MineSet™ Enterprise Edition Interface Guide

Document Number 007-3993-002

CONTRIBUTORS

Written by Helen Vanderberg and Sandra Motroni
Edited by Connie Boltz
Production by Diane Ciardelli
Engineering contributions by Barry Becker, Amit Bleiweiss, Jeff Brainerd, Cliff Brunk, Eben Haber, Ara Jerahian, Andy Kar, Ed Karrels, Eser Kandogan, Alex Kozlov, Alan Norton, Peter Rathmann, Mario Schkolnick, Dan Sommerfield, Peter Welch, and Brett Zane-Ulman.
Cover design by Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

COPYRIGHT

LIMITED RIGHTS LEGEND

TRADEMARKS AND ATTRIBUTIONS

MineSet™ Enterprise Edition Interface Guide
Document Number 007-3993-002

MineSet™ Enterprise Edition Interface Guide
Document Number 007-3993-002

Contents

List of Tables

About This Guide

The *MineSet Enterprise Edition Interface Guide* explains the advanced technical features of the MineSet suite of data mining and visualization tools. You can also find current information about MineSet online at http://www.sgi.com/software/mineset/.

This guide is written for people already familiar with the operation of the MineSet Tool Manager, and it explains how to perform the following tasks:

- Make interconnections with MineSet and the outside world.
- Export MineSet files to other applications.
- Build custom applications using the ActiveX API to the 3D visualizers.
- Perform basic system administration tasks.

Windows users will find familiar methods and pathnames given. IRIX users should be familiar with UNIX commands, and Linux users should be familiar with Linux commands.

This guide explains the tasks involved with installing and running MineSet from the command-line or configuration files. In some cases programmatic interfaces are described. A chapter-by-chapter summary can be found in "Structure of This Document" on page xvi.

For background information, see the *MineSet Enterprise Edition Reference Guide.* For information on how to use the MineSet tools, see the *MineSet Enterprise Edition User's Guide for the Windows Client*.

MineSet allows third party software such as AcPro to plug in to the application. If AcPro is installed, documentation can be found in */usr/acpro/doc*. Late-breaking information can be found on the MineSet Web page.

MineSet also allows third parties to integrate the 3D visualization tools into their own Windows applications using an ActiveX API (see Chapter 14.)

Structure of This Document

The guide begins with installing MineSet. Subsequent chapters concentrate on specific tools and processes as shown:

Chapter 1, “MineSet Overview”
Answers installation questions and provides an overview of MineSet operations.

Chapter 2, “Configuring and Setting Up MineSet”
Provides directions on installing and setting up configuration files and connecting to the database server. It also describes how to run MineSet in batch mode for complex or time-consuming calculations.

Chapter 3, “File Exchange between MineSet and SAS (IRIX only)”
Describes how to convert MineSet data files into SAS data sets, and vice versa.

Chapter 4, “MineSet Web Extensions”
Describes how to interact with MineSet over the Web, and how to set up the necessary files to achieve secure remote viewing.

Chapter 5, “Data and Configuration File Basics”
Describes basic requirements for MineSet data and configuration files, including data types, configuration file structure, and the MineSet expression language.

Chapter 6, “Flat File Support for MineSet”
Describes the files that are necessary to run MineSet, and gives examples of the *.data* and *.schema* files.

Chapter 7, “Creating Data and Configuration Files for the Tree Visualizer”
Describes the necessary files for the Tree Visualizer and gives appropriate examples.

Chapter 8, “Creating Data, Configuration, Hierarchy, and .gfx Files for the Map Visualizer”
Describes the files needed to enable the Map Visualizer to run, and includes examples.

Chapter 9, “Creating Data and Configuration Files for the Scatter Visualizer”
Describes the Scatter Visualizer configuration and data files and includes examples.

Chapter 10, “Creating Data and Configuration Files for the Splat Visualizer”
Describes the Splat Visualizer configuration and data files and includes examples.

Chapter 11, “Creating Data and Configuration Files for the Decision Table Visualizer”
Describes the Decision Table Visualizer configuration and data files and includes examples.

Chapter 12, “Format of the Evidence Visualizer’s Data File”
Describes the Evidence Visualizer and its data and configuration files.

Chapter 13, “Nulls in MineSet”
Describes how nulls are handled in MineSet.

Chapter 14, “ActiveX Visualization Control API for MineSet Visualizers”
Describes how application developers can use ActiveX controls to create custom data mining solutions for Windows platforms with MineSet’s visualizers.

Appendix A, “Further Reading and Acknowledgments”
Provides a resource for further reading, and acknowledges the resources used to establish MineSet.

Typographical Conventions

The following type conventions and symbols are used in this guide:

Italics — Executable names, filenames, program variables, tools, utilities, variable command-line arguments, and variables to be supplied by the user in examples, code, and syntax statements.

Bold — Keywords.

`Fixed-width type` — On-screen command-line text and prompts.

`Bold fixed-width type` — User input, including keyboard keys (printing and non-printing); literals supplied by the user in examples, code, and syntax statements.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please send your comments to SGI. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, you can find the document number on the back cover.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  `techpubs.sgi.com`

- Use the Feedback option on the Technical Publications Library World Wide Web page:

  `http://techpubs.sgi.com/`

- Contact your customer service representative and ask that the incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Pkwy., M/S 535
  Mountain View, California 94043-1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801

SGI values your comments and will respond to the promptly.

Chapter 1

MineSet Overview

This chapter provides a summary of MineSet tools as well as a tool execution scenario that helps you better understand how MineSet tools can be used in combination. This chapter contains the following sections:

MineSet Tools Suite Overview

The MineSet suite of tools lets you mine and graphically display quantitative information so you can better visualize, explore, and understand your data. These tools provide a highly interactive, three-dimensional (3D) visual interface that lets you manipulate visual objects on the screen, as well as search, filter and perform animations.

The MineSet suite consists of three basic components:

- A centralized control module, consisting of a graphical user interface tool called the Tool Manager, and a process called the DataMover, which runs on the server part of MineSet's client/server architecture.
- Analytical data mining, with nine data mining tools:
  - Association Rules Generator
  - Automatic Binning
  - Cluster Generator
  - Column Importance
  - Decision Table Inducer and Classifier

  - Decision Tree Inducer and Classifier
  - Evidence Inducer and Classifier
  - Option Tree Inducer and Classifier
  - Regression Tree Inducer and Regressor
- Visualization tools, which let you view your data using ten different visual metaphors:
  - Cluster Visualizer
  - Decision Table Visualizer
  - Evidence Visualizer
  - Map Visualizer
  - Record Viewer
  - Scatter Visualizer
  - Splat Visualizer
  - Statistics Visualizer
  - Tree Visualizer

For complete descriptions and directions for using the tools see the *MineSet Enterprise Edition User's Guide* for Windows or for IRIX, the *MineSet Enterprise Edition Reference Guide* or the *MineSet Enterprise Edition Tutorial* for Windows or for IRIX.

About the Tool Manager

Each of the mining and visualization tools can be configured and started using the Tool Manager, which does the following:

- Connects you to the server on which the analytical mining and transformations are performed.
- Lets you access, query and transform data.
- Creates configuration files for each tool.

The Tool Manager is installed on the MineSet client.

Understanding DataMover

DataMover is a process that runs on the server. The server performs the following actions:

- Connects to databases, or flat files (ASCII or binary), and retrieves the data.
- Invokes the mining tools.
- Performs additional data manipulation such as binning and aggregation.
- Returns the data to the Tool Manager for distribution to the visualization tools.
- Stores the data in files on the server or client for future operations.

You can run MineSet in client-only mode with limited functionality, provided you have access to the server.

MineSet Plug-in Capability

MineSet supports a plug-in architecture that allows you to add new mining tools, functions, and data transformations to MineSet. These plug-ins are installed and executed on the MineSet server, and any user interface for the plug-ins is downloaded by the Mineset client. The MineSet plug-in API is identical on IRIX, Linux, and Windows, so plug-ins can be installed on any MineSet server.

Plug-in Functions

Plug-in functions are added to the list of available functions when forming Add Column or Filter expressions. You can implement them by writing and implementing a C function with a particular prototype, compiling that into a shared library, and putting the shared library on the MineSet server computer.

Plug-In Transformations

You can access plug-in transformations are from the “Plug-in Ops” button in the Data Transformations pane of the Tool Manager. They are also visible in the View History panel. A plug-in transformation is implemented with a standalone executable that is given a data set as input, and produces a data set as output. The MineSet plug-in SDK ships with a library of functions that will assist you in reading and writing MineSet data sets. Plug-in transformations also require a Java GUI that will be sent to the MineSet client, and incorporated into the Tool Manager.

Plug-in Mining Tools

Plug-in mining tools are added as new tabs in the “Mining Tools” tabbed deck in the Tool Manager. They are implemented as standalone executables on the MineSet server that are given data sets or model files as input, and produce data sets or model files as output. Like plug-in transformations, plug-in mining tools also include a Java GUI that will be added to the Tool Manager.

Basic Tool Execution Scenario

Each of the MineSet tools is started, configured, and run in a consistent manner. The sequence of actions followed by the MineSet client and the MineSet server is shown in Figure 1-1.

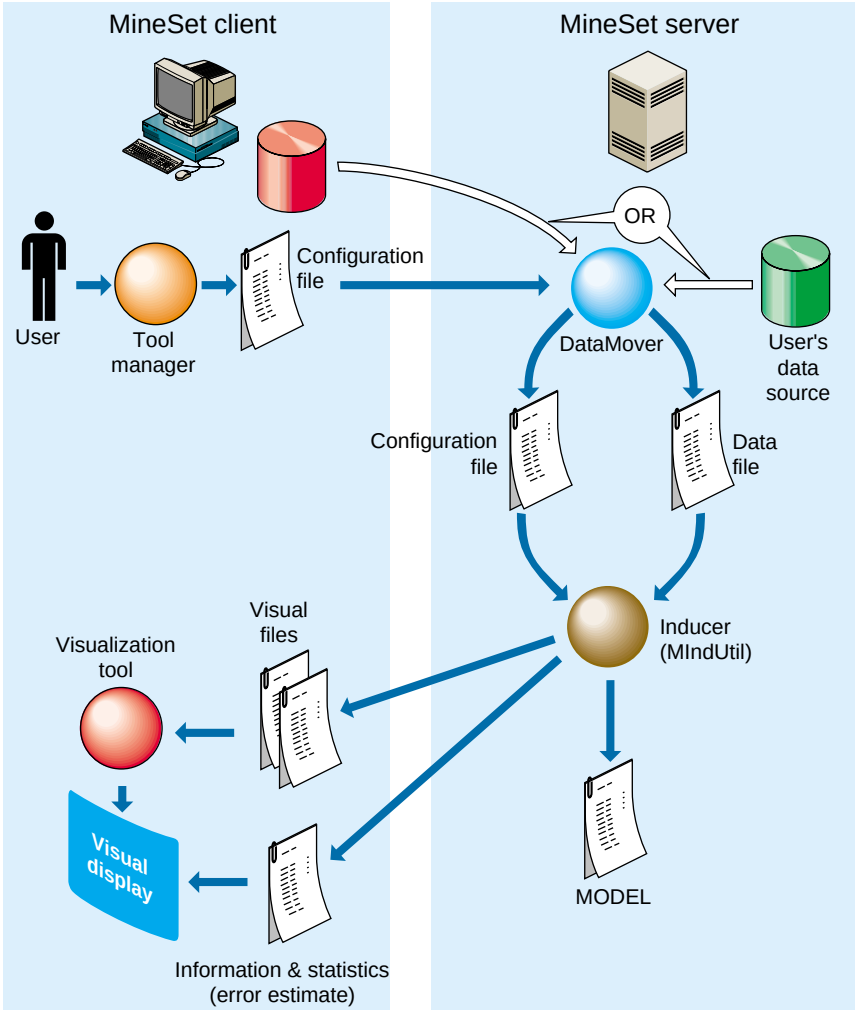

Figure 1-1 Tool Execution Sequence

The following steps describe a typical interaction with a MineSet tool, and the sequence of the tool's actions. Depending on your requirements, you might skip some steps (for example, if the data and configuration files were already generated in a previous work session).

1. Start the Tool Manager, which is the graphical interface for generating and specifying the configuration file, data file, and tools to be used. The Tool Manager runs on your MineSet client.
2. The Tool Manager opens a network connection to the DataMover, which runs on the MineSet server, which in some cases may be the same as your client workstation, and in others is a separate machine.
3. Use the Tool Manager to specify the following:
   - The database and table, or a binary or ASCII flat file containing the data on either the client or the server.
   - Which mining or visualization tools are to be applied.
   - How that data is to be displayed, through tool options.
   - A session file in which to save the history of your work.

   Information retrieved using the DataMover guides this interaction. As a result, the Tool Manager generates a configuration file. This file contains the user-defined parameters that determine the execution of the following steps.
4. The Tool Manager transmits a copy of the configuration file from step 3 to the DataMover. The DataMover processes the file by:
   - Accessing the database or flat file.
   - Performing the specified data transformations.
   - Running the mining tools when requested.
   - Generating the visualization files when requested.

   These visualization files consist of your data in a specific format readable by the MineSet tool. Then a copy of these visualization files is transferred to the MineSet client.
5. The Tool Manager invokes the appropriate MineSet visualization tool.
6. The tool accesses the visualization files and displays the data.
7. If you generated a model, that model can be applied to additional data.

Note: The MineSet client and server can run on different machines, using a network to communicate. Because network bandwidth is often scarce, you should be cautious about regularly transferring large files between client and server. If you are mining a large database or file, you can achieve greater efficiency by storing that file on the server where the DataMover runs, rather than on the client.

Configuring and Setting Up MineSet

This chapter describes how to set up MineSet on Linux, Windows, and IRIX systems. The subjects discussed are:

This chapter also explains how to load the sample files into Oracle, Sybase, and INFORMIX.

Configuring MineSet

There are two major pieces of configuration information in MineSet:

- The directory used to store the user's temporary and persistent files on the server.
- The databases to which the server can connect.

On IRIX and Linux systems, this configuration is defined by global and per-user configuration files. On Windows, these configuration files are still available, but most information is stored in the registry. Windows users therefore have less need to edit configuration files.

IRIX Systems

Setting up MineSet on IRIX systems requires modifying *dm_config* and the DataMover configuration file */usr/lib/MineSet/.datamove*. The DataMover process runs on the server, and is not directly accessible to users. The DataMover provides access to databases and data stored in flat files, and transforms data for the mining and visualization tools. The configuration has three parts:

- The configuration of kernel parameters.
- A global configuration by the system administrator.
- The configuration of an account on the server.

"Configuring the DataMover Server" on page 10 describes these files and how they may be modified.

Configuring Systune Parameters

The systune parameters on IRIX machines determine the default limits on the available system resources. Table 2-1 lists the systune parameter values that SGI recommends.

Table 2-1 systune Parameters

Parameter	Definition	Recommended Value
rlimit_pthread_cur	Current limit on the number of threads	1024
rlimit_rss_cur	Current limit on memory usage	Amount of physical memory on your machine
rlimit_vmem_cur	Current limit on virtual memory usage	Size of logical swap space on machine, or about twice the physical memory
rlimit_nofile_cur	Current limit on number of open file	1024 or the limit on the number of threads

Note: You must reboot your machine after installing the new parameters.

For more information about these parameters, see the systune(1M) reference page.

Linux Systems

To configure MineSet on a Linux server follow these steps:

1. Move to the */etc* directory and use your favorite text editor to edit the file *services*.

   ```
   # cd /etc
   # vi services
   ```

2. Remove the pound (#) sign at the beginning of the following line in *services*:

   ```
   exec 512/tcp
   ```

3. In the same directory, edit the file *inetd.conf*. Remove the pound (#) sign at the beginning of the line that contains:

   ```
   exec... in.rexecd
   ```

4. Start the service as follows:

   ```
   # killall -HUP inetd
   ```

Configuring MineSet on Windows Systems

The MineSet server uses a hybrid strategy to determine where on the server to store model, schema, and data files. At installation time, the Installer program prompts for a base directory for all users' server files. Each user is then assigned a subdirectory of this base directory, according to username. For example, if ServerFilesBase is left at its installation default of *C:\MineSet Files*, user `guest` will have MineSet server files located in *C:\MineSet File\guest*.

An individual may override this machine-level default by setting the registry entry *HKEY_CURRENT_USER\SOFTWARE\SGI\MineSet\3.0\ServerFilesDirectory*. (To set a registry entry, right-click the My Computer icon on your desktop, and choose Preferences.) If this entry exists, its value will set the server files directory for that user only.

The different versions of Windows support various filesystems, the most common of which are NTFS and FAT. In general, it is better to locate the server files directory on an NTFS filesystem, if possible. The MineSet server allows multiple instances of the MineSet client to connect to the same server files directory. This can be convenient, if one user wants to try several experiments in parallel, or if two or more individuals want to share data and model files. On FAT filesystems, the DataMover uses a coarse-grained locking mechanism to maintain consistency, which means that only one client can run a history

at one time from within a given server files directory. Other users attempting to run histories will receive a `Failed to get a lock` error message. On NTFS, the DataMover's locking implementation is more finely grained, so this problem will seldom occur.

Finally, whichever way the server files directory is determined, the MineSet server will then look for a file *.datamove* in that directory, which can contain additional configuration information, as described in the next section. The format and entries for the */usr/lib/MineSet/.datamove* file are the same on Windows and UNIX. However, the most common use of the *.datamove* file is for setting the location of server files, and this is already handled by the registry on Windows. Therefore, most users of the Windows MineSet server will not need to create a *.datamove file*.

Configuring the DataMover Server

To use the MineSet tools, two configuration files must be created on the server:

- The user must create the *.datamove* file.
- The system administrator must create the *dm_config file.*

Note: Each user must have an account on every server that can be accessed.

User Configuration File

The DataMover configuration file, *.datamove*, creates files on the server system for each user. This *.datamove* file lets you control where these user files are created and whether different classes of files are saved or discarded. On UNIX systems, the *.datamove* file is located on the server, in the user's home directory. A sample *.datamove* file called *datamove.sample* is located on the server in the following directories:

- Windows users: *MINESET_HOME\config\datamove*
- IRIX users: */usr/lib/MineSet/datamove*
- Linux users: */usr/lib/MineSet/datamove*

If the *.datamove* file is absent, or if a particular entry is not present in the *.datamove* file, the DataMover uses a default value for that entry.

Each entry in the DataMover's configuration file must be on a separate line. For example:

```
file_cache = directory_name
```

file_cache specifies the location in which the DataMover stores its output data files and models resulting from mining algorithms. If the *file_cache* directory does not exist, the DataMover attempts to create it, when first invoked. On Windows the default *file_cache* directory is determined by the registry (see "Configuring MineSet on Windows Systems" on page 9); on UNIX, the default *file_cache* directory is *./mineset_files/%U*. The *%U* is a wildcard that is filled in with the user's login name on the client machine.

The *file_cache* should be a directory in a filesystem with sufficient space to hold all of a user's output and temporary files. DataMover creates this directory if it does not already exist. Intermediate files are deleted when the DataMover no longer needs them, unless you set one of the following **keep** options:

```
keep_client_upload
keep_client_download
keep_classifier_files
keep_mlc_input
use_ascii_mlc_input
```

Table 2-2 describes these and other options.

Table 2-2 Configuration Options

Option	Description	Default
keep_client_upload	Keep files uploaded from the client for processing. If kept, the files will be in the *client_upload* subdirectory.	no
keep_client_download	Retain a copy of data files and visualizations on the server after they are downloaded to the client. If kept, the files will be in the *client_download* subdirectory.	no
keep_classifier_files	Keep the persistent classifiers (decision trees and so forth) generated by mining operations. The method is generally useful.	yes

Table 2-2 (continued) Configuration Options

Option	Description	Default
keep_classifier_options_files	Keep the options file that used when generating (inducing) the classifier. This tactic is not useful. If kept, the files will be in the *mlc_work* subdirectory.	no
keep_mlc_input	Keep input files used for mining (MIndUtil or associations) operations. If kept, the files will be in the *mlc_work* subdirectory.	no
use_ascii_mlc_input	Normally the DataMover creates MineSet binary files for MIndUtil input. If this option is set, create ASCII files instead.	no
aggregation_memory_limit	Memory limit (in bytes) for aggregation operations. This can be no larger than the system-wide limit set in the *dm_config* file.	2147483647
optimize_history=yes	The DataMover can rewrite histories to remove redundant computations. The *optimize_history* parameter controls whether or not to do this. Because this rewriting can speed up processing considerably, it is normally turned on.	N/A

A file in the *file_cache* directory is the result of a successful operation. If an operation returns an error (that is, the Tool Manager reports a message that begins with `fatal error on server`) MineSet does not change anything in the *file_cache* directory. Three examples help illustrate the point:

- A user's *file_cache* directory contains the files *cars.data* and *cars.schema*, both the result of a previous database query. The user then selects the same table, and sets the output to *server_file*, filtering for examples with mpg>55. Because no records in the dataset have mpg values this high, when the history executes, it returns no rows, which is flagged as a fatal error. After this happens, the user's *file_cache* directory will still contain the old *cars.schema* and *cars.data* files.
- A user's *file_cache* directory contains the files *cars.data* and *cars.schema*, both the result of a previous database query. The user then selects the same table, and sets the output to a visualization. The operation completes and the visualization launches successfully. Once again, the user's *file_cache* directory still contains the old *cars.schema* and *cars.data* files. The *file_cache* directory is not updated unless the user specifically chooses *server_file* as the output.

- Two client users are connected to the same server account, and the same server files directory. The first user starts a long-running mining operation on the *adult94.schema* and *adult94.data* server files. Before that mining operation completes, another user executes a database query and replaces the *adult94.schema* and *adult94.data* files with new versions. Neither user will see any errors; the first user's model will finish based on the earlier version of the data file. If the users are connected to a Windows system with the server files directory on a FAT filesystem, the second user's operation would exit with a `Failed to get a lock` error.

Global Configuration File

On IRIX, the DataMover works with Oracle versions 7.2 or later, INFORMIX, and Sybase. On Windows, the DataMover supports ODBC and OLE-DB data sources and native connections to Oracle. On a Linux server, the DataMover works with Oracle 8i.

If you are using relational databases, you may need to configure the server to find these databases. ODBC and OLE-DB data sources are listed in the Windows system registry and not in the *dm_config* file. Therefore, a Windows server using only ODBC and OLE-DB data sources does not need a *dm_config* file. Windows users need to create a *dm_config* file only if they want to use a native connection to a local or remote Oracle database. Oracle may also be accessed with ODBC or OLE-DB, but the native connection generally gives better performance. On IRIX, the *dm_config* file may be omitted if the server will not access any database.

This section details the DataMover configuration file, *dm_config*, which is read by the DataMover server during startup. Windows users access the file from the directory in which MineSet is installed, under *config\datamove\dm_config*. UNIX users access the file from the directory in which mineset is installed, under *datamove\dm_config*.

This file is not created automatically during installation. The system administrator must create it and log in as root to edit it. It can be created using any simple text editor such as Notepad or Emacs. You can find an example file, *dm_config.sample*, in the *datamove* directory. The format of this file is as follows:

```
Oracle {
"ORACLE_SID", "ORACLE_HOME";
}

Oracle_Remote {
“SERVICE_NAME”, “ORACLE_HOME”;
}
```

```
Informix {
"INFORMIXSERVER", "INFORMIXDIR";
}
Sybase {
"DSQUERY", "SYBASE";
}
```

Replace “ORACLE_SID” and “ORACLE_HOME” with the specific information and repeat it once for each Oracle database to be accessed via the DataMover. ORACLE_SID and ORACLE_HOME are Oracle-specific parameters defining an Oracle instance.

The Oracle_Remote section is for accessing remote Oracle databases via Oracle Networking. The SERVICE_NAME entry is a logical name for the remote database. Such logical names are defined by editing an Oracle-defined configuration file (for example, *tnsnames.ora*), or, depending on the Oracle version and operating system, with an Oracle-supplied applet (for example, Net8 Easy Config). The second ORACLE_HOME parameter is the directory of your client installation. On IRIX systems only, you can often do without an Oracle client installation. In this case the second parameter is interpreted as the directory where DataMover searches for the *tnsnames.ora* file. This file is described in Oracle’s networking documentation.

Each line in the Informix section defines a database server that, in turn, can contain several databases. The server is checked at runtime to determine which databases it contains, so there is no need to record the individual databases in the *dm_config* file. The first entry is the INFORMIX server (corresponding to the *INFORMIXSERVER* environment variable), and the second is the INFORMIX directory (corresponding to the *INFORMIXDIR* environment variable).

Each entry in the Sybase section defines a database server (or, in Sybase terminology, an SQL Server). The first entry is the Sybase SQL Server name (corresponding to the *DSQUERY* environment variable); the second is the Sybase home directory (corresponding to the *SYBASE* environment variable). This may be a remote database.

Note: A Windows configuration file defines different pathnames, but the structure follows conventions similar to the following example.

An IRIX example configuration file shows:

```
Oracle {
"v81", "/ora/app/oracle/product/8.1.5";
"wrhse", "/opt/oracle";
}

Oracle_Remote {
   “lifeseq”, “/usr/lib/MineSet2/datamove/”;
}

#native connections to Informix and Sybase supported on IRIX only
Informix {
"learn_online", "/u5/informix";
}
Sybase {
"MINESET", "/usr/sybase/11.5.1";
}

#  Following parameter is relevant only if your database is in a
#  non-English locale.  Default of TRUE allows database to perform
#  automatic translation of character encoding (if necessary) to
#  compensate for any mismatch in locale between the database and the
#  MineSet client.  Setting parameter to FALSE disables this
#  adjustment, which is sometimes necessary with very old or
#  misconfigured databases.

enable_database_locale_adjustments = TRUE;
```

The `enable_database_locale_adjustments` is common to both platforms.

This configuration file lets the DataMover access the following:

- Three Oracle databases, one named *v73* (installed in */usr/people/oracle/v73*), another named *wrhse* (installed in */opt/oracle*), and a remote database named *lifeseq*.
- An INFORMIX Server.
- A Sybase SQL Server.

Each of the INFORMIX and Sybase servers can, in turn, contain multiple databases.

For Sybase, DataMover uses vendor-supplied shared libraries as its connection to the databases. One of the purposes of the *dm_config* file is to specify where DataMover must look for its shared libraries.

DataMover looks for the shared libraries *libct.so*, *libcs.so*, *ibcomn.so*, *libintl.so*, *libtcl.so*, and *libinsck.so* on IRIX systems in the *$SYBASE/lib/* directory.

The IRIX DataMover is compiled as an *n32* program and therefore needs to load the *n32* versions of the Sybase client libraries. *n32* client libraries are included with Sybase versions 11.5 or later. If MineSet will be used with an earlier version of Sybase, you will need to install a recent version of the Sybase client.

A Windows example of the *dm_config* file shows:

```
# Connection to a local Oracle database on Windows NT.  First entry is
# the Oracle sid, and second entry is the ORACLE_HOME.  Note that if
# '\' is used as a directory separator, it must be doubled.

Oracle{
        "mse", "F:\\Oracle\\Ora81";
}

#  3-tier access to remote Oracle databases via Oracle networking.
#  First argument is the name of the data source, as created by the
#  Net8 configuration assistant or listed in the tnsnames.ora file.
#  Second argument is the local ORACLE_HOME directory.  At least a
#  client install of Oracle is required to connect to remote Oracle
#  databases.

Oracle_Remote {

    "uci_megamine", "F:/Oracle/Ora81";
    "MSE_NET8.ENGR.SGI.COM",  "F:/Oracle/Ora81";
    "DEMO_MEGA", "F:/Oracle/Ora81";
    "mega_v72", "F:/Oracle/Ora81";
}
```

Using MineSet with Existing Data Files

Sometimes it is convenient to use MineSet with data that is already stored as a file, but requires further processing before it can be mined or visualized. In this case, the data file can be made available (with a modest effort) to the Tool Manager/DataMover.

First, the data file must be in a tab-delimited format, with the same number of fields in each line. A numeric or string field with a single "?" character appearing between delimiters is loaded as a Null value. You can import data files automatically, using the MineSet Tool Manager as detailed in "Importing Data Files" on page 18. You can find a detailed example of the MineSet *.schema* file format in Chapter 6, "Flat File Support for MineSet."

You must describe the contents of the data file to the Tool Manager/DataMover using a file with the *.schema* extension. The format of the *.schema* file is as follows:

```
#
#  A line beginning with a "#" is a comment
#
input {

#  The first line lists the data file which is described.  It
#  must be a simple filename, not a path.

     file "carmodels.data";

#  Fields are listed left to right in the line, legal
#  types are float, double, int, string, date, fixedString and
#  dataString
#  Be sure to end every line with a semicolon ";"

     float mpg;
     int cylinders;
     float cubicinches;
     int horsepower;
     int weightlbs;
     double timeaccelerate;
     date when_introduced;
     string origin;
     fixedString(3) manufacturer_code;
     dataString model;
}
```

The schema and data files must be located in the same directory. If you prepare a dataset this way on the client machine, you can open it with the Tool Manager's Find File dialog. If the file requires any additional processing, it is copied to the server. Sometimes this is not convenient, especially if the file already exists on the server, or is large. In this case, the *.schema* and *.data* files must be copied (or symbolically linked) into your *file_cache* directory on the server.

Importing Data Files

This section shows you how to import files from other formats using the Tool Manager.

To import a file, choose the Import option from the File menu. A file dialog appears, allowing you to choose the file you wish to import, along with a number of import options.

The format of the imported file may be selected from a box at the top of the options portion of the file dialog. Table 2-3 lists the supported file formats.

Table 2-3 Importable Data File Types

File Type	Description of Source
ASCII File - Delimited	A text file with fields separated by a delimiter character.
ASCII File - Fixed Format	A text file wherein each value for a particular column has the same width.
1-2-3	Lotus 1-2-3.
dBase III	dBase II, III+, IV, and compatible systems such as Clipper or Alpha Four.
Epi Info	Epi Info through version 6.
Excel 97	Microsoft Excel 97.
Microsoft Excel	Microsoft Excel.
FoxPro	Microsoft Visual FoxPro.
Gauss	Either Gauss 89 or Gauss 96.
JMP	SAS JMP statistics files.
Limdep LIMDEP version 7 for Windows	
MATLAB	MATLAB matrices.
MINITAB	MINITAB versions 8 through 12.
Osiris	Osiris type 1 data sets.

Table 2-3 (continued) Importable Data File Types

File Type	Description of Source
Paradox	
Quattro Pro	
SAS data file	SAS version 6.08 and above.
SAS transport file	
SigmaPlot	
S-PLUS	32-bit S-PLUS data sets.
SPSS data file	
SPSS portable file	
Stata	Any version of Stata.
Statistica	Statistica version 5.
SYSTAT	Either double or single precision SYSTAT files.

When importing ASCII files (both delimited and fixed format), you may specify the following options:

- Maximum line width (default 4096): This is the maximum width for lines in the file to be imported. If the file is wider than 4096 characters, you must specify the width here.
- Maximum number of columns (default 255): This is the maximum number of columns in the file. If the file has more than 255 columns, specify how many.
- Two-digit year cutoff (default 69): When processing date values from an ASCII file, if a year is specified with only two digits, years greater than or equal to this number are considered to be in the 20th century. For example, given the default value of 69, the date 2/3/69 becomes 2/3/1969, but 2/3/68 becomes 2/3/2068.

- Decimal separator: The character used as a decimal point when processing numbers. The default value is taken from the current locale, for instance a period in the USA and a comma in most European countries.
- Separator every three digits: Sometimes numbers have punctuation every three digits, for instance, 1,000,000,345. This field allows you to specify that character. The default value is taken from the current locale, for instance, a comma in the USA and a period in most European countries.

When importing delimited ASCII files, you can also specify the delimiter between columns. If you select the "Outshines delimiter" radio button, the delimiter will be automatically detected from either comma, tab, semicolon, or space. If you select the "Specify delimiter" radio button, you can enter a different delimiter character.

When importing fixed-width ASCII files, you must provide a *.SCH* file to indicate where the fields begin and end. An example of the format follows:

```
[alcoholSample]
Filetype=Fixed
Date-Order=MDY
Date-Punctuation=/
Dec-Point=.
CharSet-asci
Field1=household_id,Float,16,0,0
Field2=pdate, Date,10,0,16
Field3=name,Char,255,0,26
Field4=age,Float,16,0,281
```

It starts with the name of the table, followed by the file type, date order, date punctuation, decimal point, character set, and a list of field names, types, length, zero, and the starting location for the field.

Exporting Files

Exporting files is possible from the Tool Manager Data File Destination panel, which has three radio buttons:

- MineSet ASCII file
- MineSet binary file
- Other file types

The first two buttons allow the user to specify whether the output file (on the client or server) should use the ASCII or binary formats.

The third button, available only when creating client files, also converts the output client file to a different data format specified in the combo box next to the "Other file types" button. Available file types are the same as those in the Import dialog, with the exception of Osiris.

Loading Sample Datasets

This section describes how to load the sample datasets included with the MineSet distribution into one of the supported relational databases.

The following are installed on the server:

- All the sample data, along with a brief description of what it contains.
- Directions on how to load the data using the provided scripts.

Load the sample datasets into a database that has been set up on your server. Windows users will find the data and directions (*README.server*) a in the directory in which MineSet is installed, under *\Examples\DBexamples*. UNIX users find them in */usr/lib/MineSet/DBexamples* on the server.

This directory contains scripts for loading the complete set of data files into one of the supported databases. To load the complete set of data, run one of the following loader scripts, depending on which database you have. (This assumes your database and environment are already set up.)

`sh load_all_Oracle.sh` <*userid*> <*passwd*>

`sh load_all_Sybase.sh` <*userid*> <*passwd*>

If you are going to work with an INFORMIX database, use the `dbaccess` interface to select:

```
create_all_Informix.sql
```

Then select:

```
load_all_Informix.sql
```

Loading Individual Datasets

Alternatively, you can load, or reload, the sample data separately. Each data directory in *DBexamples* on the server contains files necessary to load the data into any of the supported databases. Windows users can find these files in the directory in which MineSet is installed, under *\Examples\DBexamples*. UNIX users can find them in */usr/lib/MineSet/DBexamples*. These files are:

README (explains the data)

**.sql* (sets up an Oracle table)
**.ctl* (control file for loading into Oracle)

**_syb.sql* (sets up a Sybase table)
**.bcf.fmt* (Sybase format file)

**_inf.sql* (sets up an INFORMIX table)
**_load.sql* (loads the data into the INFORMIX table)

In the **.ctl* file, the separator is declared in the following line:

```
" fields terminated by X'20'  "
```

The separator is specified in ASCII hexadecimal; therefore:

`X'20'` is used for ‘ ’
`X'2c'` is used for ‘,’
`X'09'` is used for ‘\t’

Loading into Oracle

Perform the following steps on the server with an Oracle database:

1. Ensure the following environment variables are set correctly:

   `ORACLE_HOME`

   `ORACLE_SID`

2. Type:

   ```
   sqlplus <userid>/<passwd>
   SQL> @<dataset>.sql
   ```

   dataset is the name of the dataset being loaded. *userid/passwd* are your assigned username and password for the Oracle database.

   To delete an already existing table, type:

   ```
   SQL>  drop table <dataset>;
   ```

3. Type:

   ```
   sqlload control = <dataset>.ctl userid = <userid>/<passwd>
   log = /tmp/<dataset>.log  direct = true
   ```

4. Check the resulting *dataset.log* to ensure the data was loaded correctly.

Loading Into Sybase

Perform the following steps on the server with a Sybase database:

1. Ensure that the following environment variables are set:

   ```
   SYBASE
   DSQUERY
   ```

2. To create the table, type:

   ```
   isql -U<userid> -P<passwd> -i <dataset>_syb.sql
   ```

   dataset is the name of the dataset being loaded. *userid/passwd* are your assigned username and password for the Sybase database.

   To delete an already existing table, type:

   ```
   isql -U<userid> -P<passwd>
   drop table <dataset>
   go
   ```

3. To load the data, type:

   ```
   bcp <dataset> in <dataset>.data -U<userid> -P<passwd> -f
   <dataset>.bcp.fmt
   ```

   dataset is the table name (created using *<dataset>_syb.sql)*, `in` means "load into the dbms," *<dataset>*`.data` refers to the name of the ASCII data file, and **-f** points to the format file that was already created. (When reading in from a file, the data types are character.)

Loading Into INFORMIX

Perform the following steps on the server with an INFORMIX database:

1. Ensure the following environment variables are set:

   ```
   ONCONFIG
   INFORMIXSERVER
   INFORMIXTERM
   ```

2. To create the table, type:

   ```
   dbaccess
   ```

3. If necessary, log in to the appropriate database.
4. Choose *Query-language*, and then choose the appropriate database from those listed.
5. Choose <*dataset*>*_inf.sql*, and run it.
6. Choose <*dataset*>*_load.sql*, and run it (where <*dataset*> is the name of the dataset being loaded).

Logging in to a DBMS

To verify that you have the proper configuration, start MineSet. From the Tool Manager File menu, choose Open New DBMS Table.

1. Verify you have the correct server or change servers if necessary.
2. Click Change DBMS.
3. Select demo-Oracle from the DBMS menu.

   ```
   login: demo
   Password: demo
   ```

4. Click OK.

This loads the table from your selected list. You can then manipulate the table to develop a classifier or visualization as detailed in the *MineSet Enterprise Edition User's Guide*.

Running MineSet in Batch Mode

When running MineSet in batch mode, all actions from a MineSet.session file are performed without user intervention. You can use the MineSet client Tool Manager to prepare the MineSet.session file(s). Batch mode can be particularly useful in projects requiring lengthy computations that need to be done frequently. For instance, multiple or long running MineSet data mining algorithms can be run overnight so the data will be ready the next morning.

Session Files

MineSet batch mode requires a session file, which consists of the following information:

- A single data source (a file, RDBMS table or query).
- One or more MineSet transformations.
- A single data destination (a mining tool, viz tool or file).

The following is an annotated example of a session file designed to run a Decision Tree classifier:

```
MineSet 3.1
# Starts with the Preferences options
preferences {use_binary_files, parallelize, num_threads -1, max_attr_vals 1000}

# Next the single data source file, RDBMS table or query
datafile {localserv
     "E:\\mineset_demos\\marketing\\mineset_mellon\\respond2.schema"}

# Next one or more transformations
transformations {
remove_col { "owcustid"};
remove_col { "Cluster"};
}

# Next single data destination (mining tool, viz tool or file)
classify classify_and_error decision_tree "foo" "foo"

# Finally, all the OPTION settings
options {options_list "test_model_in:", "test_show_viz:0", "operation:predict",
     "predictor:classifier", "name:respond2", "model_version:", .... }
```

Creating the saved_session.mineset File On NT

To create the saved_session.mineset file:

1. Use the MineSet Tool Manager to prepare the session file.
2. Do not click the Invoke Tool or Create File button.
3. Save the file with a descriptive name (for instance, *datafile-name_classifier-name.mineset*).

Creating the saved_session.mineset File On IRIX or Linux

1. Use the MineSet Tool Manager to prepare the session file.
2. Do not click the Invoke Tool or Create File button.
3. Save the file with a descriptive name (for instance, *datafile-name-classifier-name.mineset*).
4. Transfer the file to your UNIX *$HOME/mineset_files* directory.

Running MineSet in Batch Mode on NT

Running MineSet in batch mode on NT is similar to running it on IRIX ("Step-by-step Example for Running MineSet Batch on IRIX" on page 27). If you want to run the batch directly on the NT MineSet server, follow the steps in that example, except instead of creating a shell script, create a MS-DOS .bat script:

```
mineset_batch -s USER_NAME -d RDBMS_LOGIN_NAME saved_session.mineset
```

For example:

```
mineset_batch -s NT_passwd -d Oracle_passwd saved_session.mineset
```

Running MineSet in Batch Mode on UNIX

To run MineSet in batch mode on IRIX or Linux, enter the following command at the command line:

```
MIndUtil_p30 saved_session.mineset session.schema
```

Step-by-step Example for Running MineSet Batch on IRIX

To run MineSet in batch mode on IRIX:

1. Either save your dataset on the IRIX MineSet server or verify that your RDBMS query will create the dataset you need for the visualization or data mining activity. Use the naming conventions, DATAFILE.schema and DATAFILE.data.

   For example, if *contract* is an Oracle table, then save the data file as *contract.data* on the IRIX MineSet server.

2. On the MineSet Client, create all the MineSet session files for each type of analysis or visualization you want to perform in batch. Use the naming convention, *datafilename-toolname.mineset*.

   For example:

   - If you want to run Column Importance on the *contract* database in batch mode, then save the MineSet session file as:

     `contract-import.mineset`

   - If you want to run Evidence on the *contract* database in batch mode, then save the MineSet session file as c*ontract-evi.mineset*.

3. Transfer the **.mineset* saved sessions to your *$HOME/mineset_files* directory on the MineSet IRIX server:

   - Verify that the *contract.schema* and *contract.data* files exist in your *$HOME/mineset_files* directory.

   - Transfer the *datafilename-toolname.mineset* file you created on the MineSet client to this directory on the MineSet server.

4. Modify the *batch.sh* sample script below to match your file and MineSet tool names.

   The batch.sh script performs Column Importance, Evidence, Decision Tree, and Cluster analyses on the *contract* data file. See the instructions in the batch script for more details.

5. Run the shell script.

   ```
   $ nohup sh batch.sh &
   ```

   This will run the shell script batch.sh in the background so you can log off the IRIX MineSet server without killing the batch job.

6. Monitor the progress of the batch (optional).

   ```
   $ tail -f results.out <--- this command will display the output as
                              it is generated
   $ top <----- this command displays the top UNIX processes
   ```

Sample UNIX Shell Script

The following shell script, *batch.sh*, runs the Column Importance, Evidence, Decision Tree, and Cluster analyses on the *contract* database:

```
#!/bin/sh
# This is a UNIX shell script that runs and times MineSet batch.
# This sample runs Column Importance, Evidence, Decision Tree, and
# Cluster on the "contract" dataset.

# Save the timings and any errors in results.out
RESULT="results.out"
echo > $RESULT

# Make sure you have named the Mineset session filess contract-import.mineset,
# contract-evi.mineset contract-dt.mineset, and contract-cluster.mineset and transfered them
# to the IRIX MineSet server in the same directory as the datafiles.

# Run the calculations using the MineSet analytics, /usr/sbin/MIndUtil_s (32-bit single
# threaded) or # /usr/sbin/MIndUtil_p30 (64-bit parallel), and print the run times to the
# results.out file. ("timex" is a UNIX utility to time how long the analysis took.)
timex /usr/sbin/MIndUtil_p30 contract-import.mineset contract.schema >>$RESULT 2>>$RESULT
timex /usr/sbin/MIndUtil_p30 contract-evi.mineset contract.schema >>$RESULT 2>>$RESULT
timex /usr/sbin/MIndUtil_p30 contract-dt.mineset contract.schema >>$RESULT 2>>$RESULT
timex /usr/sbin/MIndUtil_p30 contract-cluster.mineset contract.schema >>$RESULT 2>>$RESULT
```

File Exchange between MineSet and SAS (IRIX only)

This chapter describes the support for file exchanges between MineSet and SAS on IRIX. The subjects discussed are:

Systems other than IRIX and Windows must use third-party solutions for file exchange.

You can exchange data sets between MineSet and SAS with two utilities: *mineset2sas* and *sas2mineset*. To convert a MineSet *.schema* and *.data* file pair to an SAS data set, use *mineset2sas*. To convert an SAS data set to MineSet *.schema* and *.data* files, use *sas2mineset*. Both *mineset2sas* and *sas2mineset* invoke the SAS executable. SAS must be installed on the machine on which these conversion utilities are used.

Converting MineSet Data Files to SAS Data Sets

Use *mineset2sas* to convert MineSet data files to SAS data sets. The syntax for this is:

`mineset2sas` *MineSet file SAS libref.datafile* [*options*]

The two options are as follows:

- **-svsc** to save the script sent to SAS. The script is normally deleted after use.
- **-names** *namefile* to save trimmed column names in *namefile*. The script is normally deleted after use.

For example:

```
mineset2sas cars sasuser.cars -svsc -names cars.names
```

mineset2sas converts the MineSet *.schema* and *.data* files (in this case, *cars.schema* and *cars.data*) to an SAS data file. Currently, only string and numeric data types are supported. The MineSet *.data* file must be in ASCII format; binary format is not supported. To save MineSet data files in ASCII, deselect Use binary data files in Tool Manager Preferences.

SAS column names (or, in SAS terminology, variable names) can consist of only letters and underscore characters. The first character in a column name cannot be a number. Furthermore, SAS column names can be up to eight characters long. Because any character string can be a legal MineSet column name, *mineset2sas* maps MineSet column names to legal SAS column names. The rules for this mapping are:

- Any invalid character is replaced with an underscore.
- If the first character is a digit, an underscore is prepended to the column name.
- Column names are truncated to eight characters. If this truncation results in non-unique column names, the ends of the conflicting column names are replaced with sequential numbers, thus creating unique column names.

To preserve as many of the full column names as possible, *mineset2sas* also saves the first 40 characters of each column name as the column label.

-names *namefile* Command Line Option

To see a listing of the column names before and after conversion to SAS format, specify the **-names** *namefile* command line option. When *mineset2sas* executes, it writes out a mapping of the column name changes to the specified file. For example:

```
`date of birth` -> `date_of_`
`92census` -> `_92censu`
`# of days to end of quarter` -> `__of_da0`
`# of days to end of year` -> `__of_da1`
`# of davenports` -> `__of_da2`
```

-svsc Option

The *mineset2sas* utility reads the schema for the specified data file, and writes a customized SAS script. SAS is invoked with this script to read and convert the data. The script sent to SAS is normally deleted after use. With the **-svsc** option, the script is saved as the file *mineset2sas.sas*. If there is an error in the script processing, the SAS error log is saved as *mineset2sas.log*. The SAS script must be installed in */usr/sbin/sas*.

Converting SAS Data Sets to MineSet Data Files

Use *sas2mineset* to convert SAS data sets to MineSet data files. The syntax for this is:

```
sas2mineset <SAS libref.datafile> <MineSet file> [options]
```

The options are as follows:

- `-nodata` creates only a *.schema* file, no *.data* file.
- `-svsc` saves the scripts sent to SAS.
- `-nolabel` indicates that you do not want labels used for column names.
- `-names` <*namefile*> restores long column names from <*namefile*>, created by *mineset2sas*.

For example:

```
sas2mineset sasuser.houses -svsc -names houses.names
```

The *sas2mineset* utility converts a SAS data file into MineSet *.schema* and *.data* files. Currently, this utility supports only string, numeric, and date data types.

-nolabel Option

SAS only supports eight-character column names, but allows optional 40 character labels for each column. MineSet sets no limit on the column name length, so, by default, *sas2mineset* uses the column labels to name the columns in the output file, if labels have been defined. To force *sas2mineset* to use the SAS column name for each column, even if a label is specified, add the `-nolabel` option to the command line.

-names *namefile* Option

If you convert a MineSet data file to SAS with *mineset2sas* and then back to MineSet format with *sas2mineset*, you can create a column name map file to keep track of the original column names. To have *sas2mineset* use a name map file created by *mineset2sas*, add the `-names <namefile>` option to the command line, and specify the same map file as specified when the file was converted to SAS format with *mineset2sas*. This option is useful only for data files with column names longer than 40 characters, since *mineset2sas* saves up to 40 characters in the column label.

Note: The `-name <namefile>` option overrides the `-nolabel` option.

-nodata Option

To create only a MineSet schema file without downloading the data from an SAS data file, add the `-nodata` option to the command line.

-svsc Option

The *sas2mineset* utility writes two customized SAS scripts to retrieve the specified data file. The first script extracts column descriptions; the second extracts the data. The scripts are normally deleted after use. With the `-svsc` option, the scripts are saved as *getschema.sas* and *getdata.sas*, respectively. If there is an error in the script processing, the SAS error logs are saved as *getschema.log* and *getdata.log*, respectively.

MineSet Web Extensions

This chapter describes the MineSet extensions that are provided to let you create or view visualizations or interact with MineSet over the Web. The following subjects are discussed:

Overview

The MineSet Web extension allows the visualizing of files and data generated by MineSet software over the Web. You can do this by using the MineSet *.mtr* extension. The MineSet *.mtr* extension lets you place MineSet visualization, schema and data files into an archive file, which you can embed in a Web page as an HTML tag.

Once the user clicks the hyperlink in Internet Explorer, the browser automatically invokes the MineSet ActiveX control which brings up the visual tool within the browser's window. Any *.viz* or *.mtr* file embedded within an HTML document seamlessly launches the ActiveX control within the rest of the web page. The user can then click the hyperlink in Internet Explorer, or type the filename in the URL box, and the ActiveX control will be launched as well.

The machine that the browser is running on must have the MineSet client software installed.

MineSet Web Extension Files

Table 4-1 lists the MineSet Web Extension files, which are located in the */MineSet/examples* subdirectory.

Table 4-1 MineSet Web Extension Files

File	Purpose
adult-salary.eviviz.mtr	*.mtr* file of *adult-salary.eviviz*.Windows users find the file in the directory in which MineSet is installed, under *examples*. UNIX users find the file in */usr/lib/MineSet/eviviz/examples.*
nl.births.mapviz.mtr	*.mtr* file of *nl.births.mapviz.* Windows users find the file in the directory in which MineSet is installed, under *examples*. UNIX users find the file in */usr/lib/MineSet/mapviz/examples.*
company.scatterviz.mtr	*.mtr* file of *company.scatterviz.*Windows users find the file in the directory in which MineSet is installed, under *examples*. UNIX users find the file in */usr/lib/MineSet/scatterviz/examples.*
cars-dt.treeviz.mtr	*.mtr* file of *cars-dt.treeviz.*Windows users find the file in the directory in which MineSet is installed, under *examples*. UNIX users find the file in */usr/lib/MineSet/treeviz/examples.*
cars-odt.treeviz.mtr	*.mtr* file of *cars-odt.treeviz.*Windows users find the file in the directory in which MineSet is installed, under *examples*. UNIX users find the file in */usr/lib/MineSet/treeviz/examples.*
churn-dt.treeviz.mtr	*.mtr* file of *churn-dt.treeviz.*Windows users can find the file in the directory in which MineSet is installed, under *examples*. UNIX users find the file in */usr/lib/MineSet/treeviz/examples.*

MineSet Web Installation (Client)

During the MineSet client installation the client part of the Web extension is automatically installed. The system must have Internet Explorer 4.0 or greater installed for full Web support. Netscape will launch the visualizer tool externally, not within its browser window.

MineSet .mtr Files

MineSet *.mtr* files are archives of MineSet files generated by the viz tool. Once created, the files can be used as a hyperlink in an HTML page. The *.mtr* files are very effective in sharing multiple visualizations over the Web, eliminating the need for attaching huge files in mails, remote copies, or file transfers (ftp). The *.mtr* files can also be launched directly by dragging them into a browser, or by setting the File property in the HTML PARAM tag.

Since *.mtr* files are in a compressed format and use the underlying HTTP protocol, the transfer of an *.mtr* file is very fast and does not require an administrator to perform a cumbersome setup.

You can create an *.mtr* file from any MineSet visualizer by selecting File > Publish on the Web. This creates an *.mtr* file for the current visualization.

Publishing on the Web

Having created an *.mtr* file, you then can make it available on the Web in the following ways:

1. Create a hyperlink to the *.mtr* file.

   After you create the *.mtr* file, move it to the directory that contains all your *.html* files. For Internet Explorer or Netscape Navigator to launch an *.mtr* file, you can invoke it directly by entering the following in the Internet Explorer or Netscape Location window:

   ```
   http://yourserver/directory/foo.treeviz.mtr
   ```

   Or you can create a link to it from a page by adding the following line in the *.html* file for that page:

   ```
   <A HREF="foo.treeviz.mtr">foo.treeviz.mtr </A>
   ```

2. In Internet Explorer, you can set the OBJECT tag as follows:
   - PARAM tag with CLASSID

     You can set the File property of the ActiveX control directly:

     ```
     <OBJECT CLASSID=clsid:911D5D2F-906D-11D2-9A61-00104BD33DDB
     WIDTH=804 HEIGHT=627 ID=vizCompositeCtrl>
     <PARAM NAME="File" VALUE="foo.scatterviz.mtr">
     </OBJECT>
     ```

   - Alternatively, you can use MIME type instead of CLASSID:

     ```
     <OBJECT TYPE = "application/x-mineset-tar"
     WIDTH = 804 HEIGHT = 627 ID=vizCompositeCtrl>
     <PARAM NAME = "File" VALUE = "foo.scatterviz.mtr">
     </OBJECT>
     </P>
     ```

3. Embed the ActiveX control using an EMBED tag.

   ```
   <EMBED SRC="cars.scatterviz">
   ```

   You can also use these techniques c for embedding *.viz* and *.mtr* files.

Data and Configuration File Basics

This chapter covers data and configuration file basics that are the same for all MineSet tools. The following topics are discussed:

Data Types

MineSet supports integer, floating-point number, string, and date data types, as well as arrays of these types. The following data types are supported:

- **int** represents a 32-bit signed integer.
- **float** represents a single-precision floating point number. The decimal point is optional. Numbers in exponential "e" notation are also accepted.
- **double** represents a double-precision floating point number. The decimal point is optional when representing a floating point number. Numbers in exponential "e" notation are also accepted. The superior precision of **double** can be useful for accurately representing large numbers, because **float** can represent only seven or eight significant digits accurately. This superior accuracy, however, consumes twice the memory space of **float**.

- **dataString** represents a string that is unlikely to appear multiple times. If it appears multiple times, several copies are made. A **dataString** can be used to store a memory address. Addresses are unlikely to be compared, and each record can have a different address.
- **string** represents a string of characters that can appear multiple times in the data file. Unlike a **dataString**, only a single copy of a given string is stored in memory, no matter how many times it appears in the data. This saves memory for strings appearing many times.

  Comparing **strings** is also much quicker than comparing **dataStrings**. Reading in **strings** can be slower than reading in **dataStrings** because it is necessary to look for duplications. An example of **string** use is a division name that appears once for each department in the division. If you are unsure whether to use a **string** or a **dataString**, use a **string**.
- **fixed string** represents a string of fixed length. Like a **dataString**, if a **fixed string** appears multiple times, multiple copies are made. In general, **fixed strings** are used internally for representations of data from data bases, and are generally better to use than **strings** or **dataStrings**.
- **date** represents a date and time. In the data file, a **date** appears in the format MM/DD/YYYY HH:MM:SS. Output from MineSet always represents dates with four-digit years, although two-digit years are acceptable for input. MineSet follows the X-OPEN standard for interpreting two-digit years. Fields with values 69 or greater are considered to be from the 20th century (1969-1999), and values from 0 to 68 are considered to be from the 21st century (2000-2068).

Enumerations

The syntax for declaring an enumeration is:

```
enum type name { value, value...};
```

For example:

```
enum string state {
   "Alabama",
   "Alaska",
   ...
   “Wyoming"
};
```

The word “string” indicates that the enumeration maps integers to strings; they can also be mapped to other types.

Once you declare an enumeration, you can declare a column to be of that enumeration using the following syntax:

```
enum enumname columnname;
```

For example, the following declares `st` to be a variable of state enumeration:

```
enum state st;
```

The input file corresponding to this column must contain values from 0-49 (or “?” representing null); however, the output shows the state name.

You can also use Enumerations to declare enumerated arrays (see “Enumerated Arrays” on page 49).

Arrays

In MineSet, you can use one-dimensional or two-dimensional arrays of fixed or variable size.

In a fixed-sized array, all entries of the given type have the same number of values. For example, the budgets of the 50 states can be represented by a separate **float** column for each state, or by a single array with 50 floats.

A special form of a fixed array is an “enumerated array.” Like the normal fixed array, there are a fixed number of values in the array; however, the values are associated with an enumeration. For each value in the enumeration, there is a single entry in the array. For example, if an enumeration represents the 50 states, an enumerated array based on this enumeration has 50 values.

A variant of the enumerated array is the “null enumerated array,” which has an additional entry at the beginning for null (represented as a “?”). For example, with the enumeration of the 50 states, the null enumerated array has 51 values, one for NULL, and the remaining 50 for the 50 states. The null array element could be used for entries where the state is unknown.

The tree visualizer also supports variable length arrays (see Chapter 6, “Creating Data and Configuration Files for the Tree Visualizer,” for details).

As with the columns, arrays are represented as value separated by tabs or other separators. For a fixed-sized array, you can use the same separator for columns and for individual array elements (in which case, array elements are not visually distinguished from separate columns). You can also define a different separator. In the sales example, for instance, you can treat the location as a four-element array, rather than as four columns. It then could be represented like this:

Eastern:Maryland:Baltimore:1816 appliances 72 115 138

In this example, the array is separated by colons, and the columns are separated by tabs. (For clarity, the rest of this document uses this format.)

For a variable-length array, you must use different separators for the array and for the columns; otherwise, it is impossible to determine where the variable-length array ends and the other columns begin.

Any field or array element in the data file can also have the value "?" (question mark), indicating an unknown or null value (see the discussion of nulls in Chapter 12, "Nulls in MineSet.")

Fixed Arrays

You can also declare arrays using data declarations. The simplest form is the fixed array. The declaration syntax is:

```
type name [ number ] ;
```

For example:

```
float revenue [50];
```

You can also override the separator by declaring it as follows:

```
type name [ number ] separator 'char';
```

For example:

```
float revenue [50] separator ':';
```

If no separator is specified, the default column separator (usually a tab) is used.

Enumerated Arrays

To declare an enumerated array, first declare the enumeration (see “Enumerations” on page 47). Then declare the array using one of the following syntaxes:

```
type name [ enum keyname ];
type name [enum keyname ] separator ‘char’;
```

For example:

```
float revenue [enum state];
```

As with the normal fixed array, you can also specify a separator. To declare a null enumerated array, use one of the following syntaxes:

```
type name [ null enum keyname ];
type name [ null enum keyname ] separator `char';
```

For example:

```
float revenue [null enum state];
```

This indicates that the array contains one additional value at the beginning, corresponding to null.

Variable Names

A variable name can appear in two formats:

- In the first format, it is a letter followed by a number of letters, digits, or underscores. It cannot be a keyword, and should not be quoted.
- In the alternate format, the variable name should be surrounded by back quotes (‘). The variable name can match a keyword and can contain non-alphanumeric characters. This second format is primarily for *.schema* files generated automatically by the Tool Manager.

There is no scoping of variable names; a given variable name can only be declared once in the *.schema* file.

Strings and Characters

Strings and characters in the *.schema* file follow C-language conventions. Strings are in double quotation marks ("), and characters are in single quotation marks (’). All standard backslash conventions are followed (for example, \n represents a new line).

Comments

Comments begin with a pound (#) symbol at the beginning of a line; anything after this symbol is ignored, up to the end of the line.

MineSet Expression Language

The expression language used in the Filter and Add Column panels is similar to expressions in C, C++, and Java. The basic operators are the same, as listed in Table 5-2:

Table 5-1 Operators Used With Expressions

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
==	Equals
!=	Not equals
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
&&	AND

Table 5-1 Operators Used With Expressions

Operator	Description
`||`	OR
!	NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
`A?B:C`	If (A), then B else C

The expression language also provides the functions listed in Chapter 5, " Expression Language Functions":

Table 5-2 Expression Language Functions

Function	Description
isNull( )	Determines if the value in parentheses is null
if ( ) then ( ) else ( )	Standard if/then/else
( ) ? ( ) : ( )	C syntax if/then/else
divide(x, y, z)	Divide x by y, and give value z if y is 0

Also, the following functions are available:

- **modulus**(x, y, z) is similar to **divide**, but for modulus.
- **hierarchy**(*string*) is valid only within a hierarchy. It produces a string describing the components of the hierarchy, separated by *string*. For example:

  ```
  hierarchy(":")
  ```

  This might produce:

  ```
  Western:California:Mountain View
  ```

  The hierarchy function is most useful in the execute statement, passing the hierarchy information to the command being executed.
- **isSummary**() returns 1 if the expression is being applied to base information; otherwise, it returns zero. Often, this is useful with the **?:** operator, particularly in message and execute statements.

Type handling is similar to that in C. Expressions using **int** and **float** promote both sides to **float**. Expressions using **int** and **double**, or **float** and **double**, promote both sides to **double**. The result of a relational expression (for example, ==, <) is always an **int**. Type casting is also supported.

Unlike in C, strings can be compared using relational expressions; the strings are compared lexicographically.

Keywords

The currently recognized keywords are listed in Table 5-1. Variables cannot have these names unless they are surrounded by back quotes (‘). Tokens appearing only in option statements are not keywords, and can be used for variable names.

Table 5-3 Keywords for the Tree Visualizer

aggregate	disk	int	normalize
any	divide	isSummary	off
ascending	double	key	on
average	enum	label	options
back	execute	landscape	scale
base	expressions	legend	separator
buckets	file	levels	sort
color	filter	max	string
colors	float	message	sum
count	height	min	view
dataString	hierarchy	modulus	
descending	input	none	

Configuration File Basics

This section discusses the various parts that make up a MineSet configuration file.

Sections

The configuration file consists of a series of sections, each of which has this syntax:

```
sectionKeyword
{
    statements...
}
```

sectionKeyword names the section. A semicolon (;) can follow the closing brace (}) but is not required. The order of the sections is significant, since sections can refer to variables defined in previous sections. The sections found in a treeviz configuration file are “Input Section,” “Expressions Section,” “Hierarchy Section,” and “View Section.”

Options Files

As each section is encountered, a special configuration file (referred to as an “options file”) is also read in. Options files have names in the following form:

```
sectionName.treeviz.options
```

Options files normally contain options statements. These files are read in the following order:

1. The directory containing system defaults.
   - For Windows: from the directory in which MineSet is installed, under *\config\treeviz*.
   - For UNIX: */usr/lib/MineSet/treeviz*.
2. Your home directory, where you set up personal defaults.
3. The current directory, which lets you set up defaults for each directory.

Files with the same name can appear in more than one of the above-named directories; in this case, the order given is the one in which the directories are read. If the same option is found in multiple files, the last option read is used. Note that the appropriate section

in the configuration file is read after all the options files have been read in; thus, options in the configuration file override those in the options files.

Statements

A statement has the following syntax:

```
statementKeyword info;
```

statementKeyword defines the statement, and *info* varies according to the keyword. A statement can be another section (using the brace format defined under "Sections").

Option Statements

Many sections have **options** statements, which have this syntax:

```
options key info, key info...;
```

key defines the specific option, and *info* depends on the key. In some cases, the *key* can be more than one word. To maximize the number of allowable variable names, most option keys are meaningful only within the appropriate option statement; keys do not conflict with variable names. You can declare several options on the same line, separating them by commas or placing them in several options statements. For example, the following two examples are equivalent:

```
options home angle 30, shrinkage 10.0;
```

```
options home angle 30;
options shrinkage 10.0;
```

If two conflicting values for the same option appear, the last value is taken.

Include Statements

The configuration file can contain lines of the following form:

```
include "filename"
```

These lines can appear anywhere in the configuration file, but each must be on its own line. The filename must be in quotation marks; anything after the closing quotation mark is ignored. Include statements can be nested. If a relative pathname (one not beginning with a slash) is specified, the file is first sought relative to the directory containing the current configuration file. (If **include** statements are present, this might not be the same

as the initially loaded configuration file.) If it is not found in the directory containing the current configuration file, the **include** file is sought in the current directory. If the file is not found, an error message appears.

Sinclude Statements

A statement similar to an include is **sinclude**, which has the syntax:

```
sinclude "filename"
```

This is identical to the **include** statement, except that no error appears if the file does not exist; instead, the **sinclude** statement is ignored.

Input Options

The input section of a data file has several options. All options statements begin with the word *options* and have one or more comma-separated options.

- The separator option defines the separator between columns in the data file. The default separator is a tab. The syntax is:

  ```
  options separator 'char';
  ```

  For example:

  ```
  options separator ':';
  ```

  Note: Arrays can override the separator.

- The backslash option controls whether backslashes in the input data are treated specially or like other characters. The syntax is:

  ```
  options backslash off;
  options backslash on;
  ```

  The default is off. If backslash processing is on, separators in the input data preceded by backslashes are treated as regular characters rather than separators. Also, within strings, standard C-style backslash processing is done.

Flat File Support for MineSet

This chapter describes the *.schema* and the *.data* files that are required to define the MineSet flat files. The following subjects are discussed:

The Tool Manager also generates *.schema* files to include the *.schema* files for Tree Visualizer, Map Visualizer, Scatter Visualizer, and Splat Visualizer.

Data File

In its simplest form, the data file consists of a list of lines, each containing a set of fields, each separated by one tab. (Other separators are also allowed but only one, per file, can separate each field.) All lines must contain the same fields. (The interpretation of the fields is specified by the .schema file, described in the next section.) For example, the first few lines of retail store data might look like this:

```
Eastern  Maryland  Baltimore  1816  appliances  72  115  138
Eastern  Maryland  Baltimore  1816  clothing  355  344  395
Eastern  Maryland  Baltimore  1816  electronics  156  182  209
Eastern  Maryland  Baltimore  1816  furniture  78  75  82
Eastern  Massachusetts  Boston  1331  appliances  48  68  81
Eastern  Massachusetts  Boston  1331  clothing  307  258  296
Eastern  Massachusetts  Boston  1331  electronics  38  183  210
Eastern  Massachusetts  Boston  1331  furniture  52  69  75
Eastern  Massachusetts  Boston  1220  appliances  37  63  75
Eastern  Massachusetts  Boston  1220  clothing  233  240  276
Eastern  Massachusetts  Boston  1220  electronics  175  208  239
Eastern  Massachusetts  Boston  1220  furniture  35  53  58
```

In this example, the first five columns are strings: region, state, city, store ID, and product. These are followed by three numbers, representing current sales, last year's sales, and the sales target.

The data file cannot contain blank lines or comments. Missing or extra data on a line causes an error.

Note: One tab (the default separator) separates each field. Do not insert multiple tabs to line up the fields visually; this generates blank fields. You can use other characters, such as a colon (:), as a separator. In this case, the first line appears as follows:

```
Eastern:Maryland:Baltimore:1816:appliances:72:115:138
```

The order of the columns must match the format of the *.schema* file. For some visual tools, the order of the rows can affect the layout of the final graphic. See the tool-specific chapters for details.

Any field in the data can also be a "?", indicating that the data is null (unknown). See Chapter 13, "Nulls in MineSet."

Note: MineSet also supports a binary format, which currently is not documented.

.schema File

The *.schema* file consists of an input section, which defines the name and format of the file. The *.schema* files generated by the Tool Manager can also contain a history section, which is a copy of the *.mineset* file. This section is used by drill through and would normally not be present in manually generated *.schema* files.

A typical input section might look like this:

```
input {
   file "store";

   string region;
   string state;
   string city;
   string storeId;
   string product;
```

```
   float sales;
   float lastYear;
float target;

   options separator ':';
}
```

This example states that the input file is called *store*, and that there are eight fields: five of type **string** and three of type **float**.

File Statements

The file statement names the data file to be read. This statement is required. Its syntax is:

```
file "filename";
```

Filename must be in double quote marks. If it is a relative pathname (no leading slash), it is first sought in the directory containing the current *.schema* file. If it is not found in the current *.schema* file's directory, the file is sought in the current directory.

Data Statements

The data statements declare the columns in the data file. The columns must be declared in the order they appear in the data file. The format of most data statements is:

```
type name;
```

Type is **int**, **float**, **double string**, **dataString**, **date**, and **fixedString**(*n*), where *n* is an integer representing the width of the string, and *name* is the variable name. Unlike in C, only one variable can be declared per statement.

Other supported types include enumerations, fixed arrays, and enumerated arrays. You must declare these data types must inside the input section, before the declaration of the specific column.

You can also override the separator by declaring it as follows:

```
type name [ number ] separator ‘char’;
```

For example:

```
float revenue [50] separator ‘:’;
```

If no separator is specified, the default column separator (usually a tab) is used.

Exceptions

The following exceptions apply to the *.schema* and *.data* files:

- The Tree Visualizer supports only one-dimensional arrays.
- The Tree Visualizer supports variable-length arrays.
- The Map Visualizer and the Scatter Visualizer support a special enum format for dates.
- The Tree Visualizer and the Map Visualizer support the Monitor option.

Note: These exceptions are discussed in detail in the respective viz tool’s chapters.

Creating Data and Configuration Files for the Tree Visualizer

This chapter describes the types and formats of data supported by the Tree Visualizer and how the Tree Visualizer reads in and graphically displays the data file. The topics discussed are:

Both the data and configuration files can be generated automatically by the Tool Manager. Read the Tree Visualizer chapter in the *MineSet Enterprise Edition User's Guide.* before reading this chapter.

Data File

Data input to the Tree Visualizer must be provided as a single file containing raw data, usually in ASCII text form. An example of this data file is described in Chapter 6, "Flat File Support for MineSet." Special conditions for the Tree Visualizer are noted here.

The only restrictions on data files for the Tree Visualizer are in the use of arrays. With the Tree Visualizer, you can use one-dimensional arrays of fixed or variable size.

With the Tree Visualizer, variable length arrays are particularly useful for describing the tree organization. Often this can represent organizations in which different parts have different depths. For example, one department could be represented by Gomez:Shapiro:Lacy (three entries), while another is Gomez:Wong:McMartin:Singe (four entries).

A variable-length entry with zero values can also be declared by passing an empty string. This can be used to specify data for the top level of a hierarchy.

When representing an organization with variable-length arrays, be careful. The Tree Visualizer computes the height for each level of the hierarchy separately, giving the highest bar on each level a user-specified height and normalizing the other bars accordingly.

For example, imagine a U.S.-based organization with a domestic and an international sales force. Domestic sales are divided into amounts for each state, which are further divided into amounts for each city. International sales are divided into amounts for each continent, which are then divided into amounts for each country and city.

You can have sales amounts for locations such as domestic:California:Mountain View, and international:Europe:Italy:Rome. When displaying organizational hierarchies of this type, it is best to normalize heights at each level. Otherwise, small parts of the organization (for example, Mountain View) would be dwarfed by large parts of the organization (for example, domestic).

When the system tries to match up the levels, the normalization process might introduce anomalies. Usually, this is not the case at the highest level (domestic is matched with international); however, at lower levels this correspondence is no longer valid. Domestic cities (for example, Mountain View) are at the third level, but the third level for international is a country (for example, Italy). Comparing domestic cities against foreign countries usually has little validity. In this case, you might introduce artificial levels to balance the hierarchies (for example, domestic:USA:California:Mountain View), thus matching cities to cities.

Variable-length arrays might also be useful when some of the regions being compared are subdivided further than others. For example, an organization might have USA:California:San Francisco and USA:California:Los Angeles, but only USA:Wyoming. There is no need to construct an artificial third level(such as a city in Wyoming) just to keep the arrays balanced, as long as each level in the array matches the same level in other arrays.

Starting the Tree Visualizer takes longer when variable-length arrays are read in than when fixed-length arrays or individual columns are read in. Unless the data is variable length, it is best not to use variable-length arrays.

Configuration File Overview

The configuration file format is flexible. Words in it must be separated by spaces, and it is case-sensitive. Except for the include statement and text within quoted strings, spacing and line breaks are irrelevant. See Chapter 5, “Data and Configuration File Basics,” for basic information about configuration files.

Configuration File Input Section

The first section of a configuration file is normally the **input** section. It defines the name and format of the file. A typical **input** section might look like this:

```
input {
   file "store";

   string region;
   string state;
   string city;
   string storeId;
   string product;
   float sales;
   float lastYear;
   float target;

   options separator ‘:’;
}
```

This example states that the input file is called *store*, and that there are eight fields: five of type **string** and three of type **float**.

When the **input** section is entered, the options file, *input.treeviz.options*, is read in.

The syntax for the **input** section is the same as that for *.schema* files and is described in Chapter 6, “Flat File Support for MineSet.” For treeviz configuration files generated by the MineSet Tool Manager, the *.schema* file is used for the **input** section using an **include** statement.

Input Options

Treeviz does support one option not found in the standard input section of a *.schema* file. All options statements begin with the word *options* and have one or more comma-separated options.

- The monitor option allows a dynamic update of the data displayed. When the specified file is changed (for example, through the *touch* command), the data file (not the configuration file) is reread. The data file should not be used to trigger the updates. This prevents the data file from being read at the same time it is being updated.

  The syntax of the monitor option is:

  ```
  options monitor "filename";
  options monitor "filename" timeout;
  ```

  filename is the file to watch, and the optional *timeout* specifies the number of seconds to wait after the file changes. If the user interacts with the application in any way during this timeout (via the mouse or keyboard), the timeout restarts. Updating the file can take a few seconds. If you specify a timeout, the chances of an update occurring while the user is interacting with the tool are minimized (but the update is delayed). If you do not specify a timeout, the update occurs immediately.

  The file being monitored must exist at the start of the program. When this file is being updated, you must not remove and re-create it. Instead, you should update only its modify time (for example, through the *touch* command). If the file is deleted, subsequent updates are not shown.

  Suppose a program extractor extracts data from a database into a data file. If you want the program to update the data file every 10 minutes, the script might look like this:

  ```
  extractor > dataFile;        # create first data file
  touch trigger;               # create the trigger file
  while (sleep 600)            # sleep 10 minutes
  do
     extractor > dataFile;     # create new data file
     touch trigger;            # force a reread
  done &                        # this loop goes in the
                               # background
  treeviz configFile;          # run treeviz
  kill $!                       # when treeviz exits, kill the
                               # update loop
  ```

You can use the monitor option only if the file alteration monitor */usr/etc/fam* is installed. This can be found in the subsystem *desktop_eoe.sw.fam*.

The **input** section of a configuration file might look like this:

```
input
{
   file "dataFile"
   #data declarations here
   options monitor "trigger" 15;
}
```

Configuration File Expressions Section

The **expressions** section of a data file lets you define additional columns that are expressions of existing columns. For example, you can define one column as the sum of two other columns. The expressions are calculated before the definition of the hierarchy. In many cases, it is more appropriate to apply the expressions after creating the hierarchy. You should then define the expressions within the **hierarchy** section (see "Configuration File Hierarchy Section" on page 58), and you can omit the **expressions** section.

The following is a sample **expressions** section. This section assumes two existing columns of type **float**, "male" and "female"; these represent spending by males and females on various goods. Two columns are added: "total" represents the total dollars spent, and "pctFemale" represents the percentage of dollars spent by females.

```
expressions
{
float total = male+female;
float pctFemale = divide (female*100, total, 50.0);
}
```

Note: The pctFemale calculation uses "total," defined in the previous statement. Also, note the use of the divide function rather than the **/** operator. This results in 50% for the case in which no dollars are spent at all; using the **/** operator generates an error, because it results in division by zero.

The format of the **expressions** section is:

```
expressions
{
   expressionDeclaration;
   ...
}
```

expressionDeclaration has the following syntax:

```
type name = expression;
```

Because the **expressions** section has no options, no options file is read in for it.

Configuration File Hierarchy Section

The **hierarchy** section of a data file describes how the previously read table is converted into a hierarchy. The following is a sample **hierarchy** section:

```
hierarchy
{
   levels region, state, city, storeId;
   key product;
   aggregate
   {
      sum sales;
      sum lastYear;
      sum target;
   }
   expressions
   {
      float pctLastYear = divide(sales*100, lastYear, 100.0);
      float pctTarget = divide(sales*100, target, 100.0);
   }
}
```

When entering the **hierarchy** section, the *hierarchy.treeviz.options* options file is read in.

The parts of the **hierarchy** section are described in subsequent sections.

Levels Statements

The **levels** statement defines how the table is converted into a hierarchy. The format is:

```
levels name, name...;
```

name represents a column previously defined in the **input** or the **expressions** section. How the hierarchy is created depends on the types of the columns defined.

If the columns represent simple types (for example, strings or numbers), each column is converted into a single level of the hierarchy. The top level of the hierarchy is a single, all-inclusive node. The next level contains one node for each unique value in the first column. The third level contains one node for each unique value in the second column, and so on. Hierarchies created in this way are always balanced: all branches in the hierarchy go to the same depth (namely one greater than the number of columns specified in the levels statement).

If the column is an array, you can specify only a single column in the **levels** statement. Each value in the array is mapped to one level in the hierarchy. The top level is a single node representing the total aggregation. The next level contains one node for each unique value of the first value in the array; the third level contains one node for each unique value of the first two values of the array, and so on.

If the array is of fixed type, this hierarchy is balanced. If a variable array is used, the hierarchy is not necessarily balanced (some branches can go deeper than others).

You can use a variable-length array to specify the hierarchy, even if the hierarchy is balanced to a fixed depth. When using columns or fixed arrays to specify the levels, you can specify data associated only with those levels at the bottom (or leaf) nodes. In this case, all higher nodes in the hierarchy must be aggregated. However, rather than relying on automatic aggregation, you might want to supply your own data for each level of the hierarchy (if, for example, the calculation cannot be performed automatically by the Tree Visualizer). In that case, use variable-length arrays to specify levels and provide separate data for each level.

For example, the data file might contain lines such as:

```
Domestic:Western  43
Domestic:Eastern  57
Domestic    85
Intl:Europe  52
Intl:Asia  39
Intl   94
      133
```

Note: The last line has an empty value for the location; the number 133 is translated to the top of the hierarchy.

Key Statements

The **key** statement specifies those keys that are used to select the bars at each node in the hierarchy. The key corresponds to the bars displayed in the final view. The syntax of the **key** statement is:

```
key name [sort [ascending|descending]};
```

name is the name of one of the previously defined columns. It cannot be the name of a column used in the **levels** statement. Only a single **key** statement can be made.

By default, the bars generated by the key statement appear in the order first encountered. If the key is an enumerated array, the bars appear in the order of the enumeration; otherwise they appear in the order in which values are first encountered in the data file.

Adding the word **sort** at the end of the **key** statement sorts the bars. Sorting depends on the type: strings are sorted alphabetically, and numbers are sorted numerically. Enumerations are sorted on the index of the enumeration, not the string to which the enumeration refers. If, however, the key is an enumerated array, the sorting occurs according to the enumeration string. To sort based on the enumeration index, leave it unsorted. Optionally, the word **sort** can be followed by **ascending** or **descending** to specify the sort order; the default is ascending.

If the key column is a simple type (for example, a string), the unique values of that key are looked up in the original table. The order of the values is the same as the one in which the key values appear in the original input table. Although it is not required, the same keys are often repeated in the same order. For example, in the following table, the fifth column is the key, and has the values “appliances,” “clothing,” “electronics,” and “furniture.”

```
Eastern  Maryland  Baltimore  1816 appliances  72  115  138
Eastern  Maryland  Baltimore  1816 clothing  355  344  395
Eastern  Maryland  Baltimore  1816 electronics  156  182  209
Eastern  Maryland  Baltimore  1816 furniture  78  75  82
Eastern  Massachusetts  Boston  1331 appliances  48  68  81
Eastern  Massachusetts  Boston  1331 clothing  307  258  296
Eastern  Massachusetts  Boston  1331 electronics  38  183  210
Eastern  Massachusetts  Boston  1331 furniture  52  69  75
Eastern  Massachusetts  Boston  1220 appliances  37  63  75
Eastern  Massachusetts  Boston  1220 clothing  233  240  276
Eastern  Massachusetts  Boston  1220 electronics  175  208  239
Eastern  Massachusetts  Boston  1220 furniture  35  53  58
```

The key can also be any column of the enumerated array type. In this case, the enumeration is used as the key for specifying the bars. Other columns in the input can also be enumerated array types, as long as they use the same enumeration. For example, this table can also be input as:

```
Eastern  Maryland  Baltimore  1816
              72:355:156:78  115:344:182:75  138:395:209:82
Eastern  Massachusetts  Boston  1331
              48:307:38:52  68:258:183:69  81:296:210:75
Eastern  Massachusetts  Boston  1200
              837:233:175:35  63:240:208:53  75:276:239:58
```

For clarity, each line has been wrapped onto two lines; however, in the file these should be on single lines. The **input** section for this data appears as:

```
input
{
   file "...";
   key string product {
   "appliances", "clothing", "electronics", "furniture"
   }
   string region;
   string state;
   string city;
   string storeId;
   float sales [ enum product ] separator ‘:’ ;
   float lastYear [ enum product ] separator ‘:’ ;
   float target [ enum product ] separator ‘:’ ;
}
```

Note: Because the arrays are fixed, the use of a colon separator for the arrays is not required; however, it might make it easier for the user to read the input.

In this example, the **hierarchy** section appears as follows:

```
hierarchy
{
   levels region, state, city, storeId;
   key sales;
   ...
}
```

Because *sales* is an enumerated array, it used its key type (product) as the key to generating the bars; thus, each graph in the final view has four bars. Note that *lastYear* and *target* must use the same key type for their arrays.

Arrays other than enumerated arrays cannot be specified as the key.

Aggregate Subsection

The *aggregate* subsection of the *hierarchy* section describes how values are aggregated at higher levels of the hierarchy. An example is:

```
aggregate
{
   sum sales;
   sum lastYear;
   sum target;
}
```

This indicates that *sales*, *lastYear*, and *target* are to be summed at higher levels of the hierarchy (each level summing the values in the level below it). In addition to the **sum** aggregation, the aggregations **average**, **min**, **max**, **count**, and **any** are allowed. All are self-explanatory, except for **any**, which indicates that any of the values can be used. This aggregation is used if you expect the same value (for example, a string) to appear everywhere in the hierarchy and if you just want it to populate the entire hierarchy.

A special case is when the key is an enumerated array. Here, the key is normally also aggregated.

IF a variable-length array specifies data for all levels of the hierarchy simultaneously (as opposed to merely specifying the data at the leaf nodes), the **aggregate** section cannot be used.

An **aggregate** statement can take either of the following two forms:

```
agg name;
name1 = agg name2;
```

In both cases, the aggregate (agg) is one of **sum**, **average**, **min**, **max**, **count**, and **any**. The first form is illustrated in the preceding paragraphs; it aggregates a column, and the result is given the same name as the original column being aggregated. The second form aggregates the column *name2*, but names the result *name1*. This second form is useful if the same value is being aggregated multiple times. Because using the first form creates two aggregations with the same name, the second form can be used to differentiate the aggregations.

For example, if you have a column named *expenses* and want to aggregate it to show the maximum and minimum expenses, you can use:

```
aggregate
{
   maxExpenses = max expenses;
   minExpenses = min expenses;
}
```

Aggregate Base Subsection

This subsection specifies how values in the base are aggregated. It can be used only if the **aggregate** subsection is not present. If the **aggregate** section is present, the base is aggregated using the aggregations specified in it.

A sample **aggregate base** subsection is:

```
aggregate base
{
    sum sales;
    sum lastYear;
}
```

An **aggregate** statement takes the form:

```
agg name;
```

where the aggregate (agg) is one of **sum**, **average**, **min**, **max**, **count**, and **any**, (similar to the aggregate section). The aggregation is applied to all the bars on that base to give the appropriate value for the base. After the base is aggregated, its values correspond to all of the columns used in specifying the bars. Any column not specified in the **aggregate base** section has a value of zero. Because the base values correspond to the bar values, the second form of the **aggregate** statement (using the =) cannot be used in the **aggregate base** section.

Expressions Subsection

An **expressions** subsection of the **hierarchy** section is similar to the **expressions** section described previously, except that it is applied after the hierarchy is created and aggregated. The syntax is identical, but it is declared within the **hierarchy** section, not external to it.

To give an example of the difference between calculating the expressions before and after creating the hierarchy, take the example of male and female dollars spent. Assume you want to calculate the percentage of dollars spent by women. The expressions might be:

```
expressions
{
   float total = male+female;
   float pctFemale = divide (female*100, total, 50.0);
}
```

Assume you calculated these variables before creating the hierarchy. Then, when aggregating the data up the hierarchy, summing the percentages is not useful. Averaging the percentages results in a believable number; however, it averages percentages of large dollars with percentages of small dollars, and produces incorrect results. (To make this clearer, suppose that on one product, males spent $99 and females spent $0. On another product, males spent $0 and females spent $1. On the first product females spent 0%, and on the second they spent 100%. Averaging these gives 50%, but in reality, females spent only 1% of the dollars spent on the two products combined.)

The base data should be aggregated first, and then the expressions should be applied. (In the example, after aggregating, the result is a combined spending of $99 for males and $1 for females; if the percentage is calculated after the aggregation, the correct value of 1% results.)

Sort Statements

By default, the order of the nodes within each level of the hierarchy is based on the order of the data in the input file. However, sometimes it is desirable to sort the hierarchy. The **sort** statement can appear in one of two forms:

```
sort name [, ascending|descending];
sort key [, ascending|descending];
```

In the first form, one column name (not used in the level statement) is used for sorting. The column can be the result of an aggregation or an expression. In the second form, the value used in the **level** statement is the one used in laying out the hierarchy.

The hierarchy can be sorted in ascending or descending order. If neither option is specified, the default is descending order if the first form of the sort is used (this places the largest columns on the left); the default is ascending order if the second form is used (this typically sorts alphabetically).

sort statements affect the sorting of only the branches of the hierarchy; they do not affect the bars within each node of the hierarchy.

Hierarchy Options

There are two options in the hierarchy section: *skipMissing* and *organization*. The format for the *skipMissing* option is:

```
options skipMissing;
```

If this option is off (the default) and some values of the key are not present for a given hierarchy node, dummy entries are created with values of 0. This guarantees that all graphs in the hierarchy have the same number of bars, and the same layout. If this option is on, no such entries are generated. This results in variable-length tables in the hierarchy, and bars exist only for items in the input. The position of these bars, however, is not meaningful. This option is not useful if the key is an enumerated array (for which all values are supplied).

The *skipMissing* option increases memory usage and should be avoided, if possible.

The format for the *organization* option is:

```
options organization same;
options organization contains;
options organization unknown;
```

The **organization** option provides hints about the hierarchy organization that allow for more efficient algorithms. This option is most useful if no hierarchy aggregation is done. The possible values for this option are as follows:

- **same**: specifies that all nodes in the hierarchy contain entries for the same item (for example, all nodes could contain "appliances," "clothing," "electronics," and "furniture").
- **contains**: indicates that a parent node contains entries for all values that its children contain. For example, if a node contains "appliances," its parent node must also contain "appliances," although not all of its child nodes must contain appliances.
- **unknown**: indicates that no assumptions should be made regarding the contents of individual nodes.

If you do not specify an organization, the Tree Visualizer determines the organization as follows:

- If there is no **aggregate** subsection, **unknown** is used.
- If there is an **aggregate** section, but the **skipMissing** option is provided, **contains** is used; otherwise, **same** is used. Because this is normally correct when an **aggregate** subsection is provided (unless **skipMissing** is used but nothing is missing), there normally is no need to provide an organization if the **aggregate** subsection is present.

If the organization specified does not match the data, the results are unspecified. For example, you should not specify **same** unless all nodes have the same entries.

Configuration File View Section

The **view** section of a data file describes how the hierarchy is displayed, including the mapping of heights, colors, labels, and so forth. A sample **view** section is:

```
view hierarchy landscape
{
   height sales, normalize levels, max 2.0;
   height legend label "Height: Total sales";
   base height max 1.0;
   disk height target, legend label "Disk height: Target
            sales";
   color pctTarget, scale 0 100 200 500;
   color colors "red" "gray" "green" "blue";
   color legend label "Color: % of target" "0%" "100%"
            "200%" "500%";
   options columns 4;
   message "$%,.2f, %.0f%% of target, %.0f%% of last year",
            sales, pctTarget, pctLastYear;
}
```

The first words of the **view** section (before the opening brace) describe the type of view. The only view type supported is **view hierarchy landscape**; therefore, these words must introduce the **view** section.

When entering the **view** section, the *viewHierarchyLandscape.treeviz.options* options file is read in.

Note: There is not a simple *view.treeviz.options* options file; you must use the full name, *viewHierarchyLandscape*.

Height Statements

The **height** statement describes how the columns are mapped to the height of objects. It consists of a series of clauses separated by commas. Alternatively, you can specify multiple **height** statements. The following three examples are equivalent:

- ```
  height sales, normalize levels, max 2.0;
  ```
- ```
  height sales;
  height normalize levels;
  height max 2.0;
  ```
- ```
  height sales, normalize levels;
  height max 2.0;
  ```

The first clause normally contains the name of a column to be mapped to height ("sales," in the example). The column must be of a number type (**int**, **float**, or **double**); **float** is the most efficient. If you do not specify a height column, all bars are flat, and the remaining height clauses have no effect.

Normalize Clause

The **normalize** clause determines the maximum value of the height variable; it normalizes all values relative to that height. Therefore, if the maximum value is 30.0, and that bar was given a height of 1.0 (in arbitrary units), a value of 15.0 would be mapped to a value of 0.5.

The syntax of the **normalize** clause can be any of the following:

```
normalize
```

This normalizes all values against one another, throughout the hierarchy.

```
normalize levels
```

This performs independent normalization at each level of the hierarchy.

```
normalize none
```

This performs no normalization, and is the default.
```

The second form is particularly useful if the data is aggregated up the hierarchy. For example, assume the sales data is aggregated up the company. Comparing the sales of the company as a whole to the sales of a single individual has little meaning; in a large company, the heights of the bars for the individuals are so small as to be indistinguishable from zero. It makes more sense to compare salespeople to salespeople, offices to offices, regions to regions, and so on. Normalizing levels does this.

Regardless of which form of normalization is used, the base (if shown) is always normalized independently of the bars. By default, the same normalization mechanism for the bars is used for the base.

Scale Clause

The **scale** clause scales the height of all objects; all values are multiplied by the scale. The syntax of the **scale** clause is:

```
scale float
```

float is a floating point number (the decimal point is optional). For example, to double the heights, specify:

```
scale 2
```

Filter Clause

Large datasets can contain many graphics. This results in poor performance. In many cases, the data values are small and of little informative value. The **filter** clause prefilters the data based on the height variable, so that only the nodes with the highest bars are shown. The syntax of the **filter** clause is:

```
filter > float%
```

You must type in the > and % characters. For example:

```
filter > 5%
```

This example filters out all charts containing no bars greater than 5% of the maximum bar height, except for those containing descendants in the hierarchy containing such bars. If a chart contains just one bar that meets this criterion, the entire chart is shown.

You can also change the **filter** value interactively through the Filter panel.

You can use the **filter** clause only on the **height** statement.

Legend Clause

The **legend** clause defines the meaning of the height mappings. Any string can be placed in the height legend. The **legend** clause has the following syntaxes:

- `legend off`

  This turns off the height legend (this is the default).

- `legend on`

  This turns on the height legend.

- `legend label labelstring`

  This changes the legend. If legend label is used, legend on is unnecessary.

  By default, the legend has the following syntax:

- `legend:varname`

  varname is the name of the variable that is mapped to height.

You can declare separate legends for the height, the base height, and the disk height.

Base Height Statements

The **base height** statement specifies how the height of the base is calculated. The format is similar to the height statement, except that it is preceded by the word **base**. If you omit the **base height** statement, the height of the base is calculated using the same values as in the **height** statement (the same variable, normalization mechanism, max value, and so on). You also can specify only some of the clauses for the base, in which case everything else is the same as the **height** statement. For example:

```
height sales, normalize levels, max 2.0;
base height max 1.0;
```

In this case, the base height is based on *sales*, and it is normalized by *levels*. The maximum height, however, is only 1.0 instead of 2.0. Usually, the visual effect is better if the base height max is less than the max for the bars. You cannot use the filter clause with the **base height** statement.

The On and Off Clauses

You can turn the initial value of the base height on and off via the **on** and **off** clauses. To turn it off, use:

```
base height off
```

To turn it on, use the default:

```
base height on
```

You can change the base height interactively using the Base Height option in the Display menu. The **on** and **off** clauses are valid only with **base height**. Do not use them with the **height** or **disk height** statements.

Disk Height Statements

With the **disk height** statement, you can place a disk on each bar to indicate an additional item of data. The syntax of the **disk height** statement is similar to that of the **height** statement, but it is preceded by the word **disk**. For a disk to be displayed, there must be a clause specifying the column to be mapped to the disk. Other clauses are optional; if you omit these, the height statement's defaults are used.

If the **height** statement has a normalize clause, and the disk height statement has no normalize or max clause, then the disks are normalized with the bars (they are drawn to the same scale). If the disk height statement has either a normalize clause or a max clause, the disks are normalized independently of the bars. For example:

```
height sales, normalize levels, max 2.0;
disk height target;
```

In this case, the bars are mapped to the variable "sales," and the disks are mapped to "target." Both are normalized, with the maximum value of sales or target on each level mapped to a value of 2.0. If instead this example is written as:

```
height sales, normalize levels, max 2.0;
disk height target, normalize levels;
```

In this case, the bars are mapped so the highest bar at each level is 2.0, and the highest disk on each level is 2.0, but the bars and disks are not mapped to the same scale. You can use this, for example, if the bars represent dollars and the disks represent head count.

You cannot use the **filter** clause with the **disk height** statement.

Color Statements

The **color** statement describes how values are mapped to colors. The format is similar to that of the **height** statement, consisting of several clauses that you can separate by commas or enter as multiple statements.

Color Naming

Color names follow the conventions of the X Window System™, except that the names must be in quotation marks. Examples of valid colors are “green,” “Hot Pink,” and “#77ff42.” The last one is in the form “#rrggbb”, in which the red, green, and blue components of the color are specified as hexadecimal values. Pure saturation is represented by ff, a lack of color by 00. For example, “#000000” is black, “#ffffff” is white, “#ff0000” is red, and “#00ffff” is cyan. (A list of available colors is found in the file *rgb.txt*.)

- Windows users should only use the “#ff0000” form, as only some of the named colors are supported (for example, white, black, gray, red, yellow, green, and so on.).
- UNIX users can find the color list file at */usr/lib/X11/rgb.txt*.

Color Variable

As with height, you also can specify a single column to be mapped to a color. The column must be a number type. Unlike for height, there is no normalization of colors.

Key Clause

Instead of specifying a variable, you can specify the word **key**. This assigns a different color based on each key, normally for each bar. For example, if the 50 states were the keys, key assigns a different color to the bar for each state. Because the base is not keyed, when the **key** clause is used, the base is always gray.

Colors Clause

The **colors** clause specifies the colors to be used. The *colors* clause syntax is:

```
colors "colorname" "colorname"...
```

The format for *colorname* is described under “Color Naming” on page 72. There are no commas between the colors. This is because commas are used to separate clauses in the **color** statement. A sample colors clause is:

```
colors "red" "gray" "blue"
```

Colors in the list are subsequently referred to by their index, starting at zero. In the preceding example, red is color 0, gray is color 1, and blue is color 2.

If there is no **colors** statement, colors are chosen randomly; however, if there is a **colors** statement, you must specify at least as many colors as are to be mapped. If you use a key, you must specify one color for each key value.

Scale Clause

The **scale** clause allows assignment of values to a continuous range of colors. For example, when displaying a percentage, red can be assigned to 0%, gray to 50%, and blue to 100%. Intermediate values are interpolated; for example, 25% is pinkish, and 55% is a slightly bluish gray.

The syntax for the **scale** clause is:

```
scale float float ...
```

The first value is mapped to color 0, the second to color 1, and so forth. The **colors** statement must contain at least as many colors as are to be mapped to the largest index.

Values in this statement must be in increasing order. Any value less than the first color is assigned the value of the first color. Any value greater than the last value is assigned the last color. Intermediate values are interpolated.

For example, assume the pctFemale column indicates what percentage of the group is female, and you want to map a group that is 100% female to red, 100% male to blue, and 50% each to gray. The **colors** statement for this is:

```
colors pctFemale, colors "blue" "gray" "red", scale 0 50 100;
```

Buckets Clause

The **buckets** clause is similar to the **scale** clause without interpolation. All values are rounded down to the highest value in the clause, and that exact color is used. Values less than the first value use the first color.

The syntax for the **buckets** clause is:

```
buckets float float ...
```

The syntax and assignment of colors is the same as for the **scale** clause.

If, in the pctFemale example, you used the **buckets** clause instead of the **scale** clause, the statement would be:

```
colors pctFemale, colors "blue" "gray" "red", buckets 0 50 100;
```

All values greater than or equal to 100 are colored red. Values greater than or equal to 50, but less than 100, are gray. All other values are blue.

Legend Clause

The **legend** clause creates a legend of the colors. By default, a legend is on for the bar colors, and off for base and disk colors, although separate legends are permitted for each. The **legend** clause syntax can be any of the following:

```
legend off
legend on
legend "string" "string" ...
legend label "string"
legend "string" "string" ... label "string"
```

The **legend off** clause turns the legend off. The **legend on** clause turns the legend on. You can omit it if you include other **legend** statements. Specifying only **legend on** generates the default legend.

The default legend includes a single label to the left (with the name of the column that is mapped to color), and a list of colored labels on the right (with values obtained from the scale clause, the buckets clause, or from the keys). To override the strings in the colored labels, specify the strings as:

```
legend "string" "string.
```

To override the label on the left, specify it following the word label. To eliminate this label, specify an empty string as follows:

```
legend label ""
```

Base Color Statements

The **base color** statement controls the color of the base. Its syntax is similar to the **color** statement, except that it is preceded by the word **base**. If this word is omitted, the base has the same color as the bars. If you include the **base color** statement, any omitted clauses default to the values of the **color** statement.

Disk Color Statements

The **disk color** statement controls the color of the disk. The syntax is similar to the **color** statement, except that it is preceded by the word **disks**. If you omit the **disk color** statement, the disk has the same color as the bars. If you include the statement, any omitted clauses default to the values of the color statement.

Because disks are drawn only if a **disk height** statement is present, a **disk color** statement has no effect without a **disk height** statement.

Label Statements

Label statements specify the labels used when labeling objects in the scene. Normally, you can omit these statements. By default, each bar is labeled with its key; each base is labeled with its position in the hierarchy. The types of syntax for the **label** statements are:

```
label name
base label name
line label name
back label name
```

name is the name of the column to be used as the label. The first form is used as the label on the bars. The second form is the label on the bases. The third form labels the lines connecting the bases. The fourth places labels behind the bases. (Note that bases often obscure the back labels, so this form is less useful; however, there might be occasions when it is appropriate.)

Message Statements

The **message** statement specifies the message displayed when the pointer is moved over an object or when an object is selected. The syntax is similar to that of the C **printf** statement. A sample **message** statement is:

```
message "%s: $%f, %.0f%% of target, %.0f%% of last year",
      product, sales, pctTarget, pctLastYear;
```

This could produce the following message:

```
furniture: $2425.37, 23% of target, 87% of last year
```

The formats must match the type of data being used:

- **Strings** must use %s.
- **Ints** must use integer formats (such as %d).
- **Floats** and **doubles** must use floating point formats (such as %f).

For details of the **printf** format, see the printf (1) reference (man) page (type **`man printf`** at the shell prompt).

A special format type has been added to **printf**. If the percent sign is followed by a comma (for example, "%,f"), commas are inserted in the number for clarity. Currently, only the U.S. convention of d,ddd,ddd.dddd is supported, with the decimal point represented by a period, and commas separating every three places to the left of the decimal point. For example:

```
message "%s: $%,f, %,.0f%% of target, %,.0f%% of last year”,
      product, sales, pctTarget, pctLastYear;
```

In this case, it would produce the following message:

```
furniture: $2,425.37, 23% of target, 87% of last year
```

The **$**, *, h, l, ll, L, and n **printf** format options are not supported.

All values, including the format string, are expressions. Therefore, if you had a pctFemale column, but wanted a more gender-neutral message, you could use:

```
message pctFemale>50?"%f%% females":"%f%% males",
      pctFemale>50?pctFemale:100-pctFemale;
```

If pctFemale is 70, the message “70% females” is displayed; if pctFemale is 30, the message “70% males” is displayed. In this case, you can also achieve the same result with a single format string:

```
message "%f%% %s", pctFemale>50?pctFemale:100-pctFemale,
      pctFemale>50?"females":"males";
```

By default, the same message is used for the base as for the bars. You can specify a different message by using a **base message** statement, which has the same syntax.

If no message is specified, a default message containing the names and values of all the columns is used.

Execute Statement

The **execute** statement lets you execute a shell command by double-clicking an object. The syntax is similar to that of the **message** statement; however, because hierarchy information is not displayed on a separate line, it is useful to include the hierarchy information and to pass the key information as arguments.

The following is a sample **execute** statement that uses **xconfirm** to show a window with information about the item. The first line, the string, is broken into multiple lines to fit on a single page. In an actual file, it should be on a single line. Multi-line strings are not supported.

```
execute "xconfirm -t ‘%s’ -t ‘sales of %s’ -t ‘$%,.0f’
      -t ‘target $%,.0f (%.0f%% of target)’
      -t 'last year $%,.0f, %.0f%% of last year'>/dev/null",
   hierarchy(" "), isSummary()?"everything":product,
   sales, target, pctTarget, lastYear, pctLastYear;
```

This might produce a dialog with the following message:

```
Eastern Connecticut Milford
sales of clothing
$348
target $427 (81% of target)
last year $372 (94% of target)
```

Note the use of hierarchy (" ") to produce a blank-separated description of the hierarchy. Also note the `isSummary()?”everything”:product`; this produces the word “everything” if the base was selected, but otherwise produces the product. An alternative to this is using separate **execute** and **base execute** statements.

If there is no **execute** statement, double-clicking an object has the same effect as single-clicking it.

This sample **execute** statement will work on Windows as well as UNIX, because a simple *xconfirm* utility is provided with the installation on Windows.

View Options

The **view** section has many options. Like other options statements, the options can be separated by commas, or they can appear on separate lines.

Sky and Ground Colors

You can specify the sky and ground color using the following syntax:

```
options sky color colorname
options sky color colorname colorname
options ground color colorname
options ground color colorname colorname
```

The syntax for color names is the same as that for color naming.

For both the sky and the ground, you can specify either one or two colors. If you specify only one color, the sky or ground is solid. If you specify two colors, the sky or ground is shaded between the colors. For the sky, the first color is for the top of the sky, the second for the bottom. For the ground, the first color is for the far horizon, the second for the near ground.

For example, to have a solid black background, specify:

```
options sky color "black", ground color "black";
```

Bar Layout

By default, bars in each chart are laid out as close to a square as possible. You can override this by using either the rows or the columns option:

```
options rows number
options columns number
```

You can specify only one of these.

Overview

Although you can use the Show menu to see the overview, you can also configure it to display automatically at startup. The overview syntax is:

```
options overview on
options overview off
```

The first form causes the overview to be displayed at startup. The second form (the default) turns the overview off.

Shrinkage

Hierarchies normally have a large aspect ratio, having greater width than depth. In their unaltered form, it is impossible to view the entire hierarchy, except from such a far distance that no detail would be visible. To see the hierarchy more clearly, distant objects can be shrunk more than perspective normally dictates. The shrinkage option lets you control the shrinkage for a given graph. The shrinkage option syntax is any of the following:

```
options shrinkage auto
options shrinkage float
options shrinkage off
```

The first form (the default) automatically calculates a shrinkage value. Its results are usually reasonable, but not necessarily optimal in unusual hierarchical layouts. Thus, you might want to explicitly set the shrinkage using the second form. For hierarchies in which some parts are deeper than others, automatic calculation does not work well. The best shrinkage value depends on the graph being displayed, as well as various layout options such as margins. You should experiment with each graph. Start with a value of 10.0, and then make adjustments. Smaller values result in a narrower hierarchy and increased distortion. The shrinkage value must be positive; avoid values smaller than 5.0.

Shrinkage can be turned off. This is recommended only for very small hierarchies, as it produces hierarchies with very large aspect ratios.

Root Label

By default, the root node of the hierarchy receives a label based on the name of the configuration file. You can override this by using the **root label** option. The format is:

```
options root label string
```

This option also affects the string displayed when an object is selected, as well as the result of the **hierarchy()** function.

Note: The root label option has no effect if the **base label** statement was used (that statement defines the base label for the root as well as for all other bases).

Font

The font option controls the font used for drawing the labels. The syntax is:

```
options font "fontname"
```

fontname can be any font in the directory */usr/lib/DPS/outline/base*.

It also can be the string *default*. This attempts to use Helvetica, or the default Inventor font (if Helvetica is not available). Different systems can have different fonts installed.

Base Label Color

The **base label color** option controls the color of the labels in front of the bases. The syntax is:

```
options base label color "color"
```

Bar Label Color

The **bar label color** option controls the color of the labels in front of the bars. The syntax is:

```
options bar label color "color"
```

Line Color

The **line color** option controls the color of the lines connecting the nodes in the hierarchy. The syntax is:

```
options line color "color"
```

Zero

The **zero** option lets you determine whether bars, disks, and bases of height zero are drawn solid, as an outline, or hidden completely. In the last case, space is left for the object, but it is not drawn. The default value is **solid**. You can change this option at run time by using the Display menu.

The syntax for the zero option is:

```
options zero solid
options zero outline
options zero hidden
```

Null

The **null** option lets you determine whether bars, disks, and bases of height null (see Chapter 13, "Nulls in MineSet") are drawn solid, drawn outline, or hidden completely. In the last case, space is left for the object, but it is not drawn. The default value is **outline**. You can change this option at run time by using the Display menu. The syntax is:

```
options null solid
options null outline
options null hidden
```

Other Options

There are 10 other options to control the layout of the display, level of detail, and other parameters. Generally, it is not necessary to adjust these parameters. The values of many of the options are in arbitrary units. Adjust the options by increasing or decreasing the value. For the default values of these parameters, see the file *viewHierarchyLandscape*. Windows users find this file in *Program Files\Sgi\MineSet\config\treeviz*. UNIX users find this file in */usr/lib/MineSet/treeviz*.

- `options speed floatvalue`

  Controls the speed during free-form (middle-mouse) horizontal navigation (forward, backward, and side to side). The larger the value, the faster the motion.

- `options climb speed floatvalue`

  Controls the speed when moving up and down using Shift + middle mouse. The larger the value, the faster the motion.

- `options leaf leaf margin floatvalue`

  Controls the distance between adjacent nodes in the hierarchy. Larger values move the nodes farther away.

- `options root leaf margin floatvalue`

  Controls the distance between a node and its children. Larger values move the nodes farther away.

- `options leaf edge margin floatvalue`

  Adds margin space next to nodes at the edge of a subhierarchy.

- `options initial position floatvalue1 floatvalue2 floatvalue3`

  Provides the initial *x*, *y*, and *z* position from which the scene is viewed. A value of 0 0 0 positions the viewer at the root of the hierarchy; because the user is looking forward, the root probably is not visible. Increasing *x*, *y*, and *z* moves the camera to the right, up, and back, respectively. A typical position has a zero *x*, positive *y*, and positive *z*. If unspecified, the initial position depends on the layout of the hierarchy.

- `options initial angle floatvalue`

  Provides the initial angle, measured in degrees, from which the hierarchy is viewed. The value must be between 0 and 90. A value of 0 looks at the scene horizontally; a value of 90 looks straight down.

- `options bar label size floatvalue`

  Specifies the size of the labels in front of the bars. Larger values result in larger labels.

- `options base label size floatvalue`

  Specifies the size of the labels in front of the bases. Larger values result in larger labels.

- ```
  options lod [bar floatvalue floatvalue] [bar label floatvalue
    [floatvalue]] [base float floatvalue] [base label floatvalue
    [floatvalue]] [disk floatvalue] [motion floatvalue]
  ```

  Controls the level of detail. The parameters can appear in any order, be omitted, or placed in multiple **lod** options. These options control the changing form, or disappearance of, objects, thus providing better system performance.
  ```

Except for the **motion** parameter, all **float** values represent the size of the object when the form change or disappearance occurs. The smaller the value specified, the smaller and farther away the object is when the change occurs. Smaller values provide nicer graphics but slower system performance. The numbers of the different parameters cannot be compared directly because the size of the object also determines when the change occurs. A value of 0.0 means no level of detail changes for that parameter. This setting can significantly slow the rendering process.

bar controls when a bar is drawn with less detail. The first value specifies when the object is drawn as a pair of planes; the second value specifies when the object is drawn as a single line.

bar label controls when the labels on the bars disappear. If two values are specified, the first value specifies when the label is drawn in a lower-quality, fast font; the second value controls when it disappears.

base controls when the bases, and the bar charts in front on top of them, disappear. The first number is based on the width of the base; the second on the height of the base plus the tallest bar on it.

base label controls when the label in front of the base disappears. If two values are specified, the first value specifies when the label is drawn in a lower-quality, fast font; the second value controls when it disappears.

initial depth controls the initial depth to which the hierarchy is viewed. At the top of the hierarchy, you see only the number of hierarchical levels specified by the slider. The nodes in the rows are arranged to optimize their visibility. When the user is navigating to nodes lower in the hierarchy, additional rows become visible automatically. The nodes above them automatically adjust their locations to accommodate the newly added nodes; thus, some nodes might seem to move. Note that the overview shows all nodes in the hierarchy, not just the top nodes, so the layout of the overview might not match the layout of the main view. The X in the overview approximates the corresponding location in the main view; there is no exact mapping between the two layouts.

An initial depth of zero, or one greater than the depth of the hierarchy, shows the entire hierarchy. Once the Tree Visualizer is running, you can change the depth the Filter panel.

disk controls when the disk disappears.

motion controls changes in some of the level of detail calculations when the scene is animated. A value greater than 1.0 defaults to 1.0. A value of 1.0 specifies that motion has no effect on the level of detail. Smaller values change the level of detail at a proportional distance. For example, a value of 0.5 means that during animation, level of detail changes occur at half the normal distance.

Creating Data, Configuration, Hierarchy, and .gfx Files for the Map Visualizer

This chapter describes the data and configuration files that are required for the Map Visualizer. You can also generate these files automatically by using the Tool Manager. The subjects discussed are:

Read the Map Visualizer chapter of the *MineSet Enterprise User's Guide for Windows* before reading this chapter.

Data File

Data input to the Map Visualizer must be provided as a single file containing raw data, usually in ASCII text form. In its simplest form, the data file consists of a list of lines, each containing a set of fields separated by one tab. Other separators are also allowed, but only one can separate each field. See "Input Options" on page 92. All lines must contain the same fields. The interpretation of the fields is specified by the configuration file, described in "Configuration File Overview" on page 87. Using the U.S. population data in the *examples* directory (the *population.usa.data* file), provided as part of the Map Visualizer package, the first few lines of this input file appear as follows:

```
AL    0 0 0 1000 9000 127901 309527 590756 771623 964201 996992 1262505
1513401 1828697 2138093 2348174 2646248 2832961 3061743 3266740 3444354
3894025 4040587    51705
```

```
AR    0 0 0 0 1000 14000 30000 98000 210000 435000 484000 803000
1128000 1312000 1574000 1752000 1854000 1949000 1910000 1786000 1923000
2286000 2351000    53187
AZ    0 0 0 0 0 0 0 0 0 0 10000 40000 88000 123000 204000 334000 436000
499000 750000 1302000 1775000 2717000 3665000    114000
CA    0 0 0 0 0 0 0 0 93000 380000 560000 865000 1213000 1485000
2378000 3427000 5677000 6907000 10586000 15717000 19971000 23668000
29760021    158706
```

In this example, the first column is a two-character string identifying the graphical object—the state. (This string locates a record in a *.gfx* file containing information about the shape of the graphical object.) The tab separator is followed by a grouping of 23 numeric values, which represent the state's population from 1770 through 1990, in 10-year increments. The next tab separator is followed by a single numeric value, which specifies the state's area in square miles.

The data file cannot contain blank lines or comments. Missing or extra data on a line causes an error.

Note: One tab (the default separator) separates each field. Do not insert multiple tabs to line up the fields visually; this generates blank fields. The order of the columns must match the format specified by the configuration file.

Any field in the data can also be a "?", indicating that the data is null (unknown). See Chapter 13, "Nulls in MineSet."

Data Types

The Map Visualizer supports integer, floating point number, and string data types, as well as arrays of these types. See Chapter 5, "Data and Configuration File Basics," for details about data types.

Fixed Arrays

With the Map Visualizer, you can use one- or two-dimensional arrays of fixed size. In a fixed-sized array, all entries of the given type have the same number of values. Arrays contain the data values across one or two independent variables, that is, those dimensions controlled by the sliders.

A variant of the enumerated array is the “null enumerated array.” This is a variant of the enumerated array with an additional entry at the beginning for null, which is represented by “?”.

Configuration File Overview

The configuration file format is flexible. You must separate words in it by spaces, and it is case-sensitive. Except for the include statement and text within quoted strings, spacing and line breaks are irrelevant.

Comments begin with a pound (#) symbol at the beginning of a line; anything after this symbol to the end of the line is ignored.

The configuration file’s structure and grammar are explained in the following sections.

As each section is encountered, a special configuration file (referred to as a *defaults file*) is also read in. The defaults file has the same name as the section. Defaults files contain options statements. These files are searched in the following order:

1. Windows searches for all the files in the directory in which MineSet is installed, under *\config\mapviz*.

   UNIX searches for the files as specified by the X-resource *Mapviz*configPath* in the *Mapviz* file, in the directory *X11/app-defaults,* for example in */usr/lib/MineSet*.
2. The *mapviz* directory. This directory contains system defaults.
3. The *~/.MineSet* directory (the tilde, ~, indicates your home directory). You can set up personal defaults in this directory.
4. The current directory, which lets you set up defaults for each directory.

Files with the same name can appear in more than one of the above-named directories; in this case, the order given is the one in which the directories are read. If the same option is found in multiple files, the last option read is used. Note that the appropriate section in the configuration file is read after all the defaults files; thus, options in the configuration file override those in the defaults files.

Configuration File Input Section

The first section of a data file is normally the **input** section. It defines the name and format of the data file. A typical IRIX **input** section might look like this:

```
input
   {
   file
      "/usr/lib/MineSet/mapviz/examples/population.usa.data";
   enum int Year from 1770 to 1990 by 10;
   string states;
   float population[enum Year] separator ' ';
   float sqMiles;
   }
```

This example specifies that the input data file is called *population.usa.data*, and that there are three tab-separated (the default) fields as follows:

- Type **string**.
- Fixed-length vector of type **float**, with each value separated by a space.
- Scalar value of type **float**.

When the **input** section is entered, the defaults file, *input.mapviz.options*, is read in.

- For Windows users this file is in the directory in which MineSet is installed, in *\config\mapviz\input.mapviz.options*.
- For UNIX users the file is in */usr/lib/MineSet/mapviz/input.mapviz.options*.

File Statements

The **file** statement names the data file to be read. This statement is required. Its syntax is:

```
file "filename";
```

The file name must be in double quotation marks. If it is a relative pathname (no leading slash), it is first sought in the directory containing the current configuration file. If **include** statements are present, this might not be the same as the initially loaded configuration file. If it is not found in the current configuration file's directory, the file is sought in the current directory.

Enum Statements

enum statements declare enumeration variables that index into array fields. The **enum** statement has three forms:

- `enum type name from value1 to value2 by increment;`

  This declares an **enum** with values starting at *value1* and incremented by *increment* until they reach or exceed *value2*. For example, the following statement declares age as an array dimension with the values 20, 30, 40, 50, 60, and 70:

  ```
  enum int age from 20 to 70 by 10;
  ```

  Type must be a number type (**int**, **float**, or **double**) or **date** (see "Dates" on page 89).

- `enum type name from value1 to value2 across numberOfValues;`

  This declares an **enum** with values ranging from *value1* to *value2*. The *numberOfValues* is an integer specifying the number of values. For example, the following statement declares age as an **enum** with the values 20, 30, 40, 50, 60, and 70:

  ```
  enum int age from 20 to 70 across 6;
  ```

  Type must be a number type (**int**, **float**, or **double**) or **dat**e (see "Dates" on page 89).

- `enum type name {value1, value2, ..., valueN};`

  Type can be any type or date (see "Dates" on page 89).

Dates

The **enum** statement includes special support for a date type that handles date and time values starting January 1, 1753. The date type is valid only within **enum** statements. A date **enum** statement can have the following types of syntax:

```
enum date "format" name from "value1" to "value2" across
        numberOfValues;
enum date "format" name { value1, value2, ..., valueN };
enum date "format" name from "value1" to "value2" by
        "increment";
```

The *format* string specifies the format of the values; it is useful for controlling how dates are displayed in the animation control panel. The syntax of the *format* string is similar to the scanf function in C. Various units of time are represented by special characters preceded by the percent symbol (%). For example:

```
enum date cq “Calendar Q%Q, %Y” from “Calendar Q1, 1980” to “Calendar
Q3, 1985” by “1 quarter”;
```

The “Calendar Q” in the *format* string matches the “Calendar Q” in *value1* and *value2*. The %Q in the *format* string indicates that the next number in *value1* and *value2* is the calendar quarter. The comma and space in the *format* string match the commas and spaces in the values. Finally, the %Y in the *format* string specifies that the year values are next.

Table 8-1 lists the characters that can follow the percent symbol and the units of time they represent.

Table 8-1 Characters That Can Follow the Percent Symbol in the Format String

Character	Time Unit	Precision
Y	year	4
Q	calendar quarter	1
M	month	2
N	month name	>= 3
D	day	2
h	hour	2
m	minute	2
s	second	2

With the exception of N, each character matches an integer of the specified precision. N matches 3 or more characters giving the English name of the month.

The from-to-by form of the **enum** statement includes an increment value. For dates, the increment is a quoted string containing an integer, an optional space, and one of the special characters in Table 8-1 or one of the symbols **year**, **quarter**, **month**, **day**, **hour**, **minute**, and **second**. The plural forms of these symbols are also accepted. Note that these symbols are not keywords, because they have special meaning only in the increment string. The following are examples of valid increments:

```
“1 year”
“7 days”
“4h”
```

Data Statements

The data statements declare the columns in the data file. You must declare the columns must in the order they appear in the data file. The format of most data statements is:

```
type name;
```

type is **int**, **float**, **double string**, **dataString**, **date**, or **fixedString**(n); *n* is an integer representing the width of the string; *name* is the variable name. Unlike in C, you can declare only one variable per statement.

Fixed Arrays

You can also declare fixed arrays by using simple numeric data declarations; however, if you also are going to declare a slider, you must use the **enum** declaration form. The declaration syntax is:

```
type name [ number ];
```

For example:

```
float revenue [50];
```

You can also override the separator by declaring it as:

```
type name [ number ] separator ’char’;
```

For example:

```
float revenue [50] separator ’:’;
```

If no separator is specified, the default separator (usually a tab) is used.

Fixed arrays can also be two-dimensional, such as the following which might be used for an array of prices for a set of 10 products over a 20-year period.

```
enum string products {“bread”,“milk”,”cheese”,”cereal”,
”apples”,”lettuce”,”juice”,”toothpaste”,”soap”,”eggs”};

enum year from 1985 to 1994 by 1;

float prices[enum products][enum year];
```

or

```
float prices[10][20];
```

Using the *prices* array, for example, if you specified in the Tool Manager that data was to be retrieved from the database in “wide” mode (with a bin for null values), the enumerated *products* are declared as:

```
float prices[null enum products][enum year];
```

The first column contains the prices for unknown products (products not in the enumerated list of ten known products) declared in the *enum string products* statement.

Input Options

The **input** section of a data file has several options. All **options** statements begin with the word **options** and have one or more comma-separated options.

- The **separator** option defines the separator between columns in the data file. The default separator is a tab. The syntax is:

  ```
  options separator ’char’;
  ```

  For example:

  ```
  options separator ’:’;
  ```

  Note: Arrays can override the separator.

- The **monitor** option allows a dynamic update of the data displayed. When you change the specified file (for example, through the UNIX *touch* command), the data file (not the configuration file) is reread. Although the data file could be used to trigger the updates, it is better to use a different file so that the data file is not read while it is being updated. The syntax of the monitor option is:

  ```
  options monitor "filename";
  options monitor "filename" timeout;
  ```

  filename is the file to watch, and the optional *timeout* specifies the number of seconds to wait after the file changes. If the user interacts with the application in any way during this timeout (via the mouse or keyboard), the timeout restarts. Updating the file can take a few seconds. If you specify a timeout, the chances of an update occurring while the user is interacting with the tool are minimized. This might delay the update. If you do not specify a timeout, the update occurs immediately.

  On Windows, the timeout value has no effect. In other words, the monitor update always occurs immediately.

  The file being monitored must exist at the start of the program. When this file is being updated, you must not remove and re-create it; instead, you should only update only its modify time (for example, through the *touch* command). If the file is deleted, subsequent updates are not shown.

  Suppose a program extractor extracts data from a database into a data file. If you want the program to update the data file every 10 minutes, the script you write might look like this:

  ```
  extractor > dataFile;          # create first data file
  touch trigger;                 # create the trigger file
  while (sleep 600)              # sleep 10 minutes
  do
  extractor > dataFile;          # create new data file
  touch trigger;                 # force a reread
  done &                  # this loop goes in the
  # background
  mapviz configFile;             # run mapviz
  kill $!                        # when mapviz exits, kill
                                 # the update loop
  ```

  The **monitor** option on UNIX systems can be used only if the file alteration monitor */usr/etc/fam* is installed (this can be found in the subsystem *desktop_eoe.sw.fam*).

The **input** section of configuration file might look like this:

```
input
{
   file "dataFile:
   #data declarations here
   options monitor "trigger" 15;
}
```

- The **backslash option** controls whether backslashes in the input data are treated specially or like other characters. The syntax is:

  ```
  options backslash off;
  options backslash on;
  ```

  The default is off. If backslash processing is on, separators in the input data preceded by backslashes are treated as regular characters rather than as separators. Also, within strings, standard C-style backslash processing is done.

Configuration File Expressions Section

The **expressions** section of a data file lets you define additional columns that are expressions of existing columns. For example, you can define one column as the sum of two other columns.

The following is a sample **expressions** section. This section assumes two existing fixed-length columns of type **double**: male and female. These represent spending by males and females on various goods over time (one independent dimension). Two columns are added: total represents the total dollars spent, and pctFemale represents the percentage of dollars spent by females.

```
expressions
{
double total[enum month] = male+female;
double pctFemale[enum month] = divide(female*100,total,50.0);
}
```

Note: The pctFemale calculation uses total, defined in the previous section. Also, the use of the divide function rather than the **/** operator. This results in 50% for the case in which no dollars are spent; using the **/** operator generates an error because it results in division by zero. (The divide function is described in Chapter 5, “Data and Configuration File Basics,” in “MineSet Expression Language” on page 42.)

The format of the **expressions** section is:

```
expressions
{
   expressionDeclaration;
   ...
}
```

expressionDeclaration has the following syntax:

```
type name = expression;
```

Because the **expressions** section has no options, no defaults file is read in for it.

Configuration File View Section

The **view** section of a data file describes how the graphic objects are displayed, including the mapping of heights, colors, labels, and so forth. The following is a sample **view** section:

```
view map
{
   map objects "usa.states.hierarchy";
   slider Year;
   height population;
   height legend label "Height: U.S. Population (1770-1990)";
   color density, scale 0 250 500 750 1000;
   color colors "white" "#ffc0c0" "#ff8080" "#ff4040" "red";
   color legend label "Color: Pop. Density" "0/sq-mile"
            "250/sq-mile" "500/sq-mile" "750/sq-mile"
            "1000/sq-mile";
   message "population %,.0f   %,.1f per sq mile",
   population, density;
   execute "xconfirm -t 'Population %,.0f'
                     -t 'averaging %,.1f per sq mile'
                     -t 'across %,.0f sq-miles' > /dev/null",
        population, density, sqMiles;
   }
```

The first words of the **view** section (before the opening brace) describe the type of view. The only view type supported is **view map**; therefore, these words must introduce the **view** section.

When entering the **view** section, the *viewMap.mapviz.options* defaults file is read in.

Note: There is no simple view defaults file, so you must use the full name *viewMap.mapviz.options*.

Title Statement

The **title** statement inserts a title string at the bottom of the main window. The syntax is:

```
title string;
```

string is a string enclosed by double quotation marks.

Map Statement

The **map** statement specifies how the graphical objects will be drawn in the main window. The **map** statement has three possible types of syntax: one required, the other two optional. The required syntax is:

```
map objects hierarchy_filename;
```

objects is a keyword, and *hierarchy_filename* is a filename enclosed in double quotation marks. This statement names the *.hierarchy* file that describes the 3D graphical objects that exhibit heights and colors.

The following **map** statements are optional:

- `map outlines hierarchy_filename;`

  This declares graphical objects that are drawn as flat lines on which the **map** objects are placed. See the samples provided in the *examples* directory, *population.usa.cities.mapviz*.

- `map level column_name;`

  This specifies an alternative level of the geographical hierarchy for initial display. For example, in the *examples* directory, the *population.usa.mapviz* file, the unstated default is:

  ```
  map level states;
  ```

The main window initially displays individual states. If, instead, the configuration file specifies the following, the main window initially displays the United States as two halves: East and West:

```
map level eastWest;
```

Slider Statement

The **slider** statement identifies a key to be used as a slider dimension. Its syntax is:

```
slider [enum] enumName;
```

enumName is the name of an enum variable declared in the **input** section. The **enum** keyword is optional.

There can be 0, 1, or 2 **slider** statements. The first **slider** statement applies to the horizontal slider. The second applies to the vertical slider. If there is no **slider** statement, the resulting display does not include animation.

No **slider** statement is required if height and color map to non-array variables. One **slider** statement can be included if height and color map to one-dimensional arrays. Two **slider** statements can be included if height and color map to either of the following:

- Two-dimensional arrays.
- One-dimensional arrays, where the dimensions are **enum** variable names that one of the sliders controls.

Height Statement

The **height** statement describes how the columns of data are mapped to the height of objects. It consists of a series of clauses separated by commas. The first clause normally contains the name of a column to be mapped to height (such as population in the example in “Configuration File View Section” on page 95). The column must be of a number type (**int**, **float**, or **double**), of which **float** is the most memory-efficient. If the column is a fixed-length array, the **view** section also must contain at least one, and no more than two, **slider** statements.

If no height column is specified, all bars are flat, and the remaining **height** clauses have no effect.

The **scale** clause lets you scale the height values. Normally, the height variable is mapped directly to the height of the graphical objects, so that the tallest object (with the largest numeric value) rises toward the top of the view window. With the optional **scale** clause, all values are multiplied by the scale. The **scale** clause syntax is:

```
scale float
```

The **legend** clause defines the meaning of the height mappings. Any string can be placed in the height legend. The **legend** clause has the following types of syntax:

- `legend off`

  This turns off the height legend (this is the default).

- `legend on`

  This turns on the height legend. The legend can be changed by using the legend label form, in which case **legend on** is unnecessary. The legend's default syntax is:

  `height:`*`varname`*

  varname is the name of the variable that is mapped to height.

- `legend label string`

  string is the name of the variable that is mapped to height. You can change the legend by using the **legend label** form. If **legend label** is used, **legend on** is unnecessary.

Color Statement

The **color** statement describes how values are mapped to colors. The format is similar to that of the **height** statement, consisting of several clauses that can be separated by commas or entered as multiple statements.

Color naming follows the conventions of the X Window System, except that the names must be in quotation marks. Examples of valid colors are "green," "Hot Pink," and "#77ff42." The last one is in the form "#rrggbb," in which the red, green, and blue components of the color are specified as hexadecimal values. Pure saturation is represented by ff, a lack of color by 00. For example, "#000000" is black, "#ffffff" is white, "#ff0000" is red, and "#00ffff" is cyan.

- Windows users should use only the "#ff0000" form, as only some of the named colors are supported (for example, white, black, gray, red, yellow, green, and so on.)
- UNIX users can find the color list file at */usr/lib/X11/rgb.txt*.

The **color** variable lets you specify a single column to be mapped to a color (as with height). The column must be a number type.

The **colors** clause specifies the colors to be used. The **colors** clause's syntax is:

```
colors "colorname" "colorname"...
```

This is the format for *colorname*. There are no commas between the colors. This is because commas are used to separate clauses in the **color** statement. A sample **colors** clause is:

```
colors "red" "gray" "blue"
```

Colors in the list are subsequently referred to by their index, starting at zero. In the above example, red is color 0, gray is color 1, and blue is color 2.

If there is no **colors** statement, colors are chosen randomly; however, if there is a colors statement, you must specify at least as many colors as will be mapped.

The **scale** clause allows assignment of values to a continuous range of colors. For example, when displaying a percentage, red can be assigned to 0%, gray to 50%, and blue to 100%. Intermediate values are interpolated; for example, 25% is pinkish, and 55% is a slightly bluish gray.

The syntax for the **scale** clause is:

```
scale float float ...
```

The first value is mapped to color 0, the second to color 1, and so forth. The **colors** statement must contain at least as many colors as will be mapped to the largest index.

Values in this statement must be in increasing order. Any value less than the first color is assigned the value of the first color. Any value greater than the last value is assigned the last color. Intermediate values are interpolated.

For example, assume the pctFemale column indicates what percentage of the group is female, and you want to map a group that is 100% female to red, 100% male to blue, and 50% each to gray. The **colors** statement for this is:

```
colors pctFemale, colors "blue" "gray" "red", scale 0 50 100;
```

The **buckets** clause is similar to the **scale** clause without interpolation. All values are rounded down to the highest value in the clause, and that exact color is used. Values less than the first value use the first color.

The syntax for the **buckets** clause is:

```
buckets float float ...
```

The syntax and assignment of colors is the same as for the **scale** clause.

If, in the pctFemale example, you used the **buckets** clause instead of the **scale** clause, the statement would be:

```
colors pctFemale, colors "blue" "gray" "red", buckets 0 50 100;
```

All values greater than or equal to 100 are colored red. Values greater than or equal to 50, but less than 100, are gray. All other values are blue.

The **normalize** clause controls a form of color normalization, analogous to height normalization. By default, color normalization is off. The syntax is:

```
normalize off;
normalize on;
```

When color normalization is on, the color scale (or buckets) list of values must range between 0 and 100. These color values then represent relative percentages of the range from the minimum to the maximum for a given viewed scene. For example, the following generates colors in the range of “white” to “red,” where “white” corresponds to the minimum “totalSales” and “red” corresponds to the maximum “totalSales” for the particular set of graphical objects being viewed:

```
color totalSales;legend off
color scale 0 100, colors “white” “red”, normalize on;
```

See the file *variations.articles.france.mapviz* for a more elaborate example.

Windows users can find the file in the directory in which MineSet is installed, under *Examples\mapviz\examples*, and UNIX users can find the file in */usr/lib/MineSet/mapviz/examples/variations.articles.france.mapviz*.

The **legend** clause creates a legend of the colors. By default, the color legend is off. The **legend** clause syntax can be any of the following:

```
legend off
legend on
legend "string" "string" ...
legend label "string"
legend "string" "string" ... label "string"
```

The **legend off** clause turns the legend off. The **legend on** clause turns the legend on. You can omit it if other **legend** statements are included. Specifying only **legend on** generates the default legend.

The default legend includes a single label to the left (with the name of the column that is mapped to color), and a list of colored labels on the right (with values obtained from the **scale** clause, the **buckets** clause, or from the keys). To override the strings in the colored labels, specify the strings as:

```
legend "string" "string
```

To override the label on the left, specify it following the word label. To eliminate this label, specify an empty string as follows:

```
legend ""
```

Message Statement

The **message** statement specifies the message displayed when an object is selected. The syntax is similar to the C **printf** statement. A sample **message** statement is:

```
message "%s: $%f, %.0f%% of target, %.0f%% of last year",
      product, sales, pctTarget, pctLastYear;
```

This could produce the following message:

```
furniture: $2425.37, 23% of target, 87% of last year
```

The formats must match the type of data being used:

- **Strings** must use %s.
- **Ints** must use integer formats (such as %d).
- **Floats** and **doubles** must use floating point formats (such as %f).

For details of the **printf** format on UNIX systems, see the printf (1) man page (type **`man printf`** at the shell prompt).

A special format type has been added to **printf**. If the percent sign is followed by a comma (for example, "%,f"), commas are inserted in the number for clarity. Currently, only the U.S. convention of d,ddd,ddd.dddd is supported, with the decimal point represented by a period, and commas separating every three places to the left of the decimal point. For example, the preceding format could be:

```
message "%s: $%,f, %,.0f%% of target, %,.0f%% of last year",
         product, sales, pctTarget, pctLastYear;
```

In this case, it would produce the following message:

```
furniture: $2,425.37, 23% of target, 87% of last year
```

The $, *, h, l, ll, L, and n **printf** format options are not supported.

All values, including the format string, are expressions. Therefore, if you had a pctFemale column, but wanted a more gender-neutral message, you could use:

```
message pctFemale>50?"%f%% females":"%f%% males",
         pctFemale>50?pctFemale:100-pctFemale;
```

If pctFemale is 70, the message "70% females" is displayed; if pctFemale is 30, the message "70% males" is displayed. In this case, you can also achieve the same result with a single format string:

```
message "%f%% %s", pctFemale>50?pctFemale:100-pctFemale,
         pctFemale>50?"females":"males";
```

If no message is specified, a default message containing the names and values of all the columns is used.

Execute Statement

The **execute** statement lets you execute a shell command by double-clicking an object. The syntax is similar to that of the **message** statement.

The following is a sample UNIX **execute** statement that uses *xconfirm* to show a window with information about the item. The command line entry is shown as three lines. In an actual file, this should be on a single line. Multi-line strings are not supported.

```
execute "xconfirm -t '%s' -t 'population %,.0f' -t '%,.0f per
    sq mile' -t '%,.0f sq-miles' > /dev/null", states,
    population, density, sqMiles;
```

This might produce a dialog with the following message:

```
CA
64 per sq mile
266,807 sq-miles
```

If there is no **execute** statement, double-clicking an object has the same effect as single-clicking it.

This sample **execute** statement will work on Windows as well as UNIX, because a simple *xconfirm* utility is provided with the installation on Windows.

Summary Statement

The **summary** statement specifies the initial setting of the Show Data Points pulldown menu option. The syntax is:

```
summary datapoints on;
summary datapoints off;
```

The **summary** statement is optional, and the default setting is **off**.

Hierarchy File

The hierarchy file defines the object hierarchy, allowing objects to be displayed at different levels of aggregation. It enables the drill up and drill down capabilities of the Map Visualizer. The hierarchy file is specified in the *.mapviz* configuration file with the `map object` *`hierarchy_filename`* statement (see “Configuration File View Section” on page 95 and “Map Statement” on page 96).

These are the first few lines of the *usa.states.hierarchy* file:

```
states        regions        eastWest        USA
usa.states.gfx        usa.states.gfx
     usa.states.gfx        usa.states.gfx
AL        E_S_CENTRAL        USA_E        USA_ALL
AR        W_S_CENTRAL        USA_W        USA_ALL
AZ        MOUNTAIN        USA_W        USA_ALL
CA        PACIFIC        USA_W        USA_ALL
CO        MOUNTAIN        USA_W        USA_ALL
CT        NEW_ENGLAND        USA_E        USA_ALL
DE        MID_ATLANTIC        USA_E        USA_ALL
```

This defines how states combine into regions, sectors, and into a single object encompassing all states.

The first record is a list of column names of the hierarchy; each name must be separated by a single tab ('\t') character. One of the column names must match a type **string** column in the **data file**, as declared in the configuration file's **input** section in "Configuration File Input Section" on page 88. In this example, the first column name, states, is also the name of a data column in the example *population.usa.mapviz*. The number of column names in this record must be the same as the number of columns of hierarchy data, beginning at the third record of the *.hierarchy* file. If there is only one column name (for example, *gfx_files/canada.provinces.hierarchy*), then there are only two records in the *.hierarchy* file.

The second record is a list of *.gfx* file pathnames, where each pathname is separated by a single tab ('\t') character. Each column name in the first record must have a matching *.gfx* file pathname.

If there is a single column name (and *.gfx* file pathname), then only these two records must be in the file. If there are multiple column names and pathnames, then starting at the third record in the *.hierarchy* file is an N-column table of keywords of graphical objects, where N is the number of column names in the first record. Looking at the sample file, the first column contains "states" keywords, the second contains "regions" keywords, the third contains the "eastWest" keywords, and the fourth contains the "USA" keyword. The matching *.gfx* files contain the positions and shapes of each column's graphical objects.

The third and remaining records in the hierarchy file are the hierarchy data. These records define how objects at one level correspond to objects at other levels.

.gfx File

The *.gfx* files define the geometry of each object used by the Map Visualizer when displaying the objects. Each *.gfx* file contains multiple records, one for each object being displayed. Each record contains the following:

- gfx keyword name
- gfx full name
- vertex pair count
- shape hint
- vertex pairs

The following steps explain how to build *.gfx* files on UNIX systems. On Windows, an application for generating your own *.gfx* files, including instructions for using the utility to create maps, is downloadable from the MineSet Web page: http://mineset.sgi.com.

1. Using a digitizing scanner, convert a geographical image into an RGB image file format. Note that the image itself is not used by the Map Visualizer; it is just used as a template for defining the graphical objects in step 5.
2. Launch the *i3dm* application in */usr/demos/bin/*. (If this application is not currently installed, it can be installed from the IRIX 5.3 or 6.2 distribution, in the subsystem *demos.sw.tools*.) This creates windows on your screen: a Menu window on the left, an Input window across the bottom, and four windows (labeled TOP, Pers, Front, and Right) on the right. All *i3dm* windows must remain displayed (not iconified) for *i3dm* to work.
3. Right-click the Front window to display options. Hold down the right mouse button, scroll to the Image Background option, and then scroll to the Load Image option. The Input window (at the bottom of your screen) prompts you for a name to apply to this image.
4. Enter the name of the RGB image file. The image appears in the Front window.

5. Delineate the shape of each object in the image by clicking significant points on the boundary of each object. Do this in a clockwise sequence for each object. Each identified point is called a "vertex" and is represented by numeric x- and y-axis values. These values are assigned by the *i3dm* application and exist in a relative frame of reference for that RGB image file. The follow these steps to delineate each object's shape:
   - Use the middle mouse button to drag the image in the Front window so that the object you will delineate is completely exposed. If this is not possible, see step 7.
   - Go to the Menu window, and right-click the Create pulldown menu.
   - Choose the Line option.
   - Start clicking vertices in the Front window. The more vertices you select, the more accurate the resulting graphical image will be.
   - Note the red line crosshairs as you move the cursor over the image. As you click each vertex, a small red box appears at that point. The box of the previous vertex changes to a small "x," and a yellow line connects the new vertex to the previous vertices. As you move clockwise around the object, stop selecting vertices immediately before you are about to close the shape (that is, before clicking the first vertex you selected when starting to delineate the object).
   - Go to the Menu window, and right-click the Attrib pulldown menu.
   - Scroll to the Name option. The Input window (at the bottom of your screen) prompts you for a name.
   - Enter a unique identifier for the object you have just delineated. Do not use spaces. The becomes the object's gfx keyword name. For example, in *population.usa.mapviz* the gfx column is specified as the first column in the data file. This first column contains strings such as "CA" and "NY." These are the keyword names for the states. These keyword names are the gfx keyword names in the associated gfx file.
   - Go to the Menu window, and click the right mouse button on Done.

6. Repeat step 5 for every other object in the same image. If the object adjoins a previously identified object, you must reuse common vertices by selecting them with the middle mouse button instead of with the left mouse button. Using the middle mouse button while the crosshairs are positioned close to a previously selected vertex ensures that the newly selected vertex will be identical to the previously selected one.

   Caution: If a graphical object is too large to fit into the Front window, you must select the vertices in sections. After all the objects are declared and the vertex information is written to an ASCII file, you must edit this output file to join the sections of each subdivided object.

7. When all objects are identified, save the recorded vertices in a file as follows:
   - Go to the Menu window and right-click the File pulldown menu.
   - Scroll down to the File i3dm format option and choose it. The Input window (at the bottom of your screen) prompts you for a filename.
   - Enter a filename, specifying the *.i3dm* suffix.
8. Exit the *i3dm* application as follows:
   - Go to the Menu window, and choose the File pulldown menu.
   - Scroll to the Exit option, and choose it.
9. Convert the *i3dm* format file into a *.gfx* file format by using the *convert.i3dm* utility, using the following syntax:

   ```
   /usr/lib/MineSet/mapviz/convert.i3dm inputFilename
   outputFilename.gfx
   ```

   For each object, the utility prompts you to do the following:
   - Confirm the object's keyword name (which defaults to the Attrib name you supplied in step 5, substep 8, when selecting the vertices).

- Declare the object's full name (which is the name the user sees in the Map Visualizer's Selection window when using the mouse to select a geographical object).
- Declare if the object has a concave shape that requires special handling.

Note: Declaring an object to be concave results in an accurate graphical display, but at the cost of slower performance. One strategy is to declare no objects as concave, examine the display to determine which objects are inaccurately drawn, and then manually edit the *.gfx* files for those objects, changing the string "convex" to "concave." Another strategy is to declare all objects as "concave" (assuming there are few objects), and then determine if the resulting performance is acceptable.

Creating Data and Configuration Files for the Scatter Visualizer

This chapter describes the data and configuration files that are required for the Scatter Visualizer. You can also generate these files automatically by using the Tool Manager. The subjects discussed are:

Read about the Scatter Visualizer in the *MineSet Enterprise Edition User's Guide* before using this chapter.

Data File

In its simplest form, the data file consists of a list of lines, each containing a set of fields, each separated by one tab. (Other separators are also allowed, but only one per file can separate each field. See "Input Options" on page 117.) All lines in the file must contain the same fields. The interpretation of the fields is specified by the configuration file, described in the next sections.

Sample data files are provided as part of the Scatter Visualizer package. For Windows these are in the directory in which MineSet is installed, under the *Examples\scatterviz* directory. For UNIX, they are in the */usr/lib/MineSet/scatterviz/examples/* directory. One such file is the store sales data file. In this file the first few lines appear as:

```
LIQUOR STORE    4300,4460,4800,4900,4700,4200,4250,4200
2700,2800,2750,3000,2900,2600,2500,2650
1600,1650,1900,1950,2000,2200,2300,2300
```

```
GROCERY STORE    700,900,600,800,877,755,800,600
3000,2900,3100,2800,2899,2950,3400,3300
10000,11000,9000,9800,9700,9650,9770,9700
```

In this file listing, each line consists of four fields, separated by tabs. The first field is a string that identifies a store type. The second field is an array of eight numbers, separated by commas, which might be sales of alcohol over an eight-day period. The third and fourth fields are also arrays of eight numbers that could represent sales of tobacco and food, respectively, over the same eight-day period.

The sample data file has other fields in the same format, but these are not shown. These additional fields correspond to sales of other products (see the configuration file *store-type.scatterviz* for a listing of all the fields). Windows users find this file in the directory in which MineSet is installed, under *\Examples\scatterviz*. UNIX users find the file in */usr/lib/MineSet/scatterviz/examples/store-type.scatterviz*.

The data file cannot contain blank lines or comments. Missing or extra data on a line causes an error.

Note: One tab (the default separator) separates each field. Do not insert multiple tabs to line up the fields visually; this generates blank fields. The order of the fields must match the format specified by the configuration file.

Data Types

The Scatter Visualizer supports integer, floating-point number, and string data types, as well as arrays of these types. See Chapter 5, “Data and Configuration File Basics,” for more information on data types.

Arrays

With the Scatter Visualizer, you can use fields that are one- or two-dimensional arrays of fixed size. In a fixed-sized array field, all entries of the given field are arrays with the same number of values. Arrays contain the data values across one or two independent variables (those dimensions controlled by the sliders). In the listing from the file *store-type.data*, the second, third, and fourth fields are arrays.

Null Values

Any field or array element in the data file can also have the value "?" (question mark), indicating an unknown or null value.

Configuration File Overview

The configuration file format is flexible. You must separate the words with spaces, and the file is case-sensitive. Except for the **include** statement and text within quoted strings, spacing and line breaks are irrelevant.

Defaults Files

As each section is encountered, a special configuration file (referred to as a *defaults file*) is also read in. Defaults files normally contain **options** statements. These files are read in the following order:

1. The *scatterviz* directory which usually contains system defaults. Windows users can find this in the directory in which MineSet was installed, under *\config\scatterviz*. UNIX users can find the directory in */usr/lib/MineSet/scatterviz*.
2. The *~/.MineSet* directory in which you can set up personal defaults (the tilde, ~, indicates your home directory).
3. The current directory, which lets you set up defaults for each directory.

Files with the same name can appear in more than one of the above-named directories; in this case, the order given is the one in which the directories are read. If the same option is found in multiple files, the last option read is used. Note that the appropriate section in the configuration file is read after all the defaults files; thus, options in the configuration file override those in the defaults files.

Configuration File Input Section

The first section of a configuration file is normally the **input** section. It defines the name and format of the data file. A typical **input** section might appear as follows:

```
input  {
   file "company.data";
   string company;
   slider int income from 20000 to 60000 by 10000;
   slider date “%N %Y” purchaseDate from “Jan 1990” to “Dec
   1992” by “1 month”;
   options array separator ‘,’;
   float lifeSales[income][purchaseDate];
   float autoSales[income][purchaseDate];
   float homeSales[income][purchaseDate];
   string location;
   }
```

This example states that the **input** file is called *company.data*, and that there are five fields: *company*, *lifeSales*, *autoSales*, *homeSales*, and *location*. The *company* and *location* fields are of type **string**, and the other three fields are two-dimensional arrays of type **float**. Two **slider** dimensions are declared:

- *income*, which is of type **int**, ranges from 20000 to 60000 in increments of 10000.
- *purchaseDate*, which is of type **date** and ranges from January 1990 to December 1992 in increments of one month.

The arrays *lifeSales*, *autoSales*, and *homeSales* contain values for each income and purchase date. Individual values within the arrays are separated by commas.

When the **input** section is entered, the defaults file *inputDefaults* is read in.

File Statements

The **file** statement names the data file to be read. This statement is required. Its form is as follows:

```
file "filename";
```

filename must be in double quotation marks. If it is a relative pathname (no leading slash), it is first sought in the directory containing the current configuration file. If **include** statements are present, this might not be the same as the initially loaded configuration file. If it is not found in the current configuration file's directory, the file is sought in the current directory.

Enumeration Statements

Enumeration statements declare enumerations, or **enums**, that index into array fields. The **enum** statement has three forms:

- `enum type name from value1 to value2 by increment;`

  This declares an **enum** with values starting at *value1* and incremented by *increment* until they reach or exceed *value2*. For example, the following statement declares age as an enum with the values 20, 30, 40, 50, 60, and 70:

  ```
  enum int age from 20 to 70 by 10;
  ```

  Type must be a number type (**int**, **float**, or **double**) or **date** (see "Dates" on page 114).

- `enum type name from value1 to value2 across numberOfValues;`

  This declares an **enum** with values ranging from *value1* to *value2*. The *numberOfValues* is an integer specifying the number of values. For example, the following statement declares age as an enum with the values 20, 30, 40, 50, 60, and 70:

  ```
  enum int age from 20 to 70 across 6;
  ```

  Type must be a number type (**int**, **float**, or **double**) or **date** (see "Dates" on page 114).

- `enum type name {value1, value2, ..., valueN};`

  Type can be any type or date (see "Dates").

Dates

The **enum** statement includes special support for a **date** type that handles date and time values from January 1, 1753 forward. The **date** type is valid only within **enum** statements. A **date enum** statement can have the following types syntax:

```
enum date “format” name from “value1” to “value2” across
        numberOfValues;
enum date “format” name { value1, value2, ..., valueN };
enum date “format” name from “value1” to “value2” by
        “increment”;
```

The *format* string specifies the format of the values; it is useful for controlling how dates are displayed in the animation control panel. The syntax of the *format* string is similar to the **scanf** function in C. Various units of time are represented by special characters preceded by the percent symbol (%). For example:

```
enum date cq “Calendar Q%Q, %Y” from “Calendar Q1, 1980” to “Calendar
Q3, 1985” by “1 quarter”;
```

The “Calendar Q” in the *format* string matches the “Calendar Q” in *value1* and *value2*. The %Q in the *format* string indicates that the next number in *value1* and *value2* is the calendar quarter. The comma and space in the *format* string match the commas and spaces in the values. Finally, the %Y in the *format* string specifies that the year values are next.

Table 9-1 lists the characters that can follow the percent symbol and the units of time they represent.

Table 9-1 Characters That Can Follow the Percent Symbol in the Format String

Character	Time Unit	Precision
Y	year	4
Q	calendar quarter	1
M	month	2
N	month name	>= 3
D	day	2
h	hour	2

Table 9-1 Characters That Can Follow the Percent Symbol in the Format String

Character	Time Unit	Precision
m	minute	2
s	second	2

With the exception of N, each character matches an integer of the specified precision. N matches 3 or more characters giving the English name of the month.

The from-to-by form of the **enum** statement includes an increment value. For dates, the increment is a quoted string containing an integer, an optional space, and one of the special characters in Table 9-1 or one of the symbols year, quarter, month, day, hour, minute, or second. The plural forms of these symbols are also accepted. These symbols are not keywords, because they have special meaning only in the increment string. The following are examples of valid increments:

```
“1 year”
“7 days”
“4h”
```

Data Statements

The data statements declare the fields in the data file. You must declare the fields in the order they appear in the data file. The format of most data statements is as follows:

```
type name;
```

type is **int**, **float**, **double string**, **dataString**, **date**, or **fixedString**(n), where *n* is an integer representing the width of the string; *name* is the variable name. Unlike in C, you can declare only one variable per statement.

You can also base a data field an enumeration. The syntax is as follows:

```
enum enumName name;
```

The field must contain **ints** corresponding to the values of the **enum**. You can declare the **enum** ageGroup as:

```
enum string ageGroup {"below 30", "30-39", "40-49", "50-59",
        "60 or above"};
```

In this case, you can declare the field age:

```
enum ageGroup age;
```

The field should contain **ints** between 0 and 4, where 0 is displayed as "below 30," 1 as "30-39," and so forth.

You can declare only one variable per statement.

Arrays

Arrays are also declared using data declarations. The declaration syntax for one-dimensional arrays is one of the following:

```
type name [ number ] ;
type name [ enumName ] ;
type name [ null enumName ] ;
```

For example:

```
float revenue [50];
```

The declaration syntax for two-dimensional arrays is one of the following:

```
type name [ number1 ][ number2 ] ;
type name [ enumName1 ][ enumName2 ] ;
type name [ null enumName1 ][ null enumName2 ] ;
```

For example:

```
float revenue [50][10];
```

When **enums** are used, the number of values in the array is taken from the declaration of the enum. For example, given the following statements, the array *clothingPurchases* must have six values, corresponding to the enum values 20, 30, 40, 50, 60, and 70:

```
enum int age from 20 to 70 by 10;
float clothingPurchases[age];
```

The keyword **null** indicates an extra value at the beginning of the array, corresponding to null. Thus, the following statements declare *clothingPurchases* as an array with seven values: the first value corresponding to null or unknown age values, and the remaining six values corresponding to age values 20, 30, 40, 50, 60, and 70:

```
enum int age from 20 to 70 by 10;
float clothingPurchases[null age];
```

You can override the separator between values in an array by declaring it as:

```
type name [ number ] separator 'char';
```

For example:

```
float revenue [50][10] separator ':';
```

If you do not specify a separator, the default separator (usually a tab) is used.

Input Options

All **options** statements begin with the word "options" and have one or more comma-separated options.

- The separator option defines the separator between fields in the data file. The default separator is a tab. The syntax is:

  ```
  options separator 'char';
  ```

  For example:

  ```
  options separator ':';
  ```

  Note: The separator is used also to separate values within arrays; however, arrays can override the separator.

- The **backslash** option controls whether backslashes in the input data are treated specially or like other characters. The syntax is:

  ```
  options backslash off;
  ```

  ```
  options backslash on;
  ```

  The default is off. If backslash processing is on, separators in the input data preceded by backslashes are treated as regular characters rather than separators. Within strings, this causes standard C-style backslash processing.

Configuration File Expressions Section

The **expressions** section of a configuration file lets you define additional fields that are expressions of existing fields. For example, you can define a new field as the sum of two other fields.

The format of the **expressions** section is:

```
expressions
{
expressionDeclaration;
...
}
```

expressionDeclaration has the following form:

```
type name = expression ;
```

The following is a sample **expression** section. This section assumes two existing array fields of type **double**: "male" and "female". These represent spending by males and females on various goods over time (one independent dimension). Two fields are added: "total" represents the total dollars spent, and "pctFemale" represents the percentage of dollars spent by females.

```
expressions
{
double total[36] = male+female;
double pctFemale[36] = divide (female*100, total, 50.0);
}
```

Note: The pctFemale calculation uses "total," defined in the previous statement. Also, note the use of the divide function rather than the **/** operator. Using the **/** operator generates an error due to an attempted division by zero.

The **expressions** section has no options; thus, no defaults file is read in for it.

Configuration File View Section

The **view** section of a configuration file describes how the data is displayed, including the mapping of sizes, colors, axes, and so on. The default values for these options are in *view.scatterviz.options*. The Windows options are in the directory in which MineSet is installed under *\Programs\scatterviz\view.scatterviz.options*. UNIX users can find the file in */usr/lib/MineSet/scatterviz/view.scatterviz.options*. The syntax is as follows:

```
view
{
viewStatement;
...
}
```

A sample **view** section is:

```
view {
   slider month;
   entity brand;
   axis male$, color “blue”;
   axis female$, color “red”;
   size total$, max 5;
   color pctFemale, scale 0 50 100, colors “blue” "gray"
   "red";
   message "brand %s, total sales %,.0f",brand, total$;
   }
```

When entering the **view** section, the *viewDefaults* file is read in.

Slider Statement

The **slider** statement identifies an **enum** to be used as a slider dimension. Its syntax is one of the following:

```
slider enumName;
slider null enumName;
```

The **enum** name is declared in the **input** section. If the keyword **null** is present, the slider includes a position at the beginning that corresponds to null or unknown values of the **enum**. You must declare arrays indexed by the slider to match the **null** in the **slider** statement.

There can be one, two or no **slider** statements. The first **slider** statement applies to the horizontal slider, the second applies to the vertical slider. If there is no **slider** statement, the resulting display does not include animation.

Entity Statement

The **entity** statement lets you specify a variable that uniquely identifies the entities in the display. The **entity** statement consists of a series of clauses, separated by commas:

```
entity clause1, clause2,...
```

Alternatively, the clauses can be given in separate **entity** statements.

Entity Variable

The first clause of the **entity** statement normally contains the name of the entity variable (for instance, *brand* in the example on page 119).

Label Clause

This clause defines how the entities are labeled. It has the following forms:

- `label off`

  This turns off the labels.

- label on

  This turns on the labels. The default labels use the entity variable as the label for each entity.

- `label variable`

  This turns on the labels and uses the given variable to label the entities. When this form is used, it is not necessary to specify `label on`.

Label Color Clause

This clause turns on the labels and specifies their color. It has the following form:

```
label color "colorname"
```

colorname is the name of a color in a special format. (Color naming is explained in "Color Statement" on page 123.) The default label color is gray.

Legend Clause

The **legend** clause explains what the entities are. Any string can be placed in the entity legend. The **legend** clause has the following forms:

- `legend off`

  This turns off the entity legend.

- `legend on`

  This turns on the entity legend (this is the default). The default legend is:

  `Entity: varname`

  varname is the name of the entity variable.

- `legend label "string"`

  This turns on the legend and explicitly sets the legend string. If this form is used, `legend on` is unnecessary.

Size Statement

The **size** statement describes how a field of data is mapped to the sizes of entities. The **size** statement consists of a series of clauses, separated by commas:

```
size clause1, clause2,...
```

Alternatively, the clauses can be given in separate **size** statements.

Size Variable

The first clause normally contains the name of a field to be mapped to size (for instance, *total$*, in the **view** example in "Configuration File View Section" on page 119). The field must be of a number type (**int**, **float**, or **double**), of which **float** is the most efficient. The field can be an array that is indexed by slider dimensions. If no size field is specified, all entities are the same size.

Max Clause

Normally, the size variable is mapped to the size of the entities, so that the biggest entity has a size of 5. You can change this size by specifying a different value. If there is no **size** variable, the default maximum size is 5. The **max** clause has the form:

```
max float
```

Scale Clause

Instead of the **max** clause, you can use the **scale** clause to scale size values; all values are multiplied by the scale. The **scale** clause's syntax is:

```
scale float
```

Legend Clause

The **legend** clause defines the meaning of the size mappings. You can place any string in the size legend. The **legend** clause has the following forms:

- `legend off`

  This turns off the size legend.

- `legend on`

  This turns on the size legend (this is the default). The default legend is:

  `size:`*`varname`*

  varname is the name of the variable that is mapped to size.

- `legend label "string"`

  This turns on the legend and explicitly sets the legend string. If you use this form, `legend on` is unnecessary.

Color Statement

The **color** statement describes how values are mapped to colors. The format is similar to the **size** statement, consisting of several clauses that you can separate with commas, or enter as multiple statements. The syntax is:

```
color clause1, clause2,...
```

Color Naming

Color names follow the conventions of the X Window system, except that the names must be in quotes. Examples of valid colors are “green,” “hot pink,” and “#77ff42.” The latter is in the form “#rrggbb,”, in which the red, green, and blue components of the color are specified in hexadecimal value. Pure saturation is represented by ff, a lack of color by 00. For example, “#000000” is black, “#ffffff” is white, “#ff0000” is red, and “#00ffff” is cyan.

- Windows users should only use the “#ff0000”form, as only some of the named colors are supported (for example, white, black, gray, red, yellow, green, and so on).
- UNIX users can find the color list file at */usr/lib/X11/rgb.txt*.

Color Variable

As with size, you can specify a single field to be mapped to an entity color. The field can be an array that is indexed by slider dimensions. If the field is an array, it must be a number type. If the field is a number type, you can use the scale and buckets clauses described below to map a range of colors to the values of the field. If the field is not a number type, it is sorted, and each unique value is assigned a color.

Colors Clause

The **colors** clause specifies the colors to be used. The syntax is:

```
colors "colorname" "colorname"...
```

The format for *colorname* is described in “Color Naming” on page 123. There are no commas between the colors, because commas are used to separate clauses in the color statement. A sample **colors** clause is:

```
colors "red" "gray" "blue"
```

Colors in the list are subsequently referred to by their index, starting at zero. In the above example, red is color 0, gray is color 1, and blue is color 2.

If there is no **colors** statement, colors are chosen randomly. If there is a **colors** statement, at least as many colors must be specified as will be mapped.

The scale Clause

The **scale** clause allows assignment of values to a continuous range of colors. For example, when percentage is displayed, you can assign red to 0%, gray to 50%, and blue to 100%. Intermediate values are interpolated; for example 25% is pinkish, and 55% is a slightly bluish gray.

The syntax for the **scale** clause is:

```
scale float float ...
```

The first value is mapped to color 0, the second to color 1, and so forth. The colors statement must contain at least as many colors as will be mapped to the largest index.

Values in this statement must be in increasing order. Any value less than the first color is assigned the value of the first color. Any value greater than the last value is assigned the last color. Intermediate values are interpolated.

For example, assume the pctFemale field indicates what percentage of the group is female, and you want to map a group that is 100% female to red, 100% male to blue, and 50% each to gray. The colors statement for this is:

```
colors pctFemale, colors "blue" "gray" "red", scale 0 50 100;
```

Use the **scale** clause only in conjunction with a numeric color variable.

Buckets Clause

The **buckets** clause is similar to the **scale** clause without interpolation. All values are rounded down to the highest value in the clause, and that exact color is used. Values less than the first value use the first color.

The syntax for the **buckets** clause is:

```
buckets float float ...
```

The syntax and assignment of colors is the same as for the **scale** clause.

If, in the previous example, you used the **buckets** clause instead of the **scale** clause, the statement would be:

```
colors pctFemale, colors "blue" "gray" "red", buckets 0 50 100;
```

All values greater than or equal to 100 are colored red. Values greater than, or equal to, 50 but less than 100, are gray. All other values are then blue.

Use the **buckets** clause only with a numeric color variable.

Legend Clause

The **legend** clause creates a legend of the colors. The legend **clause** syntax can be any of the following:

```
legend off
legend on
legend "string" "string" ...
legend label "string"
```

The `legend off` clause turns the off legend. The `legend on` clause turns on the legend. You can omit it if other **legend** statements are included. Specifying only `legend on` generates the default legend.

The default legend includes a single label to the left (with the name of the field that is mapped to color), and a list of colored labels on the right (with values obtained from the scale clause, the buckets clause, or from the field). To override the strings in the colored labels, specify the strings as:

```
legend "string" "string"
```

To override the label on the left, specify it following the word `label`. To eliminate this label, specify an empty string; as follows:

```
legend label ""
```

Axis Statement

The **axis** statement causes a variable to be used as an axis in the 3D landscape. The variable's values determine where the entities are positioned on the axis. There can be up to three **axis** statements. Like the **size** and **color** statements, the **axis** statement contains a series of comma-separated clauses, but you must specify all of them in a single statement.

```
axis clause1, clause2,...
```

The Axis Variable

As with size and color, you can specify a field to be used as an axis. The field can be an array that is indexed by slider dimensions. If the field is an array, it must be of numeric type. If the field is non-numeric, it is sorted, and each unique value is assigned a position along the axis.

Label Clause

The **label** clause has the following form:

```
label "string"
```

The string is used to label the axis. It appears in the landscape, at the end of the axis line. The default label is the name of the axis variable.

Max Clause

Normally, the axis variable is mapped directly to the position of the entities along the axis. The **max** clause lets you normalize the values of the axis variable, so that the maximum value is mapped to the specified max. The **max** clause's syntax is:

```
max float
```

Scale Clause

Instead of using the **max** clause to affect position values, you can use the **scale** clause to scale the values. All values are multiplied by the scale. The **scale** clause syntax is:

```
scale float
```

Color Clause

The **color** clause specifies the color used for the axis line and label. It has the following form:

```
color "colorname"
```

Extend Clause

The **extend** clause specifies whether the axis should be extended automatically to include the value zero. It has the following form:

```
extend on
extend off
```

Orderby Clause

The **orderby** clause forces a particular order by string values on an axis. By default, the strings are arranged to correspond with the **color** variable. The only alternative ordering offered currently is alphabetical (`orderby alpha`).

Summary Statement

The **summary** statement specifies a summation to be calculated over all the entities. You can use the summary to color the drawing window in the animation control panel. Like the **size** and **color** statements, the **summary** statement has several clauses that you can specify in one statement, separated by commas, or in separate statements as follows:

```
summary clause1, clause2,...
```

Summary Variable

You can specify the variable to be used in the summary. This variable must be of numeric type. Typically, the **summary** variable is an array indexed by slider dimensions, so that the summary value varies across the slider dimensions.

Color Clause

The **color** clause specifies the color used to display the summary values in the drawing window. It has the following form:

```
color “colorname”
```

Various shades of the color, from white to the specified color, are used to represent summary values. The minimum summary value is mapped to white, while the maximum summary value is mapped to the specified color. The default summary color is red.

The Legend Clause

The **legend** clause creates a legend of the summary colors. The **legend** clause syntax can be any of the following:

```
legend off
legend on
legend label "string"
```

The `legend off` clause turns off the legend. The `legend on` clause turns on the legend. You can omit it if other **legend** statements are included. Specifying only `legend on` generates the default legend.

The legend includes a single label to the left (which defaults to the aggregation function and variable used in the summary), and two colored labels on the right (with the minimum and maximum summary values). To override the label on the left, specify it following the word `label`. To eliminate this label, specify an empty string as follows:

```
legend label ""
```

Drillthrough Statement

The **drillthrough** statement specifies a column to be used to define the drill through expression. This column must be string valued, and should specify an expression that defines the object when selected (assuming each row defines an object).

The syntax for this statement is of the following form:

```
drillthrough stringColumnName;
```

Drill through mapping is used in the *.scatterviz* file generated by the associations mining algorithm. It is used here because the table in the datafile contains rules and properties of the rules, not the columns in the original dataset.

The Selections > Drill Through Columns menu in the Scatter Visualizer has no effect when a specific drill through column has been mapped.

Message Statement

The **message** statement specifies the message displayed when an entity is selected. The syntax is similar to that of the C **printf** statement. A sample message statement is:

```
message "%s: $%f, %.0f%% of target, %.0f%% of last year",
         product, sales, pctTarget, pctLastYear;
```

This could produce the following message:

```
furniture: $2425.37, 23% of target, 87% of last year
```

The formats must match the type of data being used:

- **Strings** must use %.
- **Ints** must use integer formats (such as %d).
- **Floats** and **doubles** must use floating point formats (such as %f).

For details of the **printf** format, UNIX users can see the printf (1) reference (man) page (type **`man printf`** at the shell prompt).

A special format type has been added to **printf**. If the percent sign is followed by a comma (for example, "%,f"), commas are inserted in the number for clarity. Only the U.S. convention of d,ddd,ddd.dddd is supported, with the decimal point represented by a period, and commas separating every three places to the left of the decimal point. For example, the format could be:

```
message "%s: $%,f, %,.0f%% of target, %,.0f%% of last year",
         product, sales, pctTarget, pctLastYear;
```

In this case, it would produce this message:

```
furniture: $2,425.37, 23% of target, 87% of last year
```

The $, *, h, l, ll, L, and n **printf** format options are not supported.

All values, including the format string, are expressions. Therefore, if you had a pctFemale field, but wanted a more gender-neutral message, you could use the following:

```
message pctFemale>50?"%f%% females":"%f%% males",
        pctFemale>50?pctFemale:100-pctFemale;
```

If pctFemale is 70, the message "70% females" is displayed; if pctFemale is 30, the message "70% males" is displayed. In this case, you can also achieve the same result with a single format string:

```
message "%f%% %s", pctFemale>50?pctFemale:100-pctFemale,
        pctFemale>50?"females":"males";
```

If you do not specify a message, a default message containing the names and values of all the fields is used.

Execute Statement

The **execute** statement lets you execute a shell command by double-clicking an object. The syntax is similar to that of the **message** statement.

The following is a sample **execute** statement you can use on UNIX that uses *xconfirm* to show a window with information about the item. The command line (string) is shown as three lines. In an actual file, this should be on a single line. Multi-line strings are not supported.

```
execute "xconfirm -t '%s' -t 'population %,.0f' -t '%,.0f per
     sq mile' -t '%,.0f sq-miles' > /dev/null", states, population,
     density, sqMiles;
```

This might produce a the following message:

```
CA
64 per sq mile
266,807 sq-miles
```

The following is a sample Windows **execute** statement (a simple *xconfirm* utility is provided with the installation):

```
execute "xconfirm -t '%s has a population of %,.0f' -t 'averaging %,
     .1f per sq mile' -t 'across %,.0f sq-miles'",provinces,
     population, density, sqMiles;
```

If there is no **execute** statement, double-clicking an object has the same effect as single-clicking it.

Filter Statement

The **filter** statement specifies that only entities meeting certain filter criteria are displayed initially. The filter criteria are in the form of expressions whose values must all be true or nonzero for an entity to be displayed (expressions are described Chapter 5, "Data and Configuration File Basics," in "MineSet Expression Language" on page 42).

The syntax of the **filter** statement is:

```
filter expression, expression,...
```

For example, the following statement specifies that only records from California or Washington, with sales greater than 9000 and a *pctTarget* value greater than or equal to 90, should be displayed initially:

```
filter state == "CA" || state == "WA", sales > 9000, pctTarget >= 90;
```

After the Scatter Visualizer is invoked, you can change or remove the filter criteria interactively by using the filter panel.

View Options

The **view** section of the configuration file has several options for controlling parameters of the display. These options can appear in a single **options** statement, separated by commas, or in separate **options** statements. The syntax of the **options** statement is:

```
options option, option,...
```

The following options are available:

- `entity label size float`

  Controls the size of the entity labels.

- `axis label size float`

  Controls the size of the axis labels.

- `hide entity label distance float`

  Controls the distance at which entity labels become invisible. Smaller distances might improve performance, but the labels disappear more quickly.

- `grid color "colorname"`

  Controls the color of the grid.

- `grid size float float float`

  Controls the spacing between grid lines. It applies the three values to grid lines along the x, y, and z axes, respectively.

- `entity shape shapeName`

  Specifies the shape used to display entities. *shapeName* can be "cube," "bar," or "diamond."

- `background color "colorname"`

  Specifies the initial background color.

- `orientation top right front float float float`

  Specifies the initial orientation of the scene. You may use three floating point values to specify an arbitrary orientation vector.

- `perspective on off`

  Specifies whether or not to use perspective initially. The default is to use it.

- `upvector float float float`

  Specifies a camera upvector for defining a frame of reference. The default is 0.10.1. This option combined with the orientation option allows any initial viewing orientation.

Creating Data and Configuration Files for the Splat Visualizer

This chapter describes the data and configuration files that are required for the Splat Visualizer. You can also generate these files automatically by using the Tool Manager. The subjects discussed are:

Read the about the Splat Visualizer in the MineSet Enterprise Edition User's Guide before using this chapter.

Data File

In its simplest form, the data file consists of a list of lines, each containing a set of fields, each separated by one tab. (Other separators are also allowed, but only one per file can separate each field. See "Input Options" on page 139.) All lines must contain the same fields. The interpretation of the fields is specified by the configuration file, described in the next sections. These sections use examples from the *adultJobs* data file provided as part of the Splat Visualizer package. Windows users can find the file in the directory in which MineSet is installed, under *Examples\splatviz\examples*. UNIX users can find the file in */usr/lib/MineSet/splatviz/examples*.

The first few lines of the input file appear as:

```
Bachelors   Adm-clerical     3  3    51189.4869565217     115
Bachelors   Exec-managerial  2  5    70722.6271186441     59
Bachelors   Adm-clerical     2  3    37876.328358209      134
Bachelors   Exec-managerial  3  0    34436.8              5
Bachelors   Tech-support     1  2    37583.66667          3
```

```
Bachelors   Tech-support     1   3    13711.33333          3
Bachelors   Tech-support     1   4    29878.74193          31
```

In this sample file listing, each line consists of six fields, separated by tabs. The first field is a string that identifies level of education. The second field is a string which identifies occupation. The third field identifies the age bin. The fourth field identifies the number of hours per week worked bin. The fifth field quantifies the average gross income. The sixth field is the weight of records in the aggregate (i.e. the record count, unless record weighting has been used).

This data file was derived from *adult94.data* by performing Tool Manager operations (specifically binning and aggregation). Windows users can find the file in the directory in which MineSet was installed, under *\Examples\data\adult94.data*. UNIX users can find the file in */usr/lib/MineSet/data/adult94.data*.

The data file cannot contain blank lines or comments. Missing or extra data on a line causes an error.

Note: One tab (the default separator) separates each field. Do not insert multiple tabs to line up the fields visually; this generates blank fields. The order of the fields must match the format specified by the configuration file.

Data Types

The Splat Visualizer supports data of types the int, float, double, string, dataString, fixed string, and date. See Chapter 5, "Data and Configuration File Basics," for more information about data types.

Null Values

Any field element in the data file can also have the value "?" (question mark), indicating an unknown or null value.

Configuration File Overview

The configuration file format is flexible. You must separate words in it with spaces, and the file is case-sensitive. Except for the **include** statement and text within quoted strings, spacing and line breaks are irrelevant.

Defaults Files

As each section is encountered, a special configuration file (referred to as a *defaults file*) is also read in. Defaults files normally contain options statements. These files are read in the following order:

1. The *splatviz* directory which usually contains system defaults. Windows users can find this in the directory in which MineSet is installed, under *\config\splatviz*. UNIX users can find the directory in */usr/lib/MineSet/splatviz*.
2. The *~/.MineSet* directory in which you can set up personal defaults (the tilde, ~], indicates your home directory).
3. The current directory, which lets you set up defaults for each directory.

Files with the same name can appear in more than one of the above-named directories; in this case, the order given is the one in which the directories are read. If the same option is found in multiple files, the last option read is used. Note that the appropriate section in the configuration file is read after all the defaults files; thus, options in the configuration file override those in the defaults files.

Configuration File Input Section

The first section of a configuration file is normally the **input** section. It defines the name and format of the data file. A typical **input** section might appear as follows:

```
input {
     file "adultJobs.data";
     enum string `age_bin_k` {"- 20", "20-30", "30-40",
"40-50", "50-60", "60-70", "70+"};
     enum string `hours_per_week_bin_k` {"- 20", "20-25", "25-30",
"30-35", "35-40", "40-45", "45-50", "50-55", "55-60", "60-65", "65-70",
"70+"};
     string `education`;
     string `occupation`;
```

```
    enum `age_bin_k` `age_bin`;
    enum `hours_per_week_bin_k` `hours_per_week_bin`;
    double `avg_gross_income`;
    int `count_gross_income`;
}
```

This example states that the input file is called *adultJobs.data*, and that there are six fields: education, occupation, age_bin, hours_per_week_bin, avg_gross_income, and count_gross_income. The education and occupation fields are of type **string**. The *a*ge_bin and hours_per_week_bin are of type **enum**, where the values of these **enums** are defined by age_bin_k and hours_per_week_bin_k respectively. The column avg_gross_income is of type **double** and the field count_gross_income is of type **int**.

When the **input** section is entered, the defaults file *inputDefaults* is read in.

File Statements

The **file** statement names the data file to be read. This statement is required. Its form is as follows:

```
file "filename";
```

filename must be in double quotation marks. If it is a relative pathname (no leading slash), it is first sought in the directory containing the current configuration file. If include statements are present, this might not be the same as the initially loaded configuration file. If it is not found in the current configuration file's directory, the file is sought in the current directory.

Enumeration Statements

Enumeration statements declare enumerations, or **enums**. The **enum** statement has three forms.

- `enum type name from value1 to value2 by increment;`

  This declares an **enum** with values starting at *value1* and incremented by *increment* until they reach or exceed *value2*. For example, the following statement declares age as an enum with the values 20, 30, 40, 50, 60, and 70:

  ```
  enum int age from 20 to 70 by 10;
  ```

  Type must be a number type (**int**, **float**, or **double**) or **date** (see "Dates" on page 137).

- `enum type name from value1 to value2 across numberOfValues;`

  This declares an **enum** with values ranging from *value1* to *value2*. The *numberOfValues* is an integer specifying the number of values. For example, the following statement declares age as an **enum** with the values 20, 30, 40, 50, 60, and 70:

  ```
  enum int age from 20 to 70 across 6;
  ```

  Type must be a number type (**int**, **float**, or **double**) or **date** (see "Dates").

- `enum type name {value1, value2, ..., valueN};`

  This explicitly lists the enum values.

  Type can be any type or date (see "Dates").

Dates

The **enum** statement includes special support for a date type that handles date and time values from January 1, 1753 forward. The **date** type is valid only within **enum** statements. A **date enum** statement can have any if the following types of syntax:

```
enum date "format" name from "value1" to "value2" across
        numberOfValues;
enum date "format" name { value1, value2, ..., valueN };
enum date "format" name from "value1" to "value2" by
        "increment";
```

The *format* string specifies the format of the values; it is useful for controlling how dates are displayed in the animation control panel. The syntax of the *format* string is similar to the **scanf** function in C. Various units of time are represented by special characters preceded by the percent symbol (%). For example:

```
enum date cq "Calendar Q%Q, %Y" from "Calendar Q1, 1980" to "Calendar
Q3, 1985" by "1 quarter";
```

The "Calendar Q" in the *format* string matches the "Calendar Q" in *value1* and *value2*. The %Q in the *format* string indicates that the next number in *value1* and *value2* is the calendar quarter. The comma and space in the *format* string match the commas and spaces in the values. Finally, the %Y in the *format* string specifies that the year values are next.

Table 10-1 lists the characters that can follow the percent symbol and the units of time they represent.

Table 10-1 Characters That Can Follow the Percent Symbol in the Format String

Character	Time Unit	Precision
Y	year	4
Q	calendar quarter	1
M	month	2
N	month name	>= 3
D	day	2
h	hour	2
m	minute	2
s	second	2

With the exception of N, each character matches an integer of the specified precision. N matches 3 or more characters giving the English name of the month.

The from-to-by form of the **enum** statement includes an increment value. For dates, the increment is a quoted string containing an integer, an optional space, and one of the special characters in Table 10-1 or one of the symbols year, quarter, month, day, hour, minute, or second. The plural forms of these symbols are also accepted. These symbols are not keywords, because they have special meaning only in the increment string. The following are examples of valid increments:

```
“1 year”
“7 days”
“4h”
```

Data Statements

The data statements declare the fields in the data file. You must declare the fields in the order they appear in the data file. The format of most data statements is:

```
type name;
```

type is **int**, **float**, **double string**, **dataString**, **date**, or **fixedString**(n), where *n* is an integer representing the width of the string; *name* is the variable name. Unlike in C, you can declare only one variable per statement.

You can also base a data field on an enumeration. The syntax is:

```
enum enumName name;
```

The field must contain **ints** corresponding to the values of the **enums**. For example, you could declare the **enum** ageGroup as follows:

```
enum string ageGroup {"below 30", "30-39", "40-49", "50-59",
        "60 or above"};
```

In this case, you can declare the field age as follows:

```
enum ageGroup age;
```

The field should contain **ints** between 0 and 4, where 0 is displayed as "below 30," 1 as "30-39," and so forth.

You can declare only one variable per statement.

Input Options

All **options** statements begin with the word "options" and have one or more comma-separated options.

- The **separator** option defines the separator between fields in the data file. The default separator is a tab. The syntax is:

  ```
  options separator 'char';
  ```

  For example:

  ```
  options separator ':';
  ```

- The **backslash** option controls whether backslashes in the input data are treated specially or like other characters. The syntax is:

  ```
  options backslash off;
  ```

  ```
  options backslash on;
  ```

  The default is off. If backslash processing is on, separators in the input data preceded by backslashes are treated as regular characters rather than as separators. Within strings, this causes standard C-style backslash processing.

Configuration File View Section

The **view** section of a configuration file describes how the data is displayed, including the mapping of sizes, colors, axes, and so on. The default values for these options are in *view.splatviz.options*. The Windows options are in the directory in which MineSet is installed, under *config\scatterviz\view.splatviz.options*. The UNIX options are in */usr/lib/MineSet/splatviz/view.splatviz.options*. The syntax is:

```
view
{
viewStatement;
...
}
```

A sample **view** section is as follows:

```
view {
   slider `age_bin`;
   opacity `count_gross_income`;
   color `avg_gross_income`;
   axis `education`, color "grey";
   axis `occupation`, color "grey";
   axis `hours_per_week_bin`, max 100, color "grey";
   options grid size 0 0 0;
   summary `count_gross_income`, color "red";
}
```

When entering the **view** section, the *viewDefaults* file is read in.

Slider Statement

The **slider** statement identifies an **enum** column to be used as a slider dimension. Its syntax is one of the following:

```
slider columnName;
```

Declare the *columnName* name in the input section. If this column contains nulls, the slider includes a beginning position corresponding to those null values.

There can be one, two, or no **slider** statements. The first **slider** statement applies to the horizontal slider, and the second applies to the vertical slider. If there is no **slider** statement, the resulting display does not include animation.

Opacity Statement

In the Splat Visualizer, the opacity is based on counts or, more generally, record weights. If you map a column to this requirement, it is used to weight each record (rather than using 1) when computing a value for the opacity. Therefore, if you had a column with values for population, density, or the result of a count aggregation, you might want to map this column to the opacity (weight) requirement. If you had no such column, you can leave the requirement unmapped, and a column of 1's is used by default.

The **opacity** statement describes how a field of data is mapped to the opacity of the splats. The **opacity** statement consists of a series of clauses, separated by commas:

```
opacity clause1, clause2,...
```

Alternatively, you can put the clauses in separate opacity statements.

Opacity Variable

The first clause normally contains the name of a field to be mapped to opacity. The field must be of a number type, **int**, **float**, or **double**, of which **float** is the most efficient.

Max Clause

The **max** clause allows you to alter the initial opacity setting for the scene. The most opaque splat in the scene will match the value specified in this **max** clause. The default is 1. The **max** clause has the following form:

```
max float
```

Legend Clause

The **legend** clause defines the meaning of the opacity mapping. The **legend** clause has the following forms:

- `legend off`

  This turns off the opacity legend.

- `legend on`

  This turns on the opacity legend (this is the default). The default legend is:

  `opacity:count`

  count is a column that the tool has created by counting the number of records in each aggregate. If a column is mapped to opacity, the name of this column, prepended with "sum_", is shown in the legend. This new column is computed by sum aggregating the column mapped to the opacity requirement.

- `legend label "string"`

  This turns on the legend and explicitly sets the legend string. If you use this form, `legend on` is unnecessary.

Color Statement

The **color** statement describes how values are mapped to colors. The format is similar to the **opacity** statement, consisting of several clauses that you can separate with commas, or enter as multiple statements. The syntax is:

```
color clause1, clause2,...
```

Color Naming

Color names follow the conventions of the X Window System, except that the names must be in quotes. Examples of valid colors are "green," "hot pink," and "#77ff42." The latter is in the form "#rrggbb," in which the red, green, and blue components of the color are specified in hexadecimal value. Pure saturation is represented by ff, a lack of color by 00. For example, "#000000" is black, "#ffffff" is white, "#ff0000" is red, and "#00ffff" is cyan.

- Windows users should only use the "#ff0000" form, as only some of the named colors are supported (for example, white, black, gray, red, yellow, green, and so on.)
- UNIX users can find the color list file in */usr/lib/X11/rgb.txt*.

Color Variable

As with opacity, you can also specify a column to be mapped to splat color. If the column is a number type, you can use the **scale** and **buckets** clauses described in the following paragraphs to map a range of colors to the values of the field.

Colors Clause

The **colors** clause specifies the colors to be used. The syntax is:

```
colors "colorname" "colorname"...
```

The format for *colorname* is described in "Color Naming" on page 142. There are no commas between the colors, because commas are used to separate clauses in the color statement. A sample **colors** clause is:

```
colors "red" "gray" "blue"
```

Colors in the list are subsequently referred to by their index, starting at zero. In the above example, red is color 0, gray is color 1, and blue is color 2.

If there is no **colors** statement, colors are chosen randomly. If there is a **colors** statement, at least as many colors must be specified as will be mapped.

Scale Clause

The **scale** clause allows assignment of values to a continuous range of colors. For example, when a percentage is displayed, red can be assigned to 0%, gray to 50%, and blue to 100%. Intermediate values are interpolated; for example 25% is pinkish, and 55% is a slightly bluish gray.

The syntax for the **scale** clause is:

```
scale float float ...
```

The first value is mapped to color 0, the second to color 1, and so forth. The **colors** statement must contain at least as many colors as will be mapped to the largest index.

Values in this statement must be in increasing order. Any value less than the first color is assigned the value of the first color. Any value greater than the last value is assigned the last color. Intermediate values are interpolated.

For example, assume the pctFemale field indicates what percentage of the group is female, and you want to map a group that is 100% female to red, 100% male to blue, and 50% each to gray. The colors statement for this is:

```
colors pctFemale, colors "blue" "gray" "red", scale 0 50 100;
```

Use the **scale** clause only in conjunction with a numeric color variable.

Buckets Clause

The **buckets** clause is similar to the **scale** clause without interpolation. All values are rounded down to the highest value in the clause, and that exact color is used. Values less than the first value use the first color.

The syntax for the **buckets** clause is:

```
buckets float float ...
```

The syntax and assignment of colors is the same as for the **scale** clause.

If, in the above example, you used the **buckets** clause instead of the **scale** clause, the statement would be:

```
colors pctFemale, colors "blue" "gray" "red", buckets 0 50 100;
```

All values greater than or equal to 100 are colored red. Values greater than, or equal to, 50, but less than 100, are gray. All other values are then blue.

Use the **buckets** clause only with a numeric color variable.

Legend Clause

The **legend** clause creates a legend of the colors. The **legend** clause syntax can be any of the following:

```
legend off
legend on
legend "string" "string" ...
legend label "string"
```

The **legend off** clause turns off the legend. The **legend on** clause turns on the legend. You can omit it if you include other **legend** statements. Specifying only **legend on** generates the default legend.

The default legend includes a single label to the left (with the name of the field that is mapped to color), and a list of colored labels on the right (with values obtained from the **scale** clause, **buckets** clause, or from the field). To override the strings in the colored labels, specify the strings as:

```
legend "string" "string"
```

To override the label on the left, specify it following the word `label`. To eliminate this label, specify an empty string as follows:

```
legend label ""
```

Axis Statement

The **axis** statement causes a variable to be mapped to an axis in the 3D landscape. The variable's values determine where the entities are positioned on the axis. There can be up to three axis statements. Like the **size** and **color** statements, the **axis** statement contains a series of comma-separated clauses, but you must specify all of them in a single statement.

```
axis clause1, clause2,...
```

Axis Variable

As with size and color, you can specify a field to be used as an axis. The field can be an array that is indexed by slider dimensions. If the field is an array, it must be of numeric type. If the field is not numeric, it is sorted, and each unique value is assigned a position along the axis.

Label Clause

The **label** clause has the following form:

```
label "string"
```

The string is used to label the axis. It appears in the landscape, at the end of the axis line. The default label is the name of the axis variable.

Color Clause

The **color** clause specifies the color used for the axis line and label. It has the following form:

```
color "colorname"
```

Summary Statement

The **summary** statement specifies aggregate information to be calculated for all data defined by the slider position. The summary is used to color the drawing window in the animation control panel. Like the **opacity** and **color** statements, the **summary** statement has several clauses that you can specify in one statement, separated by commas, or in separate statements as follows:

```
summary clause1, clause2,...
```

Summary Variable

You can specify the variable to be used in the summary. This variable must be of numeric type. If you do not specify a **summary** variable, *sum of counts* is used. If you do specify a variable, then the weighted average of that variable (for all the data at the slider location) is used.

Color Clause

The **color** clause specifies the color used to display the summary values in the drawing window. It has the following form:

```
color “colorname”
```

Various shades of the color, from white to the specified color, are used to represent summary values. The minimum summary value is mapped to white, while the maximum summary value is mapped to the specified color. The default summary color is red. If you do not specify a slider variable, this statement has no effect.

Legend Clause

The **legend** clause creates a legend of the summary colors. The **legend** clause syntax can be any of the following:

```
legend off
legend on
legend label "string"
```

The **legend off** clause turns off the legend. The **legend on** clause turns on the legend. You can omit it if you include other **legend** statements. Specifying only **legend on** generates the default legend.

The legend includes a single label to the left (which defaults to the aggregation function and variable used in the summary), and two colored labels on the right (with the minimum and maximum summary values). To override the label on the left, specify it following the word `label`. To eliminate this label, specify an empty string; as follows:

```
legend label ""
```

View Options

The **view** section of the configuration file has several options for controlling parameters of the display. These options can appear in a single **options** statement, separated by commas, or in separate **options** statements. The syntax of the **options** statement is:

```
options option, option,...
```

The following options are available:

- `axis label size float`

  Controls the size of the axis labels.

- `background color “colorname”`

  Controls the initial color of the background.

- `hide label distance float`

  Controls the distance at which axis labels become invisible. Smaller distances might improve performance, but the labels disappear more quickly.

- `grid color` “colorname”

  Controls the color of the grid.

- `grid size float float float`

  Controls the spacing between grid lines. It applies the three values to grid lines along the x, y, and z axes, respectively.

- `orientation top right front float float float`

  Specifies the initial orientation of the scene. You may use three floating point values to specify an arbitrary orientation vector.

- `perspective on off`

  Specifies whether or not to use perspective initially. The default is to use it.

- `shape splatType`

  Specifies the type of splat used. The *shapeName* can be "constant," "linear," "gaussian," "texture," or "sphere."

Creating Data and Configuration Files for the Decision Table Visualizer

This chapter describes the Decision Table Visualizer's data and configuration files. The *.*dtableviz* files contain a schema and an optional history section, and the *.*data* files contain the data. The format of these two files is almost exactly that described in Chapter 6, “Flat File Support for MineSet.”

The Decision Table Visualizer's *.*dtableviz* file must contain a schema of the following form: some number of columns of type **string** or **enum**; followed by a **float** column containing the weight of records; followed by a vector column, indexed by an **enum** containing possible classes, and containing the proportion of the weight of records in each class. The *.*dtableviz* and *.*data* files are automatically generated by the Tool Manager, and you should not normally need to modify them.

The schema in the *.*dtableviz* file uses two extra keywords that are not present in other schemas. They are **auto**, and **nominal**. The Decision Table Visualizer must be able to differentiate between different types of **enums**. For example, **enums** that are automatically rendered discrete by mining are handled differently in drill through than **enums** that are binned by the user in the Tool Manager. Therefore, the keyword **auto** is used to distinguish columns that have been binned by mining.

The label may be a binned column or a string column. Either way it appears as an **enum** in the schema, so that the final probs[] column can be indexed by it. If the label is **string** valued, the **enum** keyword is predicated with **nominal** to distinguish it.

The following is an example configuration file created from the adult dataset:

```
MineSet
input {
     file "adult-tmbin-dtab.dtableviz.data";
     enum string `hours_per_week_bin_k` {"- 20", "20-28", "28-36",
"36-44", "44-52", "52-60", "60+"};
     enum `hours_per_week_bin_k` `hours_per_week_bin`;
     auto enum string `gross_income_k` {"- 9598", "9598-14579",
"14579-24794.5", "24794.5-24806", "24806-29606", "29606-42049.5",
"42049.5-46306.5", "46306.5-64885", "64885+"};
     enum `gross_income_k` `gross_income`;
     auto enum string `final_weight_k` {"- 223033", "223033+"};
     enum `final_weight_k` `final_weight`;
     float `weight`;
     nominal enum string `label` {"Female  ", "Male    "};
# the label values
     float `probs[]`[ enum `label` ]; # and probabilities
}
history {
:
:
}
```

In this example, the name of the data file is *adult-tmbin-dtab.dtableviz.data,* and # denotes a comment line. The first three columns are attributes from the data. The first one, hours_per_week_bin was binned by the user using the Tool Manager. The second two, gross_income and final_weight were binned automatically by the mining algorithm. Note the use of the **auto** keyword in their **enum** definitions. The fourth column gives the weight of records that have the values given by the first three columns.

The last column, probs[], is a vector-valued column that gives the probability of each class. It is computed by dividing the number of records for each class by the value in the weight column. The sum of the entries of this vector must add to 1. The definition of the index label includes the keyword **nominal** to show that the class values are not numeric, and therefore do not have an implied ordering. The **nominal** keyword would not be present had the label been a binned numeric attribute.

Format of the Evidence Visualizer's Data File

This chapter describes the input data file of the Evidence Visualizer. This file is a textual representation of the Evidence Classifier. The data file is generated automatically through the Tool Manager. In some instances you may want to edit this file in order to change label, attribute, or value names.

The Evidence Visualizer requires a data file containing the label and attributes, along with weights and probabilities. These are used to create the graphics. The data file is created when you run the Evidence Inducer using the Tool Manager. The format of the data file is:

```
#MineSet

<type> "<label>" <L>
"<label1>" <weight1> <probability1>
"<label2>" <weight2> <probability2>
:
"<labelL>" <weightL> <probabilityL>

<M>

<type> "<attrib1>" <N1> <importance1>
"<value1_1>" <weight1_1_1> <prob1_1_1> ... <weight1_1_L> <prob1_1_L>
"<value1_2>" <weight1_2_1> <prob1_2_1> ... <weight1_2_L> <prob1_2_L>
:
"<value1_N1>" <weight1_N1_1> <prob1_N1_1> ... <weight1_N1_L> <prob1_N1_L>

<type> "<attrib2>" <N2> <importance2>
"<value2_1>" <weight2_1_1> <prob2_1_1> ... <weight2_1_L> <prob2_1_L>
"<value2_2>" <weight2_2_1> <prob2_2_1> ... <weight2_2_L> <prob2_2_L>
:
"<value2_N2>" <weight2_N2_1> <prob2_N2_1> ... <weight2_N2_L> <prob2_N2_L>

:
:
:
```

```
"<attribM>" <NM> <importanceM>
"<valueM_1>" <weightM_1_1> <probM_1> ... <weightM_1_L> <probM_1_L>
"<valueM_1>" <weightM_2_1> <probM_2_2> ... <weightM_2_L> <probM_2_L>
:
"<valueM_NM>" <weightN1_NM_1> <probM_NM_1> ... <weightM_NM_L> <probM_NM_L>

history {
:
:
}
```

L is the number of label values, *M* is the number of attributes, and *N* is the number of values or bins for attribute *i*. The <>'s indicate variables. The actual file has numbers or strings. A NULL is considered a unique value if it is present in an attribute. If NULLs exist for an attribute, they always appear as the first value (that is, the first line following the attribute header) and are represented by "?".

The `<type>` can be:

- **nominal**, which is used for the label and currently implies a string-valued attribute (or an integer attribute).
- **enum**, which is used for attributes binned in Tool Manager.
- **auto-enum**, which is used for attributes that are discretized automatically by the inducer. If a type is not present, **auto-enum** is assumed.

Lines beginning with # are comments (and ignored by the program).

You can include an optional history section at the end of the file. It is used by the Tool Manager for drill through. Without this section, drill through is not possible (in the Evidence Visualizer or any other MineSet tool).

The *weights* are the number of records (or sum of weights) in the table with that particular attribute value (or range of values). Therefore, the sum of the weights for each attribute equals the total number of records in the table (unless record weighting was used). The *probability* is the number of weights for that attribute value divided by the total number of weights. If the data file was generated with *Laplace correction* turned on, the probability is only approximately the number of weights for that attribute value divided by the total number of weights. Thus, the probability value indicates the proportion of records with a particular label that have this attribute value instead of another value.

Data files must have an *.eviviz* extension. When starting the Evidence Visualizer, or when opening a file, you must specify the data file.

The following is a sample Evidence Visualizer data file, *cars.eviviz*. Windows users can find the file in the directory in which MineSet is installed, under the *Examples\eviviz\examples* directory. UNIX users find the file in */usr/lib/MineSet/eviviz/examples/cars.eviviz*.

```
#MineSet
#automatically generated
NOMINAL "origin" 3
"Europe" 73 0.179803
"Japan" 79 0.194581
"US" 254 0.625616
6
AUTO_ENUM "mpg" 5 25.448
"?" 3 0.0410959 0 0 5 0.019685
"- 16.1" 0 0 0 0 87 0.34252
"16.1-21.05" 10 0.136986 5 0.0632911 77 0.30315
"21.05-30.95" 43 0.589041 28 0.35443 67 0.26378
"30.95+" 17 0.232877 46 0.582278 18 0.0708661
NOMINAL "cylinders" 5 29.1759
"8" 0 0 0 0 108 0.425197
"4" 66 0.90411 69 0.873418 72 0.283465
"6" 4 0.0547945 6 0.0759494 74 0.291339
"3" 0 0 4 0.0506329 0 0
"5" 3 0.0410959 0 0 0 0
AUTO_ENUM "horsepower" 4 22.3514
"?" 2 0.0273973 0 0 4 0.015748
"- 78.5" 40 0.547945 46 0.582278 25 0.0984252
"78.5-134" 31 0.424658 33 0.417722 131 0.515748
"134+" 0 0 0 0 94 0.370079
AUTO_ENUM "weightlbs" 4 28.5157
"- 2379.5" 43 0.589041 57 0.721519 30 0.11811
"2379.5-2959.5" 18 0.246575 22 0.278481 57 0.224409
"2959.5-3274" 9 0.123288 0 0 29 0.114173
"3274+" 3 0.0410959 0 0 138 0.543307
AUTO_ENUM "time_to_60" 3 10.0055
"- 13.45" 3 0.0410959 3 0.0379747 78 0.307087
"13.45-19.45" 52 0.712329 75 0.949367 162 0.637795
"19.45+" 18 0.246575 1 0.0126582 14 0.0551181
AUTO_ENUM "year" 1 2.84217e-14
"ignore" 73 1 79 1 254 1
```

Note that the sum of the probabilities corresponding to a particular label value for a given attribute always equals 1. Consider the attribute *weightlbs*, for label value US (the first one), we have .11811+.224409+.114173+.543307=1.0 . Also note that attributes *mpg* and *horsepower* have NULL values.

The eviviz datafile (**.eviviz*) also accommodates loss matrices and optionally included Laplace factor. The next example *.eviviz* file contains both of these.

```
MineSet
# MLC++ generated file for MineSet Evidence Visualizer.
NOMINAL “edibility” 2
“edible” 5 0.621212121212
“poisonous” 3 0.378787878788

TOTAL 8

LAPLACE yes 0

LOSS
10 0 20
10 10000 0


3

NOMINAL “cap-shape” 6 47.6129309656
“bell” 1 0.195652173913 0 0.0333333333333
“conical” 0 0.0217391304348 0 0.0333333333333
“convex” 2 0.369565217391 2 0.566666666667
“flat” 2 0.369565217391 0 0.0333333333333
“knobbed” 0 0.0217391304348 1 0.3
“sunken” 0 0.0217391304348 0 0.0333333333333

NOMINAL “cap-surface” 4 27.8397591211
“fibrous” 2 0.386363636364 0 0.0357142857143
“grooves” 0 0.022727272727 3 0 0.0357142857143
“scaly” 1 0.204545454545 2 0.607142857143
“smooth” 2 0.386363636364 1 0.321428571429

NOMINAL “cap-color” 10 47.6129309656
“brown” 1 0.18 1 0.264705882353
“buff” 0 0.02 0 0.0294117647059
“cinnamon” 0 0.02 0 0.0294117647059
“gray” 0 0.02 1 0.264705882353
“green” 0 0.02 0 0.0294117647059
“pink” 1 0.18 0 0.0294117647059
```

```
“purple” 0 0.02 0 0.0294117647059
“red” 1 0.18 0 0.0294117647059
“white” 1 0.18 1 0.264705882353
“yellow” 1 0.18 0 0.0294117647059
```

The Laplace matrix follows the form shown in the graphical user interface (GUI). See the description in the *MineSet Enterprise Edition Reference Guide*.

Nulls in MineSet

Nulls represent unknown data. MineSet supports nulls in the data access tools, the mining tools, and the visualization tools. This chapter will help you understand how MineSet handles nulls.

Semantics of Nulls

Unknown data values are often represented as nulls in data sources. While you can associate different semantics with nulls, usually nulls represent missing or unknown values. For example, if a data record consists of fields representing *firstname*, *middlename*, and *lastname*, and if a person's *middlename* is not known, it can be represented by the null value.

Nulls can occur in data for a various reasons. For example, they can be a way of representing unknown data, or they can result from doing certain kinds of aggregations. For example, if there are no flights between San Francisco and MineSet City, a query such as "find the average flight time from San Francisco to MineSet City" yields a null value.

MineSet generally follows the semantics of relational databases when dealing with nulls, and treats them as unknown values.

Some databases, such as Oracle RDBMS, do not distinguish between null and empty strings. In such a case, it is not possible to distinguish between an unknown middle name and a person who does not have a middle name. On the other hand, Sybase RDBMS distinguishes between null and empty strings. Therefore, MineSet can distinguish empty and null strings when reading from Sybase, but not from Oracle.

Representation of Nulls

In data files, as well as in the visual tools, nulls are represented by the string "?" (question mark). Thus, if Joe Miner's middle name is unknown, his name is represented in our example data file (with schema *firstname*, *middlename*, *lastname*) as:

```
Joe     ?     Miner
```

In general, the color gray is often associated with null values in the visualizations. The graphical representation of nulls varies from tool to tool. See the chapters on individual tools in the *MineSet Enterprise Edition Reference Guide* for this information.

Operations on Nulls

Given that nulls represent unknown values, it is straightforward to give meaning to expressions involving nulls.

Arithmetic Expressions

Arithmetic operations with nulls always give a null result. For example:

- (5 + ?) evaluates to ? (adding 5 to an unknown yields yet another unknown).
- (6 / ?) evaluates to ?.

Boolean Expressions

Boolean variables can also be null. If a Boolean-valued variable has a null (unknown) value, the result of combining it with another Boolean variable in an expression is also unknown, unless it is possible to determine the result just from the known value. For example:

- "? AND FALSE" is FALSE (because FALSE ANDed with anything is always FALSE).
- "? AND TRUE" is ?.

- “? OR FALSE” is ?
- "? OR TRUE” is TRUE (because TRUE ORed with anything is always TRUE).
- “NOT ?” is ?

Relational Operations

Relational operations (==, !=, <, >, <=, and >=) involving nulls always evaluate to null. Some particular cases worth emphasizing are:

- “? == ?” evaluates to ?, not TRUE.
- “? != ?” evaluates to ?, not FALSE.

Given two unknown values, it is unknown whether the two are equal or unequal. This can be confusing when you are using a search panel. For example, if you search for all values not equal to 0, nulls do not appear, yet neither do they appear if you search for values equal to 0. Because of this, search panels allow you to search explicitly for nulls. (Some search panels allow you to treat nulls as zeros; see the individual tool discussions in the *MineSet Enterprise Edition Reference Guide* for more information.)

Testing for Nulls

The function **isNull()** can determine whether or not a variable has the value null. For example:

- **isNull(***X***)** evaluates to TRUE if variable *X* has the null value.
- **isNull(***X***)** evaluates to FALSE if variable *X* has a non-null value.

Aggregations in the Presence of Nulls

MineSet stays close to the semantics of SQL and relational databases when aggregating columns that might have null values. Therefore, null values are ignored when SUM, AVG, MIN, MAX, and COUNT are computed. For example, consider a data file with records representing the number of pets a person has. The schema of this record is *name*, *num_pets*. Null (unknown) values are represented by "?."

Name	NUM_PETS
Tesler	3
Rathmann	?
Haber	1
Bhargava	0
Sangudi	?

The computations are as follows:

- SUM(NUM_PETS) = 4.
- COUNT(NUM_PETS) = 3 (not 5, even though there are 5 rows of data).
- AVG(NUM_PETS) = 1.33.
- MAX(NUM_PETS) = 3.
- MIN(NUM_PETS) = 0.

In these aggregations, null values are basically ignored (the value 0 is different from ?, and is not ignored).

A special case is an aggregation in which all the values being aggregated are themselves null. An even more specialized case is if there are no values being aggregated: for example, when summing an empty column. In both these cases, the SUM, AVERAGE, MIN, and MAX are ?, while the COUNT is 0.

Sort Order for Nulls

In an ascending sorted sequence, null values always appear before non-null values. In a descending sorted sequence, null values always appear after non-null values.

Bins and Arrays with Nulls

MineSet lets you bin numeric data into bins or discrete intervals. It also lets you (via the aggregation panel in the Tool Manager) create arrays on these bins. When a column of values is binned, all null values are put in a bin labeled "?". Such a bin label is always created, whether or not the data being binned contains nulls.

You can choose whether to use this bin for nulls in your application. You can do so by allowing arrays to ignore or keep bins for nulls by setting the desired option in the Tool Manager's Preferences dialog (on IRIX), or by checking the "Use nulls in aggregation" checkbox at the bottom of the aggregation operator (on Windows). For example, if you know that the column being binned has no nulls, or you intend to study only the data corresponding to non-null values, you can choose to ignore the bin for nulls.

ActiveX Visualization Control API for MineSet Visualizers

If you are a developer who wants to integrate MineSet visualization tools into your ActiveX applications, this chapter provides helpful information. The topics discussed are as follows:

API Overview

Eight ActiveX controls are included in the MineSet distribution. They are automatically registered on your system when you install MineSet. Once they are registered, you can browse them using the ActiveX test container in Microsoft Developer Studio.

Basics of Component Object Model

The Component Object Model, or COM provides an efficient methodology for modular application development. You can build stand-alone components (servers). Users or clients of COM servers, can use pre-built functionality in server objects without intimate knowledge of the server object during development. Previously, you were required to link to a component's functionality at design time, or provide a path to the component in the source code. Instead, COM can now ask the registry for the object's location. This means that as long as the registry knows where to find the server object, your client application can use the object.

COM objects are implemented either within executables (EXEs) or Dynamic Link Libraries (DLLs). COM objects implemented in EXEs are called local servers, while those implemented in DLLs are called in-process servers. An in-process COM server exposes its functions so that outside applications can use the functionality of its DLL. The visualization controls described in this chapter are of the latter type.

The COM library uses a class identifier (a CLSID value) to uniquely identify each COM object. An application then uses this identifier to determine which object the application wants to use. For example, suppose you create and register a COM object that provides functionality similar to the Windows 95 tree-view control—showing, for example, a Web site hierarchy. Once you register this object, other applications can call the functions you expose in it.

Because ActiveX controls are COM objects, this control is provided as an ActiveX control (with appropriate documentation). You can integrate it into your applications using a development environment and call methods, and you can set properties on the control.

ActiveX Architecture

ActiveX controls have become the primary architecture for developing programmable software components for use in containers ranging from software development tools to end-user productivity tools. For a control to operate well in a variety of containers, it must be able to assume some minimum level of functionality that it can rely on in all containers. This minimum level of functionality is defined in the COM and ActiveX Control specifications, published by Microsoft Corporation. These guidelines have been rigorously followed so that MineSet visualization controls would be reliable and interoperable, and, and so that they would be better and more usable for building component-based solutions.

MineSet's Visualization Controls

MineSet visualization controls consist of eight separate ActiveX controls implemented as in-process DLLs. As a developer of a client application that will incorporate the MineSet visualizations, the you only have to be concerned about the VizComposite control. The VizComposite control is an actual composite control that dynamically creates and manages the other seven ActiveX controls based on the visualization file being loaded. A client application only needs to create an instance of the VizComposite control, either during design or at run-time, in order to embed MineSet visualization capabilities within its application. Once instantiated, you can call methods and set or get property values from the control to alter its behavior and to access certain functionality.

Recommended Requirements

You can develop your client application using MineSet visualization controls in any development environment, including Visual C++, Visual Basic, and Visual J++. While Windows 95, Windows 98, or Windows NT can be your development platform, SGI highly recommends developing on Windows NT 4.0 Workstation for any serious DCOM work. Using Windows NT, you can examine processes and shut down the system with somewhat more stability and ease than with Windows 95 or Windows 98.

If you are a novice at programming in COM, consider reading some entry-level books, such as:

- *Inside COM,* by Dale Rogerson (Microsoft Press).
- *Understanding ActiveX and OLE,* by David Chappell (Microsoft Press).

Some very good books on the details of COM and ActiveX/ATL include:

- *Essential COM,* by Don Box (Addison Wesley).
- *Beginning ATL COM Programming,* by Grimes and Stockton (Wrox Press).
- *Professional ATL COM Programming,* by Richard Grimes (Wrox Press).

Note: This document does not cover detailed semantics of the OLE interfaces; these are covered by the Microsoft Platform SDK documentation.

ActiveX Controls

The ActiveX control names are in the form of MineSet *ControlName* Control, (for example, MineSet VizComposite Control). The controls are:

- VizComposite (the only one of interest to the application developer).
- SyncServer (used for synchronizing visualizations).
- VizAnimation (used for the animation panel).
- VizLegend (used for the legend).
- VizMain (contains the 3D viewer).
- VizPane (auxiliary viewer used by some visualizers).

- VizPlaneViewer (viewer inside VizPane).
- VizSelection—shows pick/select information.

The VizComposite control contains and manages some or all the other controls as needed. This composite control implements several ActiveX interfaces.

The interface you obtain from the composite control should be one of the following:

- IVizCommon
- IVizCommon2
- IScatterviz
- IScatter2
- ISplatviz
- ISplatviz2
- IMapviz
- IEviviz
- IDtableviz
- IDtableviz2
- ITreeviz

IVizCommon contains methods that are common to all the other interfaces. Also, all the other interfaces include **IVizCommon**, so methods of **IVizCommon** may be called directly from any of the other interfaces.

After you have loaded a file into **VizComposite**, you can use **GetVizType()** (a method of **IVizCommon**) to determine which of the specific tool interfaces you need to obtain, or you can just decide based on the configuration file extension.

The following tables describe all the interface methods and properties. Each description includes the C++ style declaration exposed via a COM vtable interface. When you load the control into Visual Studio and it becomes a class, the **IScatterviz** and **IVizCommon** interfaces from which it derives are exposed. These Automation interfaces are slightly different than the COM vtable interfaces, so the two interfaces are described in separate tables. If you load the control into a Java or Visual Basic environment, the resulting class will have Java or Visual Basic style declarations instead.

After inserting the **VizComposite** control into the Visual Studio workspace, a member variable representing the control can be created. The COM vtable interfaces may then be accessed via this member variable. For example, if the member variable is named m_vizCompositeCtrl, the following code shows how to pass a file to the control using the COM vtable interface for IVizCommon (the file *atlbase.h* is included in order to use the USES_CONVERSION and A2W macros.):

```
#import "d:\sdk1ib\VizComposite.dll" no_namespace
#include "atlbase.h"

...

    USES_CONVERSION;

    IVizCommon *pVizCommon = NULL;
    IUnknown *pUnk = NULL;

    pUnk = m_vizcompositectrl.GetControlUnknown();

    if(pUnk)
    {
        pUnk->QueryInterface(__uuidof(IVizCommon), (void **)&pVizCommon);
        if(pVizCommon)
        {
            pVizCommon->put_File(A2W("c:\\examples\\adult-salary-evi.eviviz"));
            pVizCommon->Release();
        }
    }
```

Note: You can view online documentation for the methods in each control by using oleview.

IVizCommon

Table 14-1 lists the **IVizCommon** methods for the COM vtable interface. These methods and properties are common to all viz tools, although their behavior for a particular tool may vary. (In rare cases, behavior has not yet been implemented for some tools, in which case E_NOTIMPL is returned. Otherwise S_OK is returned.)

Table 14-1 IVizCommon Methods—COM vtable Interface

Declaration	Description
`void PutFile(_bstr_ _arg1);`	Creates the right tool based on the configuration file extension. This is probably the single most important method. You can create an application using only this method.
`VIZCODE GetVizType();`	Returns an **enum** indicating the type of visualization tool. It returns VIZ_UNKNOWN if no configuration file has been opened yet. Other values are: VIZ_MAP, VIZ_SCATTER, VIZ_SPLAT, VIZ_EVI, VIZ_DTABLE, and VIZ_TREE (in the **VizCommon.idl** file this is the VizType property).
`HRESULT SetScale(float s);`	In the Tree Visualizer, Evidence Visualizer, Decision Table Visualizer, and Map Visualizer, this function scales the heights of objects in the scene. In the Scatter Visualizer, it scales the size of the entities. In the Splat Visualizer it scales the opacity. For the MineSet Visualizer application the scale is adjusted by the slider on the left trim of the viewer.
`HRESULT ShowDecoration(long on);`	Turns on or off (TRUE/FALSE) the Inventor window decoration based on the value of *on*.
`HRESULT ShowBackgroundColorDialog();`	Brings up the color selector dialog, so the user can select a desired background color.
`HRESULT SetBackgroundColor(float r, float g, float b);`	Explicitly sets a specific background color.
`HRESULT ShowFilterDialog();`	Brings up a filter panel of some type that can be used to reduce the amount of geometry in the scene. The Scatter Visualizer, Splat Visualizer, Decision Table Visualizer, and Map Visualizer use the same style of filter dialog. The Tree Visualizer and Evidence Visualizer have their own variations.

Table 14-1 (continued) IVizCommon Methods—COM vtable Interface

Declaration	Description

`HRESULT DoIdle(long lCount);`	Makes Inventor run smoothly. It should be called in your application's **OnIdle** member function.
	For example:
	`BOOL CVizApp::OnIdle(LONG lCount) {`
	`  if (m_bIsProcessing) {`
	`  m_pVizView->m_vizCompositeCtrl.Idle(lCount)`
	`;`
	`  }`
	`  return CWinApp::OnIdle(lCount);`
	`}`
	m_bIsProcessing must be set to true in the view's **OnInitialUpdate**:
	((CVizApp *)AfxGetApp())->m_bIsProcessing = TRUE;

`HRESULT ShowSelectionDialog();`	Brings up a table showing information about all the objects in the scene which are currently selected. For the visualization application supplied with MineSet, this method is called when the user selects Show Values from the Selection menu.
`HRESULT SetLegendHeight(short numLegends);`	Sets the height of the legend control at the bottom of the visualization tool. If the *numLegends* argument is 3, then the height of the legend control will be enough to show three legends without the help of a vertical scrollbar.
`HRESULT EnableSound(long on);`	Turns the sound effects on or off (TRUE or FALSE respectively.)
`VIEWINGCODE GetViewingMode();`	Returns the Inventor viewing mode, either VIEWING_PICK (viewer turned on) or VIEWING_GRASP (viewer turned off). When the viewer is turned on (pick mode), events are consumed by the viewer. When viewing is off, events are processed by the viewer's render area. This means events will be sent to the scene graph for processing (that is, picking can occur). These methods have no meaning for Tree Visualizer, which does not use a model viewer.
`void PutViewingMode(VIEWINGCODE pVal);`	Establishes the desired viewing mode (either VIEWING_PICK or VIEWING_GRASP). In grasp mode, the cursor appears as a hand, and viewing transformations are performed. In pick mode, no navigation is possible, but you can pick and select objects in the scene. These methods have no meaning for Tree Visualizer, which does not use a model viewer.

Table 14-1 (continued) IVizCommon Methods—COM vtable Interface

Declaration	Description
`long GetSupportsAnimation();`	Returns TRUE if the visualization tool currently instantiated supports animation. In other words, returns TRUE if sliders are present. Some tools such as the Evidence Visualizer, Decision Table Visualizer, and Tree Visualizer never have animation in their current implementations. For these visualizers GetSupportsAnimation always return FALSE.
`_bstr_t GetSelectionExpression();`	Returns the selection expression which has been created based on the current selection of objects in the scene. This string is typically used to drill through to the underlying data in some manner, but there are other potential uses. If nothing has been selected the expression is an empty string (“”).
`_bstr_t GetHistoryString();`	Returns the history from the configuration file. This history contains everything that the Tool Manager needs to recreate its state at the time the current visualization was created.
`_bstr_t GetTitleString();`	Returns an appropriate application title including the application name and current configuration file base name.
`HRESULT EnableStereo(long on);`	Turns on stereo rendering in the 3D viewer. It may be necessary to set hardware specific settings, and/or have certain peripherals like Crystal Eyes glasses before stereo is actually useful.
`HRESULT SetStereoOffset(float offset);`	Sets the amount of eye separation for the left and right stereo views.
`HRESULT SetSelectionsHeight(short height);`	Sets the height (in pixels) of the selection control which appears at the top of all tools and displays pick information.
`HRESULT EnableWarnOnExecute(long on);`	Specifies whether the user should be warned when a user-defined execute statement is invoked (this occurs when an object is double clicked and such an execute statement has been defined). The warning might be useful if you are concerned about security.
`HRESULT EnableQuiet(long on);`	Turns off the display of all error and warning popup messages. In most cases this option should be off (FALSE). See also **SetErrorFile**.

Table 14-1 (continued) IVizCommon Methods—COM vtable Interface

Declaration	Description
`HRESULT SetErrorFile(_bstr_t fileName);`	Specifies a file where all error messages should be sent. If an error file is set, error messages will still appear in dialogs on the screen. If you do not want the dialogs to appear on the screen you should use EnableQuiet(TRUE). This method is primarily used for debugging and testing of applications made using the composite control.
`HRESULT SetHideDistance(float distance);`	Sets the distance at which level of detail is hidden. The most common use of this distance factor is in changing the level of detail on 3D text displayed in a scene. This option could also change detail thresholds in other circumstances. Sometimes the text has three or four levels of detail (hidden, bounding box, wire frame, full 3D), the distance parameter sets the first threshold, and the other thresholds are scalar multiples.
`void PutNullStyle(DRAW_STYLE style);`	Allows you to set how Null Values are displayed in the scene. Possible values are SOLID, HIDDEN and OUTLINE. Outline behaves the same as hidden for all tools except the Tree Visualizer.
`DRAW_STYLE GetNullStyle();`	Returns current setting for how Null values are displayed.
`HRESULT SetFont(_bstr_t fontName);`	Sets a desired font. This might be particularly useful for certain locales.
`HRESULT SaveSceneGraph(_bstr_t fileName);`	Outputs the current scenegraph to a file in Inventor format. This *.iv* file cannot be loaded directly into another application, however, without having the right DLL's installed. This method is primarily used for debugging and testing.
`HRESULT SaveCompositeImage(_bstr_t fileName, long hWnd);`	Outputs the current scenegraph to a file in Inventor format. This saves an image of the whole composite control to a file. An offscreen renderer is not used to save the scene, so if you want improved resolution, a better choice might be **SaveViewerImage**.
`HRESULT SaveViewerImage(_bstr_t fileName, float desiredXinches, float desiredYinches, short printerDPI);`	Saves the main viewer image to a file. An offscreen renderer is used for this, so the resolution is better. If the last three arguments are not specified (or are 0), then values are chosen based on the current screen size and resolution.
`HRESULT HasLegend(long *HasLegend);`	Returns TRUE or FALSE depending on whether or not a legend is present at the bottom.

Table 14-1 (continued) IVizCommon Methods—COM vtable Interface

Declaration	Description
`HRESULT SaveLegendImage(_bstr_t fileName);`	Takes a snapshot of the legend control and saves it to a file. This method is primarily used for debugging and testing.
`HRESULT SaveDataTable(_bstr_t fileName);`	Stores the tools internal data table to a file. This method is primarily used for debugging and testing.
`HRESULT PrintCompositeImage(long hDC, long hWnd;`	Prints an image of the window pointed to by *hWnd* to a file. An offscreen rendered is not used for saving the scene, so **SaveViewerImage** might be better for improved resolution. If *hWnd* is NULL, only the main rendering window image is saved.
`HRESULT SetMouseButtonStyle(MBUTTON_STYLR style);`	Selects the desired mouse mapping mode. Possible values are, MINESET_TWO_BUTTON_STYLE and MINESET_THREE_BUTTON_STYLE Since many users do not have a three-button mouse, MineSet now allows for two-button operation. For those familiar with previous versions of MineSet on the IRIX platform, three-button style offers the same controls as before. The default, two-button style, is more consistent between tools and has less dependence on the use of modifier keys. However, no functionality is lost with either a three- or a two-button mouse. See the Navigation entry in the *MineSet Enterprise Edition Reference Guide* for mouse button navigation details.
`HRESULT SetCameraPosition(float x, float y, float z);` `HRESULT GetCameraPosition(float *v);`	Provides methods of manipulating the camera viewpoint. Normally, the user interacts with the scene using the mouse, but an application may require alternative methods for setting or getting the viewpoint and orientation. Possible uses are setting up or running animations, saving or restoring interesting viewpoints, alternative input devices, and so forth.
`HRESULT SetCameraOrientation(float x, float y, float z, float angle);` `HRESULT GetCameraOrientation(float *v);`	Changes camera orientation. In **SetCameraOrientation**, *x*, *y*, *z* are the components of the axis, and *angle* is the rotation of the camera around that axis of orientation in radians. In **GetCameraOrientation**, *x*, *y*, and *z* are the components of the axis orientation, *angle* gives the orientation of the camera about that axis in radians.

Table 14-2 lists the IVizCommon methods for the Automation interface.

Table 14-2 IVizCommon Methods—Automation Interface

Declaration	Description
`void SetFile(LPCTSTR fileName);`	Creates the right tool based on the configuration file extension. This is probably the single most important method. You can create an application using only this method.
`long GetVizType();`	Returns an **enum** indicating the type of visualization tool. It returns VIZ_UNKNOWN if no configuration file has been opened yet. Other values are: VIZ_MAP, VIZ_SCATTER, VIZ_SPLAT, VIZ_EVI, VIZ_DTABLE, and VIZ_TREE (in the **VizCommon.idl** file this is the VizType property).
`void SetScale(float s);`	In the Tree Visualizer, Evidence Visualizer, Decision Table Visualizer, and Map Visualizer, this function scales the heights of objects in the scene. In the Scatter Visualizer, it scales the size of the entities. In the Splat Visualizer it scales the opacity. For the MineSet Visualizer application the scale is adjusted by the slider on the left trim of the viewer.
`void ShowDecoration(long on);`	Turns on or off (TRUE/FALSE) the Inventor window decoration based on the value of *on*.
`void ShowBackgroundColorDialog();`	Brings up the color selector dialog, so the user can select a desired background color.

`void SetBackgroundColor(float r,`
`float g,`
`float b);`
	Explicitly sets a specific background color.

`void ShowFilterDialog();`	Brings up a filter panel of some type that can be used to reduce the amount of geometry in the scene. The Scatter Visualizer, Splat Visualizer, Decision Table Visualizer, and Map Visualizer use the same style of filter dialog. The Tree Visualizer and Evidence Visualizer have their own variations.

Table 14-2 (continued) IVizCommon Methods—Automation Interface

Declaration	Description

`void DoIdle(long lCount);`	Makes Inventor run smoothly. It should be called in your application's **OnIdle** member function.
	For example:
	`BOOL CVizApp::OnIdle(LONG lCount) {`
	`  if (m_bIsProcessing) {`
	`  m_pVizView->m_vizCompositeCtrl.Idle(lCount)`
	`;`
	`  }`
	`  return CWinApp::OnIdle(lCount);`
	`}`
	m_bIsProcessing must be set to true in the view's **OnInitialUpdate**:
	((CVizApp *)AfxGetApp())->m_bIsProcessing = TRUE;

`void ShowSelectionDialog();`	Brings up a table showing information about all the objects in the scene which are currently selected. For the visualization application supplied with MineSet, this method is called when the user selects Show Values from the Selection menu.
`void SetLegendHeight(short numLegends);`	Sets the height of the legend control at the bottom of the visualization tool. If the *numLegends* argument is 3, then the height of the legend control will be enough to show three legends without the help of a vertical scrollbar.
`void EnableSound(long on);`	Turns the sound effects on or off (TRUE or FALSE respectively.)
`long GetViewingMode();`	Returns the Inventor viewing mode, either VIEWING_PICK (viewer turned on) or VIEWING_GRASP (viewer turned off). When the viewer is turned on (pick mode), events are consumed by the viewer. When viewing is off, events are processed by the viewer's render area. This means events will be sent to the scene graph for processing (that is, picking can occur). These methods have no meaning for Tree Visualizer, which does not use a model viewer.
`void SetViewingMode(long nNewValue);`	Establishes the desired viewing mode (either VIEWING_PICK or VIEWING_GRASP). In grasp mode, the cursor appears as a hand, and viewing transformations are performed. In pick mode, no navigation is possible, but you can pick and select objects in the scene. These methods have no meaning for Tree Visualizer, which does not use a model viewer.

Table 14-2 (continued) IVizCommon Methods—Automation Interface

Declaration	Description
`Long GetSupportsAnimation();`	Returns TRUE if the visualization tool currently instantiated supports animation. In other words, returns TRUE if sliders are present. Some tools such as the Evidence Visualizer, Decision Table Visualizer, and Tree Visualizer never have animation in their current implementations. For these visualizers GetSupportsAnimation always return FALSE.
`CString GetSelectionExpression();`	Returns the selection expression which has been created based on the current selection of objects in the scene. This string is typically used to drill through to the underlying data in some manner, but there are other potential uses. If nothing has been selected the expression is an empty string (“”).
`CString GetHistoryString();`	Returns the history from the configuration file. This history contains everything that the Tool Manager needs to recreate its state at the time the current visualization was created.
`CString GetTitleString();`	Returns an appropriate application title including the application name and current configuration file base name.
`void EnableStereo(long on);`	Turns on stereo rendering in the 3D viewer. It may be necessary to set hardware specific settings, and/or have certain peripherals like Crystal Eyes glasses before stereo is actually useful.
`void SetStereoOffset(float offset);`	Sets the amount of eye separation for the left and right stereo views.
`void SetSelectionsHeight(short height);`	Sets the height (in pixels) of the selection control which appears at the top of all tools and displays pick information.
`void EnableWarnOnExecute(long on);`	Specifies whether the user should be warned when a user-defined execute statement is invoked (this occurs when an object is double clicked and such an execute statement has been defined). The warning might be useful if you are concerned about security.
`void EnableQuiet(long on);`	Turns off the display of all error and warning popup messages. In most cases this option should be off (FALSE). See also **SetErrorFile**.

Table 14-2 (continued) IVizCommon Methods—Automation Interface

Declaration	Description
`void SetErrorFile(LPCTSTR fileName);`	Specifies a file where all error messages should be sent. If an error file is set, error messages will still appear in dialogs on the screen. If you do not want the dialogs to appear on the screen you should use EnableQuiet(TRUE). This method is primarily used for debugging and testing of applications made using the composite control.
`void SetHideDistance(float distance);`	Sets the distance at which level of detail is hidden. The most common use of this distance factor is in changing the level of detail on 3D text displayed in a scene. This option could also change detail thresholds in other circumstances. Sometimes the text has three or four levels of detail (hidden, bounding box, wire frame, full 3D), the distance parameter sets the first threshold, and the other thresholds are scalar multiples.
`void PutNullStyle(long nNewValue);`	Allows you to set how Null Values are displayed in the scene. Possible values are SOLID, HIDDEN and OUTLINE. Outline behaves the same as hidden for all tools except the Tree Visualizer.
`long GetNullStyle();`	Returns current setting for how Null values are displayed.
`void SetFont(LPCTSTR fontName);`	Sets a desired font. This might be particularly useful for certain locales.
`void SaveSceneGraph(LPCTSTR fileName);`	Outputs the current scenegraph to a file in Inventor format. This *.iv* file cannot be loaded directly into another application, however, without having the right DLL's installed. This method is primarily used for debugging and testing.
`void SaveCompositeImage(LPCTSTR fileName, long hWnd);`	Outputs the current scenegraph to a file in Inventor format. This saves an image of the whole composite control to a file. An offscreen renderer is not used to save the scene, so if you want improved resolution, a better choice might be **SaveViewerImage**.
`void SaveViewerImage(LPCSTR fileName, float desiredXinches, float desiredYinches, short printerDPI0);`	Saves the main viewer image to a file. An offscreen renderer is used for this, so the resolution is better. If the last three arguments are not specified (or are 0), then values are chosen based on the current screen size and resolution.
`void HasLegend(long *HasLegend);`	Returns TRUE or FALSE depending on whether or not a legend is present at the bottom.

Table 14-2 (continued) IVizCommon Methods—Automation Interface

Declaration	Description
`void SaveLegendImage(LPCTSTR fileName);`	Takes a snapshot of the legend control and saves it to a file. This method is primarily used for debugging and testing.
`void SaveDataTable(LPCTSTR fileName);`	Stores the tools internal data table to a file. This method is primarily used for debugging and testing.
`void PrintCompositeImage(long hDC,` `long hWnd;`	Prints an image of the window pointed to by *hWnd* to a file. An offscreen rendered is not used for saving the scene, so **SaveViewerImage** might be better for improved resolution. If *hWnd* is NULL, only the main rendering window image is saved.
`void SetMouseButtonStyle(long style);`	Selects the desired mouse mapping mode. Possible values are, MINESET_TWO_BUTTON_STYLE and MINESET_THREE_BUTTON_STYLE Since many users do not have a three-button mouse, MineSet now allows for two-button operation. For those familiar with previous versions of MineSet on the IRIX platform, three-button style offers the same controls as before. The default, two-button style, is more consistent between tools and has less dependence on the use of modifier keys. However, no functionality is lost with either a three- or a two-button mouse. See the Navigation entry in the *MineSet Enterprise Edition Reference Guide* for mouse button navigation details.

`void SetCameraPosition(float x,`
`float y,`
`float z);`
`void GetCameraPosition(float *v);`
	Provides methods of manipulating the camera viewpoint. Normally, the user interacts with the scene using the mouse, but an application may require alternative methods for setting or getting the viewpoint and orientation. Possible uses are setting up or running animations, saving or restoring interesting viewpoints, alternative input devices, and so forth.

`void SetCameraOrientation(float x,`
`float y,`
`float z,`
`float angle);`
`void GetCameraOrientation(float *v);`
	Changes camera orientation. In **SetCameraOrientation**, *x*, *y*, *z* are the components of the axis, and *angle* is the rotation of the camera around that axis of orientation in radians.
	In **GetCameraOrientation**, *x*, *y*, and *z* are the components of the axis orientation, *angle* gives the orientation of the camera about that axis in radians.

IVizCommon2

Table 14-3 lists the **IVizCommon2** methods for the COM vtable interface.

Table 14-3 IVizCommon2 Methods—COM vtable Interface

Declaration	Description
`HRESULT SetSelectionExpression(_bstr_t selectionExp);`	Changes the set of currently selected objects in the scene.
`_bstr_t GetFilterExpression();`	Returns the filter expression.
`HRESULT SetFilterExpression(_bstr_t filterExp);`	Specifies the filter expression and filters the scene objects appropriately.
`HRESULT EnableDynamicBrushing(long on);`	Specifies whether selection events should be propagated between multiple visualization windows. If turned on, the selection expression of an object selected in one window is propagated to all other visualization window, and the corresponding objects are highlighted.
`HRESULT PublishOnWeb(_bstr_t fileName);`	Compresses visualization, data, and schema files into a single MTR archive for publishing on the web.
`HRESULT GenerateReport(_bstr_t fileName);`	Generates a HTML based report for the current visualization.
`HRESULT ShowDrillThroughDialog();`	Brings up a dialog which allows the user to set options for drill through and then invoke it. Uses the method **SendDrillThroughRequest** of **ISyncServer** to process the drill through request (that is, communication with Tool Manager). Alternatively, the plugin viz tool may wish to do its own drill through handling. It is up to the plugin viz tool to implement the drill through dialog and interprocess communication. **VizMain** implements a default that can be used in conjunction with Tool Manager.

Table 14-4 lists the **IVizCommon2** methods for the Automation interface.

Table 14-4 IVizCommon2 Methods—Automation Interface

Declaration	Description
`void SetSelectionExpression(LPCTSTR selectionExp);`	Changes the set of currently selected objects in the scene.
`CString GetFilterExpression();`	Returns the filter expression.
`void SetFilterExpression(LPCTSTR filterExp);`	Specifies the filter expression and filters the scene objects appropriately.
`void EnableDynamicBrushing(long on);`	Specifies whether selection events should be propagated between multiple visualization windows. If turned on, the selection expression of an object selected in one window is propagated to all other visualization window, and the corresponding objects are highlighted.
`void PublishOnWeb(LPCSTR fileName);`	Compresses visualization, data, and schema files into a single MTR archive for publishing on the web.
`void GenerateReport(LPCSTR fileName);`	Generates a HTML based report for the current visualization.
`void ShowDrillThroughDialog();`	Brings up a dialog which allows the user to set options for drill through and then invoke it. Uses the method **SendDrillThroughRequest** of **ISyncServer** to process the drill through request (that is, communication with Tool Manager). Alternatively, the plugin viz tool may wish to do its own drill through handling. It is up to the plugin viz tool to implement the drill through dialog and interprocess communication. **VizMain** implements a default that can be used in conjunction with Tool Manager.

IScatterviz

Table 14-5 lists the methods for the COM vtable interface that are available to interactively modify the Scatter Visualizer.

Table 14-5 IScatterviz Methods—COM vtable Interface

Declaration	Description
`HRESULT ShowDrillThroughColumnDialog();`	Brings up a dialog that allows you to select which columns you want to be significant on drill through. In other words you use the dialog to specify which columns could potentially be in the selection Expression (see **GetSelectionExpression** on page 170).
`HRESULT SetTrailType(TRAILCODE trailtype);`	Specifies what type of trails to use if any. Possible values of TRAILCODE are NO_TRAILS, LINE_TRAILS, FADE_OUT_TRAILS, and TUBE_TRAILS.
`HRESULT ShowAnimationPanel(long on);`	Indicates that if the argument is TRUE show the animation panel, otherwise hide it. If there is no animation panel, it cannot be shown.
`HRESULT CreateBoxSelection();`	Creates a bounding box which can be used to select entities.
`HRESULT UseSliderOnDrillThough(long on();`	Specifies whether or not the animation slider position should be used in constructing the drill through expression when you drill through.
`HRESULT Set2DSliderPositions(float slider1Pos, float slider2Pos);`	Allows you to programmatically alter the slider position. If the slider is one dimensional then the second argument is ignored.
`void PutShape(SHAPECODE pVal);`	Refers to the current entity shape type. Possible values for the Scatter Visualizer are: SHAPE_CUBE SHAPE_SPHERE SHAPE_DIAMOND SHAPE_BAR
`SHAPECODE GetShape();`	Returns the current shape type.

Table 14-5 (continued) IScatterviz Methods—COM vtable Interface

Declaration	Description
`HRESULT EnableSpinAnimation(long on);`	Specifies whether or not the automatic spin animation should be enabled in the viewer. If *on* is set to TRUE, then it is possible to set the animation continuously spinning by doing a rotation and releasing the left mouse before you are done dragging.
`HRESULT SetLabelSize(float size);`	Sets the scale size of object labels that appear in the scene. The larger the value of *size* the bigger the label.
`HRESULT SetAxisLabelSize(float size);`	Sets the scale size of axis labels that appear in the scene. This applies to the labels at the ends of the axes in Scatter and Splat Visualizers.
`HRESULT SetGridColor(float r, float g, float b);`	Sets the color for the grid.

`HRESULT SetGridSize(float axis1,`
`float axis2,`
`float axis3);`
	Sets the grid spacing in each of the 3 dimensions. Larger values indicate wider grid line spacing. A value of 0 means do not draw any grid lines in that axis dimension. If the value of all axes is 0, then no grid is drawn.

`HRESULT SetOrientation(ORIENTATIONCODE orientation);`	Specifies which of the three available orthogonal orientations the view should be set to. Possible values are: ORIENTATION_FRONT ORIENTATION_RIGHT ORIENTATION_TOP ORIENTATION_DEFAULT ORIENTATION_DEFAULT allows the viz tool to choose the orientation.
`HRESULT EnablePerspective(long on);`	Sets whether or not to use perspective in the 3D viewer. If TRUE then use perspective projection if FALSE then use orthogonal projection.
`HRESULT SaveAnimationPaneImage(_bstr_t filename);`	Takes a snapshot of the animation control and saves it to a file. This method is primarily used for debugging and testing.

Table 14-6 lists the methods for the Automation interface that are available to interactively modify the Scatter Visualizer.

Table 14-6 IScatterviz Methods—Automation Interface

Declaration	Description
`void ShowDrillThroughColumnDialog();`	Brings up a dialog that allows you to select which columns you want to be significant on drill through. In other words you use the dialog to specify which columns could potentially be in the selection Expression (see **GetSelectionExpression** on page 170).
`void SetTrailType(long trailtype);`	Specifies what type of trails to use if any. Possible values of TRAILCODE are NO_TRAILS, LINE_TRAILS, FADE_OUT_TRAILS, and TUBE_TRAILS.
`void ShowAnimationPane(long on);`	Indicates that if the argument is TRUE show the animation panel, otherwise hide it. If there is no animation panel, it cannot be shown.
`void CreateBoxSelection();`	Creates a bounding box which can be used to select entities.
`void UseSliderOnDrillThough(long on();`	Specifies whether or not the animation slider position should be used in constructing the drill through expression when you drill through.
`void Set2DSliderPositions(float slider1Pos, float slider2Pos);`	Allows you to programmatically alter the slider position. If the slider is one dimensional then the second argument is ignored.
`void SetShape(long nNewValue);`	Refers to the current entity shape type. Possible values for the Scatter Visualizer are: SHAPE_CUBE SHAPE_SPHERE SHAPE_DIAMOND SHAPE_BAR
`void GetShape();`	Returns the current shape type.
`void EnableSpinAnimation(long on);`	Specifies whether or not the automatic spin animation should be enabled in the viewer. If *on* is set to TRUE, then it is possible to set the animation continuously spinning by doing a rotation and releasing the left mouse before you are done dragging.
`void SetLabelSize(float size);`	Sets the scale size of object labels that appear in the scene. The larger the value of *size* the bigger the label.

Table 14-6 (continued) IScatterviz Methods—Automation Interface

Declaration	Description
`void SetAxisLabelSize(float size);`	Sets the scale size of axis labels that appear in the scene. This applies to the labels at the ends of the axes in Scatter and Splat Visualizers.
`void SetGridColor(float r, float g, float b);`	Sets the color for the grid.

`void SetGridSize(float axis1,`
`float axis2,`
`float axis3);`
	Sets the grid spacing in each of the 3 dimensions. Larger values indicate wider grid line spacing. A value of 0 means do not draw any grid lines in that axis dimension. If the value of all axes is 0, then no grid is drawn.

`void SetOrientation (long orientation);`	Specifies which of the three available orthogonal orientations the view should be set to. Possible values are: ORIENTATION_FRONT ORIENTATION_RIGHT ORIENTATION_TOP ORIENTATION_DEFAULT ORIENTATION_DEFAULT allows the viz tool to choose the orientation.
`void EnablePerspective(long on);`	Sets whether or not to use perspective in the 3D viewer. If TRUE then use perspective projection if FALSE then use orthogonal projection.
`void SaveAnimationPaneImage(LPCTSTR filename);`	Takes a snapshot of the animation control and saves it to a file. This method is primarily used for debugging and testing.

IScatter2

Table 14-7 lists two methods that are available to interactively modify the Scatter Visualizer.

Table 14-7 IScattert2 Methods—Automation Interface

Declaration	Description
`HRESULT EnableShadows(long on);`	Specifies whether to use projective shadow rendering.

`HRESULT SetShadowColor(float r,`
`float g,`
`float b);`
	Specifies the overall color for shadows.

ISplatviz

Table 14-8 lists the methods that are available to interactively modify the Splat Visualizer.

Table 14-8 ISplatviz Methods

Declaration	Description
`void ShowPickDragger(long on);`	Specifies whether the Splat Visualizer pick-dragger is shown, otherwise it is hidden (on=TRUE—shown, on=FALSE—hidden).
`long GetShowingPickDragger();`	Returns TRUE if the pick dragger is currently showing.
`HRESULT ShowAnimationPanel(long on);`	Specifies whether the animation panel is shown (on=TRUE—shown, on=FALSE—hidden). If there is no animation panel, it cannot be shown.
`HRESULT CreateBoxSelection();`	Creates a bounding box which can be used to select splats.
`HRESULT UseSliderOnDrillThrough(long on);`	Specifies whether the animation slider position should be used in constructing the drill through expression.
`HRESULT Set2DSliderPositions(float slider1Pos, float slider2Pos);`	Allows you to programmatically alter the slider position. If the slider is one dimensional then the second argument is ignored.
`void PutShape(SHAPECODE pVal);`	Sets the shape type for splats: SHAPE_CUBE, SHAPE_SPHERE, SHAPE_DIAMOND, SHAPE_BAR, SHAPE_LINEAR, SHAPE_GAUSSIAN, or SHAPE_TEXTURE
`SHAPECODE GetShape();`	Returns the current shape type.
`HRESULT EnableSpinAnimation(long on);`	Specifies whether the automatic spin animation should be enabled in the viewer. If *on* is set to TRUE, then it is possible to set the animation continuously spinning by doing a rotation and releasing the left mouse before you are done dragging.
`HRESULT SetAxisLabelSize(float size);`	Sets the scale size of axis labels that appear in the scene. This applies to the labels at the ends of the axes in Scatter Visualizer and Splat Visualizer and similar tools.

Table 14-8 (continued) ISplatviz Methods

Declaration	Description
`HRESULT SetGridColor(float r, float g, float b);`	Sets the color for the grid.
`HRESULT SetGridSize(float axis1, float axis2, float axis3);`	Sets the grid spacing in each of the three dimensions. Larger values indicate wider grid line spacing. A value of 0 means do not draw any grid lines in that axis dimension. If the value of all axes is 0, then no grid is drawn.
`HRESULT SetOrientation(ORIENTATIONCODE orientation);`	Specifies to which of three orthogonal orientations the view should be set Possible values are ORIENTATION_FRONT, ORIENTATION_RIGHT, ORIENTATION_TOP, and ORIENTATION_DEFAULT, where ORIENTATION_DEFAULT uses the orientation that the visualization tool thinks best.
`HRESULT EnablePerspective(long on);`	Sets whether to use perspective in the 3D viewer. If TRUE use perspective projection, if FALSE use orthogonal projection.
`HRESULT SaveAnimationPaneImage(_bstr_t fileName);`	Takes a snapshot of the animation control and saves it to a file. This method is primarily used for debugging and testing.

ISplatviz2

Table 14-9 describes a method that is available to interactively modify the Scatter Visualizer.

Table 14-9 IScattert2 Methods

Declaration	Description
`HRESULT GetUsingNominalSplats();`	Returns TRUE if a string valued attribute is mapped to color. In this case the splats will show a distribution of colors corresponding to unique string values. The type of splat used to show this distribution is called a nominal splat.

IMapviz

Table 14-10 lists the methods that are available to interactively modify the Map Visualizer.

Table 14-10 IMapviz Methods

Declaration	Description
`HRESULT ShowXYCoords(long on);`	Specifies whether (TRUE) or not (FALSE) to display the XY Coordinate grid.
`HRESULT UseRandomColors(long on);`	Specifies whether (TRUE) or not (FALSE) to assign a random color to each distinct geographical entity.
`HRESULT ShowAnimationPanel(long on);`	Specifies whether (TRUE) or not (FALSE) to show the animation panel. If there is no animation panel, it cannot be shown.
`HRESULT UseSliderOnDrillThrough(long on);`	Specifies whether (TRUE) or not (FALSE) the animation slider position should be used in constructing the drill through expression.
`HRESULT Set2DSliderPositions(float slider1Pos, float slider2Pos);`	Allows you to programmatically alter the slider position. If the slider is one dimensional, the second argument is ignored.
`HRESULT EnableSpinAnimation(long on);`	Specifies whether or not the automatic spin animation should be enabled in the viewer. If *on* is set to TRUE, then it is possible to set the animation continuously spinning by doing a rotation and releasing the left mouse before you are done dragging.
`HRESULT EnablePerspective(long on);`	Sets whether or not to use perspective in the 3D viewer. TRUE specifies perspective projection. FALSE specifies orthogonal projection.
`HRESULT SaveAnimationPaneImage(_bstr_ fileName);`	Takes a snapshot of the animation control and saves it to a file. This method is primarily used for debugging and testing.

IEviviz

Table 14-11 lists the methods that are available to interactively modify the Evidence Visualizer.

Table 14-11 IEviviz Methods

Declaration	Description
`HRESULT SetAttributeOrder(ORDERCODE on);`	Allows you to programmatically change the way the attributes are listed in the evidence visualizer. Possible values are: ORDER_BY_IMPORTANCE (the default) and ORDER_BY_DATABASE (use the database ordering).
`HRESULT SubtractMinEvidence(long on);`	Specifies whether to subtract the minimum amount of evidence from each value in the bars mode (see the *MineSet Enterprise Edition Reference Guide*.)
`HRESULT SelectAttributeValue(_bstr_t attribute, _bstr_t value);`	Programmatically selects one of the attribute values and resulting probabilities.
`HRESULT EnableEvidenceMode(long on);`	Sets whether to show an evidence representation, or a probability representation. For the Evidence Visualizer, this means switching between showing the evidence cakes on the left, or the probability pies. See the *MineSet Enterprise Edition Reference Guide* for additional information about these modes.
`long GetEvidenceMode();`	Returns TRUE if the classifier visualizer is in the Evidence mode.
`HRESULT SetNominalOrder(ORDERCODE ordercode);`	Specifies how to order attribute values. Possible values are: ORDER_ALPHABETICALLY ORDER_BY_WEIGHT, ORDER_BY_LABEL
`HRESULT SetViewerType(VIEWERCODE type);`	Specifies whether to use the landscape style viewer or examiner viewer to display the scene. VIEWERCODE values are LANDSCAPE.VIEWER and EXAMINER.VIEWER Drilling up and down is not available in the landscape viewer mode.
`HRESULT SetLaplace(long on, float value);`	Specifies whether or not to use Laplace correction. The second argument is the amount of Laplace correction to use. If the second argument is -1, then a default based on the data is used.

Table 14-11 (continued) IEviviz Methods

Declaration	Description
`HRESULT SaveLabelPaneImage(_bstr_t fileName);`	Takes a snapshot of the label probability pane and saves it to a file. This method is primarily used for debugging and testing.
`HRESULT SelectLabelValue(_bstr_t value);`	Programmatically selects one of the label values. After this method has been invoked, one of the classes in the probability pane on the right should be selected.
`HRESULT SetWeightThresh(float percent);`	Filters out objects with low weight. When the threshold is 0% all values are shown, no matter how small their weight; if *value* is set to 100% all values are removed from the scene.
`HRESULT SetDetailThresh(float percent);`	Changes the amount of detail in the scene. As the threshold is lowered, all attributes with lower importance values are removed from the scene.

IDtableviz

Table 14-12 lists the methods that are available to interactively modify the Decision Table Visualizer.

Table 14-12 IDtableviz Methods

Declaration	Description
`HRESULT EnableEvidenceMode(long on);`	Specifies whether to show an evidence representation, or a probability representation. In the Decision Table Visualizer, this determines whether the cakes show evidence (normalized conditional probabilities), or straight probabilities. See the *MineSet Enterprise Edition Reference Guide* for additional information about these modes.
`HRESULT SetNominalOrder(ORDERCODE ordercode);`	Specifies how to order attribute values. Possible values are: ORDER_ALPHABETICALLY, ORDER_BY_WEIGHT, and ORDER_BY_LABEL
`HRESULT SetViewerType(VIEWERCODE type);`	Specifies whether to use the landscape style viewer or examiner viewer to display the scene. Possible VIEWERCODE values are LANDSCAPE.VIEWER and EXAMINER.VIEWER.
`HRESULT SetLaplace(long on, float value);`	Specifies whether or not to use the Laplace correction. The second argument is the amount of Laplace correction to use.
`HRESULT SaveLabelPaneImage(_bstr_t fileName);`	Takes a snapshot of the label probability pane and saves it to a file. This method is primarily used for debugging and testing.
`HRESULT SelectLabelValue(_bstr_t value);`	Programmatically selects one of the label values. After this method has been invoked, one of the classes in the probability pane on the right should be selected.
`HRESULT SetWeightThresh(float percent);`	Filters out objects with low weight. When the threshold is 0% all values are shown, no matter how small their weight; if *value* is set to 100% all values are removed from the scene.
`HRESULT SetDetailThresh(float percent);`	Changes the amount of detail in the scene. As the threshold is increased, more detail is achieved through globally drilling down on all the cake charts until the maximum level of detail is reached at 100%.

IDtableviz2

Table 14-13 describes a method that is available to interactively modify the Scatter Visualizer.

Table 14-13 IScattert2 Methods

Declaration	Description
`HRESULT ShowHierarchyDialog();`	Causes the Hierarchy Dialog box to be displayed. This is relevant only for decision tables and regression tables.

ITreeviz

Table 14-14 lists the methods that are available to interactively modify the Tree Visualizer.

Table 14-14 ITreeviz Methods

Declaration	Description
`HRESULT ShowMarksDialog();`	Brings up a dialog which allows you to mark nodes in the tree with colored flags.
`HRESULT ShowOverviewDialog();`	Brings up a dialog which shows a two dimensional view of the entire tree.
`HRESULT ShowSearchDialog();`	Brings up a dialog which lets you search for different attribute values for all the nodes in the tree hierarchy.
`HRESULT Navigate(TREE_GOCODE code);`	The equivalent of the navigation buttons on the right-hand strip of the Tree Visualizer Viewer. *Code* can take on any of the following values: TREE_HOME—Resets the camera to the home position. TREE_SETHOME—Sets a new home position. TREE_VIEWALL—Positions the camera so the entire scene is in view. TREE_BACK—Moves the position one step back in the navigation history (undo). TREE_FORWARD—Moves the position one step forward in the navigation history (redo). TREE_PARENT—Updates the selection to be the parent of the currently selected node, repositioning the camera to view it. TREE_LEFT—Updates the selection to be the left sibling of current node. TREE_RIGHT—Updates the selection to be the right sibling of current node. TREE_FIRST—Updates selection to be the first (left) child of current node. TREE_LAST—Updates selection to be the last (right) child of current node.
`HRESULT IsValidNavigation(TREE_GOCODE c, long *enabled);`	Sets *enabled* to TRUE if the corresponding TREE_GOCODE is currently available. Otherwise sets it to FALSE.

Table 14-14 (continued) ITreeviz Methods

Declaration	Description
`HRESULT ShowBaseHeights(long on);`	Toggles whether or not to show the base heights in the scene. If base heights are not shown, they appear simply as a flat box on the ground plane.
`HRESULT ShowMarksFlags(long on);`	Specifies whether to show the marks flags, if there are any.
`void SetZeroStyle(DRAW_STYLE style);`	Allows you to set how zero values are displayed. DRAW_STYLE style can take one of the following values: SOLID, HIDDEN or OUTLINE.
`DRAW_STYLE GetZeroStyle();`	Returns the current setting for how zero values are displayed using the variable DRAW_STYLE *style. This value will be either SOLID, HIDDEN, or OUTLINE.
`HRESULT NormalizeSubtree();`	Recomputes the heights for all nodes in the tree using the currently selected node as the 1.0 value.
`HRESULT SaveInitialHierarchy(b_str_t fileName);`	Primarily used for debugging and testing.
`HRESULT SaveViewedHierarchy(b_str_t fileName);`	Primarily used for debugging and testing.
`HRESULT SaveOverviewImage(_bstr_t fileName);`	Saves a snapshot of the overview dialog image. This method is primarily used for debugging and testing.
`HRESULT SaveSearchPaths(_bstr_t fileName);`	Saves a list of all the currently shown search paths. This method is primarily used for debugging and testing.

Further Reading and Acknowledgments

Some datasets were taken from the UCI repository (Merz, C. J., and Murphy, P. M. (1996). UCI Repository of machine learning databases, Irvine, CA: University of California, Department of Information and Computer Science) found at: http://www.ics.uci.edu/~mlearn/MLRepository.html.

Further Reading

Several papers describing the technology used in MineSet are available at: http://www.sgi.com/software/mineset/mineset_data.html.

An excellent, non-technical introduction to data mining techniques is:

- Michael Berry and Gordon Linoff. *Data Mining Techniques*. New York: John Wiley & Sons, 1997. ISBN 0-471-17980-9. See http://www.data-miners.com/.

A comparative study of data mining tools, including MineSet, was done by the Two Crows Corporation. It contains a good introduction to data mining.

- Two Crows Corporation. *Data Mining: Products, Applications & Technologies.* Ordering information is available at http://www.twocrows.com.

A paper describing MLC++, the underlying analytical engine used in MineSet, is described in:

- Kohavi, R., Sommerfield, D., Dougherty, J., *Data Mining using MLC++, a Machine Learning Library in C++*. International Journal of Artificial Intelligence Tools, Vol. 6, No. 4, 1997, p. 537-566. See http://robotics.stanford.edu/users/ronnyk/.

A general and easy-to-read introduction to machine learning is:

- Weiss, S. M., and C. A. Kulikowski. *Computer Systems that Learn*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1991.

A general comparison of algorithms and descriptions is provided in:

- Taylor, C., D. Michie, and D. Spiegalhalter. *Machine Learning, Neural and Statistical Classification*. Paramount Publishing International, 1994.

An easy-to-read introduction to decision tree induction is:

- Quinlan, J. R. *C4.5: Programs for Machine Learning*. Los Altos, CA: Morgan Kaufmann Publishers, Inc., 1993.

An excellent book on decision trees from a statistical perspective is:

- Breiman, L., J. H. Friedman, R. A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Wadsworth International Group, 1984.

A good edited volume of machine learning techniques is:

- Dietterich, T. G. and J. W. Shavlik (Eds). *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990.

A summary of accuracy estimation techniques is given in:

- Kohavi, R. *A study of cross-validation and bootstrap for accuracy estimation and model selection.* In Proceedings of the 14th International Joint Conference on Artificial Intelligence, edited by C. S. Mellish. Morgan Kaufmann Publishers, Inc., 1995. Available at http://robotics.stanford.edu/users/ronnyk/.

The following papers describes the decision table metaphor:

- Kohavi, R., *The Power of Decision Tables*. In The European Conference on Machine Learning, 1995.
- Becker, B. *Visualizing Decision Tables*. In IEEEs Proceedings of Information Visualization, 1998.

A good reference to a paper explaining that no classifier can be "best" is:

- Schaffer, C. *A conservation law for generalization performance.* In Machine Learning: Proceedings of the Eleventh International Conference, 259-265. Morgan Kaufmann Publishers, Inc., 1994. Available at http://wwwcs.hunter.cuny.edu/faculty/schaffer/papers/list.html.

Further Readings About Option Trees

MineSet uses an advanced version of the Option Trees described in:

- Ron Kohavi and Clayton Kunz. *Option Decision Trees with Majority Votes*. Machine Learning: Proceedings of the Fourteenth International Conference", Morgan Kaufmann Publishers, Inc., 1997. (See http://robotics.stanford.edu/users/ronnyk). The option trees used in MineSet average the predictions and do not simply vote them as described in this paper. Option Trees were first introduced by Wray Buntine in his thesis *A Theory of Learning Classification Rules*, 1992, School of Computing Science, University of Technology, Sydney.

Further Readings About the Evidence Inducer

The following paper describes the wrapper method used to select the features for the Evidence Classifier:

- Kohavi, R., Sommerfield, D. (1995). *Feature Subset Selection Using the Wrapper Model: Overfitting and Dynamic Search Space Topology.* The First International Conference on Knowledge Discovery and Data Mining, pp. 192-197. Available at: http://robotics.stanford.edu/users/ronnyk/.

An excellent introduction to the Evidence Classifier (Naive-Bayes) is:

- Kononenko, I. (1993). *Inductive and Bayesian Learning in Medical Diagnosis*. Applied Artificial Intelligence, pp. 7:317-337.

The following paper describes conditions under which the Evidence Inducer is optimal:

- Domingos, P., Pazzani, M., *Beyond Independence: Conditions for the Optimality of the Simple Bayes Classifier.* Machine Learning, Volume 29, No. 2/3, Nov/Dec 1997, pp. 103-130.

The following paper describes the use of the wrapper method in the Evidence Inducer:

- Kohavi, R., John, G., *Wrappers for Feature Subset Selection*. In Artificial Intelligence Journal, special issue on relevance, Vol. 97, Nos 1-2, pp. 273-324.

The following paper describes the Laplace correction option:

- Cestnik, B. (1990). *Estimating Probabilities: A Crucial Task in Machine Learning*. Proceedings of the Ninth European Conference on Artificial Intelligence, pp. 147-149.

The following paper describes the automatic Laplace correction used in MineSet:

- Kohavi R., Becker B., and Sommerfield D., *Improving Simple Bayes, European Conference on Machine Learning*, 1997 (poster). Available at: http://robotics.stanford.edu/users/ronnyk/.

The following paper describes the Evidence Classifier (Naive-Bayes):

- Langley, P., Iba, W., Thompson, K. (1992). *An Analysis of Bayesian Classifiers*. Proceedings of the Tenth National Conference on Artificial Intelligence, pp. 223-228. Available at: http://www.isle.org/~langley/pubs.html.

Simple Bayesian Classifier. To appear in *Lecture Notes in Computer Science*: *Issues in the Integration of Data Mining and Data Visualization*, Springer Verlag, 1998.

The following books describe the Evidence Classifier:

- Good, I. J. *The Estimation of Probabilities: An Essay on Modern Bayesian Methods*. MIT Press, 1965.
- Duda, R., Hart, P. *Pattern Classification and Scene Analysis*, Wiley, 1973.

The following paper shows that while the conditional independence assumption can be violated, the classification accuracy of the evidence classifier (called Simple Bayes in this paper) can be good:

- Domingos P., Pazzani M (1996). *Beyond Independence: Conditions for the Optimality of the Simple Bayesian Classifier*. Machine learning, Proceedings of the 13th International Conference (ICML '96), pp. 105-112. Available at http://www.ics.uci.edu/~pedrod/.

Further Readings About the Splat Visualizer

The following paper describes and provides further references for the technical details of the Splat Visualizer.

- Becker, Barry G, *Volume Rendering for Relational Data,* to appear in Proceedings of Information Visualization '97, IEEE Computer Society Press, Los Alamitos CA, October 19-24, 1997.

The following paper explains how to use Gaussian splats for volume rendering.

- Westover, Lee, *Footprint Evaluation for Volume Rendering* in Proceedings of SIGGRAPH '90, Vol. 24, No. 4, pages 367-376).

Acknowledgments

The iris database was originally used in Fisher, R. A. 1936. The use of multiple measurements in taxonomic problems. *Annals of Eugenics* 7(1):179-188. It is a classical problem in many statistical texts.

The breast cancer database was obtained from Dr. William H. Wolberg, L. Mangasarian, and W. H. Wolberg. Cancer diagnosis via linear programming. *SIAM News* 23(5):1 & 18. University of Wisconsin Hospitals, Madison, September 1990.

The data for the mushroom sample file comes from: *Audubon Society Field Guide to North American Mushrooms*. New York: Alfred A. Knopf, 1981.

The data on congressional voting was taken from the *Congressional Quarterly Almanac, 98th Congress, 2nd session 1984, Volume XL*, Congressional Quarterly Inc.: Washington, D.C., 1985.

The adult dataset was derived from the US Census Bureau survey in 1994 (http://www.census.gov/ftp/pub/DES/www/welcome.html).

Index

B

C

D

E

F

G

K

L

M

N

O

P

Q

T

U

V