perfbase

Joachim Worringen

perfbase

Joachim Worringen

Published January 2nd, 2005

Abstract

User Manual, Command Reference and XML Reference for perfbase, the system for experiment management and analysis.

Table of Contents

Part I. User Manual

The perfbase user manual is designed to show you how to do all the things you typically might want to do with perfbase in a step-by-step way. It does so by starting with an introduction into the problem that perfbase was designed to solve, and which approach it uses for this purpose. We then look at the installation of perfbase, continue with the setup of a new experiment, the modification of an existing experiment, the import of data into an experiment, and describe how to query the data previously stored. Next to this, the management of an experiment is also described.

Note that this manual is not a reference manual. The complete reference for perfbase is provided in the other two parts, namely Part II for the syntax of the command line interface, and Part III for the document type definition of the XML files that perfbase requires for certain tasks.

Table of Contents

Chapter 1. Introduction

perfbase is a software toolkit for experiment management and analysis. perfbase is designed to efficiently support all types of scientists and engineers who perform arbitrary computer-based experiments which deliver their results as text files.

perfbase allows to define experiments with an arbitrary number of parameter and result values of different data types. Unformatted experimental data can easily be processed to extract relevant information which is then stored in a database. perfbase features a clearly structured and powerful query language which allows to structure and post-process the data to create meaningful results. The results can be created as raw data, tables, or a variety of plots.

Problems Processing Experiment Output

In order to understand why perfbase can be a valuable tool, it is best to take a look on how experimental data is gathered, managed and analysed in the typical ad-hoc fashion, and to see which problems this widespread approach bears.

The way that the typical performance evaluation and result analysis is done is to store output data of the experiments (a run of a benchmark, application or simulation) in individual files. The format of the output data is typically ASCII text, as this is the easiest and most portable format to be generated from within any kind of software. Another advantage of ASCII files is that the content is still usable even when parts of the file are corrupted. All files are then organised by their name and sorted into directories. From these files, the data is in some way, either using some custom scripts or by manually copying the required data, fed into software for visualisation or analysis.

This naive, but widespread approach has a number of problems:

- Translating the benchmark output into the presentable form is often a tedious, manual task as the data needs to be extracted and transferred between different software tools.
- It is complex and error-prone to manage all results in a (big) number of files (of a certain type, usually text). It is not easy to discover which dimensions of the parameter space have not yet been measured precisely enough, or which one may need a closer look due to irregular results.
- Access to the output files is often difficult for people different from the one who performed the experiments that need the information contained within. Even with access, the way the output files are organised and the format of the file content can be hard to understand.
- Because of this complexity, the performance measurements are often limited in the range of the applicable test parameters, and in the number of samples taken for a certain set of test parameters. This leads to results of limited usefulness due to the unknown statistical variance in the results. This is especially true for application areas with a significant variation of results like testing of I/O performance, testing performance on a non dedicated system or running simulations which include error estimation.
- It is useful to track the performance development over a longer period of time or multiple software and hardware revisions. Typically, an analysis is done for a series of experiments at one point of time. Plotting and analysing data over time requires substantial additional efforts and is thus only performed by looking at the individual analysis – if the past analysis' are still available and have been generated in a way that allows a comparison.
- Generally, it is hard to perform a more complex analysis, like filtering the data by certain parameters or result values. Because the user knows this in advance, he will limit the parameter range of experiments to what be can processed with the naive approach he uses. This potentially leads to wrong

conclusions.

Experiment Management & Analysis with perfbase

perfbase solves these problems by automating the processing of the output data, extracting and storing its relevant content in a database, and offereing sophisticated and highly flexible methods to inspect, process, analyse, visualize and export processed data for furrther usage with other software like text processors or spreadsheets. The data analysis functionality includes many statistical functions. This way, it is easy to make sure that a sufficient number of "tries" of an experiment is performed to derive statiscally solid statements on the outcome and meaning of the experiment.

It is important to note that perfbase does in no way mandate or change the way the experiments are conducted as long as the output data consists of text files. perfbase also does not rely on a complicated software infrastructure; it is possible to install and run perfbase and the backend tools in a non-root user account. On the other hand, perfbase supports multi-user scenarios where different users contribute data to a single experiment and query this data with different levels of access rights.

Example Experiment

To illustrate the application of perfbase to an existing, unmodified benchmark, we will discuss one of the examples that come with the perfbase distribution. The benchmark is `b_eff_io`, an MPI-IO based test for performance of parallel file access. This benchmark creates two text ouput files, a detailed test protcol with suffix `.prot`, and a test summary with suffix `.sum`.

Chapter 2. First Steps

This chapter will show you how to get up and running with perfbase. This includes setting up the software under your account (if necessary), testing the installation and finally taking a look at an example pb_entity experiment. By the step-by-step description of this experiment, you will get to know the basic work-flow of perfbase™.

Installation of perfbase

This chapter is only relevant for users who want or need to set up their own pb_entity working environment. In case that someone else has already done this for you, please skip this section and proceed with XXX.

pb_entity supports different working environments. The differences are in the way that the required database server is operated, and if the pb_entity instance to be installed is to be available only for you or for other users as well. pb_entity supports all modes of operation, but some small differences have to be considered during setup.

Requirements

pb_entity has been developed and mostly used under Linux. Next to this, all operating systems that meet the requirements listed below are supported as well, but not necessarily validated. Before installing pb_entity, make sure your system meets the requirements that are listed below.

pb_entity System Requirements

Python	Python™ version 2.4 or later needs to be installed. Older versions of Python™ might work, but have not been tested.
Python Modules	To process XML files, the Python™ module ElementTree™ needs to be installed. To access the SQL database server, the module psycopg™ (version 1)is used and thus needs to be installed. Both of these modules are included with the pb_entity distribution. If necessary, you can download more recent versions from the web.
PostgreSQL	The experiment data is managed by an SQL database server. pb_entity supports PostgreSQL™, versions 7.4 and later. Older versions may work, too, but have not been tested. The database server does not need to be running as pb_entity can start it up by itself if necessary, using the commands **initdb** and **postmaster**. These commands are not needed if a database server which is already running is to be used. *Other SQL database servers, esp. MySQL, can not be used.*

Running the installation script

To install and set up pb_entity, simply run the included `setup` script which guides you through the process and also checks if the requirements described above are met. Detailed installation instructions are found in the file `README.INSTALL` that is included in the pb_entity distribution.

Setting up the perfbase database server

It helps if the **perfbase** command is located within your `PATH`. To verify, use the **perfbase version** command which should print a message like this:

```
$ perfbase version
perfbase release 0.6.1 (May 6th 2005), database version 2
(c) 2004-2005 C&C Research Labs, NEC Europe Ltd.
```

If this is successful, pb_entity itself is running, and the next thing to do is to configure the required database connection.

You need to decide if you want to use an existing database server (to access experiments which are stored n this server or to create new experiments on this server), or if you want to run you own "personal" database server which, by default, can only be used by you. *It is also possible to make a "personal" database server available for use by multiple users. The required configuration, however, needs to be performed on the database level as pb_entity offers no support for this.*

Using an Existing Database Server

The minimal information you need is the hostname of the system on which the database server is running. Set the environment variable `PB_DBHOST` to contain this value. Additionally, the environment variables `PB_DBPORT`, `PB_DBUSER` and `PB_DBPASSWD` may need to be set to match the configuration of the database server. Ask the administrator of the database server for the required information.

Setting up Personal Database Server

Make sure that the PostgreSQL™ commands **initdb** and **postmaster** are found. If necessary, adjust the `PATH` environment variable accordingly. Before being used for the first time, the database needs to be initialized using the **init**. This is not necessary if you use perfbase with an already running database server.

```
$ perfbase init
*# perfbase init: creating perfbase data directory /home/joe/.perfbase_data
*# perfbase init: initializing PostgreSQL database server
*# perfbase init: logging output in /home/joe/perfbase_init.log
*# perfbase init: database initialization completed.
```

Once the database server has successfully been initialized, you can actually start the database server via the **start**:

```
$ perfbase start
*# perfbase start: starting PostgreSQL database server
*# perfbase start: database server startup completed.
```

For the default usage of this database server, which is to run it on the same host on which the perfbase commands are executed, no further action is required. Otherwise, adjust the pb_entity environment variable `PB_DBHOST` accordingly.

Installing Custom SQL Datatypes

perfbase comes with a custom SQL datatype which allows PostgreSQL to natively work with version numbers like 1.1.4, 1.2rc2, 1.4.0.10 and so on. The installation of this datatype is not mandatory, but recommended if you plan to use vesion numbers as parameters to differentiate result data obtained with different versions of soft- or hardware. In principle, a version number can also be represented as a string, but this does only allow to query for identity, not for order: selecting all result data obtained with software versions earlier or later than a specified version is only possible when using the native version dataype.

More information concerning the installation of the version datatype can be found in `pgsql/version_datatype/README`.

Testing the Installation

The execution of pb_entity commands in general has already been tested above (by calling **perfbase version**). Now, the interaction between the pb_entity commands and the database server needs to be tested. For this purpose, let perfbase show all experiments that are available on the database server using the **info--all**. By adding the `-v` option, the command will give some additional information. Here is an example for a personal database server that has just been initialized and thus does not contain any experiment yet:

```
$ perfbase info -all -v
#* Database server localhost:5432 does not contain any perfbase experiments.
```

Although no experiment has been found, the output shows that the interaction with the database server is working correctly, and you should now proceed to the next section.

In case of a problem, the output would look similar to this example:

```
$ perfbase info -all -v
could not connect to server: Connection refused
Is the server running on host "dbserver" and accepting TCP/IP connections on port
```

The error message already gives information on the problem. Verify that the settings to access this server are correctly set:

```
$ env | grep PB_DB
PB_DBHOST=dbserver
```

All necessary parameters concerning the hostname, port number, user name and password need to show up here if they need to differ from the default values (see XXX). If there is no problem with these settings, ask the database administrator for help.

If you happen to be the database administrator, you should try to access the database server via the **psql** command.

```
 $ psql template1
Welcome to psql 7.4.5, the PostgreSQL interactive terminal.
Type:
  \copyright for distribution terms
  \h for help with SQL commands
  \? for help on internal slash commands
  \g or terminate with semicolon to execute query
  \q to quit template1=#
```

If local access to the database works as shown above, you might need to set up the database server to accept connections from remote nodes. This is done via the configuration file `pg_hba.conf` which you find in `$HOME/.perfbase_data` for the default setup. In this file, add a line like

```
host all joe 172.29.152.0 255.255.255.0 trust
```

This line gives user “joe” access to all databases on this server from all hosts within the 172.29.152.* subnet without a password. *Please note that this setting is inherently insecure as anyone who supplies the username “joe” will be provided access and is thus not recommended in untrusted environments. Other authentication methods (password, etc.) are supported by PostgreSQL™ as well, but need to be set up separately. See the PostgreSQL™ documentation for more information on how to do this.*

Example Experiment

Now that pb_entity is installed and set up, it's time to follow an example of how pb_entity can be used.

Chapter 3. Experiment Management

This chapter explains how a new experiment is initially set up and subsequently modified. You'll also learn how to transfer an existing experiment to a new database server and how to delete an experiment.

Setting up a New Experiment

Before setting up a new perfbase experiment, you need to determine what the parameter values and what the result values of your experiment are. Typically, this can easily be done by looking at things like command line or envrironment options that control the execution of the experiment, and at the output file(s) which show(s) the outcome of the experiment. Additionally, things like the operating environment might need to be considered, too. For each parameter and result value, you also need to determine if it has constant content throughout the experiment, or if its content changes in the course of the execution of the experiment.

Experiment Description

An experiment is defined by an experiment description, which is an XML formatted file. The related document type definition (DTD) is found in `perfbase/dtd/pb_experiment.dtd`. Each experiment description consists of three parts which provide some meta-information on the experiment, the parameter and result values of the experiment, and optionally information that gives users defined (limited) access rights. The according XML elements are described briefly in this chapter. For a detailed description, please consult the part *XML Reference*.

Meta Information

The meta information is made up by the elements `name`, `info` and `database` and their related subelements. The name of an experiment is case-insensitive: two experiments named `Test` and `test` can not exist on the same database server. Also, pb_entity does not provide means to change the name of an experiment once it has been created.

The subelements of the `info` element give information on the background of the experiment. It is recommended to actually provide meaningful and extensive information, especially for the elements `synopsis` and `description`.

The `database` element allows to define the database server to be used. However, in most cases this information will be provided via environment variables.

Multi-User Access Control

A perfbase experiment can be accessed by more than one user. Three different user classes do exist on the level of a perfbase experiment, which differ by their access rights:

admin_access	A user of the class `admin_access` has full, unlimited access rights to the experiment. Per default, the user who sets up an experiment is included in this user class implicitly if no other user is assigned to this class explicitly.
input_access	A user of the class `input_access` is thought to be a user who uses an existing experiment by adding new data (aka runs) to it and by performing queries. Such a user can not add, delete or change parameter or result values, can not delete runs (or the whole experiment), and he can not change the meta information of an experiment.

query_access	A user of the class `query_access` is thought to be a user who does not create any input data himself, but only performs queries. Therefore, this user class has all limitations of the `input_access` class, but is not allowed to create new runs. This means, a user of this class can not alter the experiment in any way.

For each user class element, an arbitrary number of `user` and `group` sub-elements can be specified.

Important

perfbase manages the access rights of different users to an experiment. However, it does not manage the *database* users. This means, to add a new user to an experiment, you first need to make sure that this user exists as a database user. PostgreSQL™ (or third party tools) offer different ways to add a user, like the command line tool **adduser**. Please refer to the PostgreSQL ™manual for more information of how to manage database users.

If an experiment description does not contain any user class elements, only the creator of the experiment will have access on the `admin_access` level. Likewise, if no `admin_access` has been defined explicitely, the creator of an experiment will be assigned to this user class implicitely. Any database user who is not assigned to any of the three user classes can not access the experiment at all.

Parameter and Result Values

Any perfbase experiment requires at least one result value, and any number of parameter values. The definition of parameter and result values is very similar. Each value needs a `name` and a `datatype`. Other elements are optional.

The name should be chosen like a variable name in a programming language: as short as possible, but as long as necessariy. The maximum length allowed by perfbase is 256 characters and is case-sensitive. Underscores in the name will be used to mark an index for many output formats (i.e. gnuplot): a value named `v_max` will be displayed as v_{max}.

The datatype is perfbase-specific and can be any of `integer`, `float`, `string` (text of up to 256 characters), `text` (unlimited length), `date` (day, month and year), `timeofday`, `duration`, `timestamp` (combination of date and timeofday), `binary` (byte values), `boolean` and `version` (version number; see explanaition below).

The `synopsis` should be a brief description of the value made up of very few words, like `message size` for a value named `S_msg`.

The `description` might look superflous at first glance for most values, but especially for non-obvious values, it pays of to describe the actual meaning and the background. This will help lateron if the original idea for this value is no longer present.

The `unit` is the physical or logical unit of the value. It will be used for the labeling in output (charts) created via queries, and also for correct scaling and conversion (like *speed multiplied with time is distance*). Therefore, it is defined as a term of `base_unit` and `scaling` elements. Valid base units are `none` (no unit), `%` (percent), `byte`, `bit`, `flop` (floating point operation), `op` (generic operation), `process` (in the sense of an execution context), `event` (generic), `s` (second), `m` (meter), `g` (gramm), `A`, `K` (Kelvin), `mol`, `cd` (candela), `Y` (currency Yen), `$` (currency dollar), `EUR` (currency Euro). Valid scaling factors are `Kilo` (10^3), `Mega` (10^6), `Giga` (10^9), `Tera` (10^{12}), `Peta` (10^{15}), `milli` (10^{-3}), `micro` (10^{-6}), `nano` (10^{-9}), `pico` (10^{-12}), `femto` (10^{-15}), `Ki` (2^{10}), `Mi` (2^{20}), `Gi` (2^{30}), `Ti` (2^{40}), `Pi` (2^{50}).
The use of different abbreviattions for 2-base and 10-base scale factors (i.e. `Mi` and `M`) is standardized and recommended as it is simple to apply and helps to remove ambigousness.

A `default` content can be specified for parameter and result values. In case that no content will be assigned to a value, it will return this default content when being queried. If the `default` element does not contain any text, it means that the default content should be `NULL`. Although values that return

`NULL` will not be considered in queries (with two exceptions), it means that it is possible to leave this value unassigned within a run. If a value has no default content defined, an assignment is mandatory: it won't be possible to create a new run without assigning content to this value.

The element `valid` can be used for parameter values of type `string` or `text`. It indicates which content is valid for this value. Any number of `valid` elements can be used within a parameter value. Content which does not match one of the valid strings will not be assigned to this value. Specifying valid content this way does not only guarantee that only these strings will be assigned to the value, but does also allow to look for these strings directly within the input data.

The `occurrence` attribute is very important for parameter and result values. It`s default content is `multiple`, which means that this value will have different content throughout a run because it may appear multple times in the input data. On the other hand, this attribute can be set to `once` which means there will be only one content for this value per run. This difference is relevant on different occasions when working with perfbase. Therefore, the correct setting for each value is very important. It can not be changed once a value has been created.

Setup Command

Once the experiment description has been created, the **setup** can be invoked, using the name of the experiment description file as the single parameter:

```
$ perfbase setup --desc=new_experiment.xml
```

If everything works well, the setup command does not create any output unless invoked with the verbosity option `-v`. The result of the operation can be verified using the **info** command:

```
$ perfbase info --exp=experiment_name
```

Changing an Existing Experiment

It is rather the rule than the exception that an initial experiment setup needs to be changed in the course of the experimental work. This can be done in perfbase using the `update` command without loosing any of the existing information. The following sections describe all possibilities. In all cases, it is necessary to create an XML experiment update description, following the document type definition provided by `pb_experiment_update.dtd`. The update command uses this description as input to perform the modifications. For some examples of changing (updating) an experiment, see the files in `test/update`.

Changing Meta Information

The parts of the meta information that can be changed are the project information like synopsis, description and who performs the experiment. The name and the owner of an experiment can not be changed.

Adding a Parameter or Result Value

To add a parameter or result value to an experiment, provide the same description that would be used to describe it when initially setting up the experiment (see chapter XXX). After the update, all existing runs will contain the new parameter or result value either with its default content (if specified), or without any content (null content). Parameter values with null content do not match any specific query, and result values with null content do not provide any data within a query. This means that existing queries will deliver the same results for all runs before and after the update.

Modifying a Parameter or Result Value

An existing parameter or result value can be modified in different ways which are described below. All types of modifications can be applied at once, using a single experiment update description. Multiple modifications of a single parameter or result value can be specified within a single `parameter` or `result` element.

Changing the Name

To change the name of a parameter, a `new_name` element has to be used. Example:

```
<parameter>
   <name>v</name>
   <new_name>v_1</new_name>
</parameter>
```

This will assign the name `v_1` to the parameter value `v`.

Changing the Default Content

To change the default content, or to assign a default content to a value which has no default content, specify an according default element. Example:

```
<parameter>
   <name>v</name>
   <default>0</default>
</parameter>
```

This will assign the default content `0` to the parameter `v`.
It is not yet possible to fully remove a default content once it has been specified. It can only be changed, also to an empty value.

Removing a Parameter or Result Value

A parameter or result value can be removed from an experiment by adding the attribute `action="drop"` to the element `parameter` or `result`. Example:

```
<parameter action="drop">
   <name>v</name>
</parameter>
```

This will remove the parameter `v` from the experiment.

All data for the parameter or result value is removed from the database and can not be recovered. Therefore, think twice before removing a parameter or result value!

Experiment Backup and Transfer

Transferring an experiment means to dump it into a file from the database server, and restore it from this file into the same or another database server. This is also a method for backing up an experiment. pb_entity supports such operations via the **dump** and **restore** commands

Dumping an Experiment

A complete representation of an experiment can be created using the **dump** command. This requires that the **pg_dump** command from the PostgreSQL distribution is in your path.

perfbase dump `--exp=exp_name`

This will create a single, automatically named file which can be used with the restore command to restore the experiment on any PostgreSQL™ database server. The option `--file=filename` (or `-f filename`) can be used to specify a different filename.

Note

Dumping an experiment is only possible with admin rights.

Restoring an Experiment

The **restore** command will create an active experiment from a previously created experiment dump file. If the name of the dump file was automatically created by pb_entity, only the name of the dump file needs to be specified. The **pg_restore** command from the PostgreSQL distribution needs to be in your path for this operation.

perfbase restore `--file=filename`

Otherwise, it is necessary to also specify the name of the to-be-restored experiment using the `--exp exp_name` (or `-e exp_name`) option.

Note

Dumping and restoring large experiments can take a considerable amount of time, and may create large dump files.

Deleting an Experiment

An experiment can be delete via the **delete** command. The experiment name needs to be specified explicitly; the environment variable `PB_EXPERIMENT` is not evaluated here to avoid unintentional deletion of an experiment.

Additionally, a warning message is printed and a verification question needs to be answered with `yes` to actually delete the experiment. This can be overridden using the `--dontask` option.

Note

A deleted experiment can not be restored, and no data can be recovered from an experiment once it has been deleted. Therefore, it is recommended to perform a backup before deleting an experiment.

Note

Deleting an experiment is only possible with admin rights.

Chapter 4. Data Import

After an experiment has been created, result data from experiment executions can be imported. pb_entity will create a run for each set of result data, consisting of singular content for parameter and result values with `occurrence=once` attribute, and an arbitrary number of data sets that assign content to the parameter and result values with `occurrence=multiple` attribute.

Creating an Input Description

An input description is required to describe how content from the result data, found in the input file(s), is assigned to the parameter and result values defined in the experiment. An input description is an XML file with a structure that complies to the document type definition `(DTD) pb_input.dtd`. According to this DTD, it may contain an arbitrary number of elements of the same or different type per value which describe where content for a value is to be extracted from the file.

In this section, an overview on the different means provided to extract content from the input files is provided. A detailed description of all valid elements and attributes defined in the DTD can be found in XXX.

Performing the Data Import

Basic Operation

The standard import operation is very simple to perform and does only require the specification of an input description and an input file which contains the output of the experiment execution:

perfbase `input --desc=input_desc.xml output.dat`

This example will parse the file `output.dat` according to the definitions in `input_desc.xml`. If successful, a new run will be created in the related experiment.

The input file can safely be deleted after a successful import or be kept for later reference. However, if the latter is desired, it is possibly a better idea to store the original content of the input file directly in the experiment database using the `--store` option.

By default, each input file can only imported once. pb_entity verifies the identity of a new input file and any of the input files that have been used to create existing runs by actually comparing the content, not the filename. Thus, using an identically named input file with different content will cause no conflict. This mechanism serves to avoid corruption of the experiment data which multiple runs created from the same input data would create. The option `--force` can be used to override this protection mechanism for all files specified for an import operation. For a single file in a multiple-file import (according to section XXX), the protection can be deactivated using the `--enforce=filename` option.

Handling Missing Data

One principle of pb_entity is to avoid the import of false data into the experiment. Therefore, the default behaviour when no content can be assigned to a parameter or result value is to issue a diagnostic message and abort the import without storing any data in the experiment. Other methods of handling such a situation, like using default content or interactively providing content, are supported, but the user has explicitly instruct pb_entity to apply these methods. By this, pb_entity ensures that the user always is aware of where the data for a newly created run stems from.

Two situations do exist in which pb_entity is not able to assign content to a value:

1. missing definition: The input description(s) used do(es) not contain a definition for a parameter or result value. This situation is detected and handled at the start of the import operation.

2. missing content: None of the definitions provided for a parameter or result value does retrieve valid content from the input file(s). This situation can not be detected and handled until all input files have been completely parsed.

In both cases, pb_entity will list the affected values, and the user has to decide how to handle the situation by choosing any combination of the three methods described below.

Provide Fixed Values on the Commandline

To provide static content for a value, a fixed value can be defined either within the input description or on the command line. For the first variant, refer to chapter XXX. To define a fixed value on the command line, use the `-f` option. As an example, to set the value `N` to `34`, use a command line like

perfbase input -f N=34 --desc `input.xml` `input.dat`

In both cases, the definition of a fixed value overrides all other possible definitions for a value in any of the used input descriptions, and no content from the input files will be considered.

This option is also very useful to run variations of a query from within a script file: it allows to use the same XML query descriptions to perform queries for different parameters.

Default Content

In the experiment definition, a default content can be defined for each parameter or result value. To actually use this default content for the case of a missing definition, the option `--use-default` (or `-u`) needs to be provided to the **input** command. .

In case of missing content, it is necessary to provide the option `--missing=default` (or `-a default`) to assign default content to a value.

This differentiation of the two cases missing definition and missing content ensures that in the case of a missing definition, no default content will be assigned to a value for which the user assumes the input files to provide content

Note

It is valid to specify empty default content. In this case, the content of a value will be NULL.

Interactive Input

Another way to assign content to parameter or result values is interactive input. In the case of a missing definition, the option `--stdin` (or `-i`) will request content at the start of the import operation. For missing content, use the option `--missing=ask` (or `-a ask`) to have pb_entity ask for input after all input files have been parsed.

No Content

If a missing definition should be ignored, it is necessary to specify empty default content to a value and use `--use-default` (or `-u`). If missing content should be ignored, provide the option `--missing=ignore` (or `-a ignore`).

Multiple Input Files and Multiple Runs

The standard usage of the **input** command is to apply a single input description onto a single input file by which a single run is created. However, it is possible to specify multiple input descriptions and/or input files on the command line, or to create multiple runs from a single input file. All non-standard usage scenarios are covered in the following sections.

Creating Multiple Runs from a Single Input File

If a single input file contains concatenated data from multiple executions of an experiment, it is desired to create the respective number of runs from it. This can be achieved by splitting up the file and using the method from the next section. However, this overhead can usually be avoided by defining a set separation which will split the single file into multiple input sets for the import operation.

A set separation can be defined explicitly via a `<set_separation>` element in the input description. A line in the input file which is matched by this element will trigger the creation of a new run. Content for the new run will be parsed from all lines of the input file including and following this match, until a line matches a `<set_separation>` again.

Am implicit definition of a set separation defines a `<named_location>` or an `<explicit_location>` to mark the start of a new input set. This is achieved by providing the attribute `is_separator="yes"` to the definition of this location element. If this element is matched, it will create a new run, and its content will be assigned to the value in this newly created run.

Applying a Single Input Description on Multiple Input Files

If a specification of a single input description is followed by multiple input file names, this input description will be applied to each of the input files to create a new run for each of it. Example:

perfbase input -d inp_desc.xml output_1.dat output_2.dat

This also allows for a convenient import like

perfbase input -d inp_desc.xml *.dat

Applying Multiple Input Descriptions on Multiple Input Files

If an experiment executions produces multiple files which all provide content for a single run representing this execution, one solution would be to concatenate all input files to create a single input file. This overhead can be avoided by applying a dedicated input description on each input file. This can be achieved by ordering the ordering the specification of the input descriptions and the input file names on the command line:

perfbase input -d inp_desc_1.xml output_1.dat -d inp_desc_2.xml output_2.dat

This will apply the input description `inp_desc_1.xml` to `output_1.dat`, and `inp_desc_1.xml` to `output_1.dat`, and a single run will be created. It is important to understand that each input file will only be parsed according to the definitions in the related input description.

Parameter Sets

It is a typical situation that multiple runs are performed in an environment in which a part of the parameter values does not change for all these runs. An example is to perform benchmarks on a specific machine, and parameters like CPU type and speed are the same for all runs. However, it is often that just these parameters are not described in the output data of the benchmark as it is not concerned with this information. The way to provide these parameter values with valid content would be to either set them as fixed values on the command line (using the `-f` switch of the **input** command), or to gather this information in a separate file and import it from there on each run in addition to the benchmark's output file(s). The latter approach bears the problem that pb_entitiy by default refuses to import data from the same file more than once. This can be overcome by using the `--enforce` command line switch. However, both methods are error-prone and not very convenient.

To conveniently handle this situation, pb_entity offers parameter sets. A parameter set is a named set of parameter values with `occurrence=once` attribute and its respective content. This means, a parameter set belongs to an experiment and defines the content of one, multiple or all parameters of an experiment.

To define a parameter set, the input command needs to be run with an input description that will retrieve content for the parameter values to be contained in the set from one or more input files. This is similar to a standard import operation. The difference is that no result values and parameter values with `occurrence=multiple` attribute are parsed, and the option `--pset-store=`*`name`* has to be provided. It specifies the `name` of the parameter set to be created, which has to be unique within the experiment. All users with input and admin rights can create a new parameter set. However, admin rights are required to modify an existing parameter set by overwriting it.

The currently defined parameter sets of an experiment can be listed using the **info** `-v` command. To see the full definition of a parameter set `name`, use the **info** command with the `--pset=name` option.

If an **input** command refers to a parameter set `name` using the `--pset-use=name` option, it will set the content of all parameter values of the run it creates to the content of those parameter values from the parameter set for which content has been specified. All parameter values which have no defined content defined in the parameter set will have their content be retrieved from the input files. This means that content from a parameter set overrides content that eventually is found in the input files.

How do I...

Frequently asked questions and the respective answers.

Chapter 5. Information Retrieval

Retrieving Information on Experiments

The info command allows to list the experiments that are available on a database server, retrieve information on the experiments and print data stored in the individual runs of an experiment.

To list all experiments on a database server, use

perfbase info --all

As always, adding the option `-v` generates a more verbose output. Once you have determined the name of the experiment of interest, you can get more detailed information via

perfbase info --exp=experiment_name

This will also print the number of runs within the experiment. To get information on a specific run, you need its ID. The option -i will print the IDs of all runs within an experiment:

perfbase info -e experiment_name -i

For a given run ID, the option -r will print information on this run:

perfbase info -e experiment_name -r ID

To actually see the data stored within a run, specify the option `-d` with either *`once`*, *`multiple`* or *`all`* as parameter to list the content of all values with appearance of once or multiple, or all values:

perfbase info -e experiment_name -r ID -d all

More sophisticated methods to find and list data stored within an experiment are described in the next two sections.

Listing the Content of an Experiment

To get a quick overview of the content of an experiment, perfbase provides the **ls** command. It allows to list the content selected values of the runs within an experiment, similar to the **ls** command of a shell which lists the content of a directory.

By default, only the IDs of all active runs of the experiment are listed. Generally, one line is printed for each run, and the output is sorted by the first column:

perbase ls -e experiment_name

To list the content of specific values, add their names on the command line, like:

perfbase ls -e experiment_name --show=T_0 --show=S

This will list the content of the values T_0 and S. The run IDs are not listed if any value is explicitely listed. However, it is possible to specifcy meta-values to be shown alongside with the other values using the *`--meta`* parameter (abbreviated as *`-m`*). Meta values are chosen by their 4-letter acronym. The parameter *`--help`* lists all available meta acronyms. To additionally list the run ID and the synopsis of each run for the example above, specify

perfbase ls -e experiment_name -s T_0,S -m indx,syno

You will have noticed the abbreviation of the --show parameter.

The content of values which have multiple different content per run is not printed. Instead, a placeholder #Nvalues# is printed to indicate that the value has N different contents in this run. Use the option `--nvals=N` to print up to N different contents of any value. The different contents are separated by the character '|'.

A very useful feature of the **ls** command is the possibility to list all *distinct* contents that a value has within the experiment. If you would i.e. like to know all different contents of the value cpu_arch, you would do

perfbase ls -e experiment_name -s cpu_arch --distinct

With the `--distinct` parameter, one line per value is printed which lists all distinct content of this value throughout the experiment.

Finding Specific Data Sets within an Experiment

While the ls command lists data from *all* active runs within an experiment, the **find** command allows to filter the runs by a large variety of conditions. The parameters `--show` and `--meta` work the same as for the **ls** command. The filtering of the runs is done via additional parameters. Any number of conditions can be specified on the command line. *All*conditions need to be fulfilled by a run to be listed.A typical case would be to list only data from runs where the content of a filter matches a condition. I.e., to show the content of T and the run ID of all runs where the value D is greater than 1 and less than 5, specify

perfbase find -e experiment_name -s T -m indx --cond=D>1 --cond=D<5
The characters '>' and '<' need to be escaped to avoid an interpretation by the shell. Typically, you need to replace '>' with '\>' and '<' with '\<' to get the expected behaviour:

perfbase find -e experiment_name -s T -m indx -c D\>1,D\<5

The conditions can not only filter for the content of values, but also for meta information of a run. To list the content of value T for all runs performed yesterday or later and with a synopsis that starts with "NFS", do

perfbase find -e experiment_name -s T --performed-from=yesterday --synopsis="^NFS.*"

The synopsis is specified as a regular expression. All supported parameters are listed when specifying the `--help` parameter.

Chapter 6. Query Operations

Structure of a Query

Queries are a very important part of perfbase as they actually are the way to create insights from all the data that you have gathered. A query is an operation which extracts data from the experiment which satisfies formulated conditions, and furtheron performs logical, arithmetic or statistical operations on this data. Finally, the data is formatted and possbily fed into other applications to create plots or other reprensations like formatted tables. A simple example is the task "generate a line plot of the minimal latency of operation *op* on platform *p1*". Once you've seen this plot, you might want to see the *difference* of the minimal latencies of operation *op* on the platforms *p1* and *p2*. Both of these task can very easily performed with a perfbase query, and the second query will require only very few modifications of the first query due to the data-stream component-based architecture of perfbase queries.

A query is formulated as an XML formatted text file validated agains the DTD `pb_query.dtd`. This DTD defines a number of elements which perform different actions and are interconnected via references to the ID of each element. This way, a directed acyclic graph of query elements is defined which controls the data flow from the bottom (the experiment database providing the raw data) to the top (the formatted output data). The resulting graph can be a simple chain or any sort of tree.

At the bottom of the graph, one or more `source` elements extract data from the runs of the experiment. The data consists of one or more result values, each of which will contribute one vector of data, and zero or more parameter values, which also contribute one data vector each. The `source` element references zero or more `parameter` elements which define filter conditions for parameter values. Only data from runs and datasets which match the conditions of these `parameter` elements is passed up to the next level via the described data vectors. At the very top of the graph, one or more `output` elements present the final data in one of multiple available output formats, like raw text, tables, XML formatted data or graphical plots in two or three dimensions. Between the bottom and the top, any number of `operator` and `combiner` elements can be placed to process the data (apply filters, perform reductions and statistical operations, evaluate terms to derive new data, etc) and control the flow of the data.

Within this chapter, we will illustrate the use of the different elements with a simplified example experiment named PMB. The full version of this example is provided in `perfbase/examples/PMB`. This experiments processes the result data of a message passing benchmark which measures the *latency* (result value `T` wiht multiple occurence) of different *operations* (parameter value `op` with multiple occurence) across varying *message sizes* (parameter value `S_chunk` with multiple occurence) and *number of processes* (parameter value `N_proc` with multiple occurence). This benchmark is run on different platforms or in different software environments, characterized by a number of parameter values with single occurrence like the *version of the message passing library* (`MPI_version`) and the used *interconnection network* (`interconnect`).

Every XML query description needs to be nested wihin a pair of `query` tags:

```
<query>
   ... query content ...
</query>
```

This is omitted in the examples that follow to increase readability.

`source` Element

The source element extracts datasets from the database based on arbitrarily defined filters and limits. The datasets that a source element provides consists of at least one result value and any number of parameter values. I.e., to create a 2-dimensional plot, a single result value and a single parameter value (both with multiple occurrence) need to be specified. These two will make the source element generate a vector with the elements being <parameter, result> tuples. These tuples can then easily be plotted or presented as a table using an output element (discussed later in this chapter).

```
<source id="simple_source">
   <result>T</result>
   <parameter>
      <value>S_chunk</value>
   </parameter>
</source>
```

Another way to write the same query is to use "external" parameter elements. This will alllow to reuse a single parameter element within multiple source elements. This will simplify the maintenance of the query and at the same time increase the readability. The extenal parameter element is then referenced from the source element via an input element, using the content of its `id` attribute:

```
<parameter id="p_chunk">
   <value>S_chunk</value>
</parameter>

<source id="simple_source">
   <result>T</result>

   <input>p_chunk</input>
</source>
```

This basic source element will generate a vector with *all* pairs of message size (`S_chunk`) and latency (`T`) that are stored within the experiment. This is rarely the intention of the user. Instead, he wants to limit the range of the datasets to be provided by the source to match certain criterias. This can be achieved using more parameter elements as described in the next chapter.

`parameter` Element

A parameter element can not only define a parameter value to provide its content in the output vector of a source element, but can also limit the selection of datasets to match user-specified criterias. This applies to both, parameter values with single and multiple occurrence. This is again illustrated using the PMB example. We modify the query to only show datasets of benchmark runs that were performed on a system with a Myrinet interconnect using a version of the message passing library equal or greater than 1.0, but below 1.1:

```
<parameter id="p_chunk">
   <value>S_chunk</value>
</parameter>

<parameter id="p_ic">
   <value>interconnect</value>
   <filter>
      <equal>Myrinet</equal>
   </filter>
</parameter>

<parameter id="p_version">
   <value>MPI_version</value>
   <filter>
      <greaterequal>1.0</greaterequal>
      <lesser>1.1</lesser>
   </filter>
</parameter>
```

```
<source id="simple_source">
   <result>T</result>

   <input>p_chunk</input>

   <input>p_ic</input>
   <input>p_version</input>
</source>
```

The new element used here is the `filter` element within the `parameter` element. It defines the conditions which the datasets need to match to be included in the output vector of the `source` element. The filter element typically includes one or more condition sub-elements which define the conditions which may equally express numerical, boolean or string relations (depending on the data type of the parameter value). If more than one condition elements are provided, they per default are used to create a boolean `AND` condition. This behaviour can be controlled via the `boolean` attribute being set to `and` or `or`.

As the parameter values used for the filtering in the example above are both of the type *single occurrence*, the output vector will look as without filtering, only being shorter. The specified filter conditions are not visible, though. However, the available output formats provide (different) means to include this information in the final result of the query, like column titles in a table or labels in a plot. The inclusion of the filter conditions in the

`fixed` Element

`run` Element

`series` Element

`operator` Element

Operators are intensively used in queries as they process the data that the sources deliver. There are many different types of operators, and a single operator can work in different ways depending on which kind of input data it works on. This section gives a brief description and a table with the most relevant characteristics of all available operator types For a full XML reference of the `<operator>` element, see section XXX.

The available operators differ in various characteristics which are explained below. It is important to understand these differences to create queries which perform as desired.

Reduction Operators	Some operators (like `max` or `sum`) are reduction operators which calculate one value for an input vector of arbitrary length. If such an operator processes an input from a `source` object, it behaves differently from operating on input from `operator` elements. *For this reason, mixing input of `operator` and `source` elements is not supported.* If the input is generated by a `source` object, it is possible that it contains data from multiple runs which all match the specified parameter filters. Therefore, the reduction is performed elementwise across the input vectors from the different runs. The reduction operator determines the matching elements by matching the parameters

	of the elements. The result of this reduction is again a vector. If a single operator provides input for the reduction operator, the input vector is reduced to a single value (i.e. the largest value for the `max` operator). Finally, if multiple operators provide input, the reduction is again performed elementwise across all input vectors, generating a single result vector. The matching of the elements is again performed by matching the parameter values of each element. If no parameter values are provided, the matching is done via the order of the elments within the vector. In this case, it might be necessary to apply the `order` operator before.
Algebraic Operators	Algebraic operators perform an algebraic calculation on exactly one, exactly two or an arbitrary number of input vectors. An example is the `diff` operator which calculates the difference between each element of exactly two input vectors. The result of an algebraic operator is a single vector. The number of elements of the result vector depends on the number of elements of the input vector(s), and the type of the elements of the result vector depends on the type of the elements of the input vector(s).
Transforming Operators	Some operators transform the input vector(s) into a partially or totally new vector (with respect to the number or type of elements). I.e., the `distrib` operator calculates a statistical distribution of the input vector's elements. Here, the number of elements of the result vector is determined by the bin width, and the values represent propabilities. Other transforming operators cut of parts of the input vector, or filter out data from specific runs.

`null`

Supported attributes:

None

`sum` and `prod`

This operator calculates the sum respectively the product of one or more input vectors.

`max` and `min`

This operators determine the maximum respectively the minimum element of an input vector. If the object that generates the input vector is of type `source`, the operator will determine the extreme of each vector of

`scale` and `offset`

These operators perform simple algebraic operations: `scale` multiplies all result value of the input vectors with a constant factor, while `offset` adds a (positive or negative) constant to all result values.

The `scale` operator is also able to adapt the unit of the value to be scaled accordingly. I.e., if the unit of a value is `kg`, and the scale factor is 1000, the unit will be adapted to `g`. For this mechanism to work, it is necessary that the scale factor is specified in terms of a scale prefix (like `k`, `M`, `ki`, `Mi`, etc.). To scale a value down using prefix symbols, they need to be specified like `1/k` (for a scale factor of 10-3).

Operator type	Algebraic
Number of input vectors	1
Number of result values per vector	1
Supported attributes	value — The factor (for `scale`) or constant (for `offset`). Data type has to be numeric. For the `scale` operator, scaling prefixes are also supported (see description above), optionally preprended with a `1/` term.

`avg`, `stddev` and `variance`

These operators calculate the arithmetic avg (`avg`), the standard deviation (`stddev`), or the variance (`variance`), respectively.

Operator type	Reduction
Number of input vectors	1
Number of result values per vector	1
Supported attributes	None

`diff` and `div`

These operators calculate the elementwise difference (`diff`) respectively the fraction (`div`, for division) of two input vectors. The assignment of minuend and subtrahend (respectively dividend and divisor) is determined by the order of the `input` elements.

Operator type	Algebraic
Number of input vectors	2
Number of result values per vector	1
Supported attributes	None

`percentof`, `above` and `below`

These operators put two input vectors into relation by elementwise calculating the ratio of two vector's elements (percentof) or the increase (`above`) respectively the decrease (`below`) that transforms one vector's elements into the other vector's elements. The order of the input elements determines the order of the elements in the term.

For two input vectors A and B, two corresponding terms for the elements a and b are as follows:

percentof	a / b * 100

above	(a / b - 1) * 100
below	(1 - a / b) * 100

Operator type	Algebraic
Number of input vectors	2
Number of result values per vector	1
Supported attributes	None

`count`

`eval`

This operator allows to define arbitrary terms to be calculated. Both, variable values from input vectors as well as constants and singular parameter values can be used in the definition of the term.

Operator type	Algebraic
Number of input vectors	one or more
Number of result values per vector	one or more
Supported attributes	None

`median` and `quantile`

Operator type	Reduction
Number of input vectors	1
Number of result values per vector	1
Supported attributes	Value

`distrib`

`sort`

`slice`, `latest` and `oldest`

`runindex`

The `runindex` operator does not modify data, but instead replaces for each data value of the result vector with the index of the run that it belongs to. This is useful for parameter optimization, like to answer the question *Which setting of parameter P gives the maximum performance?*

It often makes sense to use the output of this operator as input for the `param` operator.
The run index of a data value is not always available. I.e., if a data value was created from multiple different data values that belong to different runs (like the `avg` operator being applied on a mix of data values), no single run index can be related to this data value. In this case, the run index is set to `-1`.

`param`

The param operator transform a run index into the content of a only-once parameter from this run. This operator requires a `runindex` operator to provide the run indexes for input.

`limit`

The limit operator removes all datasets where the content of the result value does not satisfy a specified condition. A typical application of this operator is to let only pass datasets where the numerical content of a result value is larger or smaller than a given threshold.
If you want to apply a band filter (only content above threshold A and below threshold B should pass), you need to concatenate two appropriate limit operators.

Combiner Element

A combiner element does not modify any data, but is used to direct streams of data. Especially, it allows to combine multiple output streams into one input stream for another element. The data vectors of parameter and result values are not changed; however, duplicate vectors are removed.

A combiner can also be used in conjunction with sweep queries: the independent data vectors created for each sweep variant can be merged into one single object which contains multiple vectors.

Output Element

Performing a Query

Running the query command.

Handling Error Messages

How do I...

Question & answer section on common query problems.

Part II. Command Reference

Table of Contents

Chapter 7. Generic Behaviour

All pb_entity commands share some common behaviour, which is explained within this chapter.

Environment Variables

All pb_entity commands evaluate a set of environment variables for their internal configuration.

Supported Environment Variables

`PB_DBHOST`	Hostname of the system on which the database server to be used is running (default: localhost)
`PB_DBPORT`	Port number on which the database server is accepting incoming requests (default: 5432).
`PB_DBUSER`	Username to be used to access the database server (default: login name).
`PB_DBPASSWD`	Password to be used to access the database server (default: no password).
`PB_EXPERIMENT`	Name of the default experiment to be used by all pb_entity commands that require an experiment name. The content of this environment variable can still be overridden by the command line option.

Important

The content of an environment variable is overridden by a possible setting of the same variable on the command line (see next section) or within an XML file used by any pb_entity command.

Common Options

Each pb_entity command accepts the following options:

Common Command Line Options

`--dbhost=`*`host`*	Specify the hostname of the system on which the database server to be used is running (default: localhost)
`--dbport=`*`port`*	Specify the port number on which the database server is accepting incoming requests (default: 5432).
`--dbuser=`*`user`*	Specfiy a username to be used to access the database server (default: login name).
`--dbpasswd=`*`passwd`*	Specify the password to be used to access the database server (default: no password).
`--version` or `-V`	Print the version information on this command and exit. *All commands of a pb_entity release carry the same version information.*
`--verbose` or `-v`	Give some more information on what is happening While executing the

	command.
`--help`	Show a command's synopsis and a short explanation of all available arguments.
`--sqltrace`	Print all SQL commands that are executed to standard output. This can i.e help to determine the reason if a query does not return any data at all.
`--debug`	Print debug information to standard output (including all SQL commands).

Important

Configuration information provided via one of these command line options does override a possible respective setting of an environment variable and also a possible setting within an XML file processed by this command.

Configuration File

pb_entity stores some configuration data in a configuration file (`$HOME/.pbconf`).

Important

This file must not be edited manually unless you know exactly what you are doing. pb_entity may cease to function if invalid content is found in this file.

Chapter 8. Command Syntax

init

Name

init -- Set up a (personal) database server for perfbase experiments.

Synopsis

`perfbase init` [-d *path*]

Description

The **init** command is used to set up a PostgreSQL™ database server to store perfbase experiments. While a database server set up via the **init** can be used either as a personal (single-user) server or a multi-user server, the most common case will be to only use this command for a single-user application. For a multi-user application of perfbase, it is necessary to further configure the server to support remote connections, user authentication and more.

*Internally, **init** uses the PostgreSQL™ command **initdb** to set up a database server for the current user.*

Parameters

`-d` *path*	Store the database files in the directory *path*

start

Name

start -- Start a (personal) perfbase database server.

Synopsis

`perfbase start` [-f] [-p <*port*>]

Description

The **start** command is used to start a PostgreSQL™ database server on the local machine. It is a more convenient way than launching the PostgreSQL database server manually.
If the PostgreSQL server is started by other means, this command is not needed.

Parameters

`-f`	Force the startup even if the command assumes that there's already a database server running on this machine. This is required if multiple database servers should run a single host listening to different ports.
`-p` *`port`*	Specify a port number that will be used by this instance of the database server to listen for incoming requests.

stop

Name

stop -- Stop a perfbase database server.

Synopsis

```
perfbase stop
```

Description

The **stop** command is used to stop a PostgreSQL™ database server that is running on the local machine. It is a more convenient way than stopping the PostgreSQL database server manually.
If multiple PostgreSQL database servers are running on the local machine, this command will stop all servers which are running under the user account of this command.

setup

Name

setup -- Create or modify a new or existing perfbase experiment.

Synopsis

`perfbase setup` [-d <*xml*>] [-g] [-f] [-u]

Description

The **setup** command creates a new perfbase experiment based upon an XML formatted experiment description, modifies an existing experiment, or creates an XML formatted experiment description of an existing experiment.

Parameters

`-d` *`xml`*	Specify the file name of an XML formatted perfbase experiment description or experiment update description.
`-f`	Force the startup even if the command assumes that there's already a database server running on this machine. This is required if multiple database servers should run a single host listening to different ports.
`-p` *`port`*	Specify a port number that will be used by this instance of the database server to listen for incoming requests.
`-g`	Create an XML experiment description for an existing experiment. The data is written to standard output.
`-u`	Modify an existing experiment. In this case, the XML file passed via the -d option needs to be an experiment update description.

update

Name

--

Synopsis

`perfbase update` [-d <*xml*>] [-g]

Description

The command ...

Parameters

```
-d xml
-g
```

input

Name

--

Synopsis

`perfbase` [-d <*xml*>] [-g]

Description

The command ...

Parameters

`-d` *`xml`*
`-g`

query

Name

--

Synopsis

`perfbase` [-d <*xml*>] [-g]

Description

The command ...

Parameters

```
-d xml
-g
```

info

Name

--

Synopsis

`perfbase` [-d <*xml*>] [-g]

Description

The command ...

Parameters

```
-d xml
-g
```

ls

Name

--

Synopsis

`perfbase` [-d <*xml*>] [-g]

Description

The command ...

Parameters

`-d` *`xml`*
`-g`

delete

Name

--

Synopsis

`perfbase` [-d <*xml*>] [-g]

Description

The command ...

Parameters

```
-d xml
-g
```

find

Name

--

Synopsis

`perfbase` [-d <*xml*>] [-g]

Description

The command ...

Parameters

```
-d xml
-g
```

dump

Name

--

Synopsis

`perfbase` [-d <*xml*>] [-g]

Description

The command ...

Parameters

`-d` *`xml`*
`-g`

restore

Name

--

Synopsis

`perfbase` [-d <*xml*>] [-g]

Description

The command ...

Parameters

`-d` *`xml`*
`-g`

check

Name

--

Synopsis

`perfbase` [-d <*xml*>] [-g]

Description

The command ...

Parameters

`-d` *`xml`*
`-g`

version

Name

--

Synopsis

`perfbase` [-d <*xml*>] [-g]

Description

The command ...

Parameters

`-d` *`xml`*
`-g`

help

Name

--

Synopsis

`perfbase` [-d <*xml*>] [-g]

Description

The command ...

Parameters

```
-d xml
-g
```

Part III. XML Reference

pb_entity is controlled via XML-formatted files. The pb_entity commands **setup**, **input** and **query** each process XML control files that comply to a specific Document Type Description (DTD) which are called experiment description, update description, input description and query description, respectively.

Table of Contents

Chapter 9. Common Elements

Some elements are valid for each XML control file. These common elements are described in this chapter.

Database Information

There are different ways to let the pb_entity commands know which database server is to be accessed. The method with the lowest priority is to specify the access information in the XML control file. Any setting of an according environment variable or command line argument will override the information provided within the XML control file.

`database`

element name	database
occurrence	at most once
description	Specification of the database server to be accessed.
default	See default values of sub-elements.

The valid elements within a `database` element are listed in the table below.

element name	occurrence
`host`	at most once
`port`	at most once
`user`	at most once
`passwd`	at most once

`host`

element name	host
content type	parsable text
description	hostname of the system running the database server
default	`localhost`

`port`

element name	port
content type	parsable text
description	port number on which the database server is listening
default	`5432`

`user`

element name	user
content type	parsable text
description	user name to be used to access the database server
default	*current login name*

`passwd`

element name	passwd
content type	parsable text
description	clear text password to authenticate with the database server
default	empty

Tip

It is not recommended to store the password within the XML control file. Instead, set the environment variable `PB_DBPASSWD` accordingly.

Chapter 10. Experiment Description

Each experiment requires a description that contains the name of the experiment, some meta information that describes the experiment in more detail, access right assignment and finally the description of the parameter and result values that typically make up the most part of the experiment description.

Experiment Name

element name	name
occurrence	exactly once
content type	parsable text
description	`name` does contain text representing the experiment name. The name needs to be unique within a single database server. Upper case characters within the name are mapped to their lower case counterparts - this means, two experiments named `Foo` and `foo` cause a conflict on the database server because in both cases, pb_entity will create a database named `pb_foo`.
default	no default content

Meta Information

`info`

element name	info
occurrence	exactly once
content type	sub-elements
description	Describes by whom the experiment was created, and for which purpose it has been created.
default	no default content

The valid elements within an `info` element are listed in the table below.

element name	**occurrence**
`performed_by`	exactly once
`project`	at most once
`synopsis`	exactly once
`description`	exactly once

`performed_by`

element name	performed_by
content type	sub-elements
description	Real name and association of the creator of this experiment.
default	no default value defined

name

element name	name
content type	parsable text
description	Real name of the user who created this experiment.
default	no default value defined

organization

element name	organization
content type	parsable text
description	Association of the user who created this experiment.
default	no default value defined

project

element name	project
content type	parsable text
description	Name of the project within which this experiment was conducted.
default	empty

synopsis

element name	synopsis
content type	parsable text
description	One-sentence-description of the experiment (will be shown i.e. by the **info** command when listing an experiment).
Default	no default value defined

description

element name	description
content type	parsable text
description	Arbitrary-length description of the experiment (will be shown by the **info** command for a verbose experiment listing).
default	no default value defined

Multi-User Access Control

pb_entity allows multiple users to work on the same experiment at the same time. However, by default only the creator of an experiment can access an experiment (he can do so in an unlimited way). To allow other users to access an experiment, it is necessary to explicitly assign access rights to individual users or groups they belong to. Three different levels of access rights do exist (see explanations of the different elements).

Note

The names of users and groups in this context are managed by the database server, not the operating system and thus do not necessarily relate to the login and group names of the respective users in the operating system environment!

Refer to the PostgreSQL™ administration manual for help on how to manage users and groups of the database server.

`admin_access`

element name	admin_access
occurrence	at most once
content type	sub-elements
description	Lists all individual users or groups which have administration access to this experiment. This means that all users listed here, or all members of the listed groups, have full access to the experiment database. They can add or remove parameters and values, modify the existing parameters and values, clean up the database, dump and even delete the experiment. Of course, these users can also input data and perform queries.
default	The user who created the experiment is the only administrator.

The valid elements within an `admin_access` element are listed in the table below.

element name	**occurrence**
`user`	at least once
`group`	at least once

`input_access`

element name	input_access
occurrence	at most once
content type	sub-elements
description	Lists all individual users or groups which have input access to this experiment. This means that all users listed here, or all members of the listed groups, can import new data into the experiment using the **input** command. Otherwise, they have read-only access which allows to use the **query** commands and all other command which do not modify the database.
default	Only the user who created the experiment has input access (in its role as administrator).

The valid elements within an `input_access` element are listed in the table below.

element name	**occurrence**
`user`	at least once

element name	**occurrence**
`group`	at least once

query_access

element name	query_access
occurrence	at most once
content type	sub-elements
description	Lists all individual users or groups which have query access to this experiment. This means that all users listed here, or all members of the listed groups, have read-only access which allows to use the **query** command and all other commands which do not modify the database.
default	Only the user who created the experiment has query access (in its role as administrator).

The valid elements within an `query_access` element are listed in the table below.

element name	**occurrence**
`user`	at least once
`group`	at least once

user

element name	user
content type	parsable text
description	user name to which the related access rights are granted
default	no default value defined

group

element name	group
content type	parsable text
description	group name - the related access rights are granted to all members of this group
default	no default value defined

Parameter and Result Values

The core part of an experiment description are the parameter and result values that make up the effective data of an experiment. Each value needs to have a name and a data type. A short synopsis, longer description, physical (or logical) unit, default value or listing of valid values are optional.

parameter

element name	parameter

occurrence	at least once
content type	sub-elements
description	A parameter value is a value that is set to a pre-determined content before or during the execution (run) of an experiment. An example for this is the clock frequency of the CPU on which a software is executed (this content is only set once for each run), or the size of a dataset to be processed which is set to different values within a single run.
default	no default content

The valid elements within a `parameter` element are listed in the table below.

element name	occurrence
`name`	exactly once
`synopsis`	at most once
`description`	at most once
`datatype`	exactly once
`unit`	at most once
`valid`	at most once
`default`	at most once

result

element name	result
occurrence	at least once
content type	sub-elements
description	A result value is a value for which the content depends on the individual execution (run) of an experiment. An example for this is the total execution time (this content is determined only once for each run), or the time for a single iteration which is determined multiple times during an experiment.
default	no default content

The valid elements within a `parameter` element are listed in the table below.

element name	occurrence
`name`	exactly once
`synopsis`	at most once
`description`	at most once
`datatype`	exactly once
`unit`	at most once
`valid`	at most once
`default`	at most once

name

element name	name
content type	parseable text
description	Each parameter and result value needs to have a name which is unique within the experiment. It is recommended to limit the length of the name to a few characters. This helps to avoid that i.e. plots containing labels for multiple values are hard to read because of long label strings. Many output devices (like gnuplot™) support enhanced text formatting i.e. for subscript and superscript using underscore and caret: `T_min` will be printed as T_{min}, and `N^x` will be printed as N^x.
default	no default value defined

synopsis

element name	synopsis
content type	parseable text
description	A short synopsis for this parameter or result value consisting of only a few words. The synopsis should always be supplied as it will i.e. be used to label an axis in a plot.
default	empty

description

element name	description
content type	parseable text
description	A description of arbitrary length. Feel free to describe the value as thorough as possible! This includes explanations why it seems necessary to include this value at all. Such information can often prove very useful when looking at an experiment after a longer time, or when someone else has defined it.
default	empty

Chapter 11. Update Description

Adding a Value

Changing a Value or Information

Removing a Value

Chapter 12. Input Description

Meta Information

set_separation

fixed_value

filename_location

explicit_location

named_location

tabular_location

A tabular location is used to parse a tabular arrangement of content consisting of an arbitrary number of columns and lines.

split_location

A split location is a special case, different to all other location types. While for all other location types, the content of a value is collected from a single line of the input file, a split location allows to gather partial content from two separately triggered lines and derive the content to be assigned to the value from the two partial contents. This location type was created to support the parsing of trace-like input files. A typical example is the case where a timestamp is written to a file when entering and leaving a function. Using the split location, it is possible to assign the execution duration of the function, which is the difference of leave and enter time stamp, to a value. This value is typically part of a dataset which contains other values like the function name etc.

derived_parameter

A derived parameter assigns content to a parameter value which is not directly found from the input file, but is calculated from the content of one or more other parameter values. While this does not add additional information to the experiment, it might be desired to work with the content this derived parameter value instead of multiple other parameters. I.e., if an input file contains information on the number of nodes and the number of processes per node, the total number of processes can be assigned to a derived parameter.

attachment

Chapter 13. Query Description

Chapter 14. Search Description