Crimson

version 2.1 beta

Installation and User Guide

By Stephen Fisher, Susan Davidson, and Junhyong Kim.
Biology and Computer Science Departments, University of Pennsylvania.

==========

Table of Contents:

==========

Overview:

Crimson facilitates the extraction of sub-trees from very large phylogenetic trees. Trees are loaded into a shared database and sampled according to schemes controlled by the user. Comprehensive graphical dialogs allow users to easily manage and query trees in the database. Queries can be stored in the database to be shared with other users and moved between databases. A command line interface enables users to write their own functions and scripts to manage the database, manipulate the trees and queries, and automate any of the built-in functions.

The application allows users to connect to Oracle or MySQL databases. Once connected to a database, the program will register the trees that are contained in the database as well as any stored queries and data models. If no tables exist in the database, the application will automatically create the necessary tables. The user can then use the graphical tree manager to load new trees, add or remove data partitions, delete existing trees, add notes, etc. In order to facilitate the management of trees and data partitions, Crimson stores information about the data models used and can link data models to trees and data partitions. A graphical query manager allows users to manipulate existing queries and add new queries. When users create queries they have the option of publishing the queries to the database so that they are available in future sessions and to other users. It is also possible to export queries to text files so that they can be shared between databases. Users are provided with graphical means to run existing queries and to easily set the parameters associated with the running of queries. For example, a user can specify that a query run multiple times to repeatedly sample from a tree using the same query specifications. This might be useful to users who want to take repeated random samples from a tree. The application also provides simple methods to extract a tree structure, an entire tree (containing all data partitions), or a specific data partition from the database. In order to facilitate the viewing of trees, Crimson contains routines to easily display exported trees in the 3D tree viewer Walrus (see Walrus 3D Viewer, page 31).

This application has been tested on Windows (2000 and XP), Mac OS (10.5), and Linux (RHEL 5) systems and should run on any system that contains Java 1.5 or later. Both Oracle and MySQL databases can be used, and future versions will allow for connecting to other databases such as IBM DB2 and Microsoft SQL.

Extensive Java API documentation can be found in the “documentation” subfolder.

==========

Quick Start:

This is not meant to be exhaustive but rather to give you a few pointers to get you started, if you don't wish to read the more extensive documentation below.

- To run Crimson, launch "crimson.sh" on a Mac or Linux system. Use "crimson.bat" on a Windows system.
- When run, Crimson will open a graphical window and a command line interface. You can use either interface interchangeably. Note that all output will be displayed in the Messages window in the GUI.
- To connect to the SDSC Oracle database run the command "connectCIPRES()" from the Crimson command line. It will take about 30 seconds to complete the connection and load the tree and query information from the database.
- To create and run a query, select "New…" from the Query menu in the GUI or type "newQuery("<tree ID>") at the command line. Note that you must be connected to a database. Assuming you are connected to the CIPRES database, which contains a tree called "TREE-0" then you can use the command 'newQuery("TREE-0")'.
- See the example files ("exBatchScript*.py") for use cases that can be run via the command line. While these files can be used when running Crimson without the command line (similar to the "-n" option in Paup), the commands in these files can also be entered directly into the command line.
- Review the file "startup.py" to see a list of commands and for more scripting examples. You can also enter "help()" at the command line for a list of Crimson commands. To get more documentation on a specific command use "help(<command>)" such as "help(queryManager)".
- Open "index.html" in the documentation directory to view Crimson's Java API. The API is accessible via the command line. Again, view the startup.py file for examples of accessing the Java API via the command line.

Note that Crimson needs to build an internal representation of the tree structure, the first time a tree is queried. For a million leaf tree this can take up to a minute. This only happens once per tree per session. So if you repeatedly query a tree, the tree will not be rebuilt. However, if you quit Crimson and start again, then the tree will be rebuilt the first time it is used in the new Crimson session.

==========

Installing and Running Crimson:

In order to run Crimson, it is necessary to first install java version 1.5 or later. If not already installed, java can be downloaded from the Sun java website.

http://www.java.com/en/download/manual.jsp

Installation:

Linux

Setup:

1. Untar the package in the desired directory.
   $ tar xvfj crimson.tar.bz2
2. Change to the crimson directory.
   $ cd crimson
3. Adjust the attributes for the application executable.
   $ chmod a+x crimson.sh

Running:

1. Change directories into the “crimson” directory and run "./crimson.sh".

Windows (2000 or later)

Setup:

1. Unzip the package in the desired directory (WinZip or some other pkzip based application will suffice).

Running:

1. From within the “crimson” folder double-click the "crimson.bat" application.

Macintosh OS X (10.4 or later)

*Setup from *.dmg version:*

1. Open the disk image by double clicking on it from the Finder.
2. Copy the Crimson folder to somewhere on your local drive.

Running:

1. Open /Applications/Utilities/Terminal
2. Change directories into the “Crimson” directory and run "./crimson.sh".

*Setup from *.tar.bz2 version:*

1. Open /Applications/Utilities/Terminal
2. Using the terminal, follow the Linux instructions

Running:

1. Change directories into the “crimson” directory and run "./crimson.sh".

Command Line Arguments

The Crimson executable contains various optional arguments.

-dos: Sets 'ISDOS' flag to true. Used to establish command interface for launching shell commands. This flag is automatically set by 'crimson.bat'.

-batch: Sets 'ISBATCH' flag to true. Inhibits the loading of the interactive interface. This will also disable X11.

-nox: Sets 'NOX' flag to true. When true, will restrict the loading of various GUI interfaces. This is necessary on some systems that do not contain any graphical display server (ex X11 on Linux systems).

Windows usage

> crimson.bat [-batch filename | -nox [filename] | filename]

Linux / Macintosh usage

$ crimson.sh [-batch filename | -nox [filename] | filename]

Batch Mode

Both crimson.sh (Linux/Mac) and crimson.bat (Windows) allow for the inclusion of one command line argument which is a Crimson python file that will be run. The command line argument (ie python file) is loaded into the CLI at startup using the 'loadPython()' script. When the '-batch' argument is present, Crimson will automatically quit after running the remaining batch file; that is, it will not launch the interactive interface.

For example the following command will run the script 'exBatchScript.py' in batch mode on a Linux machine, quitting Crimson upon completion of the script.

crimson.sh –batch exBatchScript.py

Crimson allows users to store queries in the database. There are a number of queries stored in the SDSC Oracle server.

Four example batch files are included with the installation package.

exBatchScript.py - This script will log into the SDSC Oracle server, create a query that will randomly sample 1000 leaves from the tree 'TREE-0'. The query will run three times and save the resulting trees to the NEXUS files called sampTree1000_0, sampTree1000_1, and sampTree1000_2. To run this batch file on a Windows machine:

exBatchScript-2.py – This script will run the query "L10.P03.R05" from the SDSC database, creating a 10 leaf sub-tree, and saving the output as a phylip file. The phylip file is used by RAxML to reconstruct the sub-tree.

exBatchScript-3.py - This script will run the query "L10.P03.R05" from the SDSC database, creating a 10 leaf sub-tree, and saving the output as a NEXUS file. The NEXUS file will contain a Paup block instructing paup to use neighbor joining to reconstruct the tree. The query is then run a second time, including the same leaves and containing a Paup block using parsimony for the reconstruction. The true sub-tree topology will not be included in the NEXUS files but will be output separately as a newick string, for comparison to the reconstructed trees.

exBatchScript-4.py - This script will generate NEXUS, phylip, and newick files for 40 of the 10 leaf queries stored in the SDSC database.

Walrus 3D Viewer

Crimson contains hooks for accessing Walrus from the built-in menu and tree manager. These features are enabled when Crimson is launched with the "crimson-walrus" script.

The Walrus 3D Viewer requires the Java 3D library to run. Users can also download the Java 3D installation files from:

http://java.sun.com/products/java-media/3D/

Windows usage

> crimson-walrus.bat [filename]

Linux / Macintosh usage

$ crimson-walrus.sh [filename]

Miscellaneous Notes

When processing large trees, Java must be run with 512 or 1024 MB of memory. This is set to 1024 by default in crimson.bat and crimson.sh. To change this value, adjust the argument:

-Xmx1024m

When using MySQL, it is recommended that servers be run in "strict" mode:

--sql-mode='STRICT_ALL_TABLES'

Users also need to set the 'max_allowed_packet' for MySQL server and client to allow for loading of large trees:

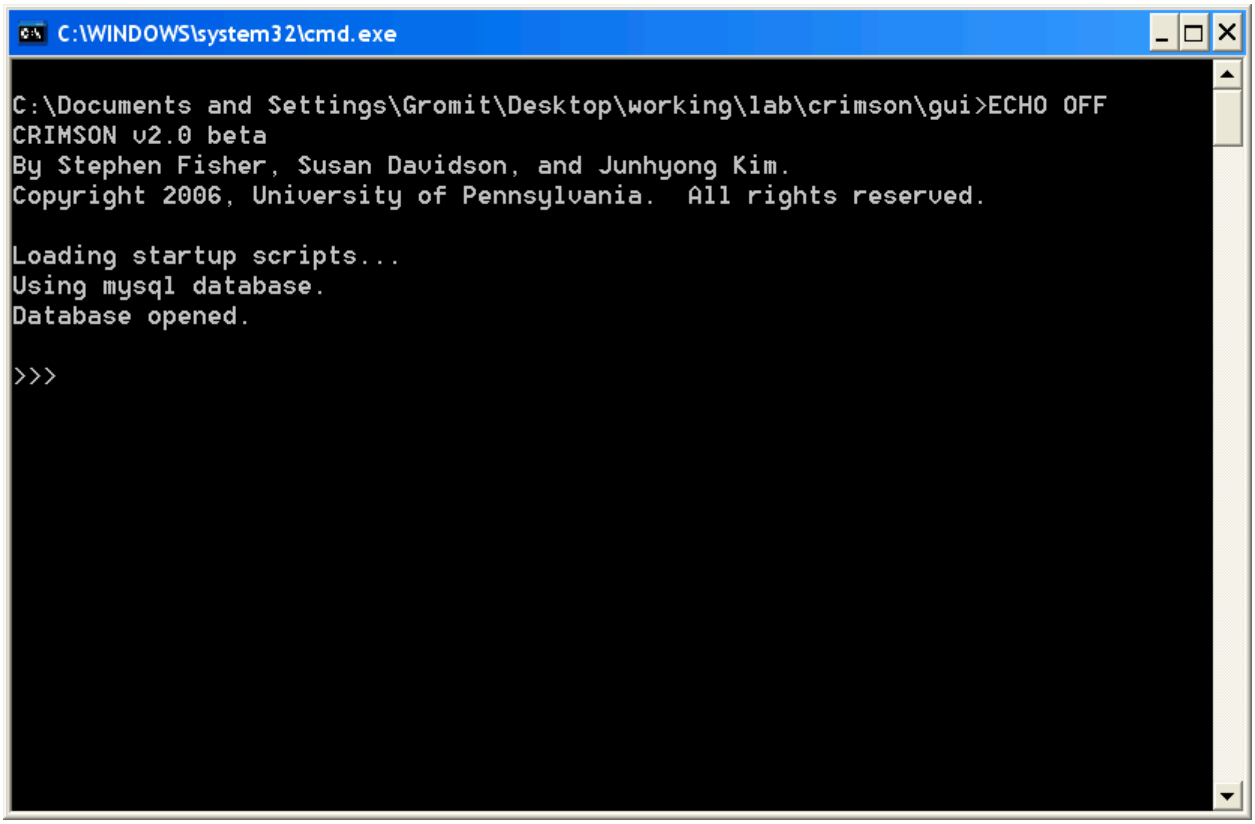
-max_allowed_packet=64M

==========

Command Line Interface:

The Command Line Interface (CLI) is based on the python scripting language and will accept most valid Python commands. The CLI can be used to load, modify, view, and save data. The CLI also provides access to the internal data structures containing the tree, partition, model, and query data. Thus, using Python commands, users can create their own scripts to automate tasks and expand upon built-in data operations. The use of scripts and user-defined data operations are discussed further in the section Scripting Language (page 23).

When the program starts it automatically loads the file called 'startup.py', containing various scripts to facilitate the use of common built-in functions. The user can modify this file to include their own scripts or to load addition files at startup.

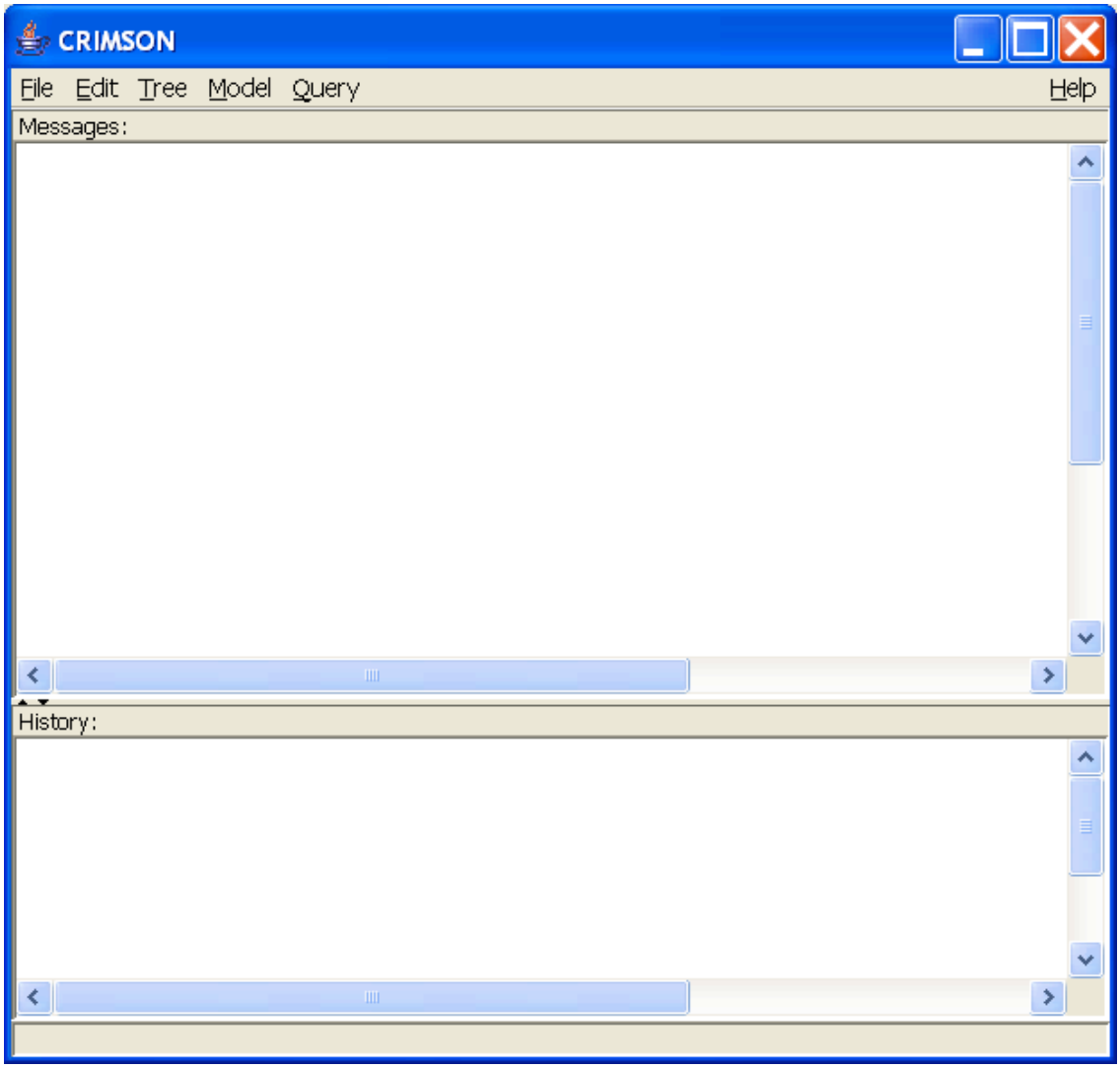


==========

Graphical User Interface:

The Graphical User Interface (GUI) can be used to perform many of the built-in operations. The GUI has a main window which contains various menus, a "Messages" window and a "History" window. The 'Messages' window will display all status, warning, and error messages. The user can disable this window via the Edit menu, in which case all output will be displayed in the CLI. The verbosity of the feedback (status, warning, and error) can be set with the "setVerbose()" command.

The 'History' window contains the CLI equivalent for all commands initiated via the GUI. Thus, the History window will contain a log of all GUI derived commands. The user can copy and paste the commands in the History window directly into the CLI or into a file to be loaded into the CLI. Future versions of the GUI will include the ability to directly load or save command histories, and convert sequences of commands from the history into macros accessible through the CLI.



Menu Options:

File:
- **Set Database Type**
- **Open Database...**
- **Close Database**
- -
- **Test Database Connection**
- -
- **Quit**

Edit:
- **Copy History**
- **Clear History**
- **Select All History**
- -
- **Enable Message Window**
- **Clear Messages**

Tree:
- **Load...**
- **Append...**
- **Export Tree...**
- -
- **Stats Tree...**
- **View Tree...**
- **3D View Tree...**
- -
- **Refresh Tree List**
- **Manage**
- -
- **Delete**

Model:
- **New…**
- -
- **Refresh Model List**
- **View**
- **Manage**
- -
- **Delete**

Query:
- **New...**
- **Load From DB**
- **Load All From DB**
- **Publish To DB…**
- -
- **Import...**
- **Export...**
- -
- **View**
- **Manage**
- -
- **Delete**

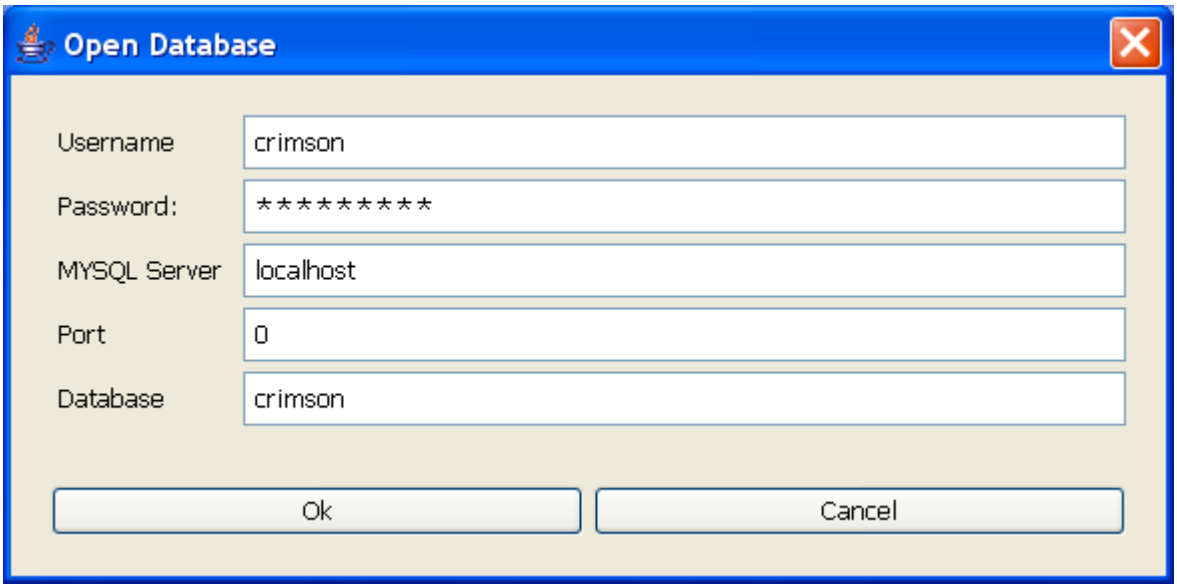
Help:
- **Script Commands**
- **API documentation**
- -
- **About**

==========

Usage Examples:

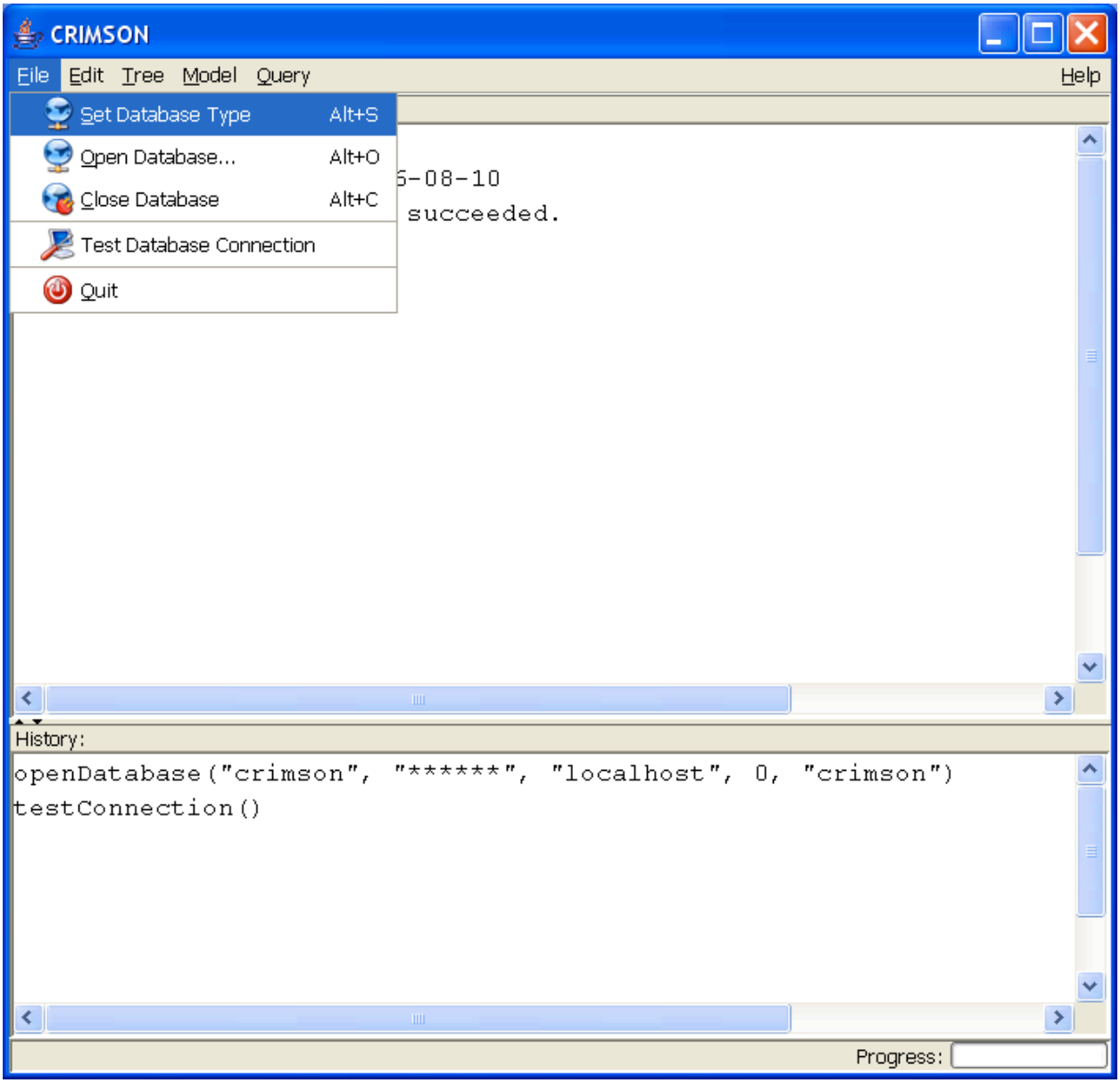
The application directory contains multiple example batch files (see “exBatchScript.py”). These files provide further usage examples and scripting hints.

1. Open database. When Crimson is first started an 'Open Database' dialog is presented. The application will remember the username, server, port, and database values across sessions. However, the password is only stored for the duration of the session. Users can use the File Menu to close an existing database and open another database at any time. When a database is opened, Crimson tests for the existence of various application specific tables (“TREES,” “PARTITIONS,” “PART_DATA,” “MODELS,” and “QUERIES”) and if they don’t already exist, it will create them. See the section Schema (page 27) for more information about the actual tables created and used by Crimson.

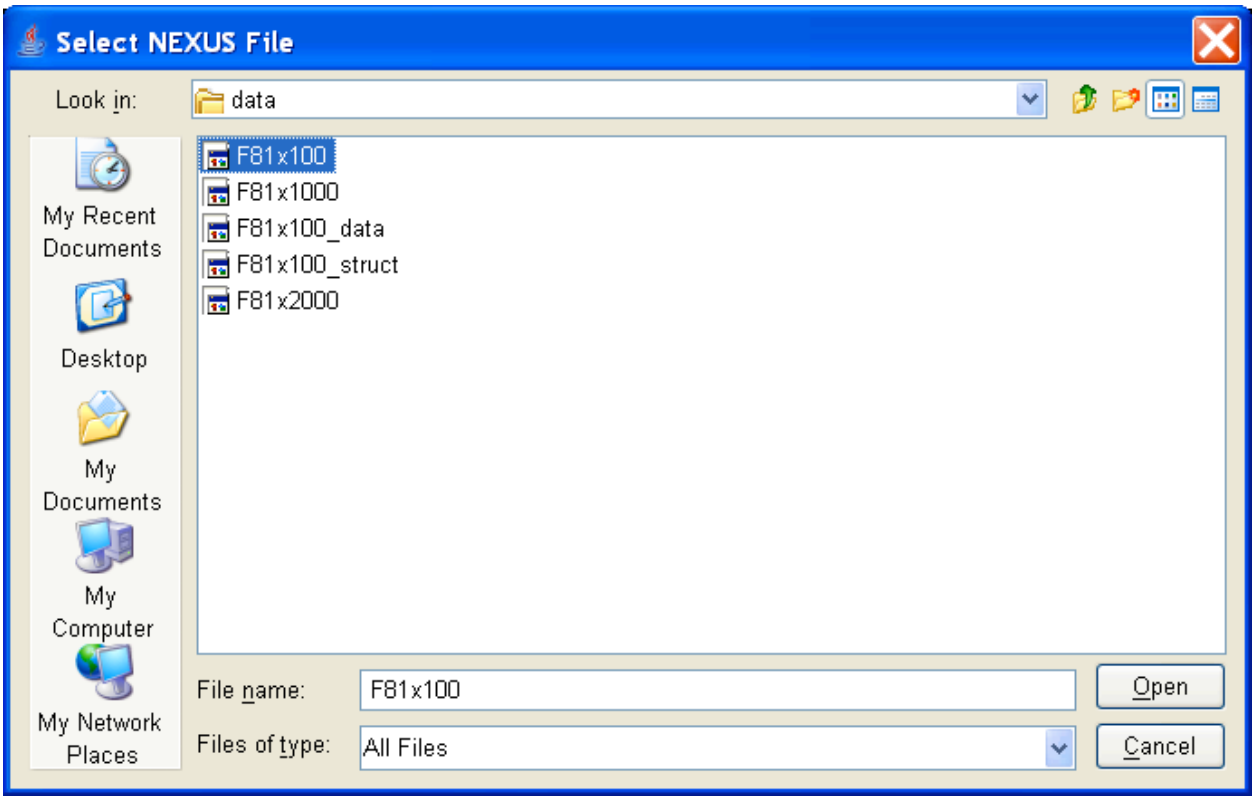


2. Test database connection. Test the connection to the database by selecting the 'File -> Test Database Connection' menu item. If the test is successful, then a message similar to the following will be printed in the message window.

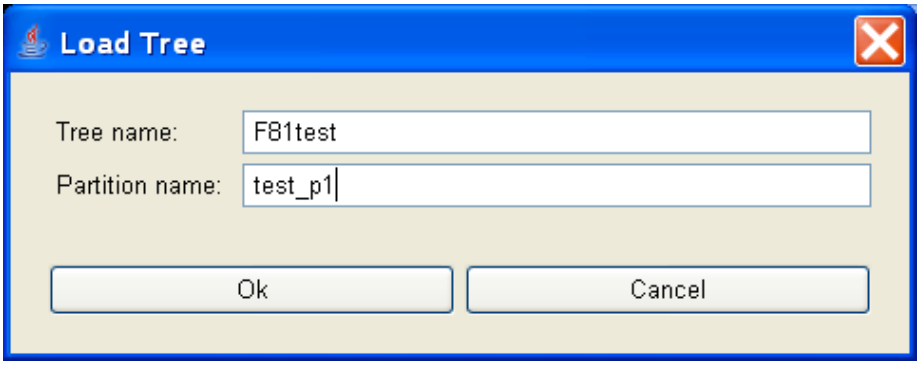
> The database date is 2005-04-01 15:30:00.0
> Database connection test succeeded.



3. Load tree. Choose the 'Tree -> Load...' menu item to load a tree. A file selection dialog will be presented. Choose the 'F81x100' file from the data directory.



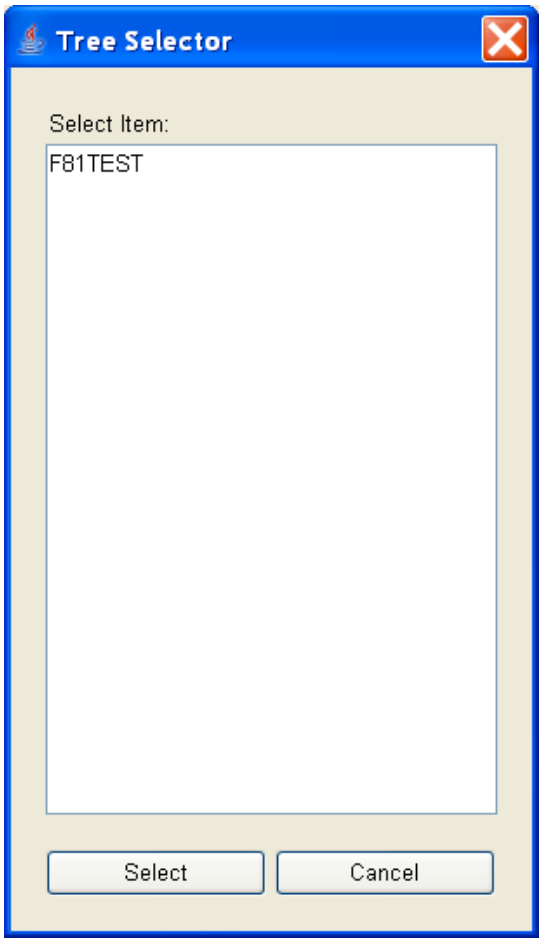
You will now be presented with a dialog to name the tree and partition. Changes the tree name to 'F81test' and the partition name to 'test_p1'. Note that every partition must have a unique name.



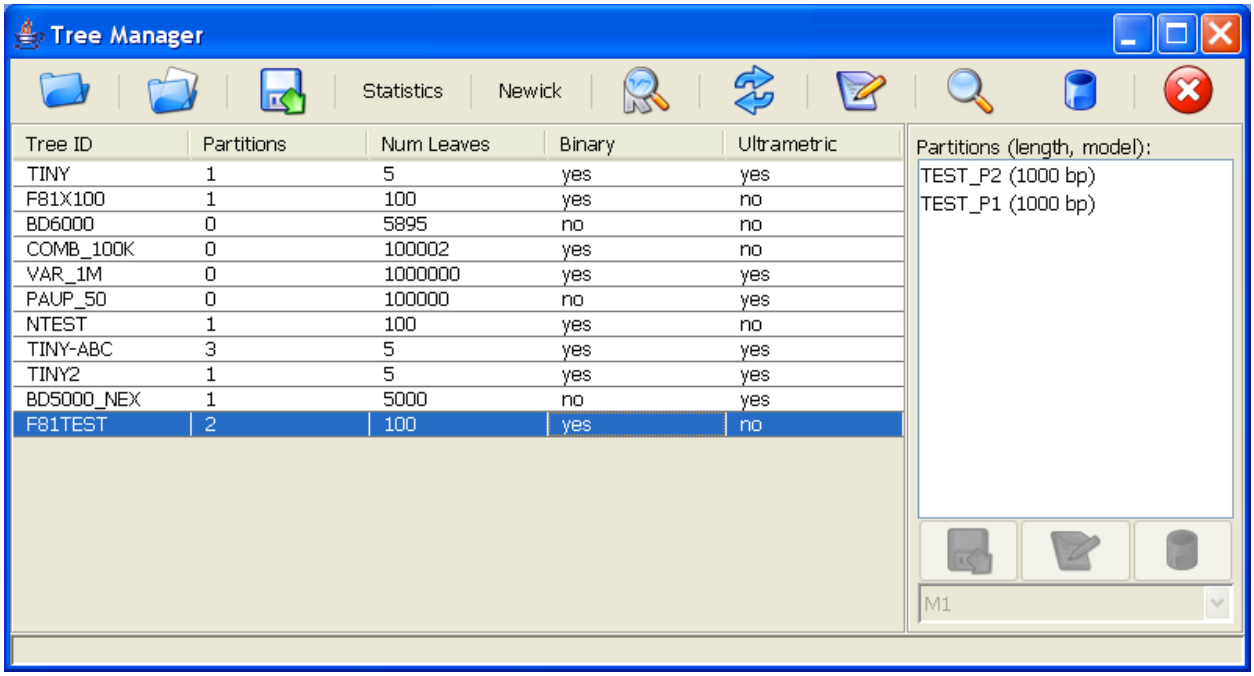
If loaded successfully you should see the following in the Messages window.

```
Building tree.
Updated TREES table.
Updated PARTITIONS table.
Finished loading CHARACTERS data.
Finished loading CRIMSON data.
```

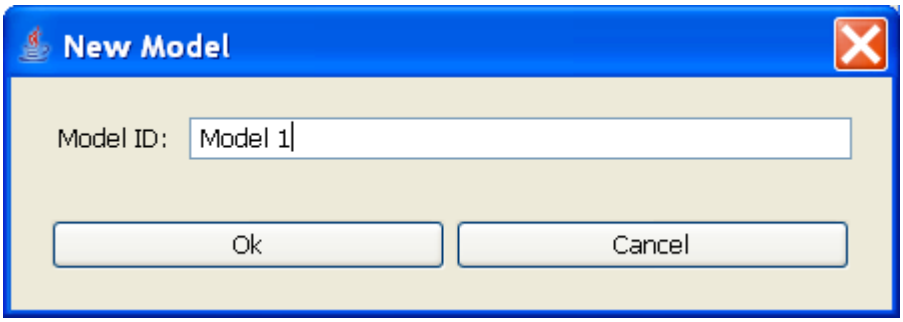
4. Append partition. Choose 'Tree -> Append...' to append a partition to the tree. A Tree Selector dialog will open and from this dialog, choose the tree to append, 'F81TEST' in this example. Next you will be presented with a NEXUS file choose. Select the file 'F81x100_data' from the data directory. After choosing the file, you will again be presented with a dialog to specify the name of the partition. Change the name to 'TEST_P2'. Note that every partition must have a unique name.



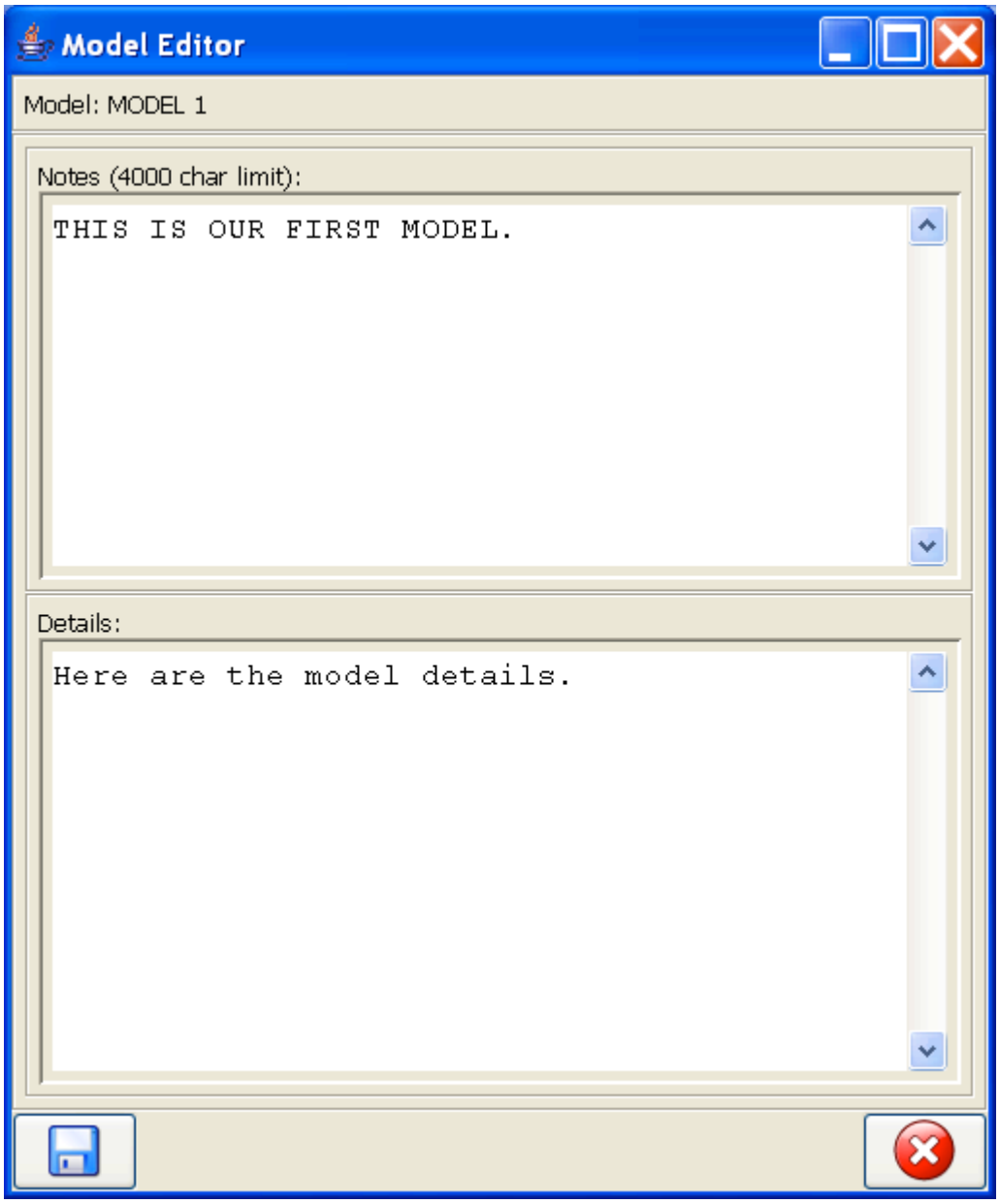
5. Manage tree. Open the Tree Manager dialog to view information about the tree (select the 'Tree -> Manage' menu item). From this dialog you can see that the tree 'F81TEST' has 2 partitions, 100 leaves, and that it's a binary tree that is not ultrametric. In the partitions panel, each partition is listed with its sequence length and model association. From this dialog, trees and partitions can be loaded or deleted. You can view and edit the notes associated with each tree or partition. You can also change the model associated with each partition.



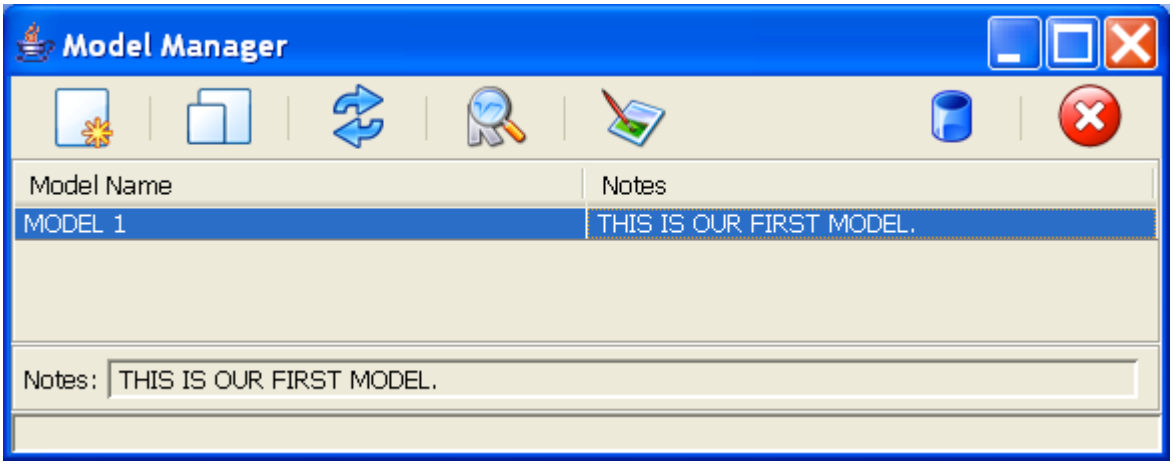
6. Managing models. Open the Model Manager dialog to add, view, edit, and delete models (select the 'Model -> Manage' menu item). Then select the "New" toolbar button (white square with a yellow start in the lower corner). Enter "Model 1" as the name of the model in the "New Model" dialog.



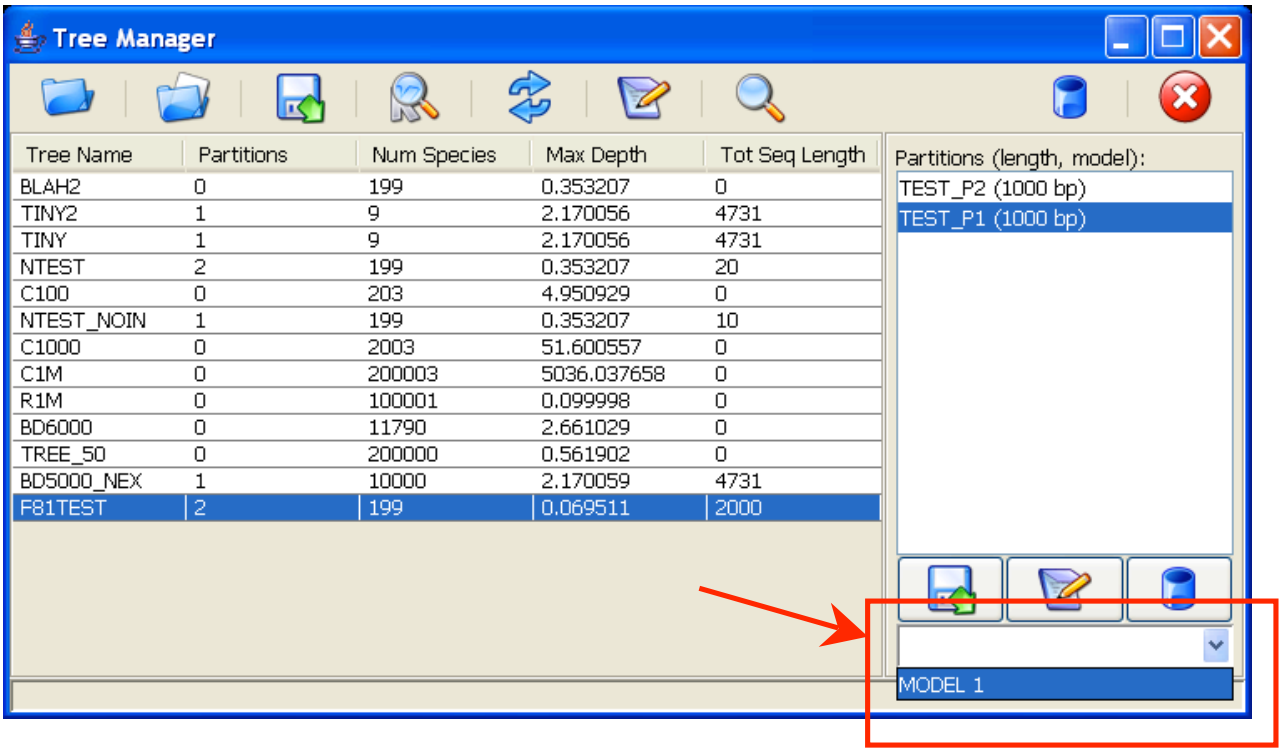
Press "Ok" and you will be presented with the Model Editor where you can edit the Notes and Details for the new model. While the Notes field is limited to 4000 characters, there isn't a limit on the Details field. Any information can be used here to distinguish or document models. When finished editing the new model press the "Save Changes" button in the lower left corner of the dialog.



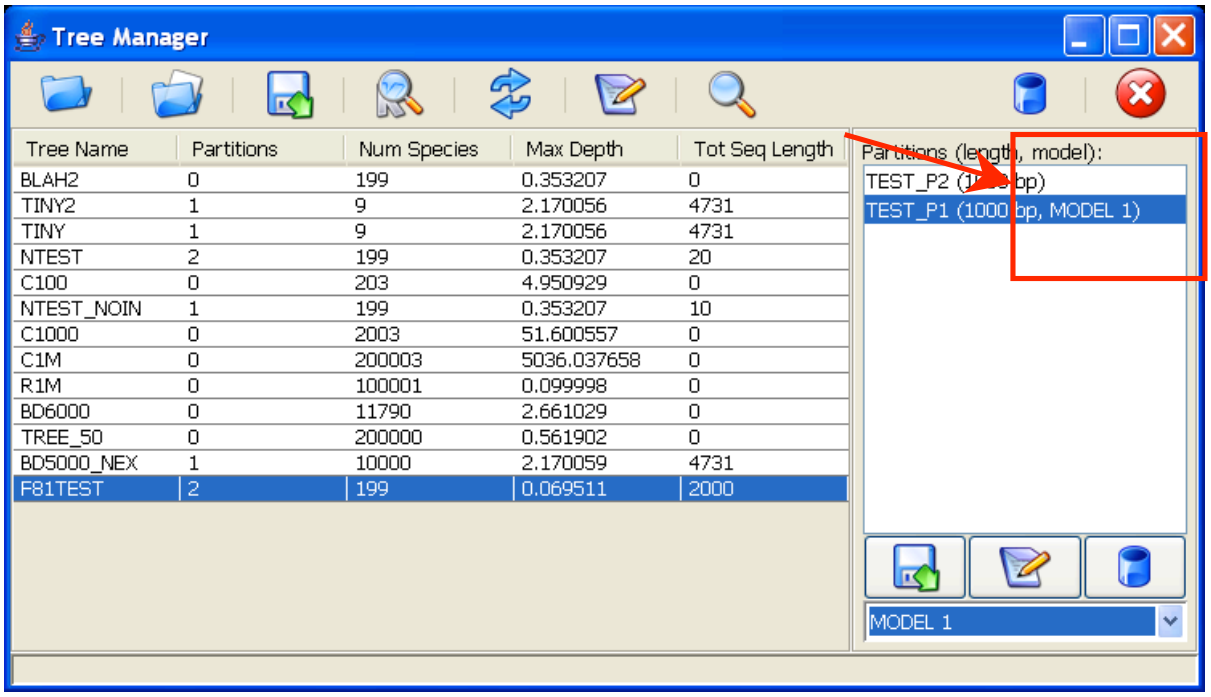
The new model called “Model 1” should now be listed in the Model Manager, as seen below.



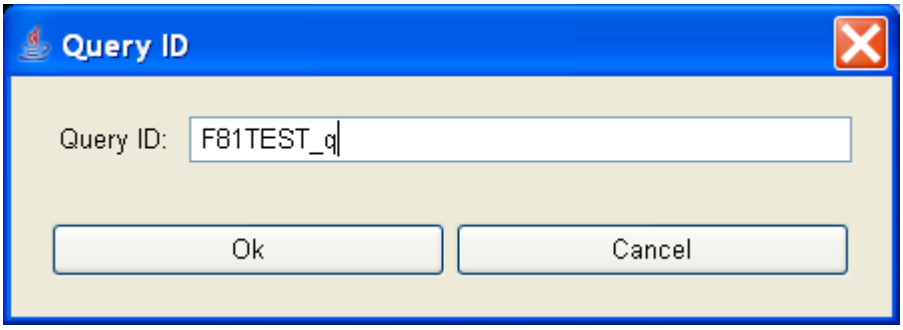
7. Associating models with partitions. From the Tree Manager, select the tree 'F81TEST'. This will cause the two partitions ("F81TEST_P1" and "F81TEST_P2") to be displayed in the Partitions panel. At this point no model is associated with either partition. Select the first partition, "F81TEST_P1" and from the pull-down menu at the bottom of the Partitions panel, select the model "Model 1."



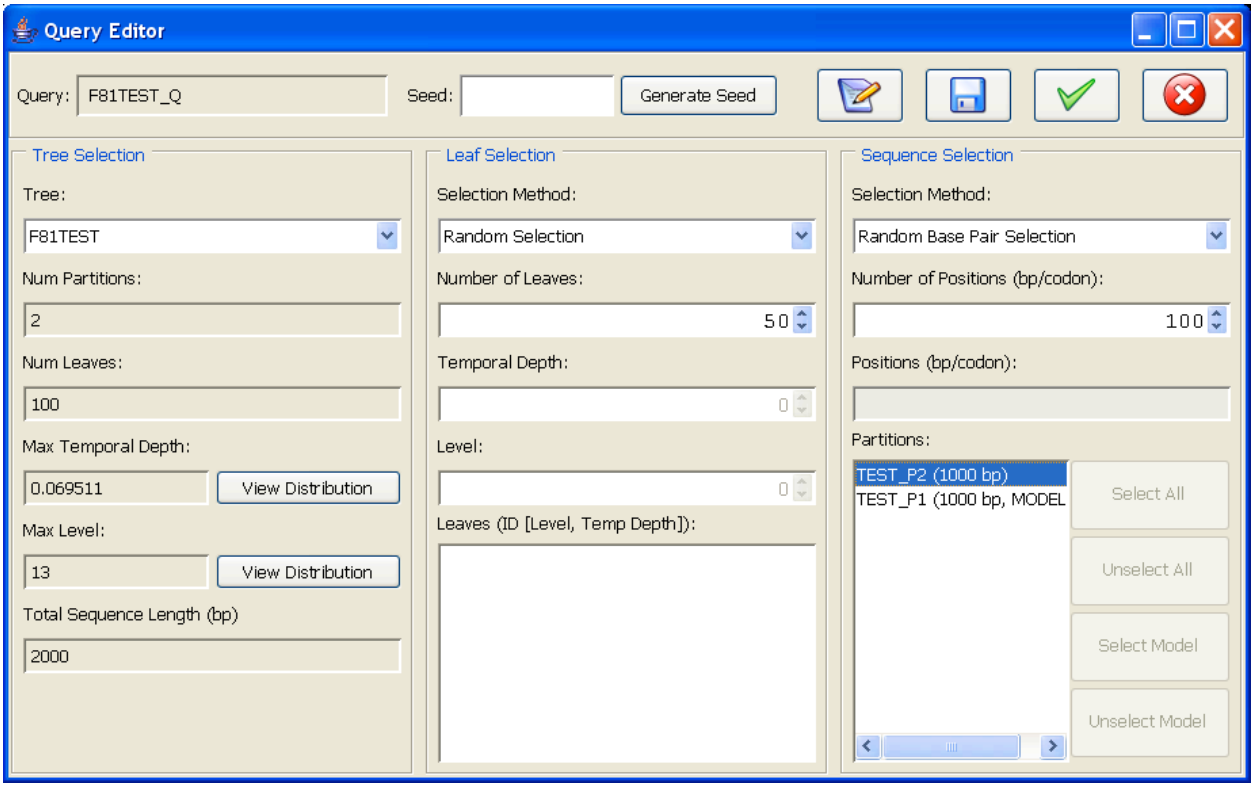
After selecting the model, the model name will be included in the description of the partition, following the length of the partition. The model can now be used to help distinguish this partition when building a query.



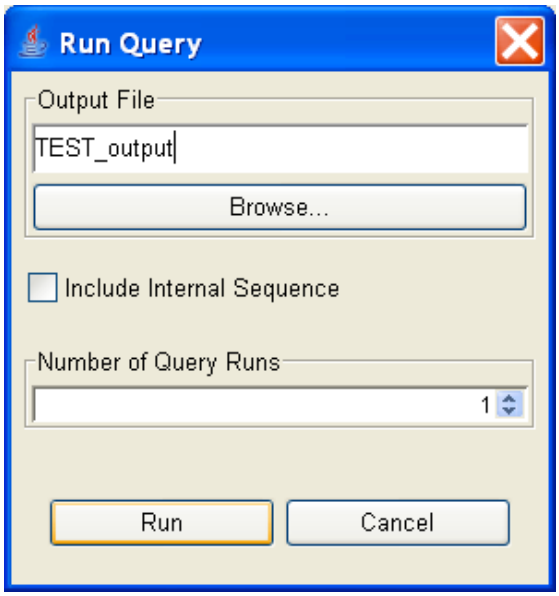
8. Query tree. From the Tree Manager, select the tree 'F81TEST'. This will cause the disabled toolbar buttons to be enabled. Then select the 'Query' toolbar button (looks like an magnification glass). Choose 'Ok' to accept the default query name 'F81TEST_q' from the Query ID dialog.



From the Query Editor, choose 'Random Selection' from the species selection method list. Enter '50' for the 'Number of Leaves' (you must first change the selection method to 'Random Selection'). Next, change the sequence selection method to 'Random Base Pair Selection' and enter '100' as the 'Number of Positions'. Then select the partition 'F81TEST_P2'. Finally, press the 'Save and Run' button (it's the green check mark next to the close button).



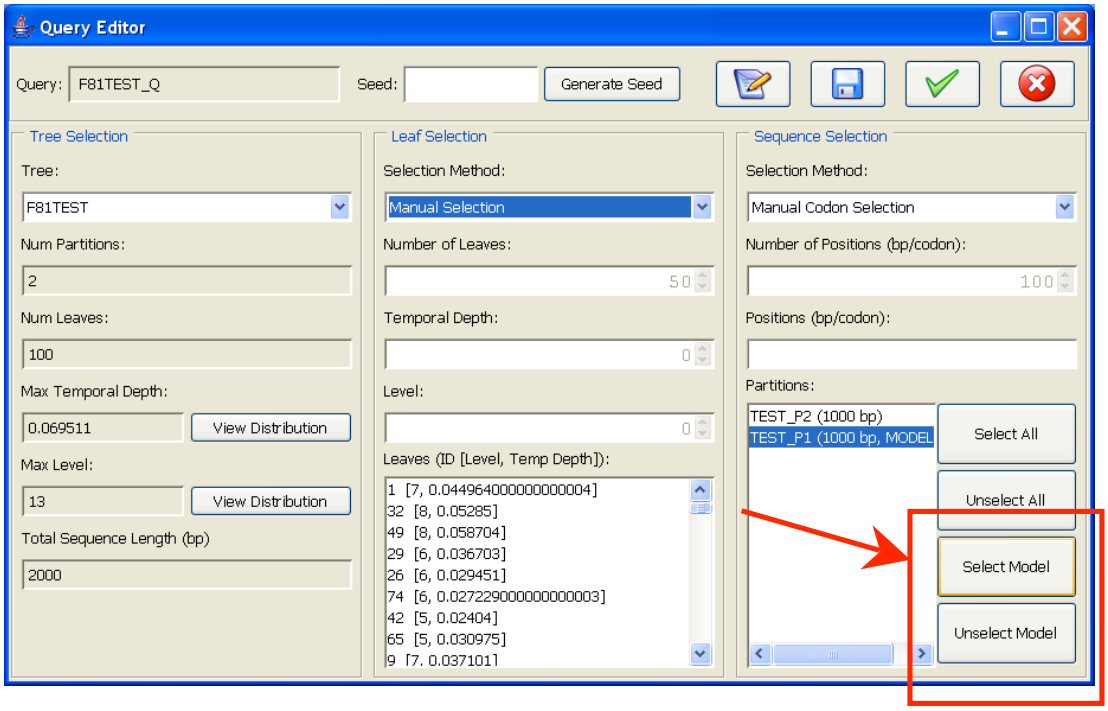
Enter 'TEST_output' as the output file from the 'Run Query' dialog and then press the 'Run' button to run the query.



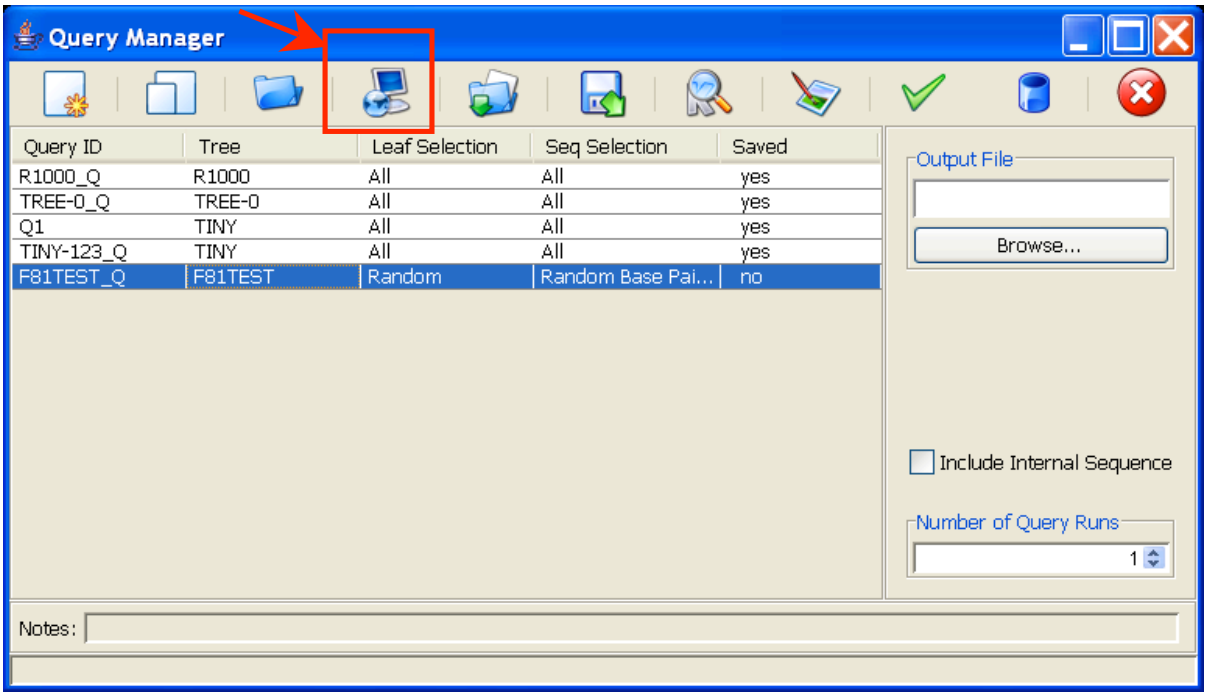
If run successfully, you should see the following in your Message window.

> Processing newick: .....
> Query finished.

When editing a query, it's possible to select tree partitions based on their associated models. To do this, use the buttons on the lower right corner of the Query Editor. When pressed, you will be asked to select a model. Depending on the button used, partitions associated with the specific model, will either be selected or unselected.



9. Manage queries. Open the Query Manager dialog (select the 'Query -> Manage' menu item). From here you can view or edit queries, create new queries, delete queries, and publish (ie save) queries to the database. Unless a query is 'published,' it will be deleted when the application is closed. Thus in order to preserve a query, it must either be exported to a text file or published to the database. To do this, select the query and press the 'Publish' toolbar button (it looks like a computer monitor with a small globe next to it). The 'Saved' column denotes whether a query has been published (ie saved) to the database. Whenever a query is modified it must be re-published in order to save the modifications to the database.



10. View trees. The Walrus display engine can be accessed from Crimson, in order to facilitate viewing trees. This section assumes the user has installed the necessary Java 3D drivers, as described in the Installation section of this manual, and launched Crimson using the "crimson-walrus" script. If the Java 3D drivers are not installed and the Walrus script not used, then Walrus these routines will not work.

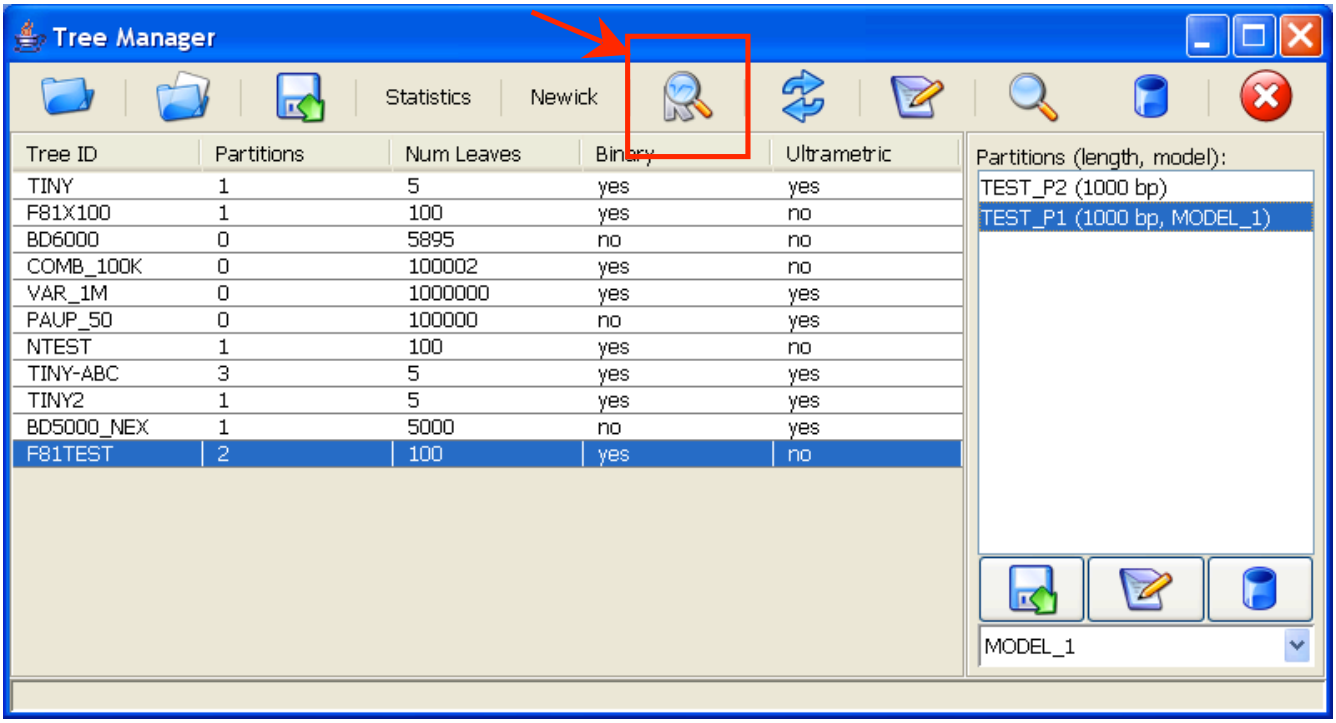
In order to view trees in Walrus, a text file must be created that contains the Walrus specific description of the tree. This text file is created from a NEXUS file using the jython function "nex2wal()".

```
>>> nex2wal("F81TEST", "F81TEST.wal")
```

In this example, the NEXUS file "F81TEST" is used to create a Walrus formatted input file called "F81TEST.wal", and then the output was displayed in Walrus. An optional argument can be given that will keep the output from being displayed in Walrus after the conversion is completed.

```
>>> nex2wal("F81TEST", "F81TEST.wal", 0)
```

Alternatively the user can use the “View Tree” button in the Tree Manager dialog. This will automatically export the tree to a Nexus file, convert the file to a Walrus format, and then display the file.



The same option can be accessed via the Menu by choosing 'Tree -> View Tree…' to access a Tree Selector dialog where the user can pick the tree to be displayed. The tree will again be exported to a Nexus file and then imported into Walrus.

==========

Query Options:

There are many different ways to query a tree. All of the options are available from the Query Editor panel and CLI. Here we describe the different options.

Leaf Selection Methods:

Select All: Selects the entire tree.

Random Selection: Randomly selects the specified number of leaves from the tree. Inner nodes will be included as needed to build the subtree specified by the leaves selected.

Select by Temp Depth (dist): The tree is sliced at the user specified temporal depth (threshold) and all resulting subtrees, below the threshold, are sampled. The user specifies the total number of leaves to be sampled and this number is divided by the number of subtrees to determine how many leaves are randomly sampled from each subtree. If the number of leaves to be sampled does not divide evenly among the subtrees, then random subtrees will be sampled an additional time. If a subtree contains fewer leaves than requested, then the additional leaves requested will not be returned. For example, if there are 3 subtrees and the user requests 10 leaves, each subtree will be sampled three times with one of the subtrees sampled a fourth time. If one of the subtrees only contains two leaves, then the resulting tree would only contain 9 leaves. If the subtree with only two leaves were randomly selected to be sampled a fourth time, then the resulting tree would only contain 8 leaves.

Select by Temp Depth (weighted): The tree is sliced at the user specified temporal depth (threshold) and all resulting subtrees, below the threshold, are computed. The leaves of all the subtrees are pooled and then collectively sampled according to the number of leaves requested by the user. Thus subtrees with more leaves are more likely to be sampled and it's possible for some subtrees to not be sampled at all.

Select by Level (dist): This is the same as "Select by Temp Depth" but the threshold is based on the level (number of ancestors to the root), instead of the temporal depth.

Select by Level (weighted): This is the same as "Select by Temp Depth" but the threshold is based on the level (number of ancestors to the root), instead of the temporal depth.

Manual Selection: The user manually chooses which leaves, based on the leaf IDs, to included in the output tree.

Sequence Selection Methods:

These methods are applied to which ever partitions the user specifies.

Select All: Selects all base pairs.

Random Codon Selection: Randomly selects the specified number of codons. If the number of base pairs is not evenly divisible by 3, then the last codon, if selected, will be incomplete.

Random Base Pair Selection: Randomly selects the specified number of base pairs.

Manual Codon Selection: The user manually specifies the codons to include. The codons are indexed from 1 (the first codon) to N (the last codon). Ranges can be used to specify a set of codons that are adjacent to one another. For example, "1:4:12-16:56" would include codons 1, 4, 12, 13, 14, 15, 16, and 56.

Manual Base Pair Selection: The user manually specifies the base pairs to include. The base pairs are indexed from 1 (the first bp) to N (the last bp). Ranges can be used to specify a set of base pairs that are adjacent to one another. For example, "1:4:12-16:56" would include base pairs 1, 4, 12, 13, 14, 15, 16, and 56.

Select None: No base pairs will be returned. The resulting Nexus file will only contain tree structure information.

==========

Scripting Language:

The command line interface is a Java implementation of Python called Jython. It contains most standard Python commands and allows the user to access all built-in Java objects. In order to facilitate the integration of the java API with the CLI, we created two jython files that load basic functions and globals into the CLI. Users are advised to review the 'header.py' and 'startup.py' files to better understand how the java API is loaded into the CLI and how to access the java API. Below are the main routines that are user accessible from these files.

Header.py:

This file loads the basic crimson objects and globals. Users are advised to include this file in any Jython files they might create.

console: This is a reference to the actual CLI object. Users can use this object to send commands to the console (ex 'console.exec("print 1+2")').

runtime: This is a reference to the runtime object for the application session.

Startup.py:

This file contains the built-in Jython functions.

General Functions:

quit(): Quits the application.

execute(cmd): Executes the command 'cmd'. If Windows OS then will prepend 'cmd /c', otherwise it will prepend 'sh -c' which is necessary for Linux.

loadPython(script): Load the python script into the command line.

viewAPI(): Launch HTML viewer to view Java API documentation.

resetGlobals(): Reset user-definable global variables to their default values.

about(): Prints version information.

time([inMillis]): Prints the current time. If inMillis is true then time is returned in milliseconds.

help([command]): Prints help for user-defined functions: 'help()' print list of commands. 'help(<UDF>)' print extended documentation for 'UDF'.

helpList(): For help on a specific built-in functions type 'help(<name>)' at the console.

Crimson Functions:

setVerbose(val): Changes the feedback verbosity of the application. The amount of detailed feedback is as follows: 2 = lots, 1 = no warnings, 0 = no feedback. This value will persist across instances of the application.

setTmpDir([val]): Changes the default directory used for temporary dataloader files (CrimsonUtils.TMP_DIR).

getTmpDir(): Returns the default directory used for temporary files.

setUsername(val): Changes the default username used to connect to database. This is also updated every time a valid database connection is made.

setDatabase(val): Changes the default database used. This is also updated every time a valid database connection is made.

getTree(id): Returns the tree object from the treePool.

getPartition(id): Returns the partition object from the partitionPool.

getModel(id): Returns the model object from the modelPool.

getQuery(id): Returns the query object from the queryPool.

importQuery(filename): This will load a query from the specified python file.

exportQuery(id, path): This will save the query to a python file that can be reload into jython. If the file already exists, it will be overwritten.

Database Functions:

setDBType(type): Sets the type of database ('Oracle' or 'MySQL').

openDatabase(username, password, server, port, database): Connects to the specified database. If username, database, or server are empty, then the default values will be used. If port is 0, then its default will be used. When a valid connection is made, the default values for username, database, server, and port will be updated.

closeDatabase(): Closes the connection to the database.

testConnection(): Performs a simple SQL query to test the database connection.

execUpdate(sql): This will execute SQL commands that update the database (i.e. CREATE, DROP, INSERT, DELETE, etc). This will return 1 if it completes or 0 on error. COMMIT will automatically be run after the execution of the sql statement.

execSQL(sql): This will execute SQL queries of the database. This can be used, for example, to run Oracle 'SELECT' statements.

loadTree(filename , treeID [, partitionID]): This will load the NEXUS file 'filename'.

appendTree(filename, treeID, partitionID): This will process the NEXUS file 'filename' and prepare the file to be appended into database tables <treeID>_P and <tree_id>_T.

removeTmpFiles(treeID): This will remove any files stored in CrimsonUtils.TMP_DIR.

exportTree(treeID): This will save the tree and all partition data as a NEXUS file.

exportStruct(treeID): This will save the tree structure as a NEXUS file.

exportPartition(treeID, partitionID): This will save the selected partition to a NEXUS file.

deleteTree(id [, delQueries]): Deleted the tree from the database, including any partition data. If any queries reference the tree and 'delQueries' is true, then the queries will also be deleted. If 'delQueries' is false (default value) and related queries are found, then the tree will not be deleted.

deletePartition(id): Deletes the partition from the database

loadAllTrees(): This will load all of the trees in the current database.

loadAllModels(): This will load all of the models in the current database.

deleteModel(id): Moves a model from the database into the Oracle recycle bin. The model can thus be restored or permanently deleted (when the recycle bin is purged).

loadAllQueries(): This will load all of the queries in the current database. If a query already exists with the same id, then the new query id will be changed so as to not conflict with the existing query.

loadQuery(id): This will load the specified query from database.

publishQuery(id): This will add the specified query to the database if it doesn't already exist in the database. If it already exists then it will update the database entry.

runQuery(id [, output, incSequence, repeat]): This will run the specified query, saving the output in the specified NEXUS file, including the CRIMSON and TREE blocks and excluding the query notes. Repeat is the number of times the query will be run. If the query.seed() is set, then runs can't be more than 1. If only the query ID is given, then a dialog will be presented to get the remaining information.

runPhylipQuery(id, output, incTree): This will run the specified query, saving the output to a phylip file. The generated tree will be returned or Null on error. There are multiple options dictated by incTree: incTree = ‘0’ -> no tree will be output; incTree = ‘1’ -> no tree will be output; incTree = ‘2’ -> tree ouptut to second NEXUS file ("<output>.tree"); incTree = ‘3’ -> rooted tree ouptut to a newick file ("<output>.newick"); incTree = ‘4’ -> unrooted tree ouptut to a newick file ("<output>.newick").

runNexusQuery(id, output, incSequence, incCrimson, incTree, incNotes): This will perform a tree query. If incCrimson is false then the CRIMSON block will not be included. If incNotes is true then the Query.notes will be appended to the end of the NEXUS file. The notes can, for example, contain a PAUP block. It is assumed that the notes are properly formatted for the NEXUS file. If Query.onlyStruct is true then incCrimson and incTree will be ignored. The generated tree will be returned or Null on error. There are multiple options dictated by incTree: incTree = 0 -> no tree will be output; incTree = 1 -> tree is included in NEXUS file; incTree = 2 -> tree ouptut to second NEXUS file ("<output>.tree"); incTree = 3 -> rooted tree ouptut to a newick file ("<output>.newick"); incTree = 4 -> unrooted tree ouptut to a newick file ("<output>.newick").

deleteQuery(id, fromDatabase): This will remove the specified query from the the queryPool. If 'fromDatabase' is true (= 1), then it will also remove the query from the database.

deleteQueryFromDatabase(id): This will remove the specified query from the database and the queryPool.

GUI Functions:

menu(): States the application's GUI. By default, this is run when startup.py is loaded.

treeManager(): Displays the tree manager.

modelManager(): Displays the model manager.

newModel(): This will create a new model object. The model object will then be opened in a ModelEditor panel.

editModel(id): This will open the model object in a ModelEditor panel.

viewModel([id]): This will display the model's parameter values.

queryManager(): Displays the query manager.

newQuery(treeID): This will create a new query object that is set to edit the specified tree. The query object will then be opened in a QueryEditor panel.

editQuery(id): This will open the query object in a QueryEditor panel.

viewQuery([id]): This will display the query's parameter values.

User Functions:

tree2wal(id): This will display a tree in the Walrus 3D viewer.

nex2wal([inFile [, outFile [, view [, min]]]]): This will convert a nexus file to a walrus format for viewing in Walrus. If the input or output file names are empty, then a graphical dialog will be presented to query the user for the file names. If 'view' is true, then after converting the file, it will be displayed in a new Walrus viewer, if not present then it is assumed to be true. If 'min' is true, then the output file will not contain any extra display info, such as species IDs or branch lengths. This will produce a significantly smaller file that will only allow for the display of the tree structure.

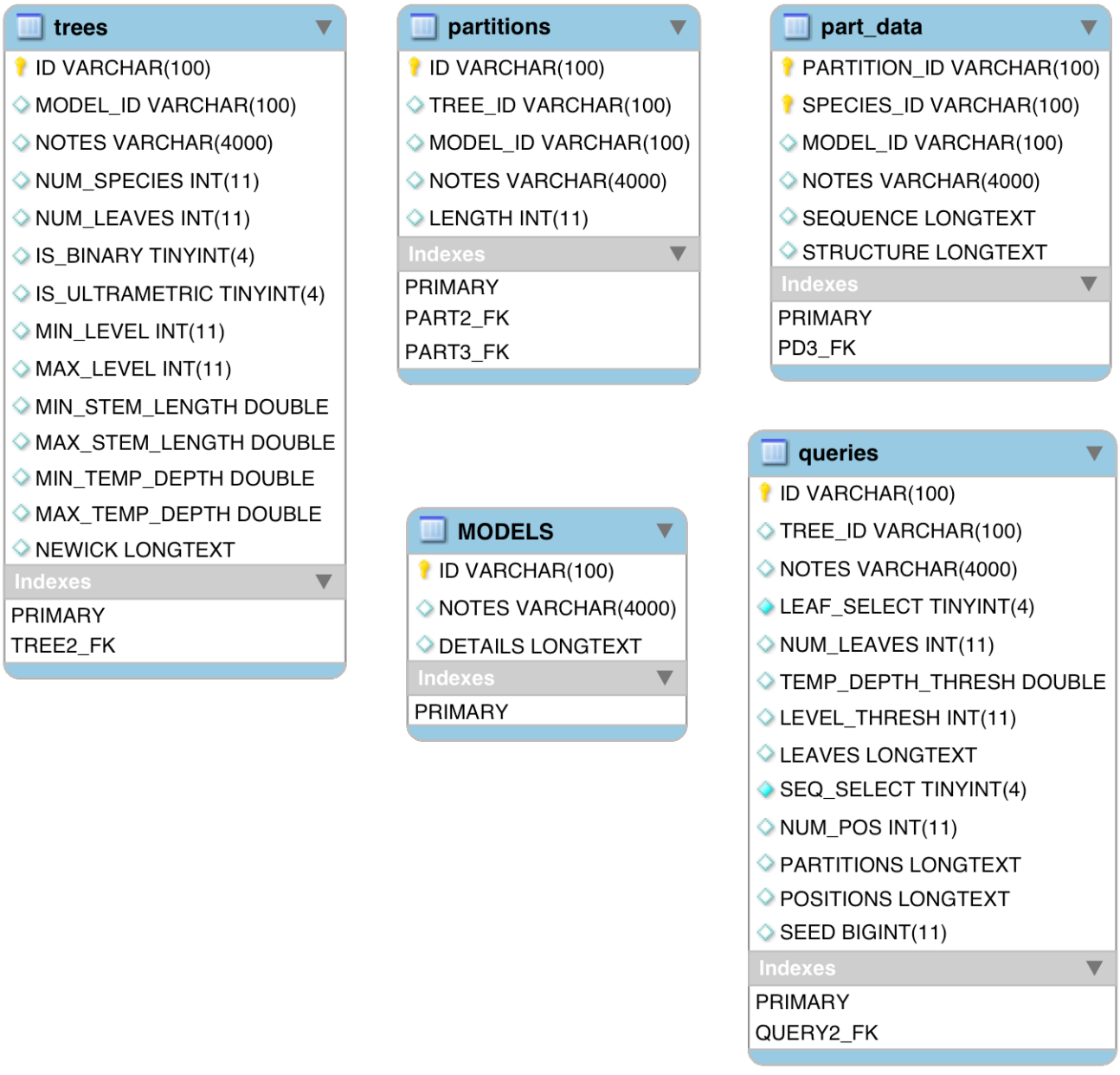
runWalrus([file]): This will display the file using the Walrus 3D viewer.

connectCIPRES(): Open a read-only connection to the SDSC Oracle server, using a web proxy.

==========

Schema:

When a connection is made to a database, Crimson checks for the existence of the necessary tables. If the tables do not exist then Crimson will create the tables. Most of the tables should be self-explanatory (TREES contains tree data, etc). However, the PARTITIONS table only contains the header info for the partitions, while the PART_DATA table contains all partition data.



==========

NEXUS File Format

Crimson uses the NEXUS file format to facilitate the exchange of data with external programs. For our purposes here, Crimson use two public blocks ('TREES' and 'DATA') and one private block ('CRIMSON'). Any other blocks in the NEXUS file will be ignored. While Crimson will output NEXUS files containing multiple data partitions, the current version is not able to import NEXUS files with more than one data partition. Below is a simple example of a valid NEXUS input file with a tree structure and one data partition containing sequence and structure data. Also included below is an example of a multiple partition NEXUS output file.

If DATA and/or CRIMSON data is included in the NEXUS file, then it must be included for every species. The DATA and CRIMSON entries must all have the same length (ie the same number of columns). Every species ID must be unique and all inner node IDs must begin with "_". To facilitate parsing of the NEXUS file spaces, not tabs, should be used to separate the species IDs from the DATA and CRIMSON data.

The CHARACTER block can be used in place of the DATA block, however the matrix formatting must be the same; that is, the CHARACTER block must include the "NEWTAXA" subcommand with relevant data formatting.

The current NEXUS parser does not properly handle in-line and multi-line comments. Lines that begin with '#' or '[' are treated as comments.

Crimson will properly quote tree and partition IDs that contain characters considered special by NEXUS. However, Crimson does not quote species IDs when exporting to a NEXUS file. Thus species IDs must not contain special characters such as '-' and '_'.

The method runNexusQuery() will include a query's notes in the NEXUS output file. This can be helpful to include PAUP blocks in query output. See the script 'exBatchScript-3.py' for an example of exporting PAUP code into NEXUS files.

For more information about the structure of NEXUS files, please see D.R. Maddison, D.L. Swofford, W.P. Maddison, *Systematic Biology*, Vol. 46, No. 4 (Dec., 1997), 590-621.

http://links.jstor.org/sici?sici=1063-5157%28199712%2946%3A4%3C590%3ANAEFFF%3E2.0.C0%3B2-H

----- Example NEXUS file with single partition (including internal sequences) -----

```
#NEXUS

BEGIN TREES;
      TREE tree1 = ((S1:0.151029, S2:0.039464)I2:0.078164, S3:0.040122,
S4:0.151029)IR:0.004586;
END;

BEGIN DATA;
      DIMENSIONS NTAX=6 NCHAR=10;
      FORMAT DATATYPE=RNA GAP=-;
      MATRIX
            IR        --ACACUGGA
            I2        --ACACUGGA
            S1        --AAAUUGAA
            S2        --UCGGUGAA
            S3        --CAUUUGGG
            S4        --ACAAGGUA
      ;
END;

BEGIN CRIMSON;
      MATRIX
            IR        ..(((()))(
            I2        ..(((()))(
            S1        ..((()))((
            S2        ..)(()))).)
            S3        ..(((())))
            S4        ..(((()))(
      ;
END;
```

----- Example NEXUS file with multiple partitions (not including internal sequences) -----

```
#NEXUS

BEGIN TREES;
      TREE tree1 = ((S1:0.151029, S2:0.039464)I2:0.078164, S3:0.040122,
S4:0.151029)IR:0.004586;
END;

BEGIN DATA;
      DIMENSIONS NTAX=4 NCHAR=10;
      FORMAT DATATYPE=RNA GAP=- INTERLEAVE;
      MATRIX
      [FOO_P1]
            S1        --AAA
            S2        --UCG
            S3        --CAU
            S4        --ACA

      [FOO_P2]
            S1        UUGAA
            S2        GUGAA
            S3        UUGGG
            S4        AGGUA
      ;
END;

BEGIN CRIMSON;
      MATRIX
      [FOO_P1]
            S1        ..(((
            S2        ..)()
            S3        ..(()
            S4        ..(((


      [FOO_P2]
            S1        )))((
            S2        ))).)
            S3        )))))
            S4        ()))(
      ;
END;

BEGIN SETS;
  CharPartition partition =
    'FOO_P1':1-5
,    'FOO_P2':6-10
;END;
```

==========

Walrus 3D Viewer

Walrus is a Java application that was developed to provide for 3D viewing of large graphs, such as trees. Crimson includes a jython script ("nex2walrus.py") to export a tree as a walrus input file.

In order to view trees in Walrus, a text file must be created that contains the Walrus specific description of the tree. This text file is created from a tree using the jython function "tree2wal()".

```
>>> tree2wal("F81X100")
```

In this example, the tree "F81X100" is used to create a Walrus formatted input file, and then the output was displayed in Walrus. Since no file name was given, a default name of "_tmpOutput.wal" was used for the Walrus file generated. Optional arguments can be given to (1) specify the output file, (2) specify whether the output will be display (as opposed to just the creation of the output file, and (3) specify whether color coding is included in the output file (this significantly increases the size of the output file).

It is also possible to view a previously created Walrus formatted tree file.

```
>>> runWalrus("F81X100.wal")
```

When viewing a tree in Walrus, it is possible to display the ID and branch length values for each species, as well as color code the branches of the tree based on branch length. See the Walrus documentation for more information about accessing these options.

Because of extra information needed to display species IDs, branch lengths, and color information, Walrus output files can be significantly larger than the initial tree files. Thus, for large trees, it is possible to create a Walrus file that doesn't have any of the extra information. These files will allow for the viewing of the tree in Walrus but without the ability to view the individual species IDs and such. To create a minimal Walrus file:

```
>>> nex2wal("F81X100", "F81X100.wal", 1, 1)
```

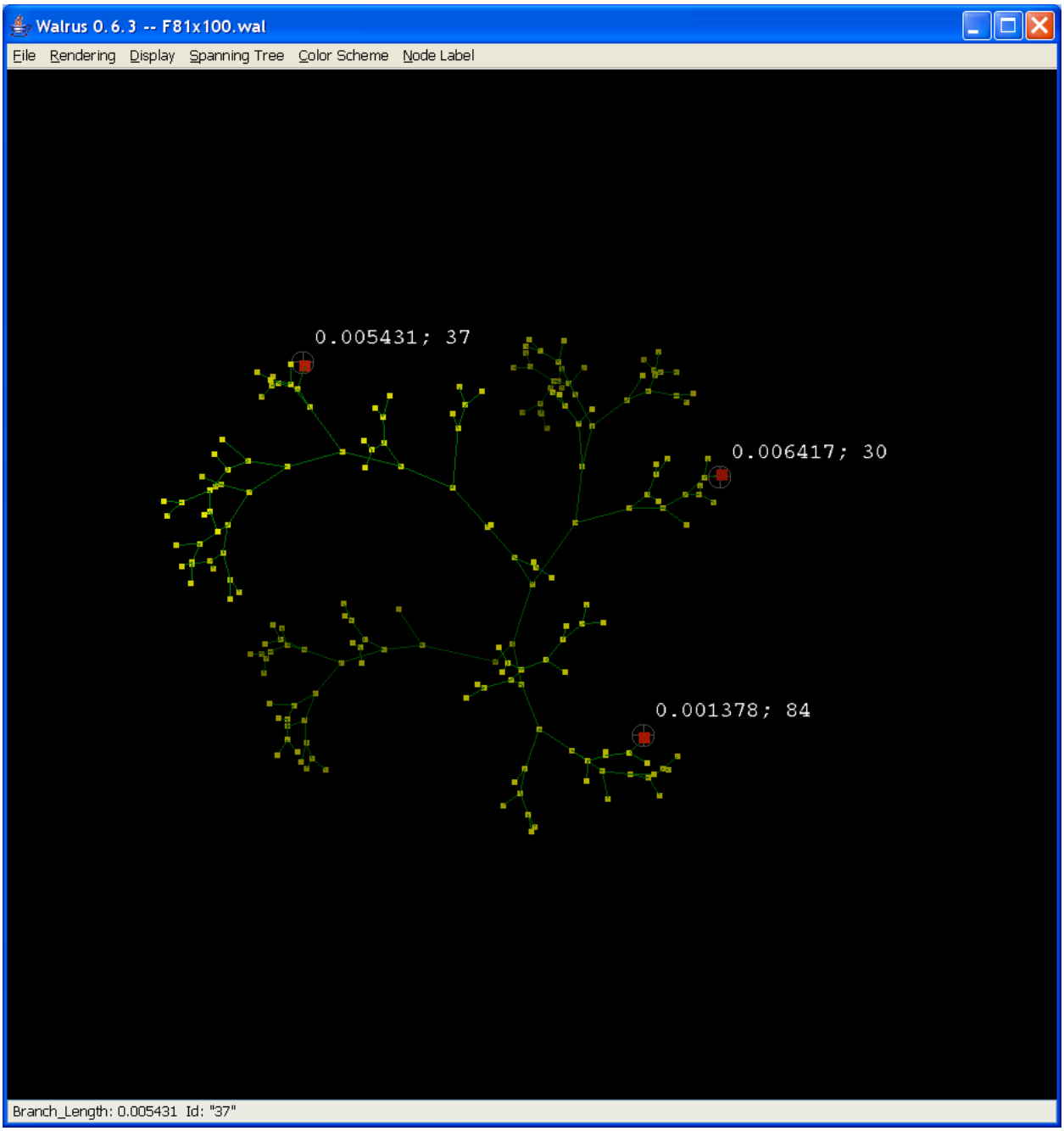
The GUI can be used to automatically export a tree into a Walrus file and then display the file in Walrus. See Section 10 of Usage Examples (page 19) for information about accessing Walrus from the GUI.

Walrus was created by Young Hyun. For more information:

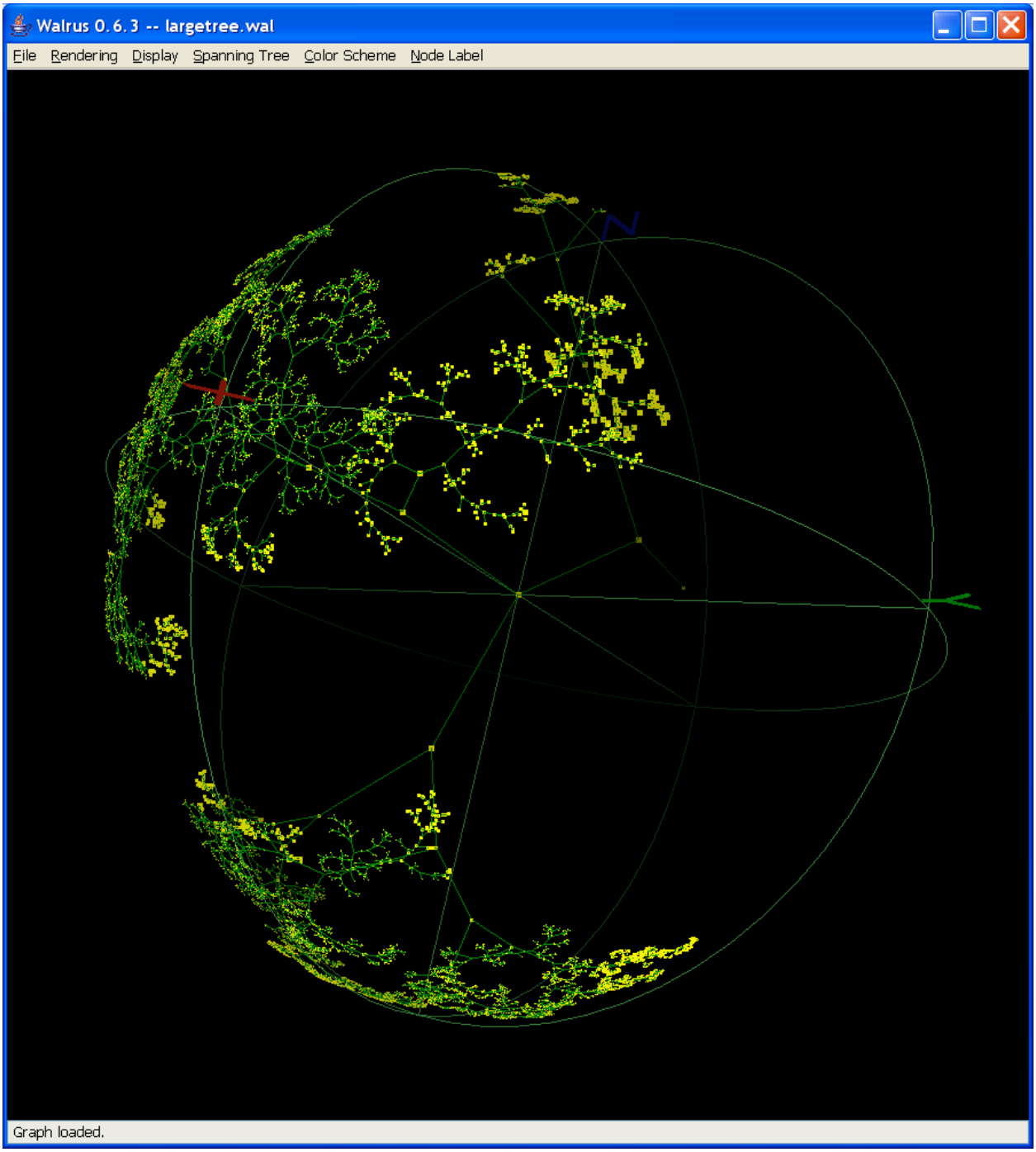
Email: youngh@caida.org
WWW: http://www.caida.org/tools/visualization/walrus

Note that most VNC servers do not have the ability to display 3D images. Thus Walrus may not work with VNC.

Below is an example of Walrus displaying a tree with 100 leafs. The species IDs and branch lengths are shown for three species that were selected.



Below is an example of Walrus displaying a tree with 100,000 leafs (200,000 total nodes).



==========

Loading Very Large Data Partitions:

The Crimson application is able to load very large trees (for example, trees with over 2,000,000 species) into Oracle and MySQL databases. However, data partitions for trees this large often contain 20 to 60 GB of data. While the application is capable of loading these large data partitions, it is much more efficient to use native utilities to load these large data partitions into the database.

The instructions below describe how to load large data partitions into Oracle and MySQL using native utilities. These instructions are for advanced users who are familiar with maintaining Oracle or MySQL databases. Users should be familiar with the database schema prior to using these data loading methods (see SCHEMA, page 27).

Oracle:

1. **Load tree structure.** Using a NEXUS file that only contains the newick tree structure, load the tree into the database. This will create the necessary entry in the TREES table and allow us to append the partition data to the tree.

2. **Format data file.** Oracle contains a native utility called "sqlldr" that very efficiently loads large data sets into an Oracle database. To use sqlldr, we must split the NEXUS file into specific data files. Crimson contains a python utility called 'split_nex.py' that will appropriately format the data.

   % split_nex.py SIM_8000.nex S8000

   This will split the NEXUS file called 'SIM_8000.nex' into three files: S8000.tax, S8000.chr, and S8000.crm. These files will contain the species labels (*.tax), character data (*.chr), and structure data (*.crm).

3. **Create control file.** Sqlldr uses a control file to configure the loading of the data. This file is a simple text file that contains information about which data files to use and how to load them into the PART_DATA table. An example control file is included below, which has been formatted to load the S8000 data partition. Note that the PARTITION_ID has been set to 'S8000_PART'. This is user definable.

----- Example SQLLDR control file -----

```
OPTIONS ( DIRECT=TRUE )
UNRECOVERABLE
LOAD DATA
INFILE 'S8000.tax'
  BADFILE 'S8000.bad'
  DISCARDFILE 'S8000.dsc'
APPEND
INTO TABLE PART_DATA
  FIELDS TERMINATED BY '\n'
  TRAILING NULLCOLS
  (
      PARTITION_ID  CONSTANT 'S8000_PART'
    , SPECIES_ID    CHAR
    , MODEL_ID      CHAR
    , NOTES         CHAR
    , SEQUENCE      LOBFILE(CONSTANT 'S8000.chr') TERMINATED BY '\n'
    , STRUCTURE     LOBFILE(CONSTANT 'S8000.crm') TERMINATED BY '\n'
  )
```

4. **Run sqlldr.** After the data files have been processed and the control file created, sqlldr is ready to be run. The following sqlldr example, assumes the control file is called ‘sqlldr.ctl’, the Oracle user login is ‘myuserid’, the Oracle user password is ‘mypasswd’, and the Oracle server is ‘mydbhost’. According to the control file, the data from the S8000.chr and S8000.crm files will be loaded into the PART_DATA table.

   $ sqlldr userid=myuserid/mypasswd@mydbhost control=sqlldr.ctl log=sqlldr.log

5. **Update PARTITIONS table.** Once the data has been loaded, it is necessary to link the data in the PART_DATA table with the tree in the TREES table. To do this, we will add a row to the PARTITIONS table. This can be done from the Crimson command line interface, using the ‘execUpdate()’ command.

   >>> execUpdate("INSERT INTO PARTITIONS (ID, TREE_ID, LENGTH) VALUES ('S8000_PART', 'S8000', 1883)")

   This command will insert a row into the PARTITIONS database that connects the data in PART_DATA identified by the ID ‘S8000_PART’ with a tree in the TREES table identified by the ID ‘S8000’. In this example the data partition sequences contain 1883 base pairs. Note that the tree must exist in the TREES table prior to running this command.

MySQL:

A utility has not been included to process data files for fast loading into MySQL databases. If loading large data files into MySQL, the Oracle instructions above would apply, although the data file would have to be structured appropriately for the MySQL ‘LOAD DATA’ command. The ‘LOAD DATA’ command would be used in place of the Oracle sqlldr command.

==========

System Requirements:

- Pentium (x586) or later processor, or equivalent
- at least 512MB of RAM (1024 MB recommended)
- at least 20 MB of hard drive space
- Java version 1.5 or later
- Oracle 10.X (or later) or MySQL 5.0 (or later)

==========

Known Issues:

- When logging into SDSC via direct connection (ie not through proxy) then it takes 10 to 15 minutes to load the queries.
- Need to speed up building of large trees. It currently takes about a minute to build a one million leaf tree.
- Need more secure handling of database passwords.
- Create GUI dialog for setting of application parameters.
- Create MySQL data preprocessor for loading large datasets.
- Update NEXUS output to use CHARACTERS/TAXA blocks instead of DATA block.

==========

License:

KIM LABORATORY SOFTWARE AND DATA LICENSE
Version 1.0, August 2007

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. **Definitions.**

   "*License*" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document. Your use of the Work and Derivative Works in Source or Object form constitutes acceptance of this License.

   "*Licensor*" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "*You*" in this License, and "*Your*" when used in the possessive, means an individual or a legal entity exercising rights under this License. For legal entities, "*You*" includes any entity that is controlling, controlled by or under common control with You. For purposes of this definition, "*control*" means the direct or indirect ownership of more than fifty percent (50%) of the outstanding voting securities of a legal entity, the right to receive fifty percent (50%) or more of the profits or earnings of a legal entity, or the right to determine the policy decisions of a legal entity.

   "*Source*" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, configuration files, and data files.

   "*Object*" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "*Work*" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work.

   "*Derivative Works*" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

2. **Grant of Copyright License.** You may use the Work and Derivative Works for any non-commercial purpose, subject to the restrictions in this License. Some purposes which can be non-commercial are teaching, academic research, and personal experimentation.

3. **Redistribution.** You may modify the Work and Derivative Works and distribute the Work or Derivative Works for non-commercial purposes, provided that You meet the following conditions:

    (a) You may not grant rights to the Work or Derivative Works that are broader than those provided by this License (for example, you may not distribute modifications of the Work under terms that would permit commercial use, or under terms that purport to require the Work or Derivative Works to be sublicensed to others); and

    (b) You must give any other recipients of the Work or Derivative Works a verbatim copy of this License; and

    (c) You must cause any modified files to carry prominent notices stating that You changed the files; and

    (d) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

    (e) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

    You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

4. **Exclusions From License Grant.** You may not use or distribute the Work or Derivative Works in any form for commercial purposes. Examples of commercial purposes would be running business operations, licensing, leasing, or selling the Software, or distributing the Work or Derivative Works for use with commercial products.

    No other rights are granted in this License. You and Your affiliates, sublicensees, employees, and agents may not use any name, logo, seal, or trademark of Licensor or any affiliate organization, employee, student or representative of the Licensor, without the prior written consent of the Licensor.

5. **Retained Rights.** The Licensor retains the right to copy the Work, prepare Derivative Works based upon the Work and distribute the Work or Derivative Works to any person or entities for any purpose.

6. **Termination.** This License is effective until terminated. You may terminate this License at any time upon notice to the Licensor. This License will terminate immediately without notice from the Licensor if You fail to comply with any provision of this License. Upon termination for any reason, You must destroy all copies of the Work and Derivative Works in Your possession and control.

7. **Government Rights and Restrictions.** You acknowledge that the License is expressly subject to reserved rights of the United States Government, if any, under all applicable statutes and regulations. You agree to comply with all laws, rules and regulations applicable to the use of the Work and Derivative Works, and You will be responsible for obtaining, at Your expense, any governmental approvals required to use the Work and Derivative Works. All rights granted to You under this License are contingent upon Your compliance with United States laws and regulations controlling the export of technical data, computer software, laboratory prototypes, and all other export controlled commodities, including, without limitation, the Arms Export Control Act and the Export Administration Act as they may be amended.

8. **Entire Agreement.** The terms and conditions contained in this License constitute the entire agreement between the parties and supersede all previous agreements and understandings, whether oral or written, between the parties with respect to the subject matter. This License may not be amended without a written agreement signed by an authorized representative of the Licensor.

9. **Severability.** If any provision of this License shall be determined to be void, invalid, unenforceable or illegal for any reason, then the validity and enforceability of all of the remaining provisions hereof shall not be affected thereby.

10. **Miscellaneous.** This License will be binding upon and inure to the benefit of the parties and their respective permitted successors and assigns. This License shall be construed and interpreted and its performance shall be governed by the laws of the Commonwealth of Pennsylvania without reference to the conflicts of law principles of any jurisdiction.

11. **Disclaimer of Warranty.** You expressly acknowledge and agree that use and distribution of the Work and Derivative Works is at Your sole risk. The Work and Derivative Works is provided "AS IS" and without warranty of any kind. THE LICENSOR MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF ACCURACY, COMPLETENESS, PERFORMANCE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, COMMERCIAL UTILITY, NON INFRINGEMENT OR TITLE. THE LICENSOR DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE WORK AND DERIVATIVE WORKS WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE WORK AND DERIVATIVE WORKS WILL BE UNINTERRUPTED OR ERROR-FREE.

12. Limitation of Liability. IN NO EVENT, INCLUDING NEGLIGENCE, WILL THE LICENSOR BE LIABLE TO YOU, YOUR SUCCESSORS OR ASSIGNS, OR ANY THIRD PARTY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES OF ANY KIND THAT RESULT FROM THE USE OR DISTRIBUTION OF THE WORK AND DERIVATIVE WORKS OR ARISE UNDER THIS LICENSE, EVEN IF THE LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

==========

Contact Information:

Susan Davidson
Computer Science Department
University of Pennsylvania
3330 Walnut Ave.
Philadelphia, PA 19104
susan@cis.upenn.edu
(215) 898-3490

Stephen Fisher
Biology Department
University of Pennsylvania
433 S. University Ave
Philadelphia, PA 19104
safisher@sas.upenn.edu
(215) 898-8395

Junhyong Kim
Biology Department
University of Pennsylvania
433 S. University Ave
Philadelphia, PA 19104
junhyong@sas.upenn.edu
(215) 746-5187

==========