NAME="linked_to"/>
   <ITEM SUBG-ID="2" ITEM-ID="9" ITEM-TYPE="O" NAME="core_page"/>
   <ITEM SUBG-ID="2" ITEM-ID="2" ITEM-TYPE="O" NAME="linked_page"/>
   <ITEM SUBG-ID="2" ITEM-ID="6" ITEM-TYPE="L" NAME="linked_to"/>
</SUBG-ITEMS>
```

Appendix D. Proximity Text Data Format

This appendix describes the plain text format for importing data into and exporting data out of Proximity. This format is commonly used to export data from Proximity and then re-import it into a new database requiring a different format, such as when upgrading to a non-compatible version of MonetDB.

The utilities that use the Proximity plain text data format perform no error checking. Although this format requires less disc space than the XML format and its use can improve import and export speed, users are solely responsible for maintaining data consistency when using this format.

For compatibility with MonetDB, we recommend that all single and double quotes and newline characters be converted to underscores. This substitution is performed automatically when importing XML data but must be performed as a pre-processing step when importing text data. Note that values such as "`&quot;`" are treated as strings and not XML entities. A string value of `&quot;` will not be converted to a double quote.

All files referenced in the set of data files must be present and in the same directory. For example, the `container.data` file references data files (e.g., `si_0_attrs.data`) that define the attributes for each container's subgraphs. If the specified container has no subgraph attributes, these files may be empty but they must still be present. The required files are created automatically during export but may need to be constructed by hand for some import applications.

In addition to handling complete databases, Proximity can import and export individual attributes and containers to and from existing databases. Users are solely responsible for ensuring that imported attribute and container data correctly matches the identifiers and data types in the existing database.

See Chapter 3, *Importing and Exporting Proximity Data* for details on using the **import-text.sh** and **export-text.sh** scripts (**import-text.bat** and **export-test.bat** for Windows) to import data using this format.

The examples in this appendix are designed to illustrate the relevant data format and are not intended to represent, in whole or in part, a valid or semantically meaningful database.

Overview

The Proximity text data format stores data in multiple files. One set of files stores structure information:

`objects.data`	stores the IDs for objects in the database
`links.data`	stores the IDs and starting and ending point IDs for links in the database

Another set of files stores attribute data:

`attributes.data`	stores the name, item type (object, link, or container) and data type for attributes in the database and includes pointers to the files containing the corresponding attribute values; subgraph attribute data are stored in a separate set of files
`O_attr_`*`attrname`*`.data`	stores the object IDs and attribute values for the *`attrname`* object attribute
`L_attr_`*`attrname`*`.data`	stores the link IDs and attribute values for the *`attrname`* link attribute

C_attr_*attrname*.data stores the container IDs and attribute values for the *attrname* container attribute

The names for the attribute data files are suggested conventions, which Proximity uses for naming exported data files. Proximity uses whatever filenames you provide in attributes.data when importing data.

Finally, container data (subgraph members and attributes) are stored in a separate series of files:

containers.data stores the ID, name, and location in the container hierarchy of the containers in the database and includes pointers to the files storing subgraph data for these containers

si_*n*.objects.data stores the object ID, subgraph ID, and label assigned by the originating query of the objects in container *n*

si_*n*.links.data stores the link ID, subgraph ID, and label assigned by the originating query of the links in container *n*

si_*n*_attrs.data stores the attribute name and data type of subgraph attributes in container *n* and points to the files containing the corresponding attribute values

si_*n*_*attrname*.data stores the subgraph IDs and values for subgraph attribute *attrname* in container *n*

File Formats

All text data files are tab-delimited, plain text files. Each line in the file corresponds to a single data item.

objects.data

The objects.data file stores the IDs for the objects in the database.

Format

```
id
```

where

- *id* is an integer-valued object ID

Example

This example defines three objects having IDs 1, 2, and 3.

```
1
2
3
```

links.data

The links.data file stores the IDs and starting and ending point information for the links in the database.

Format

```
id      o1_id      o2_id
```

where

- *id* is an integer-valued link ID
- *o1_id* is the object ID for the link's start point
- *o2_id* is the object ID for the link's end point

Example

This example defines three links having IDs 1, 2, and 3. The first line in the example specifies that link 1 starts at object 1 and ends at object 3.

```
1       1       3
2       1       2
3       3       1
```

attributes.data

The attributes.data file defines the object, link, and container attributes in the database. The file specifies the name, item type and data type for each attribute and includes pointers to the files containing the corresponding attribute values. (See "Subgraph attribute specification files" in this appendix for information on defining subgraph attributes.)

Format

```
attr-name     item-type     data-type     filename
```

where

- *attr-name* is the name of the attribute
- *item-type* is one of O (object), L (link), or C (container)
- *data-type* is one of the types listed in the DataTypeEnum class; see below for instructions for specifying attribute type for multi-dimensional attributes
- *filename* is the base name (filename without extension) of the file containing the corresponding attribute values; filenames typically follow the naming conventions described in "Attribute data files", below.

For multi-dimensional attributes (also known as multi-column attributes), the *data-type* specification is a comma-separated list of data types, one for each of the attribute's dimensions. Proximity requires that each dimension in a multi-dimensional attribute be named. For example, a location object attribute might include an x and y dimension, both of type dbl. The specification for this multi-dimensional attribute would be

```
location       O       x:dbl, y: dbl       O_attr_location
```

Example

This example defines three database attributes—an objtype object attribute, a linktype link attribute, and a count container attribute.

```
objtype        O        str        O_attr_objtype
linktype       L        str        L_attr_linktype
count          C        int        C_attr_count
```

Attribute data files

Attribute data files store the values for a single object, link, or container attribute. (See “Subgraph attribute specification files” later in this appendix for information on handling subgraph attributes.)

Attribute data files typically obey the following filename format:

```
type_attr_attrname.data
```

where

- *type* is one of `O` (object), `L` (link), or `C` (container)
- *attrname* is the name of the attribute

For example, a file storing attribute values for the object attribute objtype would be called `O_attr_objtype.data`.

The names for the attribute data files are suggested conventions, which Proximity uses for naming exported data files. Proximity uses whatever filenames you provide in `attributes.data` when importing data.

Format

```
item-id       value       [value]
```

where

- *item-id* is the ID of a database object, link, or container
- *value* is a value of this attribute for the specified database entity; string values must be quoted

Item IDs may be used more than once in the file when the corresponding entity has multiple values for the specified attribute. Additional tab-separated value columns are used when providing data for multi-dimensional attributes.

The following example provides values for the name object attribute. Note that object 2 has two values for this attribute.

Example

```
1       "Peter"
2       "James"
2       "Jimmy"
```

containers.data

The `containers.data` file defines the containers in the database. The file specifies the container ID, name, and location in the container hierarchy and includes pointers to the files storing subgraph data for these containers.

Format

```
id    cont-name    parent-id    subg-attrs-file    subg-data-file
```

where

- *id* is an integer-valued container ID
- *cont-name* is the name of the container
- *parent-id* is the ID of the parent container for the current container; a value of `-1` means that the current container is under the root container (has no parent container)
- *subg-attrs-file* is the base name (filename without extension) of the data file that defines attributes on subgraphs in this container; filenames must follow the naming conventions described in

“Subgraph attribute specification files”. This file is required but may be empty if the container has no subgraph attributes.

- *`subg-data-file`* is the base name (filename without extensions) of the data files containing the object and link data for the subgraphs in the current container; filenames must follow the naming conventions described in “Subgraph data files”.

Example

The following example describes two containers:

- A top-level container named c1
- A container under c1 named samples

The first line defines container c1 and specifies that subgraph attributes used in this container are defined in `si_0_attrs.data`, and the objects and links for these subgraphs are specified in `si_0.objects.data` and `si_0.links.data`, respectively.

```
0       c1          -1       si_0_attrs       si_0
1       samples     0        si_1_attrs       si_1
```

Subgraph data files

Subgraph data files store the IDs of objects and links in a container’s subgraphs. Note that each container has two associated subgraph data files, one for objects and one for links.

Subgraph data files obey the following filename format:

Object data: `si_`*`n`*`_objects.data`
Link data: `si_`*`n`*`_links.data`

where

- *`n`* is the ID of the encompassing container

For example, the subgraph data files for container 0 would be called `si_0_objects.data` and `si_0_links.data`.

Format

```
item-id        subgraph-id        item-label
```

where

- *`item-id`* is the object or link ID
- *`subgraph-id`* is the ID of the subgraph containing the specified object or link
- *`item-label`* is the label assigned to the object or link by the generating query; labels are strings and must be quoted

Example

The following examples define two subgraphs with IDs 0 and 1. Each subgraph contains an actor object linked to a movie in which he or she appeared.

Subgraph object data file:

```
1       0       "actor"
3       0       "movie"
2       1       "actor"
3       1       "movie"
```

Subgraph link data file:

```
1       0       "acted-in"
2       1       "acted-in"
```

Subgraph attribute specification files

A subgraph attribute specification file defines the subgraph attributes used in the corresponding container. The file specifies the name and data type for each attribute and includes pointers to the files containing the corresponding attribute values. (See “attributes.data” in this appendix for information on handling object, link, and container attributes.)

Subgraph attribute specification files obey the following filename format:

`si_`*`n`*`_attrs.data`

where

- *`n`* is the ID of the encompassing container

For example, a file defining subgraph attributes for container 1 would be called `si_1_attrs.data`.

Format

```
attr-name       data-type       subg-attr-data-file
```

where

- *`attr-name`* is the name of the attribute
- *`data-type`* is one of the types listed in the `DataTypeEnum` class
- *`subg-attr-data-file`* is the name of the file that contains the corresponding attribute values; this file may be omitted if the container has no subgraph attributes

Example

The following example defines two subgraph attributes, sample-num of type `int` with values specified in `si_0_attr_sample.data` and avg-age of type `dbl` with values specified in `si_0_attr_avg-age.data`.

```
sample-num     int       si_0_attr_sample.data
avg-age        dbl       si_0_attr_avg-age.data
```

Subgraph attribute data files

A subgraph attribute data file provides the values for a single subgraph attribute.

Subgraph attribute data files typically obey the following filename format:

`si_`*`n`*`_attr_`*`attrname`*`.data`

where

- *`n`* is the ID of the encompassing container
- *`attrname`* is name of the subgraph attribute

For example, a file storing values for subgraph attribute sample in container 2 would be called `si_2_attr_sample.data`.

The names for the subgraph attribute data files are suggested conventions, which Proximity uses for naming exported data files. Proximity uses whatever filenames you provide in the corresponding subgraph attribute specification file when importing data.

Format

```
subg-id       value       [value]
```

where

- `subg-id` is the subgraph ID
- `value` is the value of this attribute for the specified subgraph; string values must be quoted

Subgraph IDs may be used more than once in the file when the corresponding entity has multiple values for the specified attribute. Additional tab-separated value columns are used when providing data for multi-dimensional subgraph attributes.

Example

The following example provides the values for a sample-role subgraph attribute. The first line specifies that subgraph 0 has a value of `"test"` for this attribute.

```
0       "test"
1       "train"
2       "train"
3       "test"
```

Glossary

accuracy

Accuracy measures the percentage of test instances correctly classified by the model. More specifically, it computes TP/(TP+FP) where TP = true positives and FP = false positives.

adjacency

Two query vertices are adjacent if they are connected by a directed or undirected edge; a query vertex is adjacent to the edges that connect it to other vertices.

aggregation

Aggregation is the process of grouping data. For example, in Proximity the models aggregate attribute values to create features. Example aggregation functions include average, count, degree, and proportion.

ambiguous

A query is ambiguous if it admits more than one interpretation. For example, a query with adjacent annotated vertices is ambiguous because neither annotation takes precedence over the other. Ambiguous queries are not permitted in QGraph.

area under the ROC curve (AUC)

Area under the ROC (receiver operating characteristic) curve evaluates the accuracy of a ranking of all test instances according to their estimated probabilities. AUC lets you more easily compare one model to another, independent of error cost and without requiring that the prior class distribution be known. An AUC value of 0.5 indicates chance performance; values approach 1.0 as performance improves.

attribute

An attribute is a name-value pair that represents additional information about the database entity on which it appears. Proximity attribute values are sets, which may contain zero or more values and which may include specific values more than once. Proximity supports attributes for objects, links, subgraphs, and containers. Objects and links can have a variable number of attributes and attributes can have a variable number of values.

attribute constraint

Attribute constraints compare the attribute values of two database entities, such as two objects or two links.

attribute value condition

Attribute value conditions restrict query matches to objects or links having the attribute value specified in the condition.

autocorrelation

For relational data, we define autocorrelation (C') to be the correlation between the same attribute on distinct objects belonging to the same set. For example, we can compute the autocorrelation of gross receipts for movies having the same producer. If all values of the attribute are the same, C'=1. If the attribute values are independent, C'=0.

binary classification problem

A classification task that assigns items to a class or its complement.

boundary edge

A boundary edge is a query edge that crosses a subquery box.

boundary vertex

A boundary vertex is a vertex within a subquery box that is connected to query elements outside the subquery box by a boundary edge.

bounded range

A numeric annotation of the form [*i*..*j*], a bounded range specifies that there must be at least *i* and no more than *j* corresponding elements to match the query.

class label

The class label is an attribute of an item (object or link) that serves to classify the item into one of a number of distinct categories. Classification models, such as those in Proximity, seek to predict the class label.

clustering coefficient

A measure of the interrelatedness a node’s neighbors, the clustering coefficient is the fraction of existing links connecting a node’s neighbors to each other out of the maximum possible number of such links.

collective classification

Collective classification makes simultaneous statistical judgments about the same set of variables for a set of related data instances using an iterative process. Predictions about each variable are used to inform predictions for the remaining variables during each iteration.

comparable types

Conditions and constraints can compare attribute values of the same type (e.g., `STR` with `STR`). In addition, you can compare attributes of type `DBL` with `FLT` and `INT`, and attributes of type `FLT` with `INT`.

complex condition

In principle, a complex condition can be any boolean combination of simple conditions. Proximity’s implementation of QGraph restricts complex conditions to disjunctive normal form.

condition

A condition restricts query matches by requiring that database items match specified attribute values. For example, the condition ObjType = person on a vertex requires that that vertex only match objects with a ObjType attribute having a value of person. Existence conditions work similarly, but only require that the corresponding object or link have *any* value for the specified attribute without caring what that value is.

conditional probability distribution

The conditional probability measures the likelihood that one event X has occurred given the knowledge that another event Y has occurred, usually stated as the probability of X *given* Y. The conditional probability distribution provides the distribution of probabilities for X given Y. That is, it is the collection of probability distributions of X given that Y assumes a value y, for each such value of Y.

connected components

Two vertices in a graph are in the same connected component if and only if there is a path from one vertex to the other. A connected component is thus a subgraph that contains a path between all pairs of vertices in the subgraph. Link direction is ignored in Proximity’s implementation of the connected components algorithm.

constraint

Constraints compare the attribute values or identities of two distinct query elements. Only pairs of objects or links that satisfy the constraint match the corresponding query.

container

A container is a collection of subgraphs usually created as the result of executing a query.

core vertex

The core vertex is the vertex from a QGraph query that corresponds to the object to be classified. Objects and links connected to the core vertex define the *local neighborhood* to be used in classifying the core object.

cross validation

Cross validation is the practice of partitioning the available data into multiple subsamples such that one or more of the subsamples is used to fit (train) a model, with the remaining subsamples being used to test how well the model performs. Because the model is tested on data not used in training the model, cross validation can provide an useful estimate of how the model will perform on new data. In knowledge discovery, the process of cross validation is often repeated many times in order to compare different versions of the model (i.e., different values for a model’s parameters) in order to select the highest-performing version.

degree

The degree of a node in a graph is the count of edges connected to that node. In directed graphs, *in degree* counts the number of edges pointing to the target node and *out degree* counts the number of edges pointing away from the target node.

degree disparity

Degree disparity is difference in the number of items (links, objects, attributes, values) that varies systematically with the class label.

dependency network

A dependency network is a graphical model that represents dependencies among variables. Dependency networks approximate the full joint probability distribution with a set of independently learned conditional probability distributions (one for each variable given its parents). Dependency networks admit simple techniques for parameter estimation and structure learning, even though, as an approximate model, they are not guaranteed to specify a coherent probability distribution for small samples.

directed edge

A directed edge in a query requires matching links in the database follow the same direction as the query edge.

disjunctive normal form

Boolean formulae expressed as a disjunction of conjunctions are said to be in disjunctive normal form. Such formulae consist of a series of disjunctions (expressions ORed together) where each expression is either a terminal expression (a simple proposition in the case of boolean logic or a simple condition in the case of Proximity conditions), the negation of a terminal expression, or the conjunction (expressions ANDed together) of terminal expressions.

document type definition

A document type definition (DTD) is a formal specification of the structure of an XML document. It specifies which XML elements and attributes may occur in which places in the document. An XML document is valid if it obeys the structure specified in the associated DTD.

edge

Proximity uses the terms *vertex* and *edge* to refer to entities in a query and the terms *object* an *link* to refer to entities in the data. An edge in a query matches corresponding links in the data.

error cost

The cost assigned to an error can be broken down into positive error cost (the cost of a false positive) and negative error cost (the cost of a false negative).

exact annotation

An exact annotation requires a specific number of matches for example, [2], rather than a range in the number of matches such as [2-4] or [2..].

existence condition

Existence conditions check to see if the corresponding object or link has a value for the specified attribute. An existence condition is satisfied if the corresponding item has any value for the attribute, regardless of what that value is.

feature

An attribute provides us with information on the characteristics of an entity. A feature tells us which (aggregated) values of an attribute are useful for determining membership in a class. For example, links from actor objects to movie objects might have an attribute that records the salary earned by that actor for that movie. A feature of a model that estimates the likelihood of a particular movie winning a major award might require that the combined salary for its actors be above a specified threshold.

filter

A filter restricts a set of objects or links displayed in the Proximity Database Browser to those that have a specified attribute value. For example, you can filter objects in the ProxWebKB database by pagetype such that only objects having the specified pagetype are shown.

Gibbs sampling

Gibbs sampling is used to generate samples from the joint probability distribution of two or more variables when the conditional probability distribution of those variables is known. The algorithm initializes each variable to a random value and then iterates over all variables, assigning a revised value for that variable conditioned on the current values of the other variables. The process of revising these values is repeated hundreds or thousands of times, using the previous values as the new initial values. An initial “burn-in” period of a pre-determined number of iterations is typically required in order to avoid transient effects due to the initial conditions.

hubs and authorities

Hubs and authorities is the name commonly given to an algorithm developed by Jon Kleinberg to aid in the structural analysis of the World Wide Web. The algorithm uses link structure to identify objects in a graph (e.g., web pages) likely to be trusted sources of information (authorities) and objects that are likely to link to many related authorities (hubs). Although originally designed with web search in mind, the hubs and authorities algorithm has found wide use in other domains requiring the analysis of network structures.

identity constraint

Identity constraints compare the identity (OID) of two database entities.

independent and identically distributed (i.i.d.)

Values are independent and identically distributed if they come from the same probability distribution and they have been independently sampled from that distribution (the values are statistically independent).

infix notation

Prefix notation places an expression’s operator between its operands. For example, the expression $a + b$ uses infix notation.

initialization

New Proximity databases must be initialized before importing data. Initialization creates the MonetDB tables required by Proximity’s data schema.

input source

Input sources define the pieces of information (attribute values and graph structure characteristics) that a model may use to predict class labels.

instantiation

Instantiating a Proximity model creates a new instance of the model using the specified or default values for the model’s parameters.

isomorphic

Isomorphic subgraphs have the same structure in terms of nodes and edges but may have different member objects and links.

joint probability distribution

The joint probability measures the likelihood of two or more events occurring together, usually stated as the probability of *X and Y*.

knowledge discovery

Knowledge discovery seeks to find useful patterns in large and complex databases. More specifically, relational knowledge discovery focuses on constructing useful statistical models from data about complex relationships among people, places, things, and events.

label

See *name*.

lattice graph

A lattice graph arranges vertices and edges in a periodic pattern yielding a regular graph structure.

link

A link is a directed binary relation connecting two objects in a Proximity database. We use *link* to refer to relations in a database and *edge* to refer to relations in a query.

link specification

A link specification describes how to add new links to a database using a query. The specification identifies the query vertices that correspond to the new links’ starting and ending objects and may provide attribute values for the new links.

multi-dimensional attribute

A multi-dimensional (or multi-column) attribute in Proximity contains more than one value. For example, a location attribute might contain two values corresponding to the x and y coordinates of the item’s position. Proximity permits the inclusion of multi-dimensional attributes in data but does not yet support their use in queries or models.

name

Each vertex and edge in a Proximity query is assigned a name (label) that is used to identify the corresponding items in the matching subgraphs.

negated element

A negated query element (vertex or edge) has a numeric annotation of `[0]`; there must be no corresponding element in the data when matching the query.

negative instance

A negative instance is a member of the training set that is not a member of the target class.

nested synchronized table

A nested synchronized table (NST) is an internal Proximity data structure that provides a more user-friendly view of the corresponding MonetDB tables.

numeric annotation

Numeric annotations place limits on the number of isomorphic substructures that can occur in matching portions of the database. Annotations also serve to group isomorphic structures into a single subgraph rather than producing multiple matches.

object

An object is a Proximity database entity that represents things in the world such as people, places, and events.

object identifier

The object identifier (OID) is an internal identifier for an object or link in MonetDB.

optional element

Optional query elements define structures that can be, but are not required to be present in the data in order to match the query. They are annotated with `[0..`*n*`]` or `[0..]`.

positive instance

A positive instance is a member of the training set that is a member of the target class.

precedence

Precedence determines the order in which query elements are considered in matching the database. An annotated vertex has precedence over an annotated edge because the match process first finds objects that match the annotated vertex and then finds links from that object to match the corresponding edges in the query.

prefix notation

Prefix notation places an expression’s operator before its operands. For example, the expression *a* + *b* becomes + *a* *b* when using prefix notation.

prior probability distribution

The prior probability distribution of a variable provides the probability distribution of that variable’s values in the absence of other evidence.

probabilistic model

A probabilistic model assigns a probability to each potential outcome such that the sum of all the probabilities (all the outcomes) equals 1.

probability estimation tree

Probability estimation trees recursively partition the sample space. The results are represented as a graph (tree) where nodes are branch points corresponding to variables and arcs to child nodes correspond to features (specific values for that variable). Leaf nodes estimate the predicted value of an input instance. Advantages of probability estimation trees include ease of understanding and the ability to handle different types of variables.

proxy node

To avoid cluttering the display with too many objects, the graphical database browser expands objects incrementally, using proxy nodes to represents additional objects not yet shown in the graph. Clicking a proxy node adds additional objects to the display.

pseudolikelihood

Pseudolikelihood approximates the distribution of a random variable using an “intuitively plausible” function. It is used when standard calculations of likelihood are intractable.

QGraph

QGraph is a visual query language designed to support knowledge discovery in large graph databases.

relational Bayesian classifier

The relational Bayesian classifier (RBC) is a modification of the simple Bayesian classifier for relational data.

relational data

As used in this document, relational data refers to data that explicitly represent relations among objects as first-class entities. Relational data are represented by a directed graph in which nodes represent objects from the domain of interest and links represent relationships between pairs of objects.

relational dependency network

A relational dependency network (RDN) is a graphical model that extends dependency networks for relational data.

relational probability tree

A relational probability tree (RPT) is an extension of standard probability estimation trees for relational data.

root container

The root container in Proximity serves as a virtual “parent” container for other Proximity containers. It does not explicitly exist in the database and thus cannot be deleted or queried itself.

sampling

Sampling extracts a portion of a data set for use in learning or testing a model. Sampling typically uses random or systematic selection so as to create a set that is representative of the larger population. Alternately, sampling can use natural divisions in the data (e.g., temporal sampling) to create the samples.

schema

A database’s schema determines how the data are represented, i.e., which data entities are mapped to objects, which are mapped to links, and what constitutes attributes of those objects and links. Proximity also uses an internal schema that determines how Proximity database structures map to MonetDB data structures.

schema analysis

Schema analysis discovers how a database’s attributes map to object and link types, and which types of objects are connected by each link type.

social network analysis

A social network models the relationships among members in a social group. The analysis of social networks is concerned with the development and computation of indices that attempt to measure local and global characteristics of the network, such as measures of centrality (the relative structural importance of a node in the graph).

star query

A one-dimensional star query includes a *core vertex* and one or more neighboring vertices, each connected to the core vertex by a single edge. Typically, the neighboring vertices (and therefore the corresponding edges as well) are annotated with an unbounded range, permitting any number of matching neighbor objects and links. Star queries can be extended to additional dimensions through the use of subqueries.

statistical independence

Two events are statistically independent when the occurrence (or lack thereof) of one event does not change the probability of the other event occurring. More particularly for this document, statistical independence means that knowing whether the first event occurs does not affect our ability to predict whether the second event occurs.

subgraph

A subgraph is a connected portion of a graph. QGraph queries return subgraphs as matches to the query.

subquery

A subquery is a connected subgraph of vertices and edges that can be treated as a logical unit. Subqueries allow grouping and limiting of complex query structures rather than just individual query elements.

temporal attribute

Generally speaking, a temporal attribute captures some characteristic of the target entity that includes a temporal component, such as the time an event occurred or a temporal interval during which the characteristic holds. Temporal attributes in Proximity take the form of an attribute that specifies a single time stamp associated with the corresponding object or link, such as a date-of-birth attribute on a person object.

test set

A test set is a labeled data set or sample set aside to test classifier performance after training.

training set

A training set is a set of labeled instances used to learn a model.

type

A type is a label that categorizes instances in a data set, usually represented as an attribute-value pair assigned to an object or link. For example, a data set might contain objects that represent three types of entities: actors, movies, and studios. Proximity does not require a type attribute, but users may specify zero, one, or many attributes that provide type information. These attributes can be practical for the user, but in fact Proximity does not distinguish attributes representing type information from attributes representing other kinds of information.

unbounded range

A numeric annotation of the form [*i*..], an unbounded range specifies that there must be at least *i* corresponding element(s) to match the query.

undirected edge

An undirected edge in a query matches links in the database regardless of the link's direction.

update query

An update query modifies (updates) the database by adding or removing objects, links, or attributes. The QGraph language provides full update functionality; however, only a portion of that functionality has been implemented in Proximity.

validation

Validation is the process of ensuring that an XML document obeys the structure specified in the associated DTD. In Proximity, queries (which are represented internally in XML) must validate against the DTD in `graph-query.dtd`. Because DTDs cannot specify semantic content or enforce all potential syntactic requirements, a syntactically valid query may still be illegal under the rules of QGraph.

vertex

Proximity uses the terms *vertex* and *edge* to refer to entities in a query and the terms *object* an *link* to refer to entities in the data. A vertex in a query matches corresponding objects in the data.

vertical database

A vertical (or vertically fragmented) database, such as MonetDB, employs a storage model based on vertical decomposition, which stores the attribute values for a single object across multiple tables.

well formed

A well-formed query conforms to all rules governing how queries may be legally structured.

References

[Blau, Immerman, and Jensen, 2002] H. Blau, N. Immerman, and D. Jensen. *A Visual Language for Querying and Updating Graphs*. University of Massachusetts Amherst, Computer Science Department Technical Report 2002-037. 2002.

[Boncz and Kersten, 1995] P. A. Boncz and M. L. Kersten. Monet: An Impressionist Sketch of an Advanced Database System. *Proceedings Basque International Workshop on Information Technology*. 1995.

[Boncz, 2002] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.D. Thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands. 2002.

[Copeland and Khoshafian, 1985] G. Copeland and S. Khoshafian. A Decomposition Storage Model. *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*. 1985.

[Craven et al., 1999] M. Craven, D. DiPasquo, D. Freitag, A. McCallum, T. Mitchell, K. Nigam, and S. Slattery. Learning to Construct Knowledge Bases from the World Wide Web. *Artificial Intelligence*. 1999.

[Heckerman, et al., 2000] D. Heckerman, D. Chickering, R. Meek, R. Rounthwaite, and C. Kadie. Dependency Networks for Inference, Collaborative Filtering and Data Visualization. *Journal of Machine Learning Research*. Vol. 1. pp. 49-75. 2000.

[Jensen and Cohen, 1998] D. Jensen and P. Cohen. Multiple Comparisons in Induction Algorithms. *Machine Learning*. Vol. 38. No. 3. pp. 309-338. 1998.

[Jensen and Neville, *ILP*, 2002] D. Jensen and J. Neville. Autocorrelation and Linkage Cause Bias in Evaluation of Relational Learners. *Proceedings of the 12th International Conference on Inductive Logic Programming*. 2002.

[Jensen and Neville, *ICML*, 2002] D. Jensen and J. Neville. Linkage and Autocorrelation Cause Feature Selection Bias in Relational Learning. *Proceedings of the 19th International Conference on Machine Learning*. 2002.

[Jensen and Neville, *KDD*, 2002] D. Jensen and J. Neville. Schemas and Models. *Proceedings of the Multi-Relational Data Mining Workshop, 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2002.

[Jensen, Neville and Hay, 2003] D. Jensen, J. Neville, and M. Hay. Avoiding Bias When Aggregating Relational Data with Degree Disparity. *Proceedings of the 20th International Conference on Machine Learning*. 2003.

[Kleinberg, 1998] J. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. 1998.

[Neville et al., 2003] J. Neville, D. Jensen, L. Friedland, and M. Hay. Learning Relational Probability Trees. *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2003.

[Neville, Jensen and Gallagher, 2003] J. Neville, D. Jensen, and B. Gallagher. Simple Estimators for Relational Bayesian Classifiers. *Proceedings of The 3rd IEEE International Conference on Data Mining*. Available as University of Massachusetts Amherst, Computer Science Technical Report 2003-004. 2003.

[Neville and Jensen, 2003] J. Neville and D. Jensen. Collective Classification with Relational Dependency Networks. *Proceedings of the 2nd Multi-Relational Data Mining Workshop, 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2003.

[Neville and Jensen, 2004] J. Neville and D. Jensen. Dependency Networks for Relational Data.

Proceedings of The Fourth IEEE International Conference on Data Mining. 2004.

[Perlich and Provost, 2003] C. Perlich and F. Provost. Aggregation-Based Feature Invention for Relational Learning. *Proceedings of the 9th SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2003.

[Perlich and Provost, 2004] C. Perlich and F. Provost. *ACORA: Distribution-based Aggregation for Relational Learning from Identifier Attributes*. CeDER Working Paper #CeDER-04-04. Stern School of Business. New York University, New York, NY. 2004.

[Provost and Fawcett, 1997] F. Provost and T. Fawcett. Analysis and Visualization of Classifier Performance: Comparison under Imprecise Class and Cost Distributions. *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*. AAAI Press. pp. 43-48. 1997.

[Provost, Fawcett and Kohavi, 1998] F. Provost, T. Fawcett, and R. Kohavi. The Case Against Accuracy Estimation for Comparing Induction Algorithms. *Proceedings of the 15th International Conference on Machine Learning*. Morgan Kaufmann. pp. 445-553. 1998.

Index