PerfTrack

PerfTrack 3.0 Users Guide

Table of Contents

1 OVERVIEW

This manual describes the use of PerfTrack version 2.0. It is targeted to end users of the PerfTrack system. A tutorial demo is available separately in the PerfTrack Tutorial.

PerfTrack is a tool for storing, navigating, and analyzing data from parallel performance experiments, developed at UC/Lawrence Livermore National Laboratory under the direction of co-PIs John May (johnmay@llnl.gov) and Karen Karavanic (karavan@cs.pdx.edu).

1.1 The PerfTrack Source Files

The PerfTrack source repository contains the following files:

`PerfTrack/PTbugList.txt`
> This file describes known bugs and limitations of the software.

`PerfTrack/install.sh`
> This file automates the process of installing dependencies, building PerfTrack, creating and populating the database, and running data loading tests.

`PerfTrack/src/GUI`
> This directory contains sources for the PerfTrack GUI, which is implemented in C++ and Qt. This release includes an executable of the GUI but not the source code.

`PerfTrack/src/dataStore`
> This directory contains helper scripts and Python modules implementing database access.

`PerfTrack/src/data_collection`
> This directory contains helper scripts and Python modules implementing parsers for input data formats, particularly the PerfTrack data format PTDF.

`PerfTrack/demo` TODO Locate tutorial documents
> Contains the PerfTrack Tutorial document and the sample files used in the tutorial.

```
PerfTrack/db_admin
```
Contains database scripts used to install and drop the tables of the PerfTrack schema. Scripts are provided for PostgreSQL, MySQL, and Oracle databases. Files:
postgres/pdropall.sql, postgres/pcreate.sql, postgres/pgrant.sql,
oracle/dropall.sql, oracle/create.sql, mysql/mcreate.sql, mysql/dropall.sql

```
PerfTrack/docs
```
Contains the Users Guide. Files:
PerfTrackUsersGuide.doc, PerfTrackUsersGuide.pdf

```
PerfTrack/scripts
```
Contains shell scripts used to automate installation. Files:
env.py, install-dependencies-el.sh, install-dependencies-ubuntu.sh,
runtests.py

```
PerfTrack/share
```
Contains the PTDF file for the PerfTrack default and extended type hierarchies. Files:
iMDL.cfg,   PERIiMDL.cfg,   PTdefaultFocusFramework.ptdf,
dataCenterResourceHierarchyExtensions.ptdf

```
PerfTrack/tests
```
Contains unit tests for the data loading scripts and libraries.

1.2 Supported Hardware and Software platforms

The PerfTrack frontend runs on Linux, Mac OS X, and Solaris. The data collection scripts run on Linux and AIX. The following DBMS packages are supported: Oracle, PostgreSQL, and MySQL.

1.3 Contacting the PerfTrack team

PerfTrack Support   perftrack-support@cs.pdx.edu

Dr. Karen L. Karavanic   karavan@cs.pdx.edu
Rashawn Knapp   knappr@cs.pdx.ed
Kathryn Mohror   kathryn@cs.pdx.edu

On the web: http://perftrack.cs.pdx.edu/

2 PerfTrack Terminology and Resource Types

2.1 PerfTrack Terminology

In PerfTrack, the types of performance data are termed *resource types*. For example, a resource type could be the operating system, or the number of nodes running. Some resource types are naturally hierarchical. The resource type **Grid**, for example, has the *descendent* **Machine**, whose descendent is **Partition**, whose descendent is **Node**, whose descendent is **Processor**. (PerfTrack uses Unix-style path names such as grid/machine/partition/node/processor.)

A non-hierarchical resource type is **operatingSystem**. A full overview of resource types and a hierarchical diagram of them is seen in Section 4. A *resource attribute* is a characteristic of a resource, such as the clock speed for the resource type **Processor**.
The term *resource* refers to a specific object of a certain resource type. For example, a resource of resource type **operatingSystem** might be Linux.

Lastly, a *performance result* is any measured or calculated value, the *metric*, plus descriptive metadata, the *context*. For example, a performance result might include the CPU time of one out of 64 processors.

2.2 Resource Type Overview

PerfTrack's resource type system is quite flexible. At initialization time, the default resource type hierarchy. However, you can add new resource types at any time or even replace the default ones by editing the file `PTdefaultFocusFramework.ptdf` in `PerfTrack/share`. This manual will assume the default types, listed below, are being used.

Basically, there are two flavors of resource types: simple resource types are identified with a string name; and hierarchical resource types are identified with a pathname from the ancestor type down to the current type (see diagram on page 6).

- **application**
  Distinguishes unrelated applications. The database would normally contain many executions for a single application. This resource type has no attributes.
- **build**
  The particular executable used in a given application run.
- **build/module/function**
  These are the functions that can be found in the source code of the application or in the static libraries. This resource type has no attributes.
- **compiler**

The compilers used in building the executable.

- **environment/module**
  The libraries that are dynamically linked with the executable.
- **environment/module/function**
  These are the functions that can be found in the dynamically linked libraries. This resource type has no attributes.
- **execution**
  Each run of an application is represented by one execution resource.
- **execution/process**
  The processes in the execution. This resource type has no attributes.
- **execution/process/thread**
  The threads in each process. This resource type has no attributes.
- **filesystem**
  The file systems that are in use on a machine.
- **filesystem/device**
  These are the devices or mount points for each file system.
- **grid**
  A collection of one or more machine resources.
- **grid/machine**
  A machine.
- **grid/machine/partition**
  A partition of the nodes of the machine.
- **grid/machine/partition/node**
  The nodes that make up a machine partition.
- **grid/machine/partition/node/processor**
  The processors of a node.
- **inputDeck**
  Any input file that alters the parameters of the execution. Currently, this can be an input deck or even `.h` files that change attributes of the run.
- **metric**
  Types of performance measurements. This resource type has no attributes.
- **operatingSystem**
  The operating system of the build or of the execution.
- **performanceTool**
  The performance tool used to collect the performance data. This resource type has no attributes.
- **preprocessor**
  The preprocessors used in compilation of the executable.
- **submission**
  The submission contains details extracted from the launch of the execution, such as commands in the batch file submitted to the scheduler.
- **time/interval**
  Phase or iteration level data would be described in terms of a particular time/interval. This resource type has no attributes.
- **time/interval/subinterval**

Subphase or loop level data might be described in terms of a particular time/interval/subinterval. This resource type has no attributes.

```
root
├── compiler
├── grid
│   machine
│       └── partition
│               └── node
│                       └── processor
├── build
│   module
│       └── function
│               └── codeblock
├── environment
│   module
│       └── function
│               └── codeBlock
├── time
│   interval
│       └── subinterval
├── operatingSystem
├── execution
│   process
│       └── thread
├── inputDeck
├── preprocessor
├── performanceTool
├── metric
├── submission
└── application
```
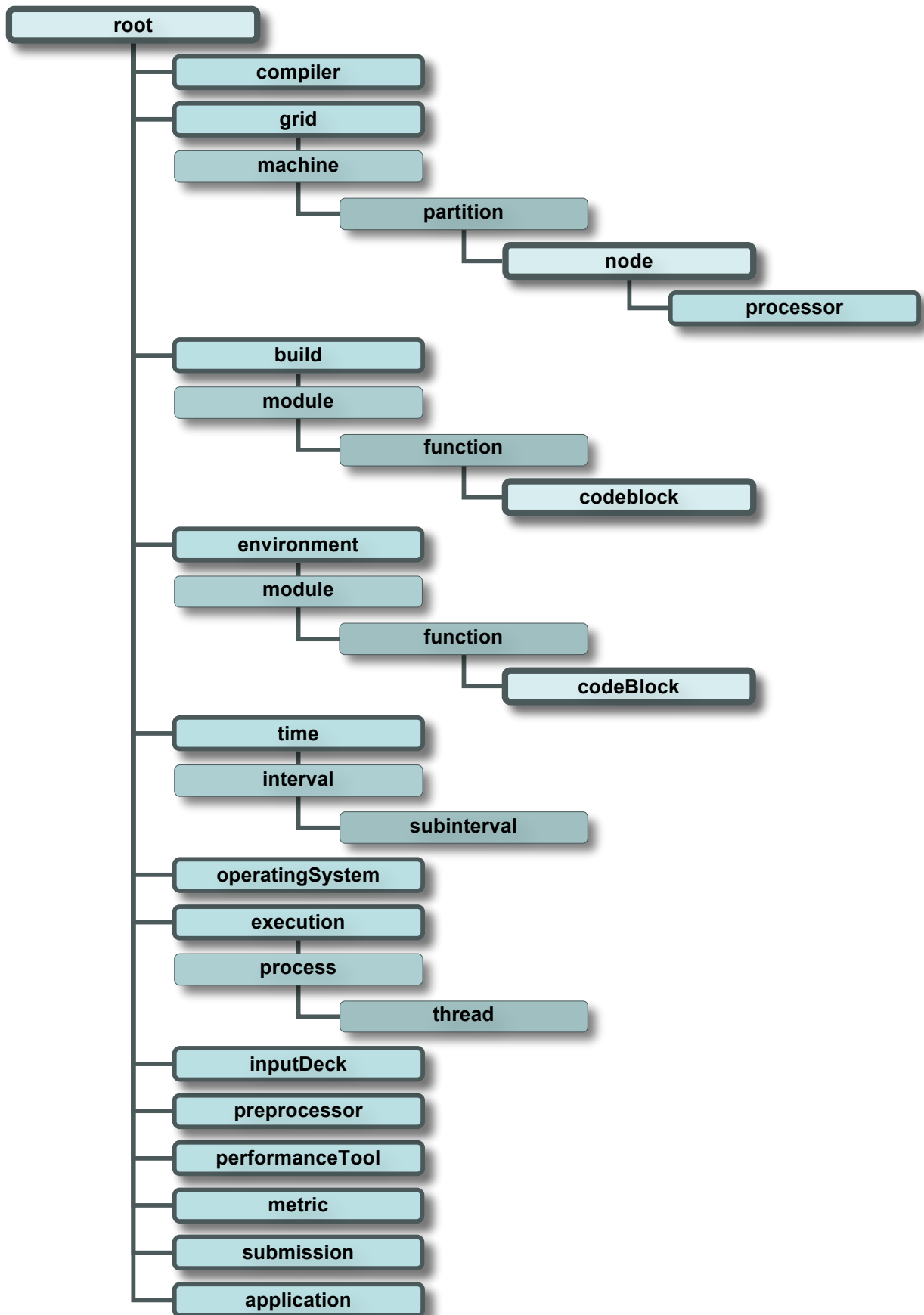
3 INSTALLING PERFTRACK

3.1 Overview

The latest version of PerfTrack has been tested using CentOS 6.4, Ubuntu Linux 12.04 LTS, and Ubuntu Linux 12.10.

Installation is a four-step process:

1. Downloading the PerfTrack source
2. Installing package dependencies
3. Building and installing PerfTrack
4. Initializing the PerfTrack database

If you have already downloaded the PerfTrack source, the script `install.sh` in the top level of the PerfTrack source conveniently automates all of the remaining steps for the Linux distributions listed above.

3.2 Downloading PerfTrack

To download PerfTrack, clone the GitHub repository:

```
git clone https://github.com/heathharrelson/PerfTrack.git
```

TODO: Update to permanent repository.

3.3 Dependencies

The following packages are needed to build and install PerfTrack:

Package	Supported Versions	Where to Get Sources
CMake	2.8	http://www.cmake.org/
Python	2.6 or 2.7	http://www.python.org/
PostgreSQL	8.4.x or 9.1.x	http://www.postgresql.org/
Psycopg	2.4	http://www.initd.org/psycopg/
PLY	3.4	http://www.dabeaz.com/ply/
Qt	Open source version 4.4.0	http://qt-project.org/
Unittest2	0.5.1	https://pypi.python.org/pypi/unittest2

The PLY and unittest2 modules are available in the EPEL repository for Enterprise Linux-based distributions (http://fedoraproject.org/wiki/EPEL), but Psycopg will need to be installed from source.

The scripts/ directory contains scripts to install dependencies for Ubuntu distributions and Enterprise Linux-based distributions (the files are `scripts/install-dependencies-ubuntu.sh` and `scripts/install-dependencies-el.sh`, respectively).

PerfTrack uses the SQL interface provided by Qt, so in addition to the distribution's Qt package and the relevant development packages, the Qt SQL support packages should also be installed.

If building Qt from source, you will need to enable database support. For instance, to configure Qt to use PostgreSQL:

```
$ ./configure -prefix /path/to/QT/Install \
    -plugin-sql-psql \
    -I/path/to/postgresql/headers/include \
    -I/path/to/postgresql/serverheaders/include/server \
    -L/path/to/postgres/libaries/lib \

$ gmake

$ gmake install
```

3.4 Building PerfTrack

PerfTrack is built using the CMake cross-platform make utility (http://www.cmake.org). To build PerfTrack, run the following commands in the PerfTrack source directory:

```
$ cd src
$ mkdir build
$ cd build
$ cmake ..
# lots of output
$ make
$ sudo make install
```

The `build.sh` script provided in the `src/` directory automates these steps.

If the commands are run as shown, all files will be installed to the default `CMAKE_INSTALL_PREFIX`, which is usually `/usr/local` on Linux. If you want to change the path where PerfTrack is installed (e.g. to install it to a user directory instead of system wide), you can set the `CMAKE_INSTALL_PREFIX`:

```
cmake -DCMAKE_INSTALL_PREFIX=/new/path ..
```

3.5 Initializing PerfTrack

Setting up PerfTrack requires the creation of a database, creating the PerfTrack tables in that database, and loading a resource hierarchy (we provide a default hierarchy and an extended default hierarchy) into the database. These directions assume that you (or your system administrator) have already created a database for PerfTrack.

3.5.1 Creating the PerfTrack Tables in PostgreSQL

To create the tables in PostgreSQL:

1. Change directories to `db_admin/postgres/` in the PerfTrack source tree.
2. Log into the database: `psql -h <host name> -d <database name> -U <username> -W`
3. At the `psql` command prompt, execute the SQL script:
   `\i pcreate.sql`

3.5.2 Creating the PerfTrack Tables in Oracle

TODO: write directions for using Oracle

3.5.3 Creating the PerfTrack Tables in MySQL

To create the tables in MySQL:

1. Change directory to `db_admin/mysql/` in the PerfTrack source tree.
2. Log into the database: `mysql -h <host name> -u <username> <database name>`
3. At the mysql command prompt, execute the SQL script: `source mcreate.sql;`

3.5.4 Loading the Default Resource Hierarchy using a MySQL or PostgreSQL Database and PTDF

As described in the section

2 PerfTrack Terminology and Resource Types, PerfTrack organizes performance data using a hierarchy of *resources*. This hierarchy of resources must be installed. We provide a default hierarchy, but you may design a different hierarchy if you wish.

Before the resource hierarchy can be loaded, the PerfTrack tables need to have been created in the database as described above. To load the default resource hierarchy (seen in Section 2.2 Resource Type Overview) call the script `ptdf_entry.py` with the file name `share/PTdefaultFocusFramework.ptdf` as a command line argument from the PerfTrack source directory:

```
$ ptdf_entry.py share/PTdefaultFocusFramework.ptdf
```

We provide a second hierarchy that extends the default hierarchy to include datacenter level resources (e.g., racks, cooling towers, airconditioners, etc). If you wish to load this hierarchy, run:

```
$ ptdf_entry.py share/dataCenterResourceHierarchyExtensions.ptdf
```

If you installed PerfTrack to a custom location, you may need to set environment variables to run the `ptdf_entry.py` script. See the Section 4.1 Environment Variables, below.
Once a hierarchy has been loaded, you can run the PerfTrack GUI even though you have not entered any performance data into the PerfTrack database. To run the GUI:

```
$ perftrack
```

3.5.5 Reloading the Database using PostgreSQL and a Backup File

In some cases, you may want to load data into PerfTrack using a backup file. To generate a backup file, execute:

```
$ pg_dump --data-only --file=FileNameOfYourChoosing
  --format=P --verbose --host=myhost.domain -U username
  -W databaseName
```

To clear out the contents of the database to get a clean start, first change directories to the perftrack/db_admin/postgres directory. Then log into the database with psql:

```
$ psql -h <host name> -U <username> -W <database name>
```

Then drop all the tables and data using this command at the psql command prompt:

```
\i pdropall.sql
```

You can check to make sure everything dropped by using this command at the psql command prompt:

```
\d+
```

To recreate the PerfTrack tables, execute the following at the psql command prompt:

```
\i pcreate.sql
```

To load the data in the back up file into the database, execute the following at the psql command prompt:

```
\i backupFileName
```

3.5.6 Loading the Default Resource Hierarchy using an Oracle Database

TODO: write directions for using Oracle

4 USING PERFTRACK

4.1 Environment Variables

If you installed PerfTrack to the default location, you should be able to run the GUI and Python scripts without having to alter your environment. If, however, you intend to run PerfTrack from the build directory or a custom install location, you may need to modify a few environment variables.

4.1.1 Environment Variables for Data Collection Scripts

Environment variables needed for the data collection scripts:

- **PYTHONPATH**: This variable needs to be set to point to the directory where the Python modules were installed. By default, these files are installed to the directory `CMAKE_INSTALL_PREFIX/lib/pythonX.Y/site-packages/perftrack`.
- **PATH**: This variable needs to include the directory where the PerfTrack executables were installed. By default, these files are installed to the directory `CMAKE_INSTALL_PATH/bin`.
- **PTDB**: This variable can have one of three values: MYSQL (for MySQL database), PG_PYGRESQL (for PostgreSQL database) or ORA_CXORACLE (for Oracle database).

4.1.2 Environment Variables for the GUI

Environment variables needed by the GUI:

- **PATH**: This variable needs to include the directory where the `perftrack` executable was installed. By default, `perftrack` is installed to the directory `CMAKE_INSTALL_DIR/bin`.

- **PYTHONPATH**: If you are running `perftrack` in the source / build directory, you will need to set this variable to include the current directory, i.e. `PYTHONPATH="."`, so that it is able to load plugins.

4.2 Gathering Data – The PerfTrack Script Interface

Python script support implements: machine, build, run, and data parsing, and data loading. There are several options available for these tasks, with more or less of the work being done automatically, and the usual tradeoff between automatic processing and flexibility for custom approaches. Here we describe the different approaches.

4.2.1 Adding Machine Data Automatically

PerfTrack has scripts for automatically collecting machine data. Currently, we provide scripts that collect data for a single node at a time. To gather data about all nodes in a cluster, use some method, e.g. rsh, mpiexec, to execute the script on all nodes.

To gather machine data, execute the script `systemScan.py` on the target node. The script takes the following options:

```
--version               show program's version number and exit
-h, --help              show this help message and exit
-d DEBUGMODE, --debug=DEBUGMODE
                        Enable debug mode
-m MODULE, --module=MODULE
                        Name of Python module containing
                        attribute classes to
                        load. The default file is pt_proc.
-f CONFIGFILE, --configfile=CONFIGFILE
                        Name of configuration file containing
                        machine
                        configuration data.  The default file for
                        PTdf is
                        iMDL.cfg. The default for PERI xml is
                        PERIiMDL.cfg.
-o OUTPUTFILE, --outfile=OUTPUTFILE
                        Name of output file. Default name is
                        systemScan.ptdf
                        (or systemScan.xml if PERI output is
                        selected).
-p, --peri              Change output format to PERI xml. Default
                        is PTdf.
```

When you execute `systemScan.py` it generates a log file called hostSystemLog that can be used to help debug any problems with the script or to see a record of what the script collected and attempted to collect.

The **module** pt_proc.py is located in the PerfTrack lib directory. It contains routines for gathering machine data on AIX and Linux machines. This module will be used if you do

not specify one on the command line. You can choose to create your own module to collect data. The module is expected to contain Python classes that inherit from the base class gm_hs_tools.baseAttrib. Each class implements the following methods:

`attribFuncReq(self)`
>   This function returns 0 or 1. It returns 0 if `attribFunc` will succeed. It returns 1 if `attribFunc` will fail. This function performs a test to determine success or failure. For example, if `attribFunc` gathers CPU information on Linux, then `attribFuncReq` might check to see if the file /proc/cpuinfo exists on the machine.

`attribFunc(self, resName="")`
>   This function gathers the actual attribute value and returns it as a string. Currently, only attributes of type string are supported.

`attribDefaultName(self)`
>   This function returns the default attribute name, e.g. 'CPU MHz'.

`attribDefaultResourceType(self)`
>   This function returns the default resource type that this class collects data for, e.g. 'processor'.

`attribDefaultResourceName(self)`
>   This function returns the default resource name that this class collects data for, e.g. '0'.

In some cases, the data collection module will not know a priori the default resource name or type. In that case, the module can return 'Unknown.' When the script is executed, the user can override the default values using the configuration file.

The **configuration file** can be used to override the default values in the data collection module. The first section of the file defines the resource types and names that the script will gather data for. The next section defines the hierarchical relationships between the resource types. The last section overrides any default values in the classes in the data collection module. Any line that begins with a # character is considered to be a comment and is ignored.

PerfTrack resources and resource types are defined in the following format:

```
#ResourceType type:ResourceName
ResourceType 'machine':'Ruby'
ResourceType 'node':'ruby0'
ResourceType 'processor':'0'
ResourceType 'processor':'1'
```

PERI resources and resource types are defined in the following format:

```
ResourceType 'peri:resourceSet'
ResourceType 'peri:resource' ''
ResourceType 'peri:nodeList' ''
```

```
ResourceType 'peri:node' 'ruby'
ResourceType 'peri:memory' ''
ResourceType 'peri:cpu' '0'
ResourceType 'peri:cpu' '1'
```

PerfTrack resource hierarchies are defined with hierarchy lines:

```
# Hierarchy resType0/resType1/resType2
Hierarchy machine/node/processor
Hierarchy machine/node/networkCard
```

PERI resource hierarchies are defined similarly:

```
Hierarchy peri:resourceSet/peri:resource/peri:nodeList/peri:node/peri:cpu
Hierarchy
peri:resourceSet/peri:resource/peri:nodeList/peri:node/peri:memory
```

Attribute names and types can be overridden. In the following example override, we specify that the class `attribFunc_linux_CPUMhz` will collect data for the attribute named 'MHz' for resources of type 'processor':

```
# className          newAttributeName          newResourceType
'attribFunc_linux_CPUMhz'    'MHz'        'processor'
```

We can also override attribute names and types for PERI xml:

```
'attribFunc_linux_amountMemKB'    'mainKB'    'peri:memory'
```

4.2.2 Adding Machine Data Manually

You can also add machine data using a manual method. First, create a machine description file in the format described below. Then, use the PerfTrack script PTdFgen.py to convert the file to PTDF, which can then be loaded into the database using ptdf_entry.py.

The format of a machine description is:

```
BEGIN <mname>
<MachineAttrs>
<PartitionAttrs>
<NodeAttrs>
<ProcessorAttrs>
END <mname>
```

`mname:` The name of the machine. Note this name is not used as data. It is there as a marker for readability.

`MachineAttrs:` A list of machine attribute value pairs separated by newlines.
Attribute names cannot contain whitespace and machine attributes must begin with the prefix "Mach" followed by their values.

The only required attribute of `MachineAttrs` is `MachName`.

Examples:
```
  MachName   Frost
  MachType   cluster of smp
```

PartitionAttrs: A list of partition attribute value pairs separated by newlines. Attribute names cannot contain whitespace and must also begin with the prefix "Partition" followed by their values. `PartitionNames` is a list of the partitions separated by spaces that make up the machine, in order of node names. In other words, if node0 is in the interactive partition, then 'interactive' should be listed first. Also, if nodes in a partition are not contiguous, then the name of the partition should be listed multiple times, each for each partition 'chunk'.

The only required attributes of `PartitionAttrs` are `PartitionNames` and `PartitionSizes`.

Examples:
```
  For a machine layout like:
    thunder0-thunder21 interactive
    thunder22-thunder644 pbatch
    thunder645-thunder657 pvtune
    thunder658-thunder1007 pbatch
    thunder1008-thunder1023 pdebug
  You would put the following into the machine file:
    PartitionNames           interactive pbatch pvtune pbatch pdebug
    PartitionSizes           22 623 13 350 16
Required Attributes:
```

NodeAttrs: A list of node attribute value pairs separated by newlines. Attribute names cannot contain whitespace and must begin with the prefix "Node" followed by their values.

The only required attributes of `NodeAttrs` are `NodeNames` and `NodeNumProcs`.

Examples:
```
  NodeNames        {frost000, frost001, ..., frost067}
  NodeNumProcs     16
```

Note on `NodeNames` attribute: This attribute is meant to give the pattern of node names in the machine. From such a description, the script can infer that node names begin with 'frost' and have 3 digit number identifiers that run from 000 to 067. Curr ently, the script only supports node names of this format: nameXXX, where 'name' is the common prefix for the node names (e.g. frost) and XXX is numeric identifier, not necessarily of consistent length (e.g. for mcr, nodes are mcr0 and mcr100).

ProcessorAttrs: A list of processor attribute value pairs separated by newlines. Attribute names cannot contain whitespace and must begin with the prefix "Proc" followed by their values.

Examples：
```
  ProcType  RS/6000 Power 3 Nighthawk 2
  ProcMHz   375
```

After your machine description file is complete, use PTdFgen.py to convert the file to PTDF:

```
PTdFgen.py -m --machine_file <description file> --machine_out <ptdf
name> -v
```

4.2.3 Building and running the code with ptGo.py

ptGo.py can be used to build your application and generate a build data file. This build data file can be used multiple times for describing the build conditions for executions. ptGo.py has some limitations in the the kinds of Makefiles and compilers it supports (see below). Make sure that you do a clean build each time to ensure that the build file contains all the information about the build.

ptGo.py calls the script ptbuild.py to build your application and generate a build data file in the build directory. The build data file will be named perftrack_build_appname_dateTime_buildnode.txt, where appname is the name of the application as supplied to ptbuild.py, dateTime is the date and time of the build, and buildnode is the name of the machine on which the build takes place. This data file can be processed by PTdFgen.py, which will put the data into ptdf format for easy entry into the PerfTrack database.

ptGo.py can be used to run your application and generate a run data file. In order to run your application, ptGo.py takes the name of a batch file as an argument. ptGo.py requires the existence of certain lines in your batch file in order to get all needed runtime information. We expect this requirement to be relaxed in the future. However, for now, see batch_lim.txt to find out what is required in the batch file.

Each time ptGo.py is run, it creates a new directory for the build and run data files. The name of the directory is perfTrackData_XXXX, where XXXX is a timestamp.
The options to ptGo.py are:

```
  -h, --help
          show a help message and exit

  --build_only
          Use this option to indicate that you only want to build the
      executable. If you don't specify this, ptGo.py will attempt to
      run the executable.
```

```
--build_file=BUILDDATAFILE
        Use this option if you don't want ptGo.py to build your
    application.  Instead, you supply the name of the build data file
    that describes the build of the executable.

--app=APPNAME
        Here, you supply the name of the application that you are
    running. Examples: "irs", "sppm", "su3_rmd"

--CVSproj=CVSPROJ
        If you want ptGo.py to check out your project from CVS,
    here is where you indicate the name of the project to check out.

--trial=TRIALNAME
        You can specify the name of the execution here. This name
    will be used to identify the execution in the PerfTrack database.
    Note: You can also give the execution a name in your submission
    batch file. ptGo.py will extract the name specified in your batch
    file. Example: #PSUB -r Mpitest

--makeCmd=MAKECMD
        Here you can specify the name of the make command to use.
    For example, you may want to use gmake instead of make.

--makefile=MAKEFILE
        This option lets you specify the name of the makefile to
    use. This is helpful if your makefile is not named "Makefile."

--makeFlags=MAKEFLAGS
        Use this option to specify any flags you want to give to
    the make command.

--exeName=EXENAME
        Here you can specify the name of the executable. This is
    useful if ptGo.py has difficulty determining the name of the
    executable from the output from make.

--srcDir=SRCDIR
        Tell ptGo.py the location of the build directory. This
    should be the directory that contains the Makefile. This defaults
    to '.'

--buildHelp
        If you only want to build your application and you just
    want to see the options needed for building, execute ptGo.py --
    buildHelp to see them.

--run_file=RUNDATAFILE
        If you don't want to run your application, but want ptGo.py
    to process your run data file, you can supply the name of the run
    data file with this option.

--runHelp
        If you only want to see the options for running
    applications use this flag.
```

```
  --launcher
          Here you specify the name of the job launcher. This
      defaults to 'pbs'. Currently accepted values are: 'lcrm', 'pbs',
      and 'mpirun'. For command line launchers, e.g. mpirun, the
      arguments to the command should be in the file given to the
      --batchFile argument.

  --batchFile=BATCHFILE
          This flag is used to specify the name of the batch file
      that will be given to the launcher for executing your
      application.

  --inputDecks=INPUTDECKS
          Here you give the names of the input decks used by the
      execution. It should be a comma separated list of input decks.
      Example: --inputDecks input1,input2

  --pathToExe=PATHTOEXE
          Use this specify the path to the executable. This is needed
      so that ptGo.py can find the executable to determine which
      dynamic libraries are used by your application. This defaults to
      /current/working/directory + name specified to --exeName option

  -v, --verbose
          be a little verbose
  -V, --very_verbose
          be more verbose
  -q, --quiet
          be quiet
  --peri
          Use this option to output PERI xml data files.
```

Here are some examples of using ptGo.py:

1. You only want to build your application.

```
ptGo.py --build_only --app app_name --srcDir ./src --exeName app -V
```

This example shows how to use ptGo.py to build your application. It indicates
that the directory ./src contains the Makefile (--srcDir ./src), that
the name of the application is app_name and that the executable is named app.
By giving the -V option, you specify that you want to see the output of the
build.

After executing this, you will have a directory called perfTrackData_XXXX,
where XXXX is a timestamp, containing a single build data file.

2. You only want to run your application and use an existing build data file.

```
ptGo.py --build_file dirname/perftrack_build.txt --batch_file
mpi.script --inputDecks ../deck1,../deck2 --pathToExe ../build/myexe --
app app_name --launcher lcrm -V
```

Here you are using the build data file "perftrack_build.txt" located in directory "dirname". The name of the batch file to be used is mpi.scipt. The input decks are located in the parent directory of the current directory and are named deck1 and deck2. The executable is located in ../build. The batch scheduler is LCRM.

After executing this, you will have a new directory called perfTrackData_XXXX, where XXXX is a timestamp. ptGo.py will copy the specified build data file to the new directory and will put the new run data file in the new directory.

4.2.4 Known Limitations

4.2.4.1 Build Limitations

1. May have problems with makefiles that define environment variables, export them, and then use them in the make commands. It currently works for makefiles that define the variables in this format:
       var1=somevalue
       var2=somevalue

2. May get incomplete information for mpxlf when it tries to determine the libraries used by the MPI script.

3. Does not get library versions.

4. The compiler flags for each compiler are a concatenation of all the flags seen for that compiler during the build. This is not a problem if all of the source files that use the same compiler also use all the same flags. However, if some source files use different flags for the compiler than others do, the flags attribute of the compiler will be a concatenation of the two sets of flags.

4.2.4.2 Batch File Limitations

ptGo.py uses the script submission.py to get needed information about the runtime configuration of your execution from the batch file. Currently, submission.py requires that some lines exist in your batch file to get all the needed information. We expect to relax this requirement in the future.

However for the time being, submission.py requires the following:

1. For LCRM: A geometry specification line. Example:
#PSUB -g 4@tpn4
OR
#PSUB -g 4

OR
#PSUB -cpn 4
  This is needed to find out the number of processes per node in your
execution. If you specify the total number of tasks with the -g option (a line like: "#PSUB
-g 4"), you must precede it with a line that specifies the number of nodes, e.g. "#PSUB -
ln 2".

2. For LCRM: A constraint line that tells the name of the machine and the partition being
used. Example:
#PSUB -c frost,pbatch
  This is needed to determine the name of the runtime machine and the
partition on which it will execute.

3. For LCRM: A specification of the number of nodes used with -ln. Example:
#PSUB -ln 1
It only supports an exact number of nodes, not a range or minimum number, etc.

4. For LCRM: To give the execution a name in the PerfTrack database, you can use the -r
option. Example:
#PSUB -r irsMPI4p

5. For mpirun: A file that contains the command arguments. This file name is
given to the --batchFile argument to ptGo.py. Example:
-np 2 --nodes=node1,node2 /path/to/my/program --argsForMyProgram

Bugs/Limitations:
1. For LCRM: Calculation of tasks per node when using the -cpn option currently
depends on the script knowing how many processors per node there are on the machine
you are using. This is currently done assuming that the machine you run the run data
collection script is the same as the machines that the job will run on. It parses
/proc/cpuinfo on Linux and executes lsdev on AIX.

2. For LCRM and PBS: In order to parse the constraints line, we need to know
information about machines, namely their names and partition names. We rely on a file in
your homedir called: .ptconfig. In this file, specify machines with a line that says:
machine <machine name>

Example:
machine MCR
machine jacquard

Specify partition names with lines that say:
partition <machine name> <partition name>

Example:

partition MCR pbatch
partition jacquard interactive
3. For PBS: To get some execution details such as job exit status and job error messages, we need to parse the stdout and stderr from PBS. When you launch your job, specify output files for stdout and stderr like so:
#PBS -o pbs.out
#PBS -e pbs.err

and redirect your application's stdout to a different file:
mpiexec -np 2 my_application > my_application.out

4. For PBS and LCRM: PerfTrack doesn't currently get any arguments given to qsub or psub on the command line. It only sees what is in the batchfile.

4.2.4.3 Run Limitations

1. In order to get the dynamic libraries used by the application, the run data collection script must be run in the environment that the application will run in. This is because the dynamic library information is gathered with ldd.

2. Currently, on Linux we use the "df" command to get filesystem information, and on AIX we parse /etc/filesystems as well as use "df". If this command (or file for AIX) does not exist on your system, we need to know some basic information written to a file in your homedir called .ptconfig. In this file, specify filesystems with a line that says:
filesystem <filesystem name>

Example:
filesystem gpfs

To get version information about the filesystem, you can provide a command with a line that says:
versionCommand <filesystem name> <command> <args>

The <args> portion can contain the word $device. If you specify devices in the .ptconfig file, $device will be replaced with the name of a device.

If you do not provide a command to get the version, the scripts will attempt to gather the version information using some built-in commands.

Example:
filesystem gpfs
versionCommand gpfs /usr/lpp/mmfs/bin/mmlsfs $device -V
device gpfs /dev/scratch
device gpfs /dev/common
device gpfs /dev/tlproject

3. Sometimes we can't figure out the name of the run machine. In these cases, this can be remedied by specifying the name of the machine during the PTdf-generation step with the -M option. For example:
PTdFgen.py -d datadir -e -v -M jacquard

4.3 Data Parsing with PTdFgen.py (Creating PTdf files)

The script PTdFgen.py is designed to read performance data files with build or run information, and output PTDF format files for input into the PerfTrack database. The build and run data files are expected to be either generated by ptGo, or be in the format generated by those scripts (see files RunData.txt and BuildData.txt in the PerfTrack docs directory). The performance data files looked for are dependent upon the application.

4.3.1 Setup for PTdFgen.py

PTdFgen.py expects the files to be organized in a single directory with a PTrunIndex.txt file indicating what executions to expect. A separate .ptdf file is generated in the directory for each execution in the PTrunIndex.txt file. The format of the PTrunIndex.txt file is (Note: buildTime and runtime fields are *optional*):
`ExecutionName AppName Concurrency NumProcs NumThreads buildTime runtime`

`ExecutionName` is user-defined. This is the name you want to use to refer to the execution.
`AppName` is the name of the application resource, e.g. SMG2000, SPPM, IRS
`Concurrency` can have one of three values: MPI, MPI_OPENMP, OPENMP, where MPI indicates that it is an MPI-only run, MPI_OPENMPI indicates that it is a mixed MPI and OpenMP run, and OPENMP indicates that it is an OpenMP-only run.
`NumProcs` is the number of processes in the run.
`NumThreads` is the number of threads per process.
`buildTime` is the timestamp of the build (part of the build data file name).
`runTime` is the timestamp of the run (part of the run data file name).

Example:
irs-50 irs MPI_OPENMP 16 4 2005-03-25T11:08:54 FriMar2511:12:422005

Each execution in the directory needs to have a line as described above in PTrunIndex.txt. The data files for each execution need to be named `ExecutionName.extension`, where `extension` can have several values:
`ExecutionName.bld` is the PerfTrack build data file.
`ExecutionName.run` is the PerfTrack run data file.
If using PBS:
`ExecutionName.pbs.out` is stdout from PBS scheduler.
`ExecutionName.pbs.err` is stderr from PBS scheduler.

PerfTrack has performance data parsers for some benchmarks and tools. The following describes the extensions expected for the data generated by each benchmark or tool:

IRS data files:

`ExecutionName.hsp` contains the data from irshsp.

`ExecutionName.tmr` contains the data from irs.tmr.

`ExecutionName.ult` contains the data from irs.ult.

SPPM data files:

`ExecutionName.outputX`, where X is the rank of the process that output the file.

SMG2000 data file:

`ExecutionName.smg` contains the stdout from the run of SMG2000.

UMT2k data file:

`ExecutionName.rtout` contains the data from the rtout file from the run.

MILC su3_rmd data file:

`ExecutionName.su3_rmd` constains the stdout for the run.

mpiP files:

`ExecutionName.mpiP` contains the mpiP data.

Gprof files:

`ExecutionName.gprofX` contains gprof data in text format. X can be either the rank of the process that output the data or can be omitted if there is a single gprof file.

4.3.2 Running PTdFgen.py

The options that PTdFgen.py takes are:

```
  --version
          show program's version number and exit
 -h, --help
          show this help message and exit
 -d DATADIR, --data_dir=DATADIR
          the name of the data directory
 -m, --machine_data
          use this flag to parse machine data.
 --machine_file=MACHINEFILE
          the name of the machine data file
 --machine_out=MACHINEPTDF
          the name of the output PTdF file for machine data
 -e, --exec_data
          use this flag to parse execution data.
 -p PERFTOOLS, --perfTools=PERFTOOLS
          use this flag to specify the performance tools used.
    Expects a comma separated list of tools. Defaults to self
    instrumentation.
 -M MACHINENAME, --machineName=MACHINENAME
```

```
             the name of the machine that the executions ran on. This is
     only necessary when the run data collection scripts can't
     determine this information automatically.
 -P, --pbsOut
             specify that the script should look for files that end in
     .pbs.out and pbs.err that will contain the messages output to
     stdout and stderr by PBS
 -v, --verbose
             use this flag to get verbose output.
```

Example usage:
```
PTdFgen.py -d dataDir -e -p "self instrumentation,mpiP" -P -v
```

This example indicates that the directory containing the data and PTrunIndex.txt file is
called dataDir, and that dataDir contains execution data. Indicating "self instrumentation"
and mpiP as the performance tools means that there is benchmark output and mpiP
performance data to parse. The -P says that there is output from the PBS scheduler
containing more information about the job.

4.2 Data Parsing with PERIxmlGen.py (Creating PERI xml files)

If you didn't create PERI xml directly from ptGo.py, you can convert the data files
generated from the build and run data collection scripts to PERI xml using the
PERIxmlGen.py script. The arguments to this script are:

```
 --version               show program's version number and exit
  -h, --help             show this help message and exit
  -r RUNFILE, --run_input=RUNFILE
                         the name of the run data file
  -m MACHINEFILE, --machine_input=MACHINEFILE
                         the name of the machine data file
  -b BUILDFILE, --build_input=BUILDFILE
                         the name of the build data file
  -R RUNXML, --run_output=RUNXML
                         the name of the output XML file for run
                         data. The
                         default is the input name with .xml
                         appended to it.
  -M MACHINEXML, --machine_output=MACHINEXML
                         the name of the output XML file for
                         machine data. The
                         default is the input name with .xml
                         appended to it.
  -B BUILDXML, --build_output=BUILDXML
                         the name of the output XML file for build
                         data. The
                         defalt is the input name with .xml
                         appended to it.
```

Using this script is pretty simple. Here is an example:

```
PERIxmlGen.py  -b perfTrackData_2007-05-
29T15:32:28/perftrack_build_su3_rmd_2007-05-29T15:32:28.jacin03.txt -r
perfTrackData_2007-05-29T15:32:28/perftrack_run_su3_rmd_2007-05-
29T15:32:43..txt
```

4.3 Loading Data – PTDF files

PerfTrack can only load data that is in PTDF format. If you used PerfTrack to build and
run your application, and then used PTdfgen.py to convert the files, then you have PTDF
formatted output.

Load the data using the `ptdf_entry.py` script. This script takes one argument: either a
file name/path in ptdf format of the data to be loaded or a directory name of ptdf files to
load.

Examples:
```
ptdf_entry.py datafile.ptdf
ptdf_entry.py dataDirectory
```

Optionally, you can write your own Python script to load PTDF files using the
PTdataStore interface (described in more detail later). The function `storePTDFdata` takes
a file name as an argument, reads each PTDF line and stores it in the database. Here's an
example Python script that takes a single argument, the name of a PTDF file:

```
#!/usr/bin/env python
from PTds import PTdataStore
import sys

ptdfname = sys.argv[1]  # the name of the PTDF file
ptds = PTdataStore()    # create a PTdataStore object
debug = False
ptds.connectToDB(debug) # connect to the DB, not in debug mode
ptds.storePTDFdata(ptdfname) # load the PTDF file into the DB
ptds.closeDB()                 # close the database
```

4.4 The Data Store Interface

There are three ways to access PerfTrack's underlying data store: directly using an SQL
tool such as SQLplus; the PerfTrack GUI; and the Data Store interface. PerfTrack's Data
Store interface is implemented with Python, as a class called PTdataStore. Currently all
member functions of this class are directly accessible to users via python scripts. You
must include the PtdataStore module in your script, and its location should be included in
your PYTHONPATH setting.

4.4.1 Querying the Data Store

Here we describe the public methods. Naturally, PerfTrack is in a development phase
and changes will occur; there are methods in the PTdataStore class that are already slated

for change, and their use should be avoided, since they will probably change in the near future.

```
# manage a database connection #

# resources #
integer getResourceId(self, type, Name, parent_id)
integer findResourceByName (self, fullName)
integer findResource(self, parent_id, type, name, attributeList=[])
integer findResourceByShortNameAndType(self, type, shortname)
findResourceByNameAndType(self, resName, resType)  finds a resource
match for name and type. does not check attributes!
string getResourceName(self, id)
integer findResType(self, resTypeName)

# resource constraints #
boolean lookupConstraints(self, from_resource, toResourceList)
boolean lookupConstraint(self, from_resource, to_resource)
# attributes #
boolean lookupAttributes(self, resource, attrList)
boolean lookupAttribute(self, resource, attr, value, type="string")
# focus #
findFocusByID (self, resource_id_list)
findFocusByName (self, focusName)
```

4.5 Analyzing Data – The PerfTrack GUI

PerfTrack uses a comprehensive graphical user interface (GUI) to find and view performance data. The GUI issues queries based on a combination of resources and attributes that you select. Once the selected data is loaded from the database, you can use PerfTrack to display data in spreadsheet-like form, present bar charts, and save the data in comma-separated value (CSV) files for spreadsheet programs to import.
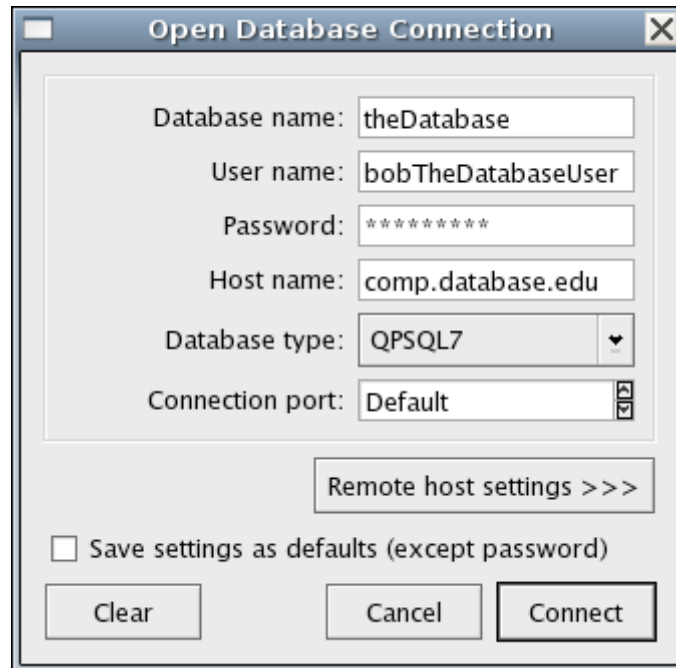
4.5.1 Connecting to a Database

Execute the PerfTrack GUI by opening the *perftrack* executable. You will first see the Open Database Connection window. (Figure 1) This allows you to set the database name, username, password, host, database type, and connection port for the database PerfTrack will access. Currently, Oracle, MySQL, and PostgreSQL databases are supported. The connection port is set by default and thus optional to specify.

In addition, you can also specify database information from just the command line by setting the following environmental variables. Upon opening PerfTrack, the values you specified on the command line appear as the default values in the Open Database Connection window.

- Database name: `PT_dbname`
- User name: `PT_dbusername`
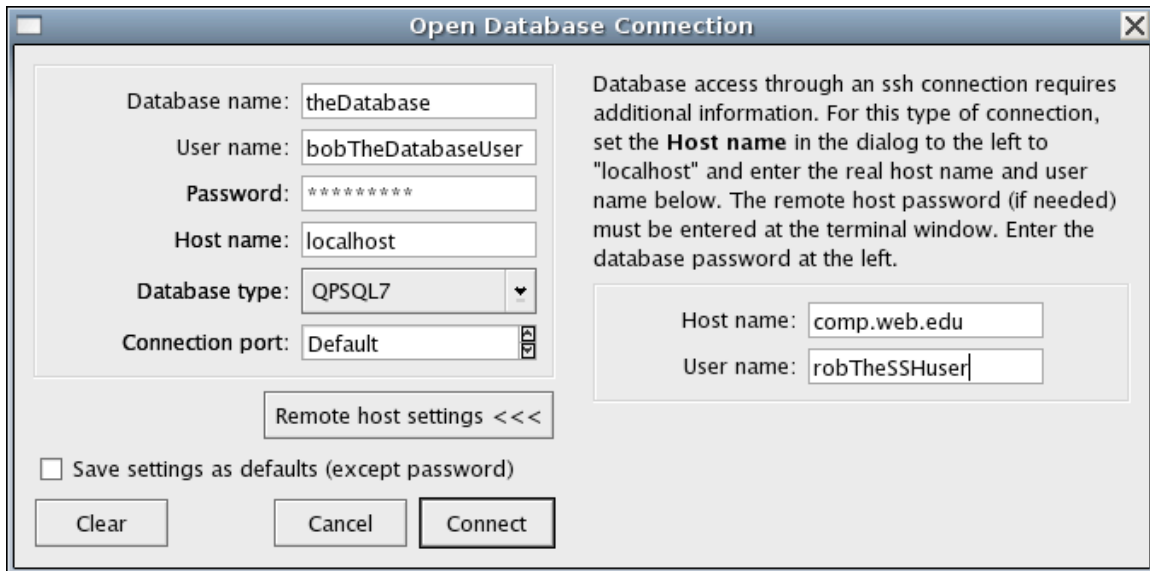
- Password: `PT_dbpass`
- Host name: `PT_dbhostname`
- Remote host name: `PT_remotehostname`
- Remote host user name: `PT_remotehostname`
- Database type: `PT_dbtype`
- Connection port: `PT_connectport`

Figure 1

To connect to a remote database via SSH, click the "Remote host settings" button below the database connection parameters. This will expand the window to allow you to enter the hostname and accompanying username of the computer on which the database is located. (Note: When connecting to a database in this way, be sure to replace the "Host name" field with "localhost" as in Figure 2) If the remote computer requires a password, it must be entered in the command line.

Figure 2

After all the information is entered, click "Connect" to access the database and display the records in it.

4.5.2 Selecting Data

Once PerfTrack connects to the database, you will see the Select Data screen. In this screen, you can view all the resources in the database and build a query to find certain data. To begin, click on the menu near the top left of the window which lists all the different resource types. After you select one, each object of that resource type will appear immediately below the menu. In the window below that, each attribute of that resource type appears. (Figure 3)

When dealing with hierarchical resources, double click the name of one of them to show *child resources*. This shows you only the child resources to that specific parent resource. Selecting the same child resource in the resource type menu would show *all* resources of that type.

In the "Selection Parameters" box, there is a 'D' displayed by default under the column "Relatives." (See below for other settings) This 'D' indicates that PerfTrack will search not only for all performance results with the specified resource name, but also for any of their descendents. This allows you to easily select related sets of performance results. For example, choosing the resource "Frost" defines a resource subset that also includes Frost's partitions, all of their nodes, and all of their processors.

Settings for "Relatives":

> **Tip:** The *question mark button*
>
>
>
> Click on the button, then click on any resource in the box below the button to see detailed information about that resource.

'**D**': Descendents
'**A**': Ancestors (Parent resources)
'**B**': Both Descendents and Ancestors
'**N**': Neither Descendents nor Ancestors, only the specified resource

Once you have selected the resource you want, click "Add to Selection Parameters." Complete this procedure to select as many types of data as you like; the "Items matching all parameters" label displays how many performance results match your specified parameters. Press "Get Data" to collect all these performance results from the database.

NOTE: Clicking "Add Resource Type" instead of "Add Selection Parameters" broadens the database query to find all performance data that have, in addition to the other any other specified Selection Parameters, *any* data of the specified resource type.
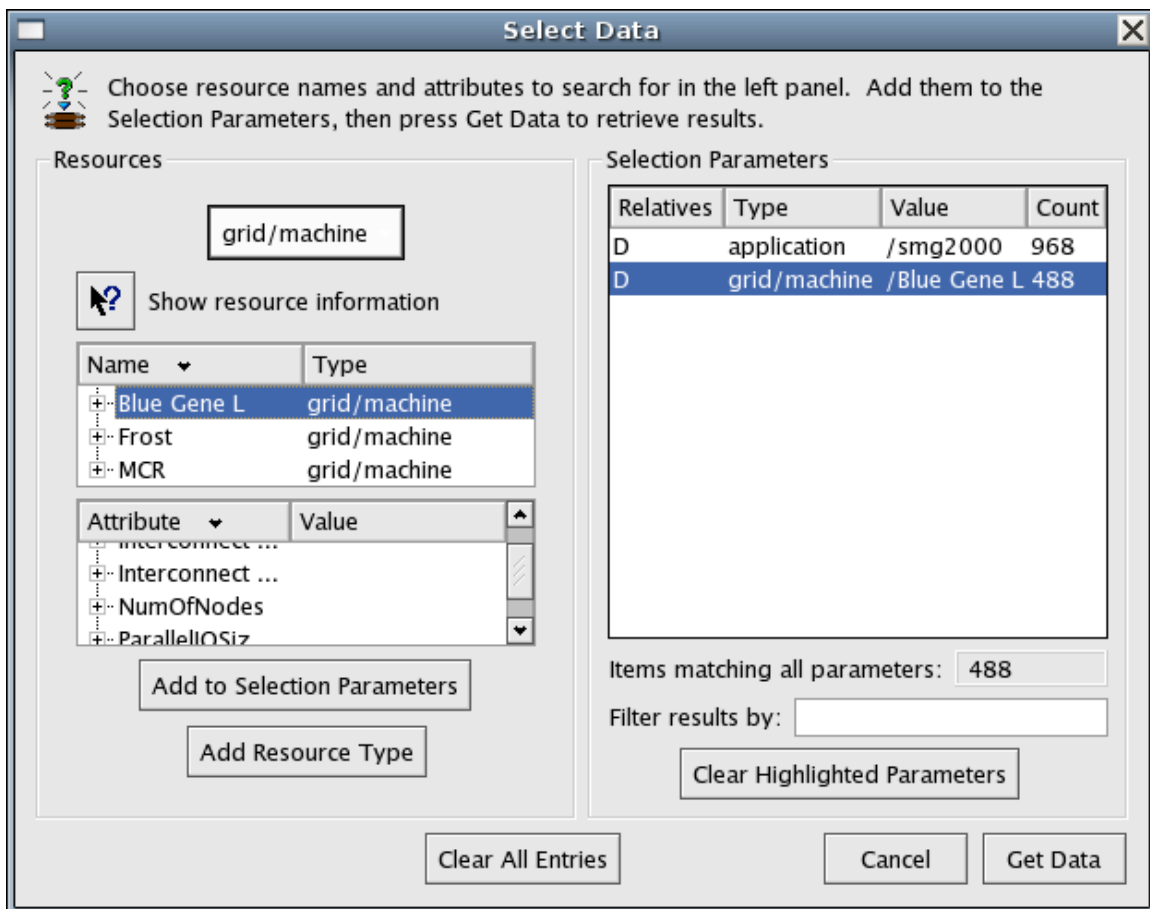
Figure 3

4.5.3 Viewing Data

When you click "Get Data", PerfTrack will search for any performance results matching all the parameters you specified, then present the data it finds in a new window: the main PerfTrack window (Figure 5).

Initially, all the resultant data is displayed but with little of the accompanying descriptive resource information. To add columns of such information, select "Add display parameters" under the "Data" menu. In the new "Add Data Columns" window, the resource and attribute types available for the data listed in the table are shown. The resources types listed include only those whose names are not identical for all the listed results. For example, if all the selected results came from applications run on Linux, the resource type **operatingSystem** would not be shown. Press "Get Data" to retrieve the additional resource information from the database the display it in additional columns in the table.

To filter data by value, select "Filter Results" under the "Data" menu. The window Data Filters appears. Select a parameter to filter, select what comparisons should be made, and what results you prefer. Then click "Apply." You can add more filters if you desire. In Figure 6, for example, only performance results whose metrics are lower than 0.5 will be shown.
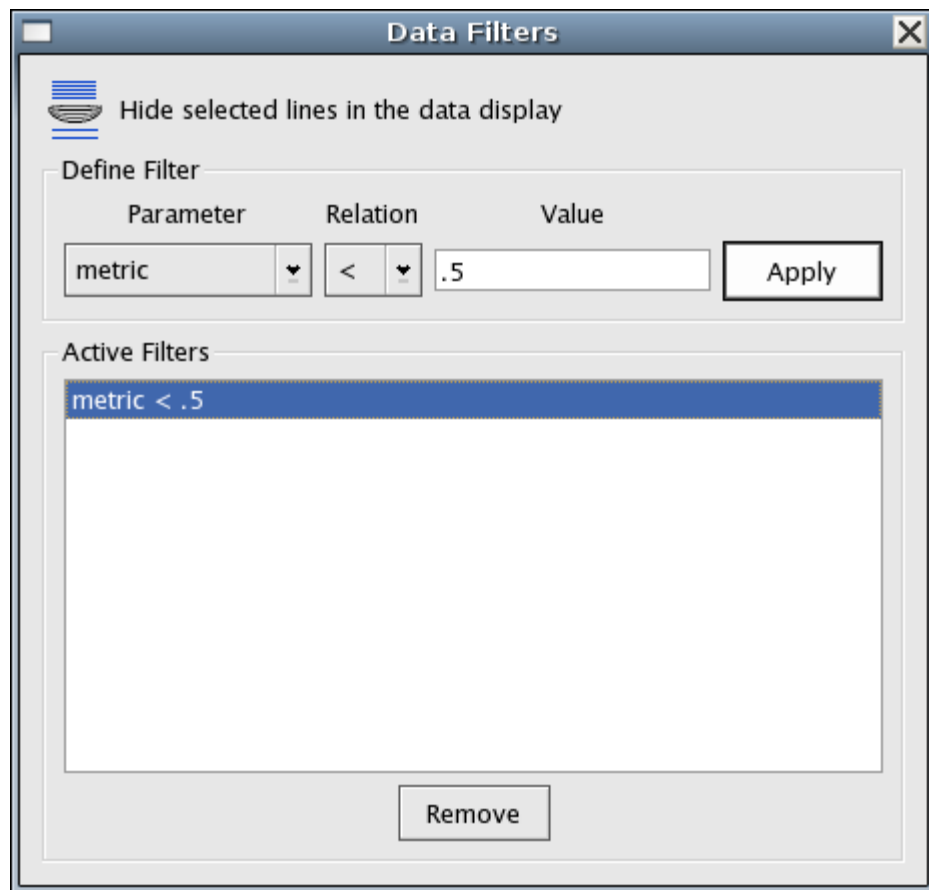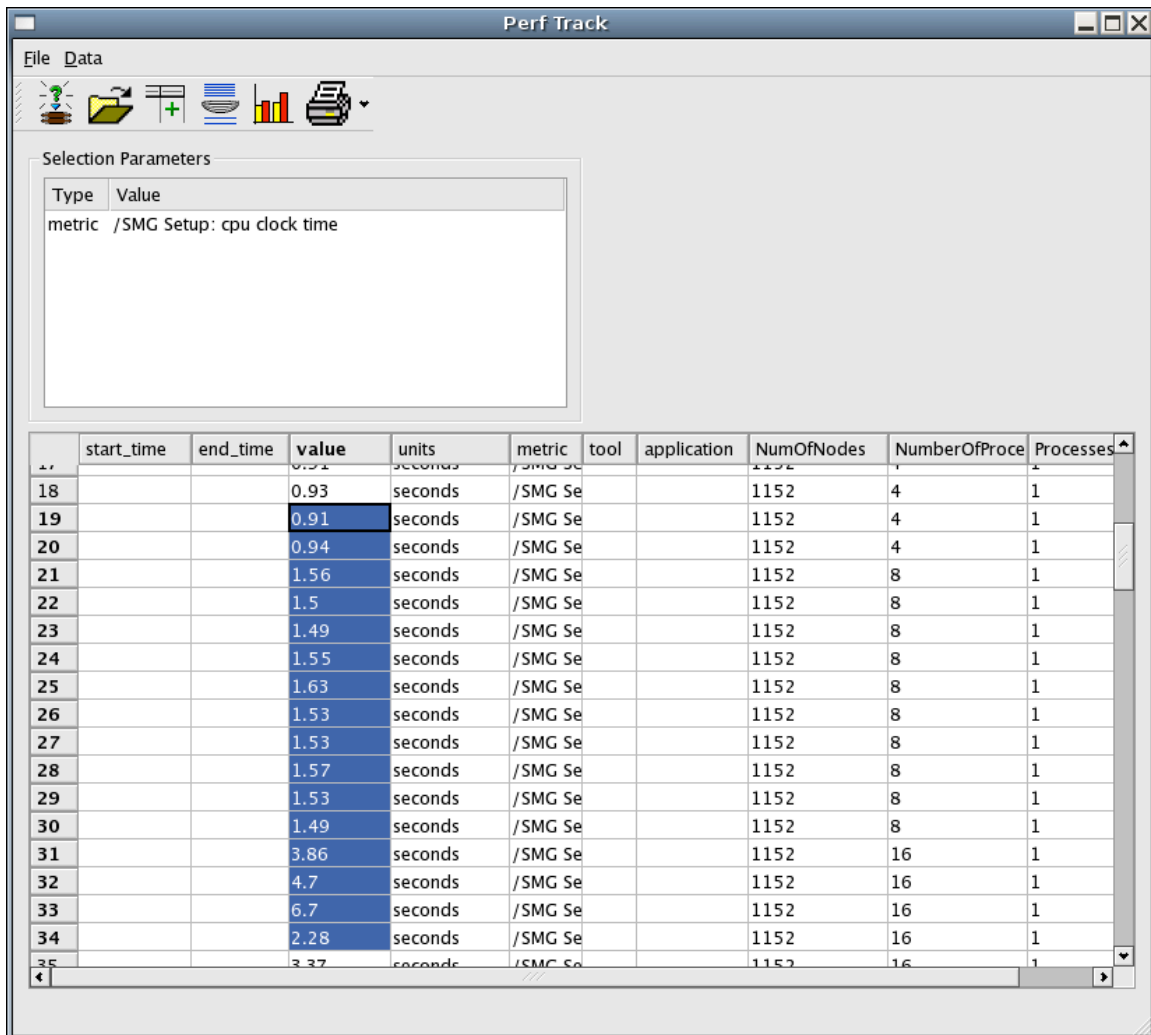
Figure 4

Figure 5

4.5.4 Plotting Data

To plot your performance results in PerfTrack, first select the data you wish to plot as in Figure 5. Then select "Plot Data" in the "Data" menu. This opens the Plot Data window (Figure 6).
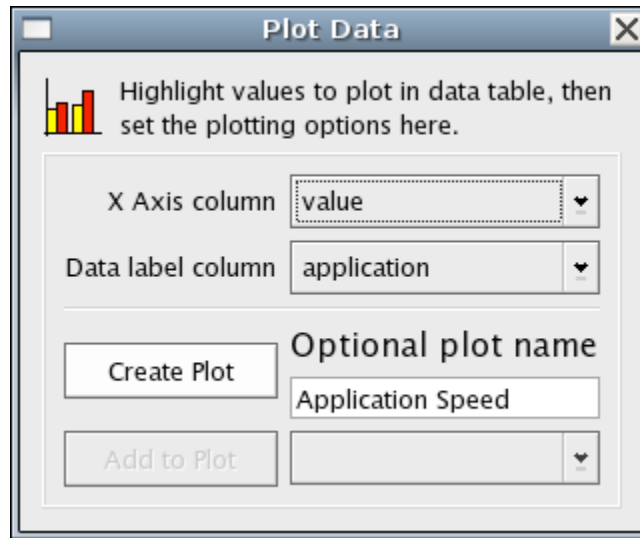

Figure 6

After you click "Create Plot" to chart the data (Figure 8), you may switch back to the PerfTrack window and select more data to plot. Then select "Plot Data" again and click "Add to Plot" (Figure 7).
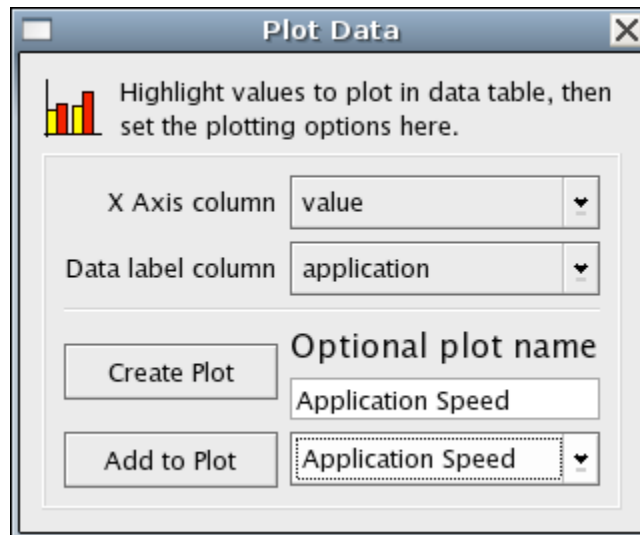

Figure 7

Figure 8

5 Index