Pegasus 4.8.2 User Guide

Pegasus 4.8.2 User Guide

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Overview and Features

Pegasus WMS [http://pegasus.isi.edu] is a configurable system for mapping and executing abstract application workflows over a wide range of execution environments including a laptop, a campus cluster, a Grid, or a commercial or academic cloud. Today, Pegasus runs workflows on Amazon EC2, Nimbus, Open Science Grid, the TeraGrid, and many campus clusters. One workflow can run on a single system or across a heterogeneous set of resources. Pegasus can run workflows ranging from just a few computational tasks up to 1 million.

Pegasus WMS bridges the scientific domain and the execution environment by automatically mapping high-level workflow descriptions onto distributed resources. It automatically locates the necessary input data and computational resources necessary for workflow execution. Pegasus enables scientists to construct workflows in abstract terms without worrying about the details of the underlying execution environment or the particulars of the low-level specifications required by the middleware (Condor, Globus, or Amazon EC2). Pegasus WMS also bridges the current cyberinfrastructure by effectively coordinating multiple distributed resources. The input to Pegasus is a description of the abstract workflow in XML format.

Pegasus allows researchers to translate complex computational tasks into workflows that link and manage ensembles of dependent tasks and related data files. Pegasus automatically chains dependent tasks together, so that a single scientist can complete complex computations that once required many different people. New users are encouraged to explore the tutorial chapter to become familiar with how to operate Pegasus for their own workflows. Users create and run a sample project to demonstrate Pegasus capabilities. Users can also browse the Useful Tips chapter to aid them in designing their workflows.

Pegasus has a number of features that contribute to its useability and effectiveness.

- **Portability / Reuse**

  User created workflows can easily be run in different environments without alteration. Pegasus currently runs workflows on top of Condor, Grid infrastrucutures such as Open Science Grid and TeraGrid, Amazon EC2, Nimbus, and many campus clusters. The same workflow can run on a single system or across a heterogeneous set of resources.

- **Performance**

  The Pegasus mapper can reorder, group, and prioritize tasks in order to increase the overall workflow performance.

- **Scalability**

  Pegasus can easily scale both the size of the workflow, and the resources that the workflow is distributed over. Pegasus runs workflows ranging from just a few computational tasks up to 1 million. The number of resources involved in executing a workflow can scale as needed without any impediments to performance.

- **Provenance**

  By default, all jobs in Pegasus are launched via the **kickstart** process that captures runtime provenance of the job and helps in debugging. The provenance data is collected in a database, and the data can be summarised with tools such as **pegasus-statistics**, **pegasus-plots**, or directly with SQL queries.

- **Data Management**

  Pegasus handles replica selection, data transfers and output registrations in data catalogs. These tasks are added to a workflow as auxilliary jobs by the Pegasus planner.

- **Reliability**

  Jobs and data transfers are automatically retried in case of failures. Debugging tools such as **pegasus-analyzer** helps the user to debug the workflow in case of non-recoverable failures.

- **Error Recovery**

When errors occur, Pegasus tries to recover when possible by retrying tasks, by retrying the entire workflow, by providing workflow-level checkpointing, by re-mapping portions of the workflow, by trying alternative data sources for staging data, and, when all else fails, by providing a rescue workflow containing a description of only the work that remains to be done. It cleans up storage as the workflow is executed so that data-intensive workflows have enough space to execute on storage-constrained resource. Pegasus keeps track of what has been done (provenance) including the locations of data used and produced, and which software was used with which parameters.

- **Operating Environments**

Pegasus workflows can be deployed across a variety of environments:

- *Local Execution*

Pegasus can run a workflow on a single computer with Internet access. Running in a local environment is quicker to deploy as the user does not need to gain access to muliple resources in order to execute a workfow.

- *Condor Pools and Glideins*

Condor is a specialized workload management system for compute-intensive jobs. Condor queues workflows, schedules, and monitors the execution of each workflow. Condor Pools and Glideins are tools for submitting and executing the Condor daemons on a Globus resource. As long as the daemons continue to run, the remote machine running them appears as part of your Condor pool. For a more complete description of Condor, see the Condor Project Pages [http://www.cs.wisc.edu/condor/description.html]

- *Grids*

Pegasus WMS is entirely compatible with Grid computing. Grid computing relies on the concept of distributed computations. Pegasus apportions pieces of a workflow to run on distributed resources.

- *Clouds*

Cloud computing uses a network as a means to connect a Pegasus end user to distributed resources that are based in the cloud.

Workflow Gallery

Pegasus is curently being used in a broad range of applications. To review example workflows, see the Example Workflows chapter. To see additional details about the workflows of the applications see the Gallery of Workflows [http://pegasus.isi.edu/workflow_gallery/].

We are always looking for new applications willing to leverage our workflow technologies. If you are interested please contact us at pegasus at isi dot edu.

About this Document

This document is designed to acquaint new users with the capabilities of the Pegasus Workflow Management System (WMS) and to demonstrate how WMS can efficiently provide a variety of ways to execute complex workflows on distributed resources. Readers are encouraged to take the tutorial to acquaint themselves with the components of the Pegasus System. Readers may also want to navigate through the chapters to acquaint themselves with the components on a deeper level to understand how to integrate Pegasus with your own data resources to resolve your individual computational challenges.

Document Formats (Web, PDF)

The main version of this document is intended to be viewed online at the Pegasus website [https://pegasus.isi.edu/doc-umentation/]. For offline viewing, a PDF version [https://pegasus.isi.edu/documentation/pegasus-user-guide.pdf] is also provided.

Chapter 2. Tutorial

Introduction

This tutorial will take you through the steps of running simple workflows using Pegasus Workflow Management System. Pegasus allows scientists to

1. **Automate** their scientific computational work, as portable workflows. Pegasus enables scientists to construct workflows in abstract terms without worrying about the details of the underlying execution environment or the particulars of the low-level specifications required by the middleware (Condor, Globus, or Amazon EC2). It automatically locates the necessary input data and computational resources necessary for workflow execution. It cleans up storage as the workflow is executed so that data-intensive workflows have enough space to execute on storage-constrained resources.

2. **Recover** from failures at runtime. When errors occur, Pegasus tries to recover when possible by retrying tasks, and when all else fails, provides a rescue workflow containing a description of only the work that remains to be done. It also enables users to move computations from one resource to another. Pegasus keeps track of what has been done (provenance) including the locations of data used and produced, and which software was used with which parameters.

3. **Debug** failures in their computations using a set of system provided debugging tools and an online workflow monitoring dashboard.

This tutorial is intended for new users who want to get a quick overview of Pegasus concepts and usage. The accompanying tutorial VM comes pre-configured to run the example workflows. The instructions listed here refer mainly to the simple split workflow example. The tutorial covers

- submission of an already generated example workflow with Pegasus.

- how to use the Pegasus Workflow Dashboard for monitoring workflows.

- the command line tools for monitoring, debugging and generating statistics.

- recovery from failures

- creation of workflow using system provided API

- information catalogs configuration.

More information about the topics covered in this tutorial can be found in later chapters of this user's guide.

All of the steps in this tutorial are performed on the command-line. The convention we will use for command-line input and output is to put things that you should type in bold, monospace font, and to put the output you should get in a normal weight, monospace font, like this:

```
[user@host dir]$ you type this
you get this
```

Where `[user@host dir]$` is the terminal prompt, the text you should type is "`you type this`", and the output you should get is "`you get this`". The terminal prompt will be abbreviated as `$`. Because some of the outputs are long, we don't always include everything. Where the output is truncated we will add an ellipsis '...' to indicate the omitted output.

If you are having trouble with this tutorial, or anything else related to Pegasus, you can contact the Pegasus Users mailing list at `<pegasus-users@isi.edu>` to get help. You can also contact us on our support chatroom [https://pegasus.isi.edu/support] on HipChat.

Getting Started

Easiest way to start the tutorial is to connect to a hosted service using SSH as shown below.

```
$ ssh tutorial@pegasus-tutorial.isi.edu
tutorial@pegasus-tutorial.isi.edu's password: pegasus123
```

Note

The workflow dashboard is not run in the hosted service. To try out the workflow dashboard use the virtual machines provided below.

OR

We have provided several virtual machines that contain all of the software required for this tutorial. Virtual machine images are provided for VirtualBox and Amazon EC2. Information about deploying the tutorial VM on these platforms is in the appendix. If you want to use the tutorial VM, please go to the appendix for the platform you are using and follow the instructions for starting the VM found there before continuing with this tutorial.

If you have already installed Pegasus and Condor on your own machine, then you don't need to use the VM for the tutorial. You can use the `pegasus-init` command to generate the example workflow in any directory on your machine. Just be aware that you will have to modify the paths referenced in this tutorial to match the directory where you generated the example workflow.

The remainder of this tutorial will assume that you have a terminal open with Pegasus on your PATH.

What are Scientific Workflows

Scientific workflows allow users to easily express multi-step computational tasks, for example retrieve data from an instrument or a database, reformat the data, and run an analysis. A scientific workflow describes the dependencies between the tasks and in most cases the workflow is described as a directed acyclic graph (DAG), where the nodes are tasks and the edges denote the task dependencies. A defining property for a scientific workflow is that it manages data flow. The tasks in a scientific workflow can be everything from short serial tasks to very large parallel tasks (MPI for example) surrounded by a large number of small, serial tasks used for pre- and post-processing.

Workflows can vary from simple to complex. Below are some examples. In the figures below, the task are designated by circles/ellipses while the files created by the tasks are indicated by rectangles. Arrows indicate task dependencies.

Process Workflow

It consists of a single task that runs the `ls` command and generates a listing of the files in the `/` directory.

Figure 2.1. Process Workflow

Pipeline of Tasks

The pipeline workflow consists of two tasks linked together in a pipeline. The first job runs the `curl` command to fetch the Pegasus home page and store it as an HTML file. The result is passed to the `wc` command, which counts the number of lines in the HTML file.

Figure 2.2. Pipeline of Tasks

Split Workflow

The split workflow downloads the Pegasus home page using the `curl` command, then uses the `split` command to divide it into 4 pieces. The result is passed to the `wc` command to count the number of lines in each piece.

Figure 2.3. Split Workflow
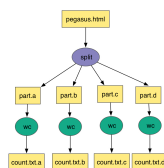
Merge Workflow

The merge workflow runs the `ls` command on several */bin directories and passes the results to the `cat` command, which merges the files into a single listing. The merge workflow is an example of a parameter sweep over arguments.

Figure 2.4. Merge Workflow
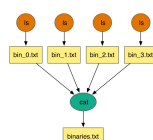
Diamond Workflow

The diamond workflow runs combines the split and merge workflow patterns to create a more complex workflow.

Figure 2.5. Diamond Workflow
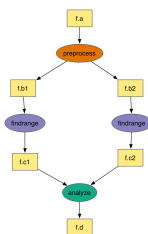
Complex Workflows

The above examples can be used as building blocks for much complex workflows. Some of these are showcased on the Pegasus Applications page [https://pegasus.isi.edu/applications].

Submitting an Example Workflow

All of the example workflows described in the previous section can be generated with the `pegasus-init` command. For this tutorial we will be using the split workflow, which can be created like this:

```
$ cd /home/tutorial
$ pegasus-init split
Do you want to generate a tutorial workflow? (y/n) [n]: y
1: Process
2: Pipeline
3: Split
4: Merge
5: Diamond
What tutorial workflow do you want? (1-5) [1]: 3
$ cd split
$ ls
README.md          input           plan_dax.sh        tc.txt
daxgen.py          output          rc.txt
generate_dax.sh    pegasus.properties sites.xml
```

Tip

The `pegasus-init` tool can be used to generate workflow skeletons from templates by asking the user questions. It is easier to use pegasus-init than to start a new workflow from scratch.

The split workflow looks like this:

Figure 2.6. Split Workflow
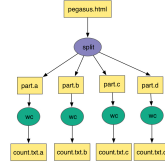
The input workflow description for Pegasus is called the DAX. It can be generated by running the `generate_dax.sh` script from the split directory, like this:

```
$ ./generate_dax.sh split.dax
Generated dax split.dax
```

This script will run a small Python program (`daxgen.py`) that generates a file with a .dax extension using the Pegasus Python API. We will cover the details of creating a DAX programmatically later in the tutorial. Pegasus reads the DAX and generates an executable HTCondor workflow that is run on an execution site.

The `pegasus-plan` command is used to submit the workflow through Pegasus. The pegasus-plan command reads the input workflow (DAX file specified by --dax option), maps the abstract DAX to one or more execution sites, and submits the generated executable workflow to HTCondor. Among other things, the options to pegasus-plan tell Pegasus

• the workflow to run

• where (what site) to run the workflow

• the input directory where the inputs are placed

• the output directory where the outputs are placed

By default, the workflow is setup to run on the compute sites (i.e sites with handle other than "local") defined in the sites.xml file. In our example, the workflow will run on a site named "condorpool" in the sites.xml file.

Note

If there are multiple compute sites specified in your sites.xml, and you want to choose a specific site, use the --sites option to pegasus-plan

To plan the split workflow invoke the pegasus-plan command using the `plan_dax.sh` wrapper script as follows:

```
$ ./plan_dax.sh split.dax
2015.10.22 19:12:10.402 PDT:
2015.10.22 19:12:10.409 PDT:
 -----------------------------------------------------------------
2015.10.22 19:12:10.414 PDT:   File for submitting this DAG to Condor        :
 split-0.dag.condor.sub
2015.10.22 19:12:10.420 PDT:   Log of DAGMan debugging messages              :
 split-0.dag.dagman.out
2015.10.22 19:12:10.426 PDT:   Log of Condor library output                 :
 split-0.dag.lib.out
2015.10.22 19:12:10.431 PDT:   Log of Condor library error messages         :
 split-0.dag.lib.err
2015.10.22 19:12:10.436 PDT:   Log of the life of condor_dagman itself      :
 split-0.dag.dagman.log
2015.10.22 19:12:10.442 PDT:
```

```
2015.10.22 19:12:10.458 PDT:
 -----------------------------------------------------------------------
2015.10.22 19:12:14.292 PDT:   Your database is compatible with Pegasus version: 4.5.3
2015.10.22 19:12:15.198 PDT:   Submitting to condor split-0.dag.condor.sub
2015.10.22 19:12:15.997 PDT:   Submitting job(s).
2015.10.22 19:12:16.003 PDT:   1 job(s) submitted to cluster 111.
2015.10.22 19:12:16.035 PDT:
2015.10.22 19:12:16.055 PDT:   Your workflow has been started and is running in the base directory:
2015.10.22 19:12:16.061 PDT:
2015.10.22 19:12:16.071 PDT:     /home/tutorial/split/submit/tutorial/pegasus/split/run0001
2015.10.22 19:12:16.078 PDT:
2015.10.22 19:12:16.084 PDT:   *** To monitor the workflow you can run ***
2015.10.22 19:12:16.090 PDT:
2015.10.22 19:12:16.098 PDT:     pegasus-status -l /home/tutorial/split/submit/tutorial/pegasus/
split/run0001
2015.10.22 19:12:16.114 PDT:
2015.10.22 19:12:16.119 PDT:   *** To remove your workflow run ***
2015.10.22 19:12:16.125 PDT:
2015.10.22 19:12:16.131 PDT:     pegasus-remove /home/tutorial/split/submit/tutorial/pegasus/split/
run0001
2015.10.22 19:12:16.137 PDT:
2015.10.22 19:12:17.630 PDT:   Time taken to execute is 1.918 seconds
```

Note

The line in the output that starts with `pegasus-status`, contains the command you can use to monitor the status of the workflow. The path it contains is the path to the submit directory where all of the files required to submit and monitor the workflow are stored.

This is what the split workflow looks like after Pegasus has finished planning the DAX:

Figure 2.7. Split DAG

For this workflow the only jobs Pegasus needs to add are a directory creation job, a stage-in job (for pegasus.html), and stage-out jobs (for wc count outputs). The cleanup jobs remove data that is no longer required as workflow executes.

Workflow Dashboard for Monitoring and Debugging

The Pegasus Dashboard is a web interface for monitoring and debugging workflows. We will use the web dashboard to monitor the status of the split workflow.

If you are doing the tutorial using the tutorial VM, then the dashboard will start when the VM boots. If you are using your own machine, then you will need to start the dashboard by running:

```
$ pegasus-service
```

By default, the dashboard server can only monitor workflows run by the current user i.e. the user who is running the pegasus-service.

Access the dashboard by navigating your browser to **https://localhost:5000**. If you are using the EC2 VM you will need to replace 'localhost' with the IP address of your EC2 instance.

When the webpage loads up, it will ask you for a username and a password. If you are using the tutorial VM, then log in as user "**tutorial**" with password "**pegasus**". If you are running the dashboard on your own machine, then use your UNIX username and password to log in.

The Dashboard's home page lists all workflows, which have been run by the current-user. The home page shows the status of each workflow i.e. Running/Successful/Failed/Failing. The home page lists only the top level workflows (Pegasus supports hierarchical workflows i.e. workflows within a workflow). The rows in the table are color coded

- **Green**: indicates workflow finished successfully.

- **Red**: indicates workflow finished with a failure.

- **Blue**: indicates a workflow is currently running.

- **Gray**: indicates a workflow that was archived.

Figure 2.8. Dashboard Home Page
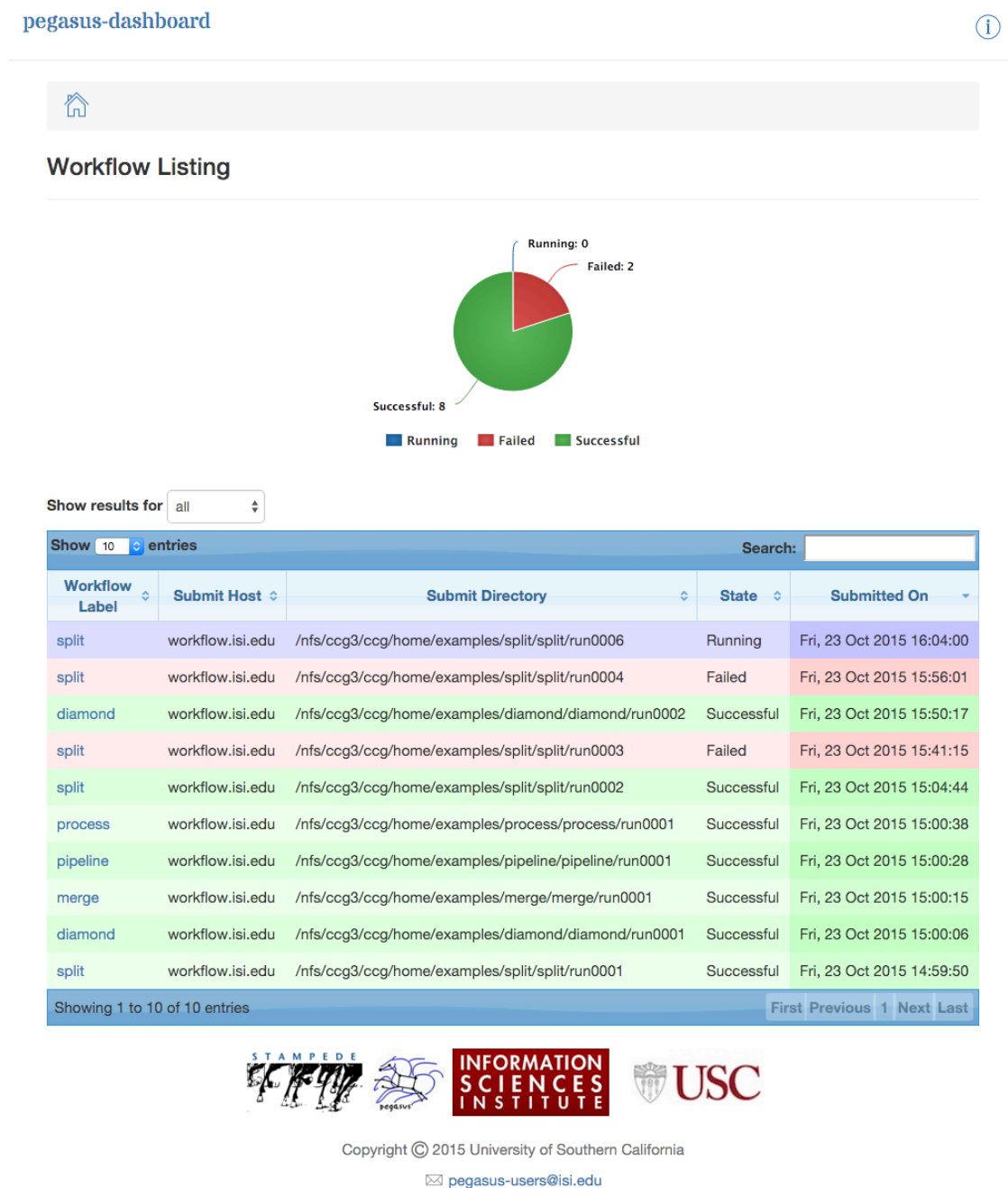
To view details specific to a workflow, the user can click on corresponding workflow label. The workflow details page lists workflow specific information like workflow label, workflow status, location of the submit directory, etc. The details page also displays pie charts showing the distribution of jobs based on status.

In addition, the details page displays a tab listing all sub-workflows and their statuses. Additional tabs exist which list information for all running, failed, successful, and failing jobs.

The information displayed for a job depends on it's status. For example, the failed jobs tab displays the job name, exit code, links to available standard output, and standard error contents.

Workflow Details 1145e2d5-ad2f-45d6-a3ce-4bd68499d8af

Summary | Files **6** | Metadata **2**

Label	diamond
Type	root-wf
Progress	Successful
Submit Host	cartman
User	bamboo
Submit Directory	/lfs1/software/bamboo/data/xml-data/build-dir/PEGASUS-WT-T39A/test/core/039-bl...
DAGMan Out File	diamond-0.dag.dagman.out
Wall Time	5 mins 9 secs
Cumulative Wall Time	5 mins 52 secs

Job Status (Entire Workflow)

Unsubmitted: 0
Failed: 0
Successful: 26

■ Unsubmitted ■ Failed ■ Successful

Job Status (Per Workflow)

Jobs: 0
Workflows: 0
Total: 0

Jobs: 0
Workflows: 0
Total: 0

Jobs: 26
Workflows: 0
Total: 26

■ Running ■ Failed ■ Successful

Charts | **Statistics**

Sub Workflows | **Failed** | **Running** | **Successful** | **Failing**

Show 10 entries   Search: [   ]

Job Name	Time Taken
analyze_ID0000004	1 min
clean_up_local_level_3_0	5 secs
clean_up_local_level_4_0	5 secs
clean_up_local_level_4_1	3 secs
clean_up_local_level_5_0	7 secs
clean_up_local_level_6_0	3 secs
cleanup_diamond_0_local	3 secs
create_dir_diamond_0_local	2 secs
findrange_ID0000002	1 min 1 sec
findrange_ID0000003	1 min

Showing 1 to 10 of 26 entries   First Previous 1 **2** 3 Next Last

To view details specific to a job the user can click on the corresponding job's job label. The job details page lists information relevant to a specific job. For example, the page lists information like job name, exit code, run time, etc.

The job instance section of the job details page lists all attempts made to run the job i.e. if a job failed in its first attempt due to transient errors, but ran successfully when retried, the job instance section shows two entries; one for each attempt to run the job.

The job details page also shows tab's for failed, and successful task invocations (Pegasus allows users to group multiple smaller task's into a single job i.e. a job may consist of one or more tasks)

🏠 / Workflow / Job

Job Details

Label	**ls_ID0000001**
Type	Compute
Exit Code	0
Working Directory	📄 /private/var/condor/execute/dir_12968
Application Stdour/Stderr	Preview
Kickstart Output	🔗 00/00/ls_ID0000001.out.000
Condor Stderr/Pegasus Lite Log	🔗 00/00/ls_ID0000001.err.000
Condor Submit File	🔗 ls_ID0000001.sub
Site	condorpool
Host	128.9.72.154 > isis.isi.edu

Job States

Submit	Thu Mar 23, 2017 01:25:53 PM (0 secs)
Execute	Thu Mar 23, 2017 01:26:08 PM (15 secs)
Image Size	Thu Mar 23, 2017 01:26:08 PM (0 secs)
Job Terminated	Thu Mar 23, 2017 01:26:08 PM (0 secs)
Job Success	Thu Mar 23, 2017 01:26:08 PM (0 secs)
Post Script Started	Thu Mar 23, 2017 01:26:08 PM (0 secs)
Post Script Terminated	Thu Mar 23, 2017 01:26:13 PM (5 secs)
Post Script Success	Thu Mar 23, 2017 01:26:13 PM (0 secs)

Job Instances

Show 10 ⏷ entries

Try ▾	Job Instance ID ⇅	Exitcode ⇅	Stdout ⇅	Stderr ⇅
1	2	0	Preview	Preview

Showing 1 to 1 of 1 entries

First Previous 1 Next Last

Job Invocations

Failed	Successful

No failed invocations.

The task invocation details page provides task specific information like task name, exit code, duration etc. Task details differ from job details, as they are more granular in nature.

Figure 2.11. Dashboard Invocation Page

pegasus-dashboard ⓘ

🏠 / Workflow / Job / Task Details

Task Details

Task Label	ID0000004
Transformation	diamond::analyze:4.0
Working Directory	📄 /var/lib/condor/execute/dir_784086
Executable	📄 /var/lib/condor/execute/dir_784086/diamond-analyze-4.0
Arguments	📄 -a analyze -T60 -i f.c1 f.c2 -o f.d
Exit Code	0
Start Time	Tue, 26 Jan 2016 09:54:16
Remote Duration	1 min
Remote CPU Time	59 secs

Task Metadata

size	2048
time	60
transformation	analyze

The dashboard also has web pages for workflow statistics and workflow charts, which graphically renders information provided by the pegasus-statistics and pegasus-plots command respectively.

The Statistics page shows the following statistics.

1. Workflow level statistics

2. Job breakdown statistics

3. Job specific statistics

Figure 2.12. Dashboard Statistics Page
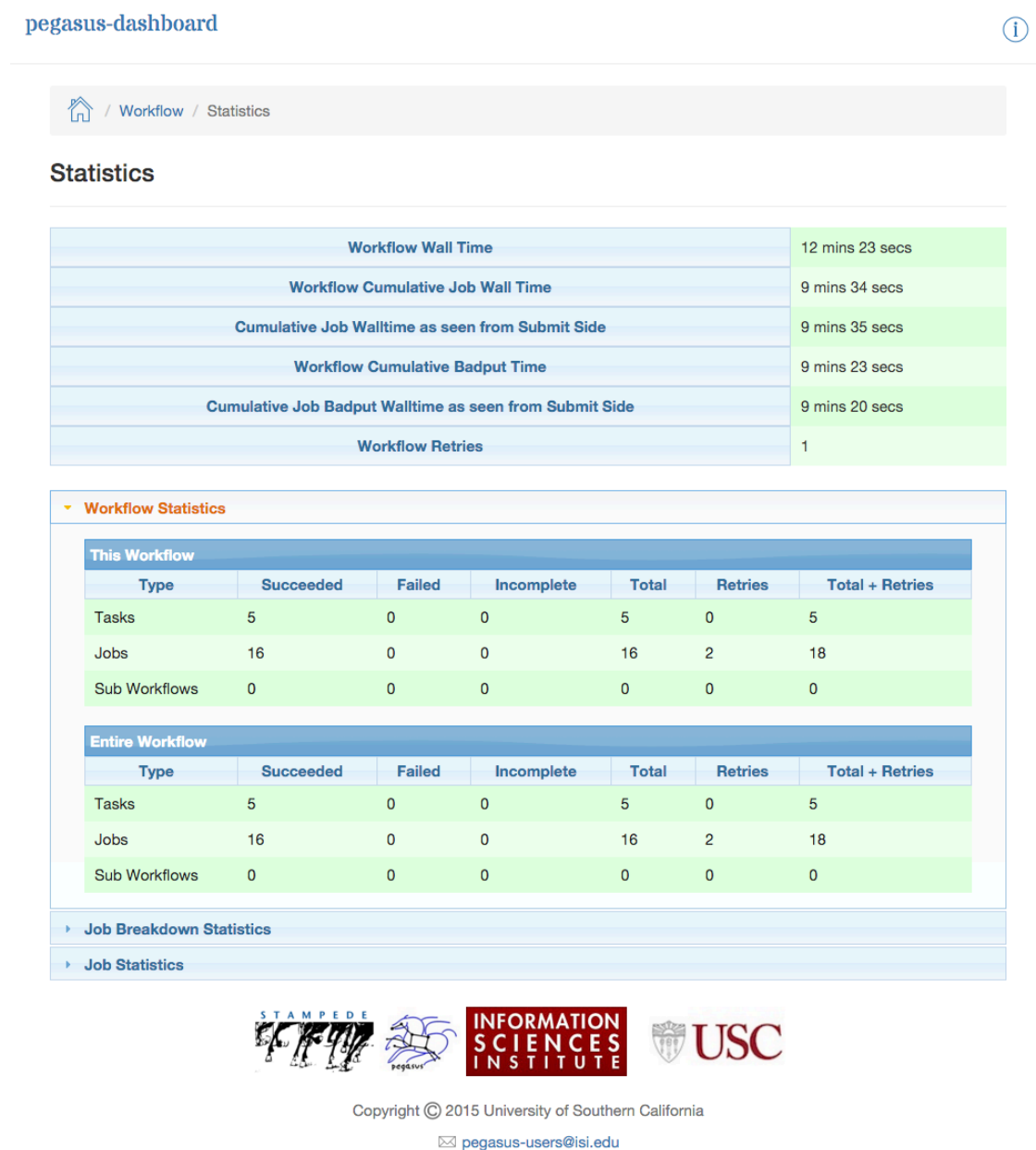
Command line tools for Monitoring and Debugging

Pegasus also comes with a series of command line tools that users can use to monitor and debug their workflows.

- pegasus-status : monitor the status of the workflow

- pegasus-analyzer : debug a failed workflow

- pegasus-statistics : generate statistics from a workflow run.

pegasus-status - monitoring the workflow

After the workflow has been submitted you can monitor it using the `pegasus-status` command:

```
$ pegasus-status -l submit/tutorial/pegasus/split/run0001
STAT   IN_STATE   JOB
Run      00:39   split-0 ( /home/tutorial/split/submit/tutorial/pegasus/split/run0001 )
Idle     00:03    ##split_ID0000001
Summary: 2 Condor jobs total (I:1 R:1)

UNRDY READY   PRE  IN_Q  POST  DONE  FAIL %DONE STATE   DAGNAME
   14     0     0     1     0     2     0  11.8 Running *split-0.dag
```

This command shows the workflow (split-0) and the running jobs (in the above output it shows the two findrange jobs). It also gives statistics on the number of jobs in each state and the percentage of the jobs in the workflow that have finished successfully.

Use the `watch` option to continuously monitor the workflow:

```
$ pegasus-status -w submit/tutorial/pegasus/split/run0001
...
```

You should see all of the jobs in the workflow run one after the other. After a few minutes you will see:

```
(no matching jobs found in Condor Q)
UNRDY READY   PRE  IN_Q  POST  DONE  FAIL %DONE STATE   DAGNAME
    0     0     0     0     0    15     0 100.0 Success *split-0.dag
```

That means the workflow is finished successfully.

If the workflow finished successfully you should see the output count files in the `output` directory.

```
$ ls output/
count.txt.a  count.txt.b  count.txt.c  count.txt.d
```

pegasus-analyzer - debug a failed workflow

In the case that one or more jobs fails, then the output of the `pegasus-status` command above will have a non-zero value in the `FAILURE` column.

You can debug the failure using the `pegasus-analyzer` command. This command will identify the jobs that failed and show their output. Because the workflow succeeded, `pegasus-analyzer` will only show some basic statistics about the number of successful jobs:

```
$ pegasus-analyzer submit/tutorial/pegasus/split/run0001
pegasus-analyzer: initializing...

****************************Summary***************************

 Total jobs          :      11 (100.00%)
 # jobs succeeded     :      11 (100.00%)
 # jobs failed        :       0 (0.00%)
 # jobs unsubmitted   :       0 (0.00%)
```

If the workflow had failed you would see something like this:

```
$ pegasus-analyzer submit/tutorial/pegasus/split/run0002

**********************************Summary*************************************

 Submit Directory   : submit/tutorial/pegasus/split/run0002
```

```
 Total jobs         :    15 (100.00%)
 # jobs succeeded   :     1 (5.88%)
 # jobs failed      :     1 (5.88%)
 # jobs unsubmitted :    15 (88.24%)


******************************Failed jobs' details******************************

=========================stage_in_local_PegasusVM_0_0=========================

 last state: POST_SCRIPT_FAILED
       site: local
submit file: stage_in_local_PegasusVM_0_0.sub
output file: stage_in_local_PegasusVM_0_0.out.001
 error file: stage_in_local_PegasusVM_0_0.err.001

-----------------------------Task #1 - Summary--------------------------------

site        : local
hostname    : unknown
executable  : /usr/bin/pegasus-transfer
arguments   :   --threads   2
exitcode    : 1
working dir : /home/tutorial/split/submit/tutorial/pegasus/split/run0002

------------------Task #1 - pegasus::transfer - None - stdout------------------

2015-10-22 21:13:50,970    INFO:   Reading URL pairs from stdin
2015-10-22 21:13:50,970    INFO:   PATH=/usr/bin:/bin
2015-10-22 21:13:50,970    INFO:   LD_LIBRARY_PATH=
2015-10-22 21:13:50,972    INFO:   1 transfers loaded
2015-10-22 21:13:50,972    INFO:   Sorting the tranfers based on transfer type and source/destination
2015-10-22 21:13:50,972    INFO:
 ------------------------------------------------------------------------------
2015-10-22 21:13:50,972    INFO:   Starting transfers - attempt 1
2015-10-22 21:13:50,972    INFO:   Using 1 threads for this round of transfers
2015-10-22 21:13:53,845    ERROR:  Command exited with non-zero exit code (1): /usr/bin/scp -r -B -
o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -i /home/tutorial/.ssh/id_rsa -P 22 '/
home/tutorial/examples/split/input/pegasus.html' 'tutorial@127.0.0.1:/home/tutorial/work/tutorial/
pegasus/split/run0002/pegasus.html'
2015-10-22 21:15:55,911    INFO:
 ------------------------------------------------------------------------------
2015-10-22 21:15:55,912    INFO:   Starting transfers - attempt 2
2015-10-22 21:15:55,912    INFO:   Using 1 threads for this round of transfers
2015-10-22 21:15:58,446    ERROR:  Command exited with non-zero exit code (1): /usr/bin/scp -r -B -
o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -i /home/tutorial/.ssh/id_rsa -P 22 '/
home/tutorial/examples/split/input/pegasus.html' 'tutorial@127.0.0.1:/home/tutorial/work/tutorial/
pegasus/split/run0002/pegasus.html'
2015-10-22 21:16:40,468    INFO:
 ------------------------------------------------------------------------------
2015-10-22 21:16:40,469    INFO:   Starting transfers - attempt 3
2015-10-22 21:16:40,469    INFO:   Using 1 threads for this round of transfers
2015-10-22 21:16:43,168    ERROR:  Command exited with non-zero exit code (1): /usr/bin/scp -r -B -
o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -i /home/tutorial/.ssh/id_rsa -P 22 '/
home/tutorial/examples/split/input/pegasus.html' 'tutorial@127.0.0.1:/home/tutorial/work/tutorial/
pegasus/split/run0002/pegasus.html'
2015-10-22 21:16:43,173    INFO:
 ------------------------------------------------------------------------------
2015-10-22 21:16:43,173    INFO:   Stats: no local files in the transfer set
2015-10-22 21:16:43,173 CRITICAL:  Some transfers failed! See above, and possibly stderr.


-------------Task #1 - pegasus::transfer - None - Kickstart stderr--------------

Warning: Permanently added '127.0.0.1' (RSA) to the list of known hosts.
/home/tutorial/split/input/pegasus.html: No such file or directory
..
/home/tutorial/split/input/pegasus.html: No such file or directory
```

In this example, we removed one of the input files. We will cover this in more detail in the recovery section. The output of `pegasus-analyzer` indicates that pegasus.html file could not be found.

pegasus-statistics - collect statistics about a workflow run

The `pegasus-statistics` command can be used to gather statistics about the runtime of the workflow and its jobs. The `-s all` argument tells the program to generate all statistics it knows how to calculate:

```
$ pegasus-statistics -s all submit/tutorial/pegasus/split/run0001

#
# Pegasus Workflow Management System - http://pegasus.isi.edu
#
# Workflow summary:
#   Summary of the workflow execution. It shows total
#   tasks/jobs/sub workflows run, how many succeeded/failed etc.
#   In case of hierarchical workflow the calculation shows the
#   statistics across all the sub workflows.It shows the following
#   statistics about tasks, jobs and sub workflows.
#     * Succeeded - total count of succeeded tasks/jobs/sub workflows.
#     * Failed - total count of failed tasks/jobs/sub workflows.
#     * Incomplete - total count of tasks/jobs/sub workflows that are
#       not in succeeded or failed state. This includes all the jobs
#       that are not submitted, submitted but not completed etc. This
#       is calculated as  difference between 'total' count and sum of
#       'succeeded' and 'failed' count.
#     * Total - total count of tasks/jobs/sub workflows.
#     * Retries - total retry count of tasks/jobs/sub workflows.
#     * Total+Retries - total count of tasks/jobs/sub workflows executed
#       during workflow run. This is the cumulative of retries,
#       succeeded and failed count.
# Workflow wall time:
#   The wall time from the start of the workflow execution to the end as
#   reported by the DAGMAN.In case of rescue dag the value is the
#   cumulative of all retries.
# Workflow cumulative job wall time:
#   The sum of the wall time of all jobs as reported by kickstart.
#   In case of job retries the value is the cumulative of all retries.
#   For workflows having sub workflow jobs (i.e SUBDAG and SUBDAX jobs),
#   the wall time value includes jobs from the sub workflows as well.
# Cumulative job wall time as seen from submit side:
#   The sum of the wall time of all jobs as reported by DAGMan.
#   This is similar to the regular cumulative job wall time, but includes
#   job management overhead and delays. In case of job retries the value
#   is the cumulative of all retries. For workflows having sub workflow
#   jobs (i.e SUBDAG and SUBDAX jobs), the wall time value includes jobs
#   from the sub workflows as well.
# Cumulative job wall time as seen from submit side:
#   The sum of the wall time of all jobs as reported by DAGMan.
#   This is similar to the regular cumulative job wall time, but includes
#   job management overhead and delays. In case of job retries the value
#   is the cumulative of all retries. For workflows having sub workflow
#   jobs (i.e SUBDAG and SUBDAX jobs), the wall time value includes jobs
#   from the sub workflows as well.
# Workflow cumulative job badput wall time:
#   The sum of the wall time of all failed jobs as reported by kickstart.
#   In case of job retries the value is the cumulative of all retries.
#   For workflows having sub workflow jobs (i.e SUBDAG and SUBDAX jobs),
#   the wall time value includes jobs from the sub workflows as well.
# Cumulative job badput wall time as seen from submit side:
#   The sum of the wall time of all failed jobs as reported by DAGMan.
#   This is similar to the regular cumulative job badput wall time, but includes
#   job management overhead and delays. In case of job retries the value
#   is the cumulative of all retries. For workflows having sub workflow
#   jobs (i.e SUBDAG and SUBDAX jobs), the wall time value includes jobs
#   from the sub workflows as well.


-------------------------------------------------------------------------
Type          Succeeded Failed  Incomplete  Total    Retries  Total+Retries
Tasks         5         0       0           5        0        5
Jobs          15        0       0           15       0        15
Sub-Workflows 0         0       0           0        0        0
-------------------------------------------------------------------------

Workflow wall time                                    : 2 mins, 6 secs
Workflow cumulative job wall time                     : 38 secs
Cumulative job wall time as seen from submit side     : 42 secs
Workflow cumulative job badput wall time              :
Cumulative job badput wall time as seen from submit side :

Summary                    : submit/tutorial/pegasus/split/run0001/statistics/summary.txt
Workflow execution statistics : submit/tutorial/pegasus/split/run0001/statistics/workflow.txt
Job instance statistics       : submit/tutorial/pegasus/split/run0001/statistics/jobs.txt
Transformation statistics     : submit/tutorial/pegasus/split/run0001/statistics/breakdown.txt
Time statistics               : submit/tutorial/pegasus/split/run0001/statistics/time.txt
```

The output of `pegasus-statistics` contains many definitions to help users understand what all of the values reported mean. Among these are the total wall time of the workflow, which is the time from when the workflow was submitted until it finished, and the total cumulative job wall time, which is the sum of the runtimes of all the jobs.

The `pegasus-statistics` command also writes out several reports in the `statistics` subdirectory of the workflow submit directory:

```
$ ls submit/tutorial/pegasus/split/run0001/statistics/
jobs.txt          summary.txt          time.txt          breakdown.txt          workflow.txt
```

The file `breakdown.txt`, for example, has min, max, and mean runtimes for each transformation:

```
$ more submit/tutorial/pegasus/split/run0001/statistics/breakdown.txt
# legends
# Transformation - name of the transformation.
# Count          - the number of times the invocations corresponding to
#                  the transformation was executed.
# Succeeded      - the count of the succeeded invocations corresponding
#                  to the transformation.
# Failed         - the count of the failed invocations corresponding to
#                  the transformation.
# Min(sec)       - the minimum invocation runtime value corresponding to
#                  the transformation.
# Max(sec)       - the maximum invocation runtime value corresponding to
#                  the transformation.
# Mean(sec)      - the mean of the invocation runtime corresponding to
#                  the transformation.
# Total(sec)     - the cumulative of invocation runtime corresponding to
#                  the transformation.


# 773d8fa3-8bff-4f75-8e2b-38e2c904f803 (split)
Transformation        Count    Succeeded Failed  Min     Max     Mean    Total
dagman::post          15       15        0       5.0     6.0     5.412   92.0
pegasus::cleanup      6        6         0       1.474   3.178   2.001   12.008
pegasus::dirmanager   1        1         0       2.405   2.405   2.405   2.405
pegasus::rc-client    2        2         0       2.382   7.406   4.894   9.788
pegasus::transfer     3        3         0       3.951   5.21    4.786   14.358
split                 1        1         0       0.009   0.009   0.009   0.009
wc                    4        4         0       0.005   0.029   0.012   0.047


# All (All)
Transformation        Count    Succeeded Failed  Min     Max     Mean    Total
dagman::post          15       15        0       5.0     6.0     5.412   92.0
pegasus::cleanup      6        6         0       1.474   3.178   2.001   12.008
pegasus::dirmanager   1        1         0       2.405   2.405   2.405   2.405
pegasus::rc-client    2        2         0       2.382   7.406   4.894   9.788
pegasus::transfer     3        3         0       3.951   5.21    4.786   14.358
split                 1        1         0       0.009   0.009   0.009   0.009
wc                    4        4         0       0.005   0.029   0.012   0.047
```

In this case, because the example transformation sleeps for 30 seconds, the min, mean, and max runtimes for each of the analyze, findrange, and preprocess transformations are all close to 30.

Recovery from Failures

Executing workflows in a distributed environment can lead to failures. Often, they are a result of the underlying infrastructure being temporarily unavailable, or errors in workflow setup such as incorrect executables specified, or input files being unavailable.

In case of transient infrastructure failures such as a node being temporarily down in a cluster, Pegasus will automatically retry jobs in case of failure. After a set number of retries (usually once), a hard failure occurs, because of which workflow will eventually fail.

In most of the cases, these errors are correctable (either the resource comes back online or application errors are fixed). Once the errors are fixed, you may not want to start a new workflow but instead start from the point of failure. In order to do this, you can submit the rescue workflows automatically created in case of failures. A rescue workflow contains only a description of only the work that remains to be done.

Submitting Rescue Workflows

In this example, we will take our previously run workflow and introduce errors such that workflow we just executed fails at runtime.

First we will "hide" the input file to cause a failure by renaming it:

```
$ mv input/pegasus.html input/pegasus.html.bak
```

Now submit the workflow again:

```
$ ./plan_dax.sh split.dax
2015.10.22 20:20:08.299 PDT:
2015.10.22 20:20:08.307 PDT:
 ----------------------------------------------------------------------
2015.10.22 20:20:08.312 PDT:   File for submitting this DAG to Condor         :
 split-0.dag.condor.sub
2015.10.22 20:20:08.323 PDT:   Log of DAGMan debugging messages               :
 split-0.dag.dagman.out
2015.10.22 20:20:08.330 PDT:   Log of Condor library output                   :
 split-0.dag.lib.out
2015.10.22 20:20:08.339 PDT:   Log of Condor library error messages           :
 split-0.dag.lib.err
2015.10.22 20:20:08.346 PDT:   Log of the life of condor_dagman itself        :
 split-0.dag.dagman.log
2015.10.22 20:20:08.352 PDT:
2015.10.22 20:20:08.368 PDT:
 ----------------------------------------------------------------------
2015.10.22 20:20:12.331 PDT:   Your database is compatible with Pegasus version: 4.5.3
2015.10.22 20:20:13.326 PDT:   Submitting to condor split-0.dag.condor.sub
2015.10.22 20:20:14.224 PDT:   Submitting job(s).
2015.10.22 20:20:14.254 PDT:   1 job(s) submitted to cluster 168.
2015.10.22 20:20:14.288 PDT:
2015.10.22 20:20:14.297 PDT:   Your workflow has been started and is running in the base directory:
2015.10.22 20:20:14.303 PDT:
2015.10.22 20:20:14.309 PDT:      /home/tutorial/split/submit/tutorial/pegasus/split/run0002
2015.10.22 20:20:14.315 PDT:
2015.10.22 20:20:14.321 PDT:   *** To monitor the workflow you can run ***
2015.10.22 20:20:14.326 PDT:
2015.10.22 20:20:14.332 PDT:     pegasus-status -l /home/tutorial/split/submit/tutorial/pegasus/
split/run0002
2015.10.22 20:20:14.351 PDT:
2015.10.22 20:20:14.369 PDT:   *** To remove your workflow run ***
2015.10.22 20:20:14.376 PDT:
2015.10.22 20:20:14.388 PDT:     pegasus-remove /home/tutorial/split/submit/tutorial/pegasus/split/
run0002
2015.10.22 20:20:14.397 PDT:
2015.10.22 20:20:16.146 PDT:   Time taken to execute is 10.292 seconds
```

We will now monitor the workflow using the pegasus-status command till it fails. We will add -w option to pegasus-status to watch automatically till the workflow finishes:

```
$ pegasus-status -w submit/tutorial/pegasus/split/run0002
(no matching jobs found in Condor Q)
UNREADY   READY     PRE  QUEUED    POST SUCCESS FAILURE %DONE
      8        0        0       0       0       2       1  18.2
Summary: 1 DAG total (Failure:1)
```

Now we can use the pegasus-analyzer command to determine what went wrong:

```
$ pegasus-analyzer submit/tutorial/pegasus/split/run0002

**********************************Summary*************************************

 Submit Directory   : submit/tutorial/pegasus/split/run0002
 Total jobs         :     11 (100.00%)
 # jobs succeeded   :      2 (18.18%)
 # jobs failed      :      1 (9.09%)
 # jobs unsubmitted :      8 (72.73%)

*****************************Failed jobs' details*****************************

===========================stage_in_remote_local_0_0===========================
```

```
 last state: POST_SCRIPT_FAILED
        site: local
submit file: stage_in_remote_local_0_0.sub
output file: stage_in_remote_local_0_0.out.001
 error file: stage_in_remote_local_0_0.err.001

-----------------------------Task #1 - Summary-------------------------------

site       : local
hostname   : unknown
executable : /usr/local/bin/pegasus-transfer
arguments  :   --threads   2
exitcode   : 1
working dir : /home/tutorial/split/submit/tutorial/pegasus/split/run0002

-----------------Task #1 - pegasus::transfer - None - stdout------------------

2016-02-18 11:52:58,189   INFO:  Reading URL pairs from stdin
2016-02-18 11:52:58,189   INFO:  PATH=/usr/local/bin:/usr/bin:/bin
2016-02-18 11:52:58,189   INFO:  LD_LIBRARY_PATH=
2016-02-18 11:52:58,189   INFO:  1 transfers loaded
2016-02-18 11:52:58,189   INFO:  Sorting the tranfers based on transfer type and source/destination
2016-02-18 11:52:58,190   INFO:
 ----------------------------------------------------------------------------
2016-02-18 11:52:58,190   INFO:  Starting transfers - attempt 1
2016-02-18 11:52:58,190   INFO:  Using 1 threads for this round of transfers
2016-02-18 11:53:00,205   ERROR:  Command exited with non-zero exit code (1): /bin/cp -f -R -L
 '/home/tutorial/split/input/pegasus.html' '/home/tutorial/split/scratch/tutorial/pegasus/split/
run0002/pegasus.html'
2016-02-18 11:54:46,205   INFO:
 ----------------------------------------------------------------------------
2016-02-18 11:54:46,205   INFO:  Starting transfers - attempt 2
2016-02-18 11:54:46,205   INFO:  Using 1 threads for this round of transfers
2016-02-18 11:54:48,220   ERROR:  Command exited with non-zero exit code (1): /bin/cp -f -R -L
 '/home/tutorial/split/input/pegasus.html' '/home/tutorial/split/scratch/tutorial/pegasus/split/
run0002/pegasus.html'
2016-02-18 11:55:24,224   INFO:
 ----------------------------------------------------------------------------
2016-02-18 11:55:24,224   INFO:  Starting transfers - attempt 3
2016-02-18 11:55:24,224   INFO:  Using 1 threads for this round of transfers
2016-02-18 11:55:26,240   ERROR:  Command exited with non-zero exit code (1): /bin/cp -f -R -L
 '/home/tutorial/split/input/pegasus.html' '/home/tutorial/split/scratch/tutorial/pegasus/split/
run0002/pegasus.html'
2016-02-18 11:55:26,240   INFO:
 ----------------------------------------------------------------------------
2016-02-18 11:55:26,240   INFO:  Stats: no local files in the transfer set
2016-02-18 11:55:26,240 CRITICAL:  Some transfers failed! See above, and possibly stderr.


-------------Task #1 - pegasus::transfer - None - Kickstart stderr--------------

cp: /home/tutorial/split/input/pegasus.html: No such file or directory
cp: /home/tutorial/split/input/pegasus.html: No such file or directory
cp: /home/tutorial/split/input/pegasus.html: No such file or directory
```

The above listing indicates that it could not transfer pegasus.html. Let's correct that error by restoring the pegasus.html file:

```
$ mv input/pegasus.html.bak input/pegasus.html
```

Now in order to start the workflow from where we left off, instead of executing pegasus-plan we will use the command pegasus-run on the directory from our previous failed workflow run:

```
$ pegasus-run submit/tutorial/pegasus/split/run0002/
Rescued /home/tutorial/split/submit/tutorial/pegasus/split/run0002/split-0.log as /home/tutorial/
split/submit/tutorial/pegasus/split/run0002/split-0.log.000
Submitting to condor split-0.dag.condor.sub
Submitting job(s).
1 job(s) submitted to cluster 181.

Your workflow has been started and is running in the base directory:

  submit/tutorial/pegasus/split/run0002/
```

```
*** To monitor the workflow you can run ***

  pegasus-status -l submit/tutorial/pegasus/split/run0002/

*** To remove your workflow run ***

  pegasus-remove submit/tutorial/pegasus/split/run0002/
```

The workflow will now run to completion and succeed.

```
$ pegasus-status -l submit/tutorial/pegasus/split/run0002/
(no matching jobs found in Condor Q)
UNRDY READY   PRE  IN_Q  POST  DONE  FAIL %DONE STATE   DAGNAME
    0     0     0     0     0    11     0 100.0 Success *split-0.dag
Summary: 1 DAG total (Success:1)
```

Generating the Workflow

The example that you ran earlier already had the workflow description (split.dax) generated. Pegasus reads workflow descriptions from DAX files. The term "DAX" is short for "Directed Acyclic Graph in XML". DAX is an XML file format that has syntax for expressing jobs, arguments, files, and dependencies. We now will be creating the split workflow that we just ran using the Pegasus provided DAX API:

Figure 2.13. Split Workflow
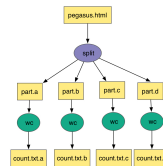
In this diagram, the ovals represent computational jobs, the dog-eared squares are files, and the arrows are dependencies.

In order to create a DAX it is necessary to write code for a DAX generator. Pegasus comes with Perl, Java, and Python libraries for writing DAX generators. In this tutorial we will show how to use the Python library.

The DAX generator for the split workflow is in the file daxgen.py. Look at the file by typing:

```
$ more daxgen.py
...
```

> ## Tip
>
> We will be using the more command to inspect several files in this tutorial. more is a pager application, meaning that it splits text files into pages and displays the pages one at a time. You can view the next page of a file by pressing the spacebar. Type 'h' to get help on using more. When you are done, you can type 'q' to close the file.

The code has 3 main sections:

1. A new ADAG object is created. This is the main object to which jobs and dependencies are added.

   ```
   # Create a abstract dag
   dax = ADAG("split")
   ...
   ```

2. Jobs and files are added. The 5 jobs in the diagram above are added and 9 files are referenced. Arguments are defined using strings and File objects. The input and output files are defined for each job. This is an important step, as it allows Pegasus to track the files, and stage the data if necessary. Workflow outputs are tagged with "transfer=true".

   ```
   # the split job that splits the webpage into smaller chunks
   webpage = File("pegasus.html")
   ```

```
split = Job("split")
split.addArguments("-l","100","-a","1",webpage,"part.")
split.uses(webpage, link=Link.INPUT)
dax.addJob(split)

...
```

3. Dependencies are added. These are shown as arrows in the diagram above. They define the parent/child relationships between the jobs. When the workflow is executing, the order in which the jobs will be run is determined by the dependencies between them.

```
# Add control-flow dependencies
dax.depends(wc, split)
```

Generate a DAX file named `split.dax` by typing:

```
$ ./generate_dax.sh split.dax
Generated dax split.dax
```

The `split.dax` file should contain an XML representation of the split workflow. You can inspect it by typing:

```
$ more split.dax
...
```

Information Catalogs

The workflow description (DAX) that you specify to Pegasus is portable, and usually does not contain any locations to physical input files, executables or cluster end points where jobs are executed. Pegasus uses three information catalogs during the planning process.

Figure 2.14. Information Catalogs used by Pegasus
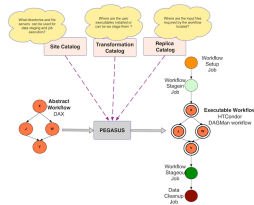
The Site Catalog

The site catalog describes the sites where the workflow jobs are to be executed. In this tutorial we assume that you have a Personal Condor pool running on localhost. If you are using one of the tutorial VMs this has already been setup for you. The site catalog for the tutorial examples is in `sites.xml`:

```
$ more sites.xml
...
  <!-- The local site contains information about the submit host -->
  <!-- The arch and os keywords are used to match binaries in the transformation catalog -->
  <site handle="local" arch="x86_64" os="LINUX">

      <!-- These are the paths on the submit host were Pegasus stores data -->
      <!-- Scratch is where temporary files go -->
      <directory type="shared-scratch" path="/home/tutorial/scratch">
          <file-server operation="all" url="file:///home/tutorial/scratch"/>
      </directory>

      <!-- Storage is where pegasus stores output files -->
      <directory type="local-storage" path="/home/tutorial/output">
          <file-server operation="all" url="file:///home/tutorial/output"/>
      </directory>
  </site>

...
```

Note

By default (unless specified in properties), Pegasus picks ups the site catalog from a XML file named sites.xml in the current working directory from where pegasus-plan is invoked.

There are two sites defined in the site catalog: "local" and "condorpool". The "local" site is used by Pegasus to learn about the submit host where the workflow management system runs. The "condorpool" site is the Condor pool configured on your submit machine. In the case of the tutorial VM, the local site and the condorpool site refer to the same machine, but they are logically separate as far as Pegasus is concerned.

1. The **local** site is configured with a "storage" file system that is mounted on the submit host (indicated by the file:// URL). This file system is where the output data from the workflow will be stored. When the workflow is planned we will tell Pegasus that the output site is "local".

2. The **condorpool** site is also configured with a "scratch" file system. This file system is where the working directory will be created. When we plan the workflow we will tell Pegasus that the execution site is "condorpool".

Pegasus supports many different file transfer protocols. In this case the Pegasus configuration is set up so that input and output files are transferred to/from the condorpool site by Condor. This is done by setting `pegasus.data.configuration = condorio` in the properties file. In a normal Condor pool, this will cause job input/output files to be transferred from/to the submit host to/from the worker node. In the case of the tutorial VM, this configuration is just a fancy way to copy files from the workflow scratch directory to the job scratch directory.

Finally, the condorpool site is configured with two profiles that tell Pegasus that it is a plain Condor pool. Pegasus supports many ways of submitting tasks to a remote cluster. In this configuration it will submit vanilla Condor jobs.

HPC Clusters

Typically the sites in the site catalog describe remote clusters, such as PBS clusters or Condor pools.

Usually, a typical deployment of an HPC cluster is illustrated below. The site catalog, captures for each cluster (site)

- directories that can be used for executing jobs

- whether a shared file system is available

- file servers to use for staging input data and staging out output data

- headnode of the cluster to which jobs can be submitted.

Figure 2.15. Sample HPC Cluster Setup
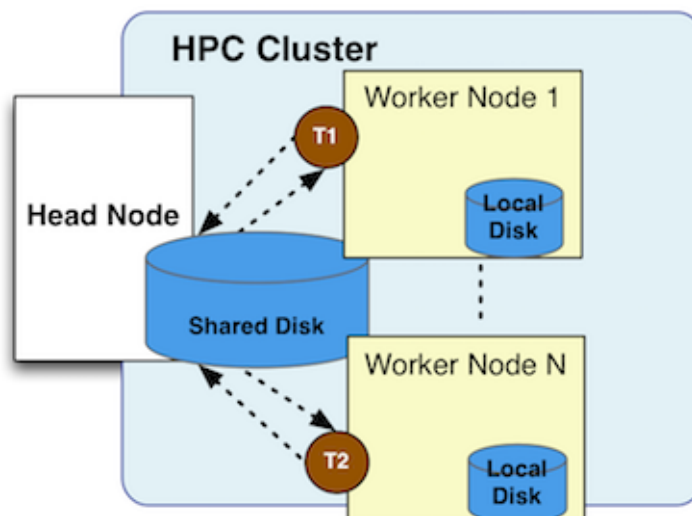
Below is a sample site catalog entry for HPC cluster at SDSC that is part of XSEDE

```
<site  handle="sdsc-gordon" arch="x86_64" os="LINUX">
        <grid  type="gt5" contact="gordon-ln4.sdsc.xsede.org:2119/jobmanager-fork" scheduler="Fork"
 jobtype="auxillary"/>
        <grid  type="gt5" contact="gordon-ln4.sdsc.xsede.org:2119/jobmanager-pbs"
 scheduler="unknown" jobtype="compute"/>

        <!-- the base directory where workflow jobs will execute for the site -->
        <directory type="shared-scratch" path="/oasis/scratch/ux454281/temp_project">
            <file-server operation="all" url="gsiftp://oasis-dm.sdsc.xsede.org:2811/oasis/scratch/
ux454281/temp_project"/>
        </directory>

        <profile namespace="globus" key="project">TG-STA110014S</profile>
        <profile namespace="env" key="PEGASUS_HOME">/home/ux454281/software/pegasus/pegasus-4.5.0</
profile>
    </site>
```

The Transformation Catalog

The transformation catalog describes all of the executables (called "transformations") used by the workflow. This description includes the site(s) where they are located, the architecture and operating system they are compiled for, and any other information required to properly transfer them to the execution site and run them.

For this tutorial, the transformation catalog is in the file `tc.txt`:

```
$ more tc.txt
tr wc {
    site condorpool {
        pfn "/usr/bin/wc"
        arch "x86_64"
        os "linux"
        type "INSTALLED"
    }
}
...
```

Note

By default (unless specified in properties), Pegasus picks up the transformation catalog from a text file named tc.txt in the current working directory from where pegasus-plan is invoked.

The `tc.txt` file contains information about two transformations: wc, and split. These two transformations are referenced in the split DAX. The transformation catalog indicates that both transformations are installed on the condorpool site, and are compiled for x86_64 Linux.

The Replica Catalog

Note: Replica Catalog configuration is not required for the tutorial setup. It is only required if you want to refer to input files on external servers.

The example that you ran, was configured with the inputs already present on the submit host (where Pegasus is installed) in a directory. If you have inputs at external servers, then you can specify the URLs to the input files in the Replica Catalog. This catalog tells Pegasus where to find each of the input files for the workflow.

All files in a Pegasus workflow are referred to in the DAX using their Logical File Name (LFN). These LFNs are mapped to Physical File Names (PFNs) when Pegasus plans the workflow. This level of indirection enables Pegasus to map abstract DAXes to different execution sites and plan out the required file transfers automatically.

The Replica Catalog for the diamond workflow is in the `rc.txt` file:

```
$ more rc.txt
# This is the replica catalog. It lists information about each of the
# input files used by the workflow. You can use this to specify locations to input files present on
 external servers.

# The format is:
```

```
# LFN      PFN    pool="SITE"
#
# For example:
#data.txt  file:///tmp/data.txt         site="local"
#data.txt  http://example.org/data.txt  site="example"
pegasus.html file:///home/tutorial/split/input/pegasus.html   site="local"
```

Note

By default (unless specified in properties), Pegasus picks ups the transformation catalog from a text file named tc.txt in the current working directory from where pegasus-plan is invoked. In our tutorial, input files are on the submit host and we used the --input dir option to pegasus-plan to specify where they are located.

This replica catalog contains only one entry for the split workflow's only input file. This entry has an LFN of "pegasus.html" with a PFN of "file:///home/tutorial/split/input/pegasus.html" and the file is stored on the local site, which implies that it will need to be transferred to the condorpool site when the workflow runs.

Configuring Pegasus

In addition to the information catalogs, Pegasus takes a configuration file that specifies settings that control how it plans the workflow.

For the diamond workflow, the Pegasus configuration file is relatively simple. It only contains settings to help Pegasus find the information catalogs. These settings are in the pegasus.properties file:

```
$ more pegasus.properties
# This tells Pegasus where to find the Site Catalog
pegasus.catalog.site.file=sites.xml

# This tells Pegasus where to find the Replica Catalog
pegasus.catalog.replica=File
pegasus.catalog.replica.file=rc.txt

# This tells Pegasus where to find the Transformation Catalog
pegasus.catalog.transformation=Text
pegasus.catalog.transformation.file=tc.txt

# Use condor to transfer workflow data
pegasus.data.configuration=condorio

# This is the name of the application for analytics
pegasus.metrics.app=pegasus-tutorial
```

Conclusion

Congratulations! You have completed the tutorial.

If you used Amazon EC2 for this tutorial make sure to terminate your VM. Refer to the appendix for more information about how to do this.

Refer to the other chapters in this guide for more information about creating, planning, and executing workflows with Pegasus.

Please contact the Pegasus Users Mailing list at <pegasus-users@isi.edu> if you need help.

Chapter 3. Installation

The preferred way to install Pegasus is with native (RPM/DEB) packages. It is recommended that you also install HT-Condor (formerly Condor) (yum [http://research.cs.wisc.edu/htcondor/yum/], debian [http://research.cs.wisc.edu/ht-condor/debian/]) from native packages.

Prerequisites

Pegasus has a few dependencies:

- **Java 1.6 or higher**. Check with:

```
# java -version
java version "1.6.0_07"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.6.0_07-164)
Java HotSpot(TM) Client VM (build 1.6.0_07-87, mixed mode, sharing)
```

- **Python 2.6 or higher**. Check with:

```
# python -v
Python 2.6.2
```

   Non-standard Python installation: Pegasus will use the system Python by default. If you want to override this behavior, please set the **PEGASUS_PYTHON** environment variable during the build. This environment variable is only for build time configuration. Once built, Pegasus will continue to use the build time specified Python install.

- **HTCondor (formerly Condor) 8.4 or higher**. See http://www.cs.wisc.edu/htcondor/ for more information. You should be able to run `condor_q` and `condor_status`.

Optional Software

- **Globus 5.0 or higher**. Globus is only needed if you want to run against grid sites or use GridFTP for data transfers. See http://www.globus.org/ for more information.

- **psycopg2**. Python module for PostgreSQL access. Only needed if you want to store the runtime database in PostgreSQL (default is SQLite)

Environment

To use Pegasus, you need to have the pegasus-* tools in your PATH. If you have installed Pegasus from RPM/DEB packages. the tools will be in the default PATH, in /usr/bin. If you have installed Pegasus from binary tarballs or source, add the bin/ directory to your PATH.

Example for bourne shells:

```
$ export PATH=/some/install/pegasus-4.8.0/bin:$PATH
```

Note

Pegasus 4.x is different from previous versions of Pegasus in that it does not require PEGASUS_HOME to be set or sourcing of any environment setup scripts.

If you want to use the DAX APIs, you might also need to set your PYTHONPATH, PERL5LIB, or CLASSPATH. The right setting can be found by using pegasus-config:

```
$ export PYTHONPATH=`pegasus-config --python`
$ export PERL5LIB=`pegasus-config --perl`
$ export CLASSPATH=`pegasus-config --classpath`
```

RHEL / CentOS / Scientific Linux

Binary packages provided for: RHEL 6 x86_64, RHEL 7 x86_64 (and OSes derived from RHEL: CentOS, SL)

Add the Pegasus repository to yum downloading the Pegasus repos file and adding it to **/etc/yum.repos.d/**. For RHEL 7 based systemes:

```
# wget -O /etc/yum.repos.d/pegasus.repo http://download.pegasus.isi.edu/pegasus/rhel/7/pegasus.repo
```

For RHEL 6 based systems:

```
# wget -O /etc/yum.repos.d/pegasus.repo http://download.pegasus.isi.edu/pegasus/rhel/6/pegasus.repo
```

Search for, and install Pegasus:

```
# yum search pegasus
pegasus.x86_64 : Workflow management system for Condor, grids, and clouds
# yum install pegasus
Running Transaction
Installing     : pegasus

Installed:
pegasus

Complete!
```

Ubuntu

Binary packages provided for: 14.04 LTS (Trusty Tahr) x86_64, 16.04 (Xenial Xerus) x86_64

To be able to install and upgrade from the Pegasus apt repository, you will have to trust the repository key. You only need to add the repository key once:

```
# wget -O - http://download.pegasus.isi.edu/pegasus/gpg.txt | apt-key add -
```

Create repository file, update and install Pegasus (currently available releases are precise and trusty):

```
# echo 'deb http://download.pegasus.isi.edu/pegasus/ubuntu trusty main' >/etc/apt/sources.list.d/
pegasus.list
# apt-get update
# apt-get install pegasus
```

Debian

Binary packages provided for: 7 (Wheezy) x86_64, 8 (Jessie) x86_64, 9 (Stretch) x86_64

To be able to install and upgrade from the Pegasus apt repository, you will have to trust the repository key. You only need to add the repository key once:

```
# wget -O - http://download.pegasus.isi.edu/pegasus/gpg.txt | apt-key add -
```

Create repository file, update and install Pegasus (currently available releases are jessie (8) and stretch (9)):

```
# echo 'deb http://download.pegasus.isi.edu/pegasus/debian stretch main' >/etc/apt/sources.list.d/
pegasus.list
# apt-get update
# apt-get install pegasus
```

Mac OS X

The easiest way to install Pegasus on Mac OS is to use Homebrew. You will need to install XCode and the XCode command-line tools, as well as Homebrew. Then you just need to tap the Pegasus tools repository and install Pegasus and HTCondor like this:

```
$ brew tap pegasus-isi/tools
$ brew install pegasus htcondor
```

Once the installation is complete, you need to start the HTCondor service. The easiest way to do that is to use the Homebrew services tap:

```
$ brew tap homebrew/services
$ brew services list
$ brew services start htcondor
```

You can also stop HTCondor like this:

```
$ brew services stop htcondor
```

And you can uninstall Pegasus and HTCondor using `brew rm` like this:

```
$ brew rm pegasus htcondor
```

> ### Note
>
> It is also possible to install the latest development versions of Pegasus using the `--devel` and `--HEAD` arguments to `brew install`, like this: `$ brew install --devel pegasus`

Pegasus from Tarballs

The Pegasus prebuild tarballs can be downloaded from the *Pegasus Download Page* [https://pegasus.isi.edu/downloads].

Use these tarballs if you already have HTCondor installed or prefer to keep the HTCondor installation separate from the Pegasus installation.

• Untar the tarball

```
# tar zxf pegasus-*.tar.gz
```

• include the Pegasus bin directory in your PATH

```
# export PATH=/path/to/pegasus-install/bin:$PATH
```

Chapter 4. Creating Workflows

Abstract Workflows (DAX)

The DAX is a description of an abstract workflow in XML format that is used as the primary input into Pegasus. The DAX schema is described in dax-3.4.xsd [schemas/dax-3.4/dax-3.4.xsd] The documentation of the schema and its elements can be found in dax-3.4.html [schemas/dax-3.4/dax-3.4.html].

A DAX can be created by all users with the DAX generating API in Java, Perl, or Python format

Note
We highly recommend using the DAX API.

Advanced users who can read XML schema definitions can generate a DAX directly from a script

The sample workflow below incorporates some of the elementary graph structures used in all abstract workflows.

- **fan-out**, **scatter**, and **diverge** all describe the fact that multiple siblings are dependent on fewer parents.

  The example shows how the **Job 2 and 3** nodes depend on **Job 1** node.

- **fan-in**, **gather**, **join**, and **converge** describe how multiple siblings are merged into fewer dependent child nodes.

  The example shows how the **Job 4** node depends on both **Job 2 and Job 3** nodes.

- **serial execution** implies that nodes are dependent on one another, like pearls on a string.

- **parallel execution** implies that nodes can be executed in parallel

Figure 4.1. Sample Workflow
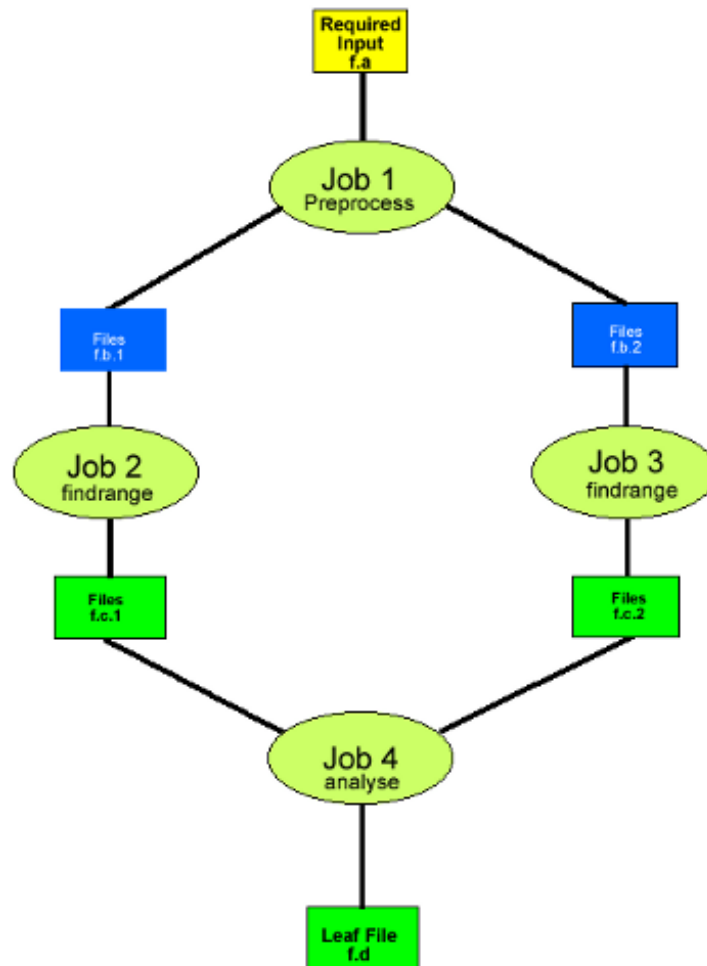
The example diamond workflow consists of four nodes representing jobs, and are linked by six files.

- Required input files must be registered with the Replica catalog in order for Pegasus to find it and integrate it into the workflow.

- Leaf files are a product or output of a workflow. Output files can be collected at a location.

- The remaining files all have lines leading to them and originating from them. These files are products of some job steps (lines leading to them), and consumed by other job steps (lines leading out of them). Often, these files represent intermediary results that can be cleaned.

There are two main ways of generating DAX's

1. Using a DAX generating API in Java, Perl or Python.

   Note: We recommend this option.

2. Generating XML directly from your script.

   Note: This option should only be considered by advanced users who can also read XML schema definitions.

One example for a DAX representing the example workflow can look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!-- generated on: 2016-01-21T10:36:39-08:00 -->
<!-- generated by: vahi [ ?? ] -->
<adag xmlns="http://pegasus.isi.edu/schema/DAX" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://pegasus.isi.edu/schema/DAX http://pegasus.isi.edu/schema/
dax-3.6.xsd" version="3.6" name="diamond" index="0" count="1">

<!-- Section 1: Metadata attributes for the workflow (can be empty)  -->

    <metadata key="name">diamond</metadata>
    <metadata key="createdBy">Karan Vahi</metadata>

<!-- Section 2: Invokes - Adds notifications for a workflow (can be empty) -->

    <invoke when="start">/pegasus/libexec/notification/email -t notify@example.com</invoke>
    <invoke when="at_end">/pegasus/libexec/notification/email -t notify@example.com</invoke>

<!-- Section 3: Files - Acts as a Replica Catalog (can be empty) -->

    <file name="f.a">
       <metadata key="size">1024</metadata>
       <pfn url="file:///Volumes/Work/lfs1/work/pegasus-features/PM-902/f.a" site="local"/>
    </file>

<!-- Section 4: Executables - Acts as a Transformaton Catalog (can be empty) -->

    <executable namespace="pegasus" name="preprocess" version="4.0" installed="true" arch="x86"
 os="linux">
       <metadata key="size">2048</metadata>
       <pfn url="file:///usr/bin/keg" site="TestCluster"/>
    </executable>
    <executable namespace="pegasus" name="findrange" version="4.0" installed="true" arch="x86"
 os="linux">
       <pfn url="file:///usr/bin/keg" site="TestCluster"/>
    </executable>
    <executable namespace="pegasus" name="analyze" version="4.0" installed="true" arch="x86"
 os="linux">
       <pfn url="file:///usr/bin/keg" site="TestCluster"/>
    </executable>

<!-- Section 5: Transformations - Aggregates executables and Files (can be empty) -->


<!-- Section 6: Job's, DAX's or Dag's - Defines a JOB or DAX or DAG (Atleast 1 required) -->

    <job id="j1" namespace="pegasus" name="preprocess" version="4.0">
       <metadata key="time">60</metadata>
       <argument>-a preprocess -T 60 -i  <file name="f.a"/> -o  <file name="f.b1"/>   <file
 name="f.b2"/></argument>
       <uses name="f.a" link="input">
          <metadata key="size">1024</metadata>
       </uses>
       <uses name="f.b1" link="output" transfer="true" register="true"/>
       <uses name="f.b2" link="output" transfer="true" register="true"/>
       <invoke when="start">/pegasus/libexec/notification/email -t notify@example.com</invoke>
       <invoke when="at_end">/pegasus/libexec/notification/email -t notify@example.com</invoke>
    </job>
    <job id="j2" namespace="pegasus" name="findrange" version="4.0">
       <metadata key="time">60</metadata>
       <argument>-a findrange -T 60 -i  <file name="f.b1"/> -o  <file name="f.c1"/></argument>
       <uses name="f.b1" link="input"/>
       <uses name="f.c1" link="output" transfer="true" register="true"/>
       <invoke when="start">/pegasus/libexec/notification/email -t notify@example.com</invoke>
       <invoke when="at_end">/pegasus/libexec/notification/email -t notify@example.com</invoke>
    </job>
    <job id="j3" namespace="pegasus" name="findrange" version="4.0">
       <metadata key="time">60</metadata>
       <argument>-a findrange -T 60 -i  <file name="f.b2"/> -o  <file name="f.c2"/></argument>
       <uses name="f.b2" link="input"/>
       <uses name="f.c2" link="output" transfer="true" register="true"/>
       <invoke when="start">/pegasus/libexec/notification/email -t notify@example.com</invoke>
       <invoke when="at_end">/pegasus/libexec/notification/email -t notify@example.com</invoke>
    </job>
    <job id="j4" namespace="pegasus" name="analyze" version="4.0">
       <metadata key="time">60</metadata>
       <argument>-a analyze -T 60 -i  <file name="f.c1"/>   <file name="f.c2"/> -o  <file name="f.d"/
></argument>
       <uses name="f.c1" link="input"/>
```

```
            <uses name="f.c2" link="input"/>
            <uses name="f.d" link="output" transfer="true" register="true"/>
            <invoke when="start">/pegasus/libexec/notification/email -t notify@example.com</invoke>
            <invoke when="at_end">/pegasus/libexec/notification/email -t notify@example.com</invoke>
        </job>

    <!-- Section 7: Dependencies - Parent Child relationships (can be empty) -->

        <child ref="j2">
            <parent ref="j1"/>
        </child>
        <child ref="j3">
            <parent ref="j1"/>
        </child>
        <child ref="j4">
            <parent ref="j2"/>
            <parent ref="j3"/>
        </child>
    </adag>
```

The example workflow representation in form of a DAX requires external catalogs, such as transformation catalog (TC) to resolve the logical job names (such as diamond::preprocess:2.0), and a replica catalog (RC) to resolve the input file f.a. The above workflow defines the four jobs just like the example picture, and the files that flow between the jobs. The intermediary files are neither registered nor staged out, and can be considered transient. Only the final result file f.d is staged out.

Data Discovery (Replica Catalog)

The Replica Catalog keeps mappings of logical file ids/names (LFN's) to physical file ids/names (PFN's). A single LFN can map to several PFN's. A PFN consists of a URL with protocol, host and port information and a path to a file. Along with the PFN one can also store additional key/value attributes to be associated with a PFN.

Pegasus supports the following implementations of the Replica Catalog.

1. **File**(Default)

2. **Regex**

3. **Directory**

4. **Database via JDBC**

5. **MRC**

File

In this mode, Pegasus queries a file based replica catalog. The file format is a simple multicolumn format. It is neither transactionally safe, nor advised to use for production purposes in any way. Multiple concurrent instances will conflict with each other. The site attribute should be specified whenever possible. The attribute key for the site attribute is **"site".**

```
LFN PFN
LFN PFN a=b [..]
LFN PFN a="b" [..]
"LFN w/LWS" "PFN w/LWS" [..]
```

The LFN may or may not be quoted. If it contains linear whitespace, quotes, backslash or an equal sign, it must be quoted and escaped. The same conditions apply for the PFN. The attribute key-value pairs are separated by an equality sign without any whitespaces. The value may be quoted. The LFN sentiments about quoting apply.

The file mode is the Default mode. In order to use the File mode you have to set the following properties

1. **pegasus.catalog.replica=File**

2. **pegasus.catalog.replica.file=*<path to the replica catalog file>***

Regex

In this mode, Pegasus queries a file based replica catalog. The file format is a simple multicolumn format. It is neither transactionally safe purposes in any way. Multiple concurrent instances will conflict with each other. The site attribute should be specified whenever possible. The attribute key for the site attribute is **"site".**

In addition users can specifiy regular expression based LFN's. A regular expression based entry should be qualified with an attribute named 'regex'. The attribute regex when set to true identifies the catalog entry as a regular expression based entry. Regular expressions should follow Java regular expression syntax.

For example, consider a replica catalog as shown below.

Entry 1 refers to an entry which does not use a regular expressions. This entry would only match a file named 'f.a', and nothing else.

Entry 2 referes to an entry which uses a regular expression. In this entry f.a referes to files having name as f<any-character>a i.e. faa, f.a, f0a, etc.

```
#1
f.a file:///Volumes/data/input/f.a site="local"
#2
f.a file:///Volumes/data/input/f.a site="local" regex="true"
```

Regular expression based entries also support substitutions. For example, consider the regular expression based entry shown below.

Entry 3 will match files with name alpha.csv, alpha.txt, alpha.xml. In addition, values matched in the expression can be used to generate a PFN.

For the entry below if the file being looked up is alpha.csv, the PFN for the file would be generated as file:///Volumes/data/input/csv/alpha.csv. Similary if the file being lookedup was alpha.csv, the PFN for the file would be generated as file:///Volumes/data/input/xml/alpha.xml i.e. The section [0], [1] will be replaced. Section [0] refers to the entire string i.e. alpha.csv. Section [1] refers to a partial match in the input i.e. csv, or txt, or xml. Users can utilize as many sections as they wish.

```
#3
alpha\.(csv|txt|xml) file:///Volumes/data/input/[1]/[0] site="local" regex="true"
```

In case of a LFN name matching multiple entries in the file, the implementation picks up the first matching regex as it appears in the file. If you want to specify a default location for all LFN's that don't match any regex expression, you can have this entry as the last entry in your file.

```
#4 all unmatched LFN's reside in the same input directory.

.*    file:///Volumes/data/input/[0] site="local" regex="true"
```

Directory

In this mode, Pegasus does a directory listing on an input directory to create the LFN to PFN mappings. The directory listing is performed recursively, resulting in deep LFN mappings. For example, if an input directory $input is specified with the following structure

```
$input
$input/f.1
$input/f.2
$input/D1
$input/D1/f.3
```

Pegasus will create the mappings the following LFN PFN mappings internally

```
f.1 file://$input/f.1  site="local"
f.2 file://$input/f.2  site="local"
D1/f.3 file://$input/D1/f.3 site="local"
```

Users can optionally specify additional properties to configure the behavior of this implementation.

1. **pegasus.catalog.replica.directory** to specify the path to the directory where the files exist.

2. **pegasus.catalog.replica.directory.site** to specify a site attribute other than local to associate with the mappings.

3. **pegasus.catalog.replica.directory.flat.lfn** to specify whether you want deep LFN's to be constructed or not. If not specified, value defaults to false i.e. deep lfn's are constructed for the mappings.

4. **pegasus.catalog.replica.directory.url.prefix** to associate a URL prefix for the PFN's constructed. If not specified, the URL defaults to file://

> ### Tip
>
> pegasus-plan has **--input-dir** option that can be used to specify an input directory on the command line. This allows you to specify a separate replica catalog to catalog the locations of output files.

JDBCRC

In this mode, Pegasus queries a SQL based replica catalog that is accessed via JDBC. To create the schema for JDBCRC use the pegasus-db-admin command line tool.

> ### Note
>
> A site attribute was added to the SQL schema as a unique key for 4.4. To update an existing database schema, use pegasus-db-admin tool.

Figure 4.2. Schema Image of the JDBCRC.
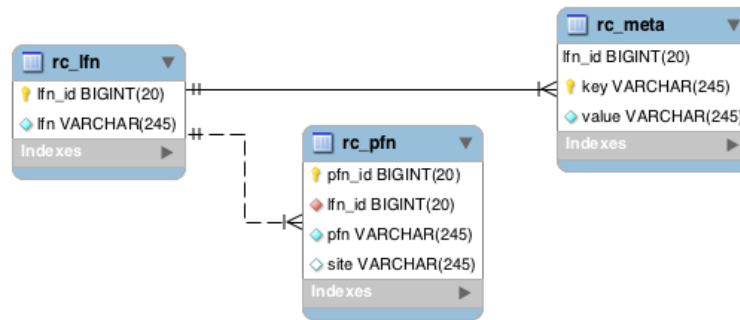
To use JDBCRC, the user additionally needs to set the following properties

1. **pegasus.catalog.replica JDBCRC**

2. **pegasus.catalog.replica.db.driver mysql | postgres |sqlite**

3. **pegasus.catalog.replica.db.url=*<jdbc url to the database> e.g jdbc:mysql://database-host.isi.edu/database-name | jdbc:sqlite:/shared/jdbcrc.db***

4. **pegasus.catalog.replica.db.user=*<database user>***

5. **pegasus.catalog.replica.db.password=*<database password>***

Users can use the command line client *pegasus-rc-client* to interface to query, insert and remove entries from the JDBCRC backend. Starting 4.5 release, there is also support for sqlite databases. Specify the jdbc url to refer to a sqlite database.

MRC

In this mode, Pegasus queries multiple replica catalogs to discover the file locations on the grid.

To use it set

1. **pegasus.catalog.replica=*MRC***

Each associated replica catalog can be configured via properties as follows.

The user associates a variable name referred to as [value] for each of the catalogs, where [value] is any legal identifier (concretely [A-Za-z][_A-Za-z0-9]*) For each associated replica catalogs the user specifies the following properties

- **pegasus.catalog.replica.mrc.[value]** - specifies the type of replica catalog.

- **pegasus.catalog.replica.mrc.[value].key** - specifies a property name key for a particular catalog

For example, to query a File catalog and JDBCRC at the same time specify the following:

- **pegasus.catalog.replica=MRC**

- **pegasus.catalog.replica.mrc.jdbcrc=JDBCRC**

- **pegasus.catalog.replica.mrc.jdbcrc.url=*<jdbc url >***

- **pegasus.catalog.replica.mrc.file1=File**

- **pegasus.catalog.replica.mrc.file1.url=<path to file based replica catalog>**

In the above example, **jdbcrc** and **file1** are any valid identifier names and **url** is the property key that needed to be specified.

Another example is to use MRC with multiple input directories. Sample properties for that configuration are listed below

- **pegasus.catalog.replica=MRC**

- **pegasus.catalog.replica.mrc.directory1=Directory**

- **pegasus.catalog.replica.mrc.directory1.directory=/path/to/dir1**

- **pegasus.catalog.replica.mrc.directory1.directory.site=obelix**

- **pegasus.catalog.replica.mrc.directory2=Directory**

- **pegasus.catalog.replica.mrc.directory2.directory=/path/to/dir2**

- **pegasus.catalog.replica.mrc.directory2.directory.site=corbusier**

Replica Catalog Client pegasus-rc-client

The client used to interact with the Replica Catalogs is pegasus-rc-client. The implementation that the client talks to is configured using Pegasus properties.

Lets assume we create a file f.a in your home directory as shown below.

```
$ date > $HOME/f.a
```

We now need to register this file in the **File** replica catalog located in **$HOME/rc** using the pegasus-rc-client. Replace the **gsiftp://url** with the appropriate parameters for your grid site.

```
$ pegasus-rc-client -Dpegasus.catalog.replica=File -Dpegasus.catalog.replica.file=$HOME/rc insert \
 f.a gsiftp://somehost:port/path/to/file/f.a site=local
```

You may first want to verify that the file registeration is in the replica catalog. Since we are using a File catalog we can look at the file **$HOME/rc** to view entries.

```
$ cat $HOME/rc
```

```
# file-based replica catalog: 2010-11-10T17:52:53.405-07:00
f.a gsiftp://somehost:port/path/to/file/f.a site=local
```

The above line shows that entry for file **f.a** was made correctly.

You can also use the **pegasus-rc-client** to look for entries.

```
$ pegasus-rc-client -Dpegasus.catalog.replica=File -Dpegasus.catalog.replica.file=$HOME/rc lookup
 LFN f.a
```

```
f.a gsiftp://somehost:port/path/to/file/f.a site=local
```

Resource Discovery (Site Catalog)

The Site Catalog describes the compute resources (which are often clusters) that we intend to run the workflow upon. A site is a homogeneous part of a cluster that has at least a single GRAM gatekeeper with a **jobmanager-fork** and *jobmanager-<scheduler>* interface and at least one **gridftp** server along with a shared file system. The GRAM gatekeeper can be either WS GRAM or Pre-WS GRAM. A site can also be a condor pool or glidein pool with a shared file system.

The Site Catalog can be described as an XML . Pegasus currently supports two schemas for the Site Catalog:

1. **XML4**(Default) Corresponds to the schema described here [schemas/sc-4.0/sc-4.0.html].

2. **XML3**(Deprecated) Corresponds to the schema described here [schemas/sc-3.0/sc-3.0.html]

XML4

This is the default format for Pegasus 4.2. This format allows defining filesystem of shared as well as local type on the head node of the remote cluster as well as on the backend nodes

Figure 4.3. Schema Image of the Site Catalog XML4
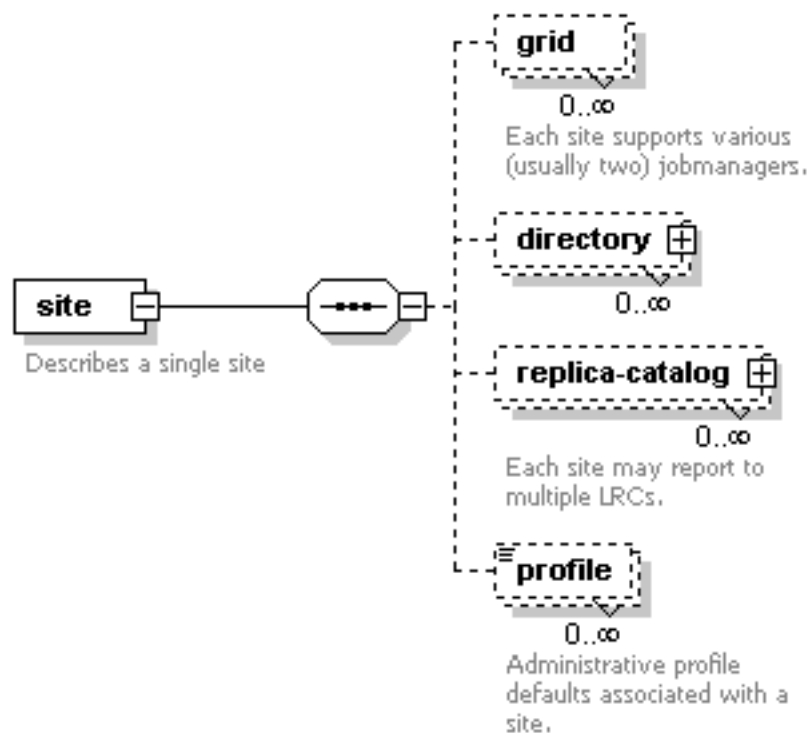
Below is an example of the XML4 site catalog

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog http://pegasus.isi.edu/
schema/sc-4.0.xsd"
             version="4.0">

    <site  handle="local" arch="x86_64" os="LINUX">
        <directory type="shared-scratch" path="/tmp/workflows/scratch">
            <file-server operation="all" url="file:///tmp/workflows/scratch"/>
        </directory>
        <directory type="local-storage" path="/tmp/workflows/outputs">
            <file-server operation="all" url="file:///tmp/workflows/outputs"/>
        </directory>
    </site>

    <site  handle="condor_pool" arch="x86_64" os="LINUX">
        <grid type="gt5" contact="smarty.isi.edu/jobmanager-pbs" scheduler="PBS"
 jobtype="auxillary"/>
        <grid type="gt5" contact="smarty.isi.edu/jobmanager-pbs" scheduler="PBS" jobtype="compute"/>
        <directory type="shared-scratch" path="/lustre">
            <file-server operation="all" url="gsiftp://smarty.isi.edu/lustre"/>
        </directory>
        <replica-catalog type="LRC" url="rlsn://smarty.isi.edu"/>
    </site>

    <site  handle="staging_site" arch="x86_64" os="LINUX">
        <directory type="shared-scratch" path="/data">
            <file-server operation="put" url="scp://obelix.isi.edu/data"/>
            <file-server operation="get" url="http://obelix.isi.edu/data"/>
        </directory>
    </site>

</sitecatalog>
```

Described below are some of the entries in the site catalog.

1. **site** - A site identifier.

2. **Directory** - Info about filesystems Pegasus can use for storing temporary and long-term files. There are several configurations:

   • **shared-scratch** - This describe a scratch file systems. Pegasus will use this to store intermediate data between jobs and other temporary files.

   • **local-storage** - This describes the storage file systems (long term). This is the directory Pegasus will stage output files to.

   • **local-scratch** - This describe the scratch file systems available locally on a compute node. This parameter is not commonly used and can be left unset in most cases.

   For each of the directories, you can specify access methods. Allowed methods are **put**, **get**, and **all** which means both put and get. For each mehod, specify a URL including the protocol. For example, if you want share data via http using the /var/www/staging directory, you can use scp://hostname/var/www for the put element and http://hostname/staging for the get element.

3. **arch,os,osrelease,osversion, glibc** - The arch/os/osrelease/osversion/glibc of the site. OSRELEASE, OSVERSION and GLIBC are optional

   ARCH can have one of the following values X86, X86_64, SPARCV7, SPARCV9, AIX, PPC.

   OS can have one of the following values LINUX,SUNOS,MACOSX. The default value for sysinfo if none specified is X86::LINUX

4. **replica-catalog** - URL for a local replica catalog (LRC) to register your files in. Only used for RLS implementation of the RC. This is optional and support for RLS has been dropped in Pegasus 4.5.0 release.

5. **Profiles** - One or many profiles can be attached to a site.

   One example is the environments to be set on a remote site.

To use this site catalog the follow properties need to be set:

1. **pegasus.catalog.site.file=<path to the site catalog file>**

XML3

Warning

This format is now deprecated in favor of the XML4 format. If you are still using the File format you should convert it to XML4 format using the client pegasus-sc-converter

This is the default format for Pegasus 3.0. This format allows defining filesystem of shared as well as local type on the head node of the remote cluster as well as on the backend nodes

Figure 4.4. Schema Image of the Site Catalog XML 3
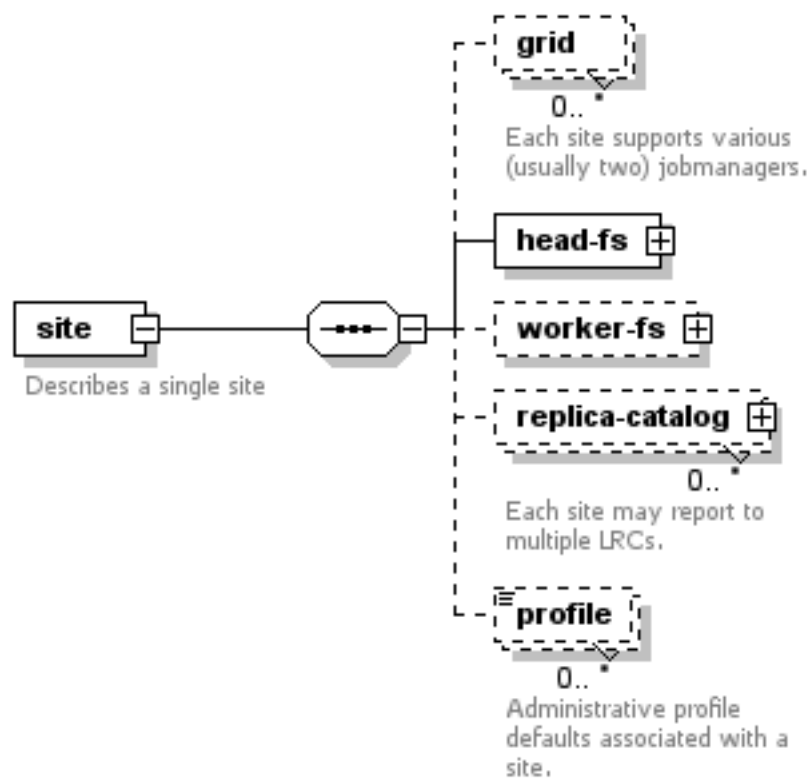
Below is an example of the XML3 site catalog

```
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog
http://pegasus.isi.edu/schema/sc-3.0.xsd" version="3.0">
  <site  handle="isi" arch="x86" os="LINUX" osrelease="" osversion="" glibc="">
      <grid  type="gt2" contact="smarty.isi.edu/jobmanager-pbs" scheduler="PBS" jobtype="auxillary"/
>
      <grid  type="gt2" contact="smarty.isi.edu/jobmanager-pbs" scheduler="PBS" jobtype="compute"/>
          <head-fs>
              <scratch>
                  <shared>
                     <file-server protocol="gsiftp" url="gsiftp://skynet-data.isi.edu"
                               mount-point="/nfs/scratch01" />
                     <internal-mount-point mount-point="/nfs/scratch01"/>
                  </shared>
              </scratch>
              <storage>
```

```
            <shared>
                <file-server protocol="gsiftp" url="gsiftp://skynet-data.isi.edu"
                             mount-point="/exports/storage01"/>
                <internal-mount-point mount-point="/exports/storage01"/>
            </shared>
        </storage>
    </head-fs>
    <replica-catalog  type="LRC" url="rlsn://smarty.isi.edu"/>
    <profile namespace="env" key="PEGASUS_HOME" >/nfs/vdt/pegasus</profile>
    <profile namespace="env" key="GLOBUS_LOCATION" >/vdt/globus</profile>
  </site>
</sitecatalog>
```

Described below are some of the entries in the site catalog.

1. **site** - A site identifier.

2. **replica-catalog** - URL for a local replica catalog (LRC) to register your files in. Only used for RLS implementation of the RC. This is optional and support for RLS has been dropped in Pegasus 4.5.0.

3. **File Systems** - Info about filesystems mounted on the remote clusters head node or worker nodes. It has several configurations

   • **head-fs/scratch** - This describe the scratch file systems (temporary for execution) available on the head node

   • **head-fs/storage** - This describes the storage file systems (long term) available on the head node

   • **worker-fs/scratch** - This describe the scratch file systems (temporary for execution) available on the worker node

   • **worker-fs/storage** - This describes the storage file systems (long term) available on the worker node

   Each scratch and storage entry can contain two sub entries,

   • SHARED for shared file systems like NFS, LUSTRE etc.

   • LOCAL for local file systems (local to the node/machine)

   Each of the filesystems are defined by used a file-server element. Protocol defines the protocol uses to access the files, URL defines the url prefix to obtain the files from and mount-point is the mount point exposed by the file server.

   Along with this an internal-mount-point needs to defined to access the files directly from the machine without any file servers.

4. **arch,os,osrelease,osversion, glibc** - The arch/os/osrelease/osversion/glibc of the site. OSRELEASE, OSVERSION and GLIBC are optional

   ARCH can have one of the following values X86, X86_64, SPARCV7, SPARCV9, AIX, PPC.

   OS can have one of the following values LINUX,SUNOS,MACOSX. The default value for sysinfo if none specified is X86::LINUX

5. **Profiles** - One or many profiles can be attached to a pool.

   One example is the environments to be set on a remote pool.

To use this site catalog the follow properties need to be set:

1. **pegasus.catalog.site.file=<path to the site catalog file>**

Site Catalog Converter pegasus-sc-converter

Pegasus 4.2 by default now parses Site Catalog format conforming to the SC schema 4.0 (XML4) available here [schemas/sc-4.0/sc-4.0.xsd] and is explained in detail in the Catalog Properties section of Running Workflows.

Pegasus 4.2 comes with a pegasus-sc-converter that will convert users old site catalog (XML3) to the XML4 format. Sample usage is given below.

```
$ pegasus-sc-converter -i sample.sites.xml -I XML3 -o sample.sites.xml4 -O XML4
```

```
2010.11.22 12:55:14.169 PST:   Written out the converted file to sample.sites.xml4
```

To use the converted site catalog, in the properties do the following:

1. unset pegasus.catalog.site or set pegasus.catalog.site to XML

2. point pegasus.catalog.site.file to the converted site catalog

Executable Discovery (Transformation Catalog)

The Transformation Catalog maps logical transformations to physical executables on the system. It also provides additional information about the transformation as to what system they are compiled for, what profiles or environment variables need to be set when the transformation is invoked etc.

Pegasus currently supports a Text formatted Transformation Catalog

1. **Text:** A multi line text based Transformation Catalog (DEFAULT)

In this guide we will look at the format of the Multiline Text based TC.

MultiLine Text based TC (Text)

The multile line text based TC is the new default TC in Pegasus. This format allows you to define the transformations

The file is read and cached in memory. Any modifications, as adding or deleting, causes an update of the memory and hence to the file underneath. All queries are done against the memory representation. The file sample.tc.text in the etc directory contains an example

```
tr example::keg:1.0 {

#specify profiles that apply for all the sites for the transformation
#in each site entry the profile can be overriden

  profile env "APP_HOME" "/tmp/myscratch"
  profile env "JAVA_HOME" "/opt/java/1.6"

  site isi {
    profile env "HELLo" "WORLD"
    profile condor "FOO" "bar"
    profile env "JAVA_HOME" "/bin/java.1.6"
    pfn "/path/to/keg"
    arch "x86"
    os "linux"
    osrelease "fc"
    osversion "4"
    type "INSTALLED"
  }

  site wind {
    profile env "CPATH" "/usr/cpath"
    profile condor "universe" "condor"
    pfn "file:///path/to/keg"
    arch "x86"
    os "linux"
    osrelease "fc"
    osversion "4"
    type "STAGEABLE"
  }
}
```

The entries in this catalog have the following meaning

1. **tr** tr - A transformation identifier. (Normally a Namespace::Name:Version.. The Namespace and Version are optional.)

2. **pfn** - URL or file path for the location of the executable. The pfn is a file path if the transformation is of type INSTALLED and generally a url (file:/// or http:// or gridftp://) if of type STAGEABLE

3. **site** - The site identifier for the site where the transformation is available

4. **type** - The type of transformation. Whether it is installed ("INSTALLED") on the remote site or is availabe to stage ("STAGEABLE").

5. **arch, os, osrelease, osversion** - The arch/os/osrelease/osversion of the transformation. osrelease and osversion are optional.

   ARCH can have one of the following values x86, x86_64, sparcv7, sparcv9, ppc, aix. The default value for arch is x86

   OS can have one of the following values linux,sunos,macosx. The default value for OS if none specified is linux

6. **Profiles** - One or many profiles can be attached to a transformation for all sites or to a transformation on a particular site.

To use this format of the Transformation Catalog you need to set the following properties

1. **pegasus.catalog.transformation=Text**

2. **pegasus.catalog.transformation.file=<path to the transformation catalog file>**

Containerized Applications in the Transformation Catalog

Users can specify what container they want to use for running their application in the Transformation Catalog using the multi line text based format described in this section. Users can specify an optional attribute named container that refers to the container to be used for the application.

```
tr example::keg:1.0 {

  #specify profiles that apply for all the sites for the transformation
  #in each site entry the profile can be overriden

  profile env "APP_HOME" "/tmp/myscratch"
  profile env "JAVA_HOME" "/opt/java/1.6"

  site isi {
    # environment to be set when the job is run in the container
    # overrides env profiles specified in the container
    profile env "HELLo" "WORLD"
    profile env "JAVA_HOME" "/bin/java.1.6"

    profile condor "FOO" "bar"

    pfn "/path/to/keg
    arch "x86"
    os "linux"
    osrelease "fc"
    osversion "4"

    # INSTALLED means pfn refers to path in the container.
    # STAGEABLE means the executable can be staged into the container
    type "INSTALLED"

    #optional attribute to specify the container to use
    container "centos-pegasus"
  }
}

cont centos-pegasus{
    # can be either docker or singularity
    type "docker"

    # URL to image in a docker|singularity hub OR
    # URL to an existing docker image exported as a tar file or singularity image
    image "docker:///rynge/montage:latest"

    # optional site attribute to tell pegasus which site tar file
```

```
      # exists. useful for handling file URL's correctly
      image_site "optional site"

      # environment to be set when the job is run in the container
      # only env profiles are supported
      profile env "JAVA_HOME" "/opt/java/1.6"
}
```

The container itself is defined using the cont entry. Multiple transformations can refer to the same container.

1. **cont** cont - A container identifier.

2. **image** - URL to image in a docker|singularity hub or URL to an existing docker image exported as a tar file or singularity image. Example of a docker hub URL is docker:///rynge/montage:latest, while for singularity shub:// pegasus-isi/fedora-montage

3. **image_site** - The site identifier for the site where the container is available

4. **Profiles** - One or many profiles can be attached to a transformation for all sites or to a transformation on a particular site. For containers, only env profiles are supported.

> ## Note
>
> Containerized Applications can only be specified in the transformation catalog, not via the DAX API.

TC Client pegasus-tc-client

We need to map our declared transformations (preprocess, findrange, and analyze) from the example DAX above to a simple "mock application" name "keg" ("canonical example for the grid") which reads input files designated by arguments, writes them back onto output files, and produces on STDOUT a summary of where and when it was run. Keg ships with Pegasus in the bin directory. Run keg on the command line to see how it works.

```
$ keg -o /dev/fd/1

Timestamp Today: 20040624T054607-05:00 (1088073967.418;0.022)
Applicationname: keg @ 10.10.0.11 (VPN)
Current Workdir: /home/unique-name
Systemenvironm.: i686-Linux 2.4.18-3
Processor Info.: 1 x Pentium III (Coppermine) @ 797.425
Output Filename: /dev/fd/1
```

Now we need to map all 3 transformations onto the "keg" executable. We place these mappings in our File transformation catalog for site clus1.

> ## Note
>
> In earlier version of Pegasus users had to define entries for Pegasus executables such as transfer, replica client, dirmanager, etc on each site as well as site "local". This is no longer required. Pegasus versions 2.0 and later automatically pick up the paths for these binaries from the environment profile PEGASUS_HOME set in the site catalog for each site.
>
> A single entry needs to be on one line. The above example is just formatted for convenience.

Alternatively you can also use the pegasus-tc-client to add entries to any implementation of the transformation catalog. The following example shows the addiition the last entry in the File based transformation catalog.

```
$ pegasus-tc-client -Dpegasus.catalog.transformation=Text \
-Dpegasus.catalog.transformation.file=$HOME/tc -a -r clus1 -l black::analyze:1.0 \
-p gsiftp://clus1.com/opt/nfs/vdt/pegasus/bin/keg  -t STAGEABLE -s INTEL32::LINUX \
-e ENV::KEY3="VALUE3"

2007.07.11 16:12:03.712 PDT: [INFO] Added tc entry sucessfully
```

To verify if the entry was correctly added to the transformation catalog you can use the pegasus-tc-client to query.

```
$ pegasus-tc-client -Dpegasus.catalog.transformation=File \
-Dpegasus.catalog.transformation.file=$HOME/tc -q -P -l black::analyze:1.0
```

```
#RESID     LTX            PFN                    TYPE            SYSINFO

clus1      black::analyze:1.0    gsiftp://clus1.com/opt/nfs/vdt/pegasus/bin/keg
                  STAGEABLE    INTEL32::LINUX
```

Note

pegasus-tc-client is no longer actively developed and is deprecated.

TC Converter Client pegasus-tc-converter

Pegasus 3.0 by default now parses a file based multi line textual format of a Transformation Catalog. The new Text format is explained in detail in the chapter on Catalogs.

Pegasus 3.0 comes with a pegasus-tc-converter that will convert users old transformation catalog (File) to the Text format. Sample usage is given below.

```
$ pegasus-tc-converter -i sample.tc.data -I File -o sample.tc.text -O Text

2010.11.22 12:53:16.661 PST:    Successfully converted Transformation Catalog from File to Text
2010.11.22 12:53:16.666 PST:    The output transfomation catalog is in file  sample.tc.text
```

To use the converted transformation catalog, in the properties do the following:

1. unset pegasus.catalog.transformation or set pegasus.catalog.transformation to Text

2. point pegasus.catalog.transformation.file to the converted transformation catalog

Variable Expansion

Pegasus Planner supports notion of variable expansions in the DAX and the catalog files along the same lines as bash variable expansion works. This is often useful, when you want paths in your catalogs or profile values in the DAX to be picked up from the environment. An error is thrown if a variable cannot be expanded.

To specify a variable that needs to be expanded, the syntax is ${VARIABLE_NAME} , similar to BASH variable expansion. An important thing to note is that the variable names need to be enclosed in curly braces. For example

```
 ${FOO}  - will be expanded by Pegasus
 $FOO    - will NOT be expanded by Pegasus.
```

Also variable names are case sensitive.

Some examples of variable expansion are illustrated below:

- **DAX**

  A job in the DAX file needs to have a globus profile key project associated and the value has to be picked up (per user) from user environment.

  ```
  <profile namespace="globus" key="project">${PROJECT}</profile>
  ```

- **Site Catalog**

  In the site catalog, the site catalog entries are templated, where paths are resolved on the basis of values of environment variables. For example, below is a templated entry for a local site where $PWD is the working directory from where pegasus-plan is invoked.

  ```
  <site  handle="local" arch="x86_64" os="LINUX" osrelease="" osversion="" glibc="">
        <directory  path="${PWD}/LOCAL/shared-scratch" type="shared-scratch" free-size="" total-
  size="">
                  <file-server  operation="all" url="file:///${PWD}/LOCAL/shared-scratch">
                  </file-server>
        </directory>
        <directory  path="${PWD}/LOCAL/shared-storage" type="shared-storage" free-size="" total-
  size="">
  ```

```
                <file-server  operation="all" url="file:///${PWD}/LOCAL/shared-storage">
                </file-server>
        </directory>
        <profile namespace="env" key="PEGASUS_HOME">/usr</profile>
        <profile namespace="pegasus" key="clusters.num" >1</profile>
</site>
```

- **Replica Catalog**

  The input file locations in the Replica Catalog can be resolved based on values of environment variables.

  ```
  # File Based Replica Catalog
  production_200.conf file://$PWD/production_200.conf site="local"
  ```

  ## Note

  Variable expansion is only supported for File based Replica Catalog, not Regex or other file based formats.

- **Transformation Catalog**

  Similarly paths in the transformation catalog or profile values can be picked up from the environment i.e environment variables OS , ARCH and PROJECT are defined in user environment when launching pegasus-plan.

  ```
  # Snippet from a Text Based Transformation Catalog
  tr pegasus::keg{
      site obelix {
          profile globus "project" "${PROJECT}"
          pfn "/usr/bin/pegasus-keg"
          arch "${ARCH}"
          os "${OS}"
          type "INSTALLED"
      }
  }
  ```

Chapter 5. Running Workflows

Executable Workflows (DAG)

The DAG is an executable (concrete) workflow that can be executed over a variety of resources. When the workflow tasks are mapped to multiple resources that do not share a file system, explicit nodes are added to the workflow for orchestrating data. transfer between the tasks.

When you take the DAX workflow created in Creating Workflows, and plan it for a single remote grid execution, here a site with handle **hpcc**, and plan the workflow without clean-up nodes, the following concrete workflow is built:

Figure 5.1. Black Diamond DAG
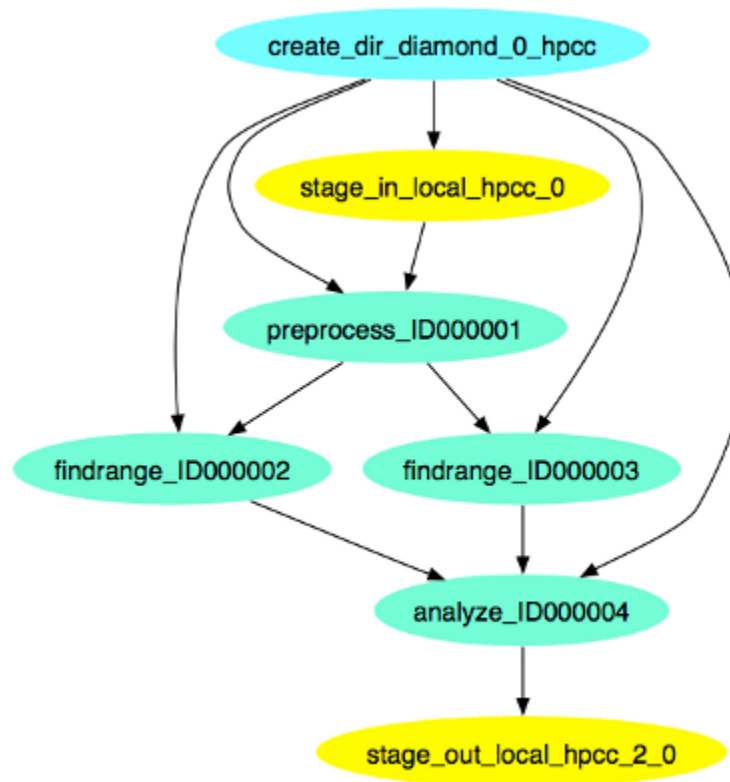
Planning augments the original abstract workflow with ancillary tasks to facility the proper execution of the workflow. These tasks include:

- the creation of remote working directories. These directories typically have name that seeks to avoid conflicts with other simultaneously running similar workflows. Such tasks use a job prefix of `create_dir`.

- the stage-in of input files before any task which requires these files. Any file consumed by a task needs to be staged to the task, if it does not already exist on that site. Such tasks use a job prefix of `stage_in`.If multiple files from various sources need to be transferred, multiple stage-in jobs will be created. Additional advanced options permit to control the size and number of these jobs, and whether multiple compute tasks can share stage-in jobs.

- the original DAX job is concretized into a compute task in the DAG. Compute jobs are a concatenation of the job's **name** and **id** attribute from the DAX file.

- the stage-out of data products to a collecting site. Data products with their **transfer** flag set to `false` will not be staged to the output site. However, they may still be eligible for staging to other, dependent tasks. Stage-out tasks use a job prefix of `stage_out`.

- If compute jobs run at different sites, an intermediary staging task with prefix stage_inter is inserted between the compute jobs in the workflow, ensuring that the data products of the parent are available to the child job.

- the registration of data products in a replica catalog. Data products with their **register** flag set to false will not be registered.

- the clean-up of transient files and working directories. These steps can be omitted with the **--no-cleanup** option to the planner.

The Data Management chapter details more about when and how staging nodes are inserted into the workflow.

The DAG will be found in file diamond-0.dag, constructed from the **name** and **index** attributes found in the root element of the DAX file.

```
######################################################################
# PEGASUS WMS GENERATED DAG FILE
# DAG diamond
# Index = 0, Count = 1
######################################################################

JOB create_dir_diamond_0_hpcc create_dir_diamond_0_hpcc.sub
SCRIPT POST create_dir_diamond_0_hpcc /opt/pegasus/default/bin/pegasus-exitcode
 create_dir_diamond_0_hpcc.out

JOB stage_in_local_hpcc_0 stage_in_local_hpcc_0.sub
SCRIPT POST stage_in_local_hpcc_0 /opt/pegasus/default/bin/pegasus-exitcode
 stage_in_local_hpcc_0.out

JOB preprocess_ID000001 preprocess_ID000001.sub
SCRIPT POST preprocess_ID000001 /opt/pegasus/default/bin/pegasus-exitcode preprocess_ID000001.out

JOB findrange_ID000002 findrange_ID000002.sub
SCRIPT POST findrange_ID000002 /opt/pegasus/default/bin/pegasus-exitcode findrange_ID000002.out

JOB findrange_ID000003 findrange_ID000003.sub
SCRIPT POST findrange_ID000003 /opt/pegasus/default/bin/pegasus-exitcode findrange_ID000003.out

JOB analyze_ID000004 analyze_ID000004.sub
SCRIPT POST analyze_ID000004 /opt/pegasus/default/bin/pegasus-exitcode analyze_ID000004.out

JOB stage_out_local_hpcc_2_0 stage_out_local_hpcc_2_0.sub
SCRIPT POST stage_out_local_hpcc_2_0 /opt/pegasus/default/bin/pegasus-exitcode
 stage_out_local_hpcc_2_0.out

PARENT findrange_ID000002 CHILD analyze_ID000004
PARENT findrange_ID000003 CHILD analyze_ID000004
PARENT preprocess_ID000001 CHILD findrange_ID000002
PARENT preprocess_ID000001 CHILD findrange_ID000003
PARENT analyze_ID000004 CHILD stage_out_local_hpcc_2_0
PARENT stage_in_local_hpcc_0 CHILD preprocess_ID000001
PARENT create_dir_diamond_0_hpcc CHILD findrange_ID000002
PARENT create_dir_diamond_0_hpcc CHILD findrange_ID000003
PARENT create_dir_diamond_0_hpcc CHILD preprocess_ID000001
PARENT create_dir_diamond_0_hpcc CHILD analyze_ID000004
PARENT create_dir_diamond_0_hpcc CHILD stage_in_local_hpcc_0
######################################################################
# End of DAG
######################################################################
```

The DAG file declares all jobs and links them to a Condor submit file that describes the planned, concrete job. In the same directory as the DAG file are all Condor submit files for the jobs from the picture plus a number of additional helper files.

The various instructions that can be put into a DAG file are described in Condor's DAGMAN documentation [http://www.cs.wisc.edu/condor/manual/v7.5/2_10DAGMan_Applications.html].The constituents of the submit directory are described in the "Submit Directory Details"chapter

Mapping Refinement Steps

During the mapping process, the abstract workflow undergoes a series of refinement steps that converts it to an executable form.

Data Reuse

The abstract workflow after parsing is optionally handed over to the Data Reuse Module. The Data Reuse Algorithm in Pegasus attempts to prune all the nodes in the abstract workflow for which the output files exist in the Replica Catalog. It also attempts to cascade the deletion to the parents of the deleted node for e.g if the output files for the leaf nodes are specified, Pegasus will prune out all the workflow as the output files in which a user is interested in already exist in the Replica Catalog.

The Data Reuse Algorithm works in two passes

First Pass - Determine all the jobs whose output files exist in the Replica Catalog. An output file with the transfer flag set to false is treated equivalent to the file existing in the Replica Catalog , if the output file is not an input to any of the children of the job X.

Second Pass - The algorithm removes the job whose output files exist in the Replica Catalog and tries to cascade the deletion upwards to the parent jobs. We start the breadth first traversal of the workflow bottom up.

```
( It is already marked for deletion in Pass 1
     OR
     ( ALL of it's children have been marked for deletion
       AND
       ( Node's output files have transfer flags set to false
         OR
         Node's output files with transfer flag as true have locations recorded in the Replica
 Catalog
       )
     )
 )
```

Tip

The Data Reuse Algorithm can be disabled by passing the **--force** option to pegasus-plan.

Figure 5.2. Workflow Data Reuse
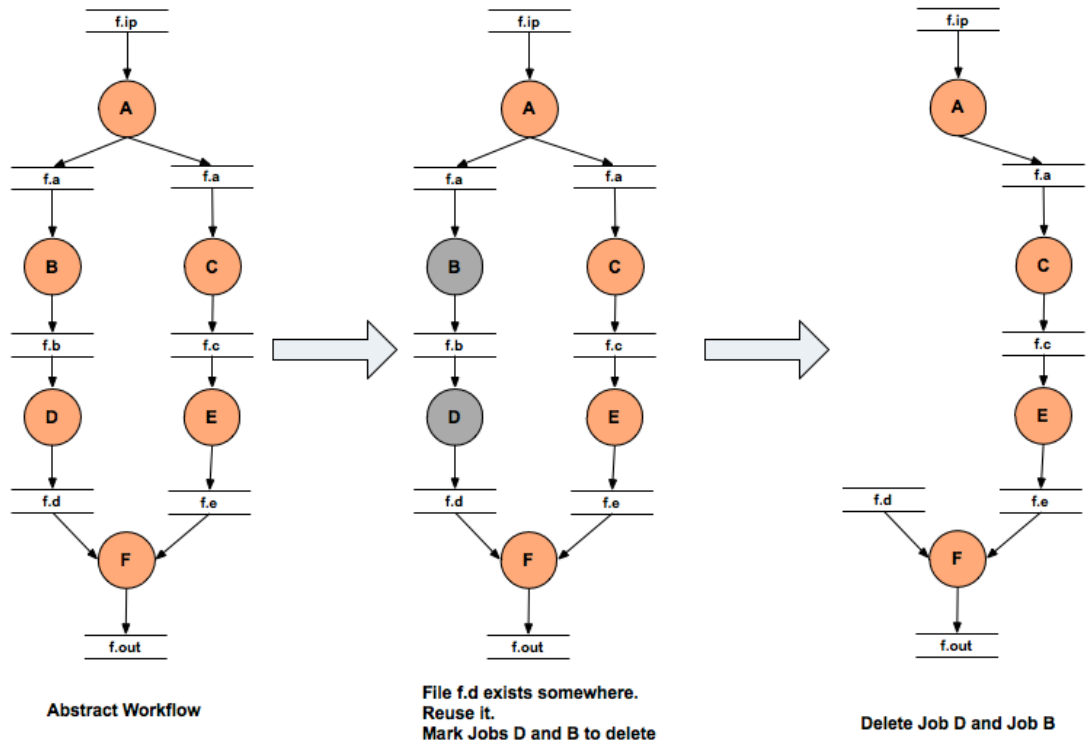
Site Selection

The abstract workflow is then handed over to the Site Selector module where the abstract jobs in the pruned workflow are mapped to the various sites passed by a user. The target sites for planning are specified on the command line using the **--sites** option to pegasus-plan. If not specified, then Pegasus picks up all the sites in the Site Catalog as candidate sites. Pegasus will map a compute job to a site only if Pegasus can

- find an INSTALLED executable on the site

- OR find a STAGEABLE executable that can be staged to the site as part of the workflow execution.

Pegasus supports variety of site selectors with Random being the default

- **Random**

  The jobs will be randomly distributed among the sites that can execute them.

- **RoundRobin**

  The jobs will be assigned in a round robin manner amongst the sites that can execute them. Since each site cannot execute every type of job, the round robin scheduling is done per level on a sorted list. The sorting is on the basis of the number of jobs a particular site has been assigned in that level so far. If a job cannot be run on the first site in the queue (due to no matching entry in the transformation catalog for the transformation referred to by the job), it goes to the next one and so on. This implementation defaults to classic round robin in the case where all the jobs in the workflow can run on all the sites.

- **Group**

  Group of jobs will be assigned to the same site that can execute them. The use of the **PEGASUS profile key group** in the DAX, associates a job with a particular group. The jobs that do not have the profile key associated with them, will be put in the default group. The jobs in the default group are handed over to the "Random" Site Selector for scheduling.

- **Heft**

  A version of the HEFT processor scheduling algorithm is used to schedule jobs in the workflow to multiple grid sites. The implementation assumes default data communication costs when jobs are not scheduled on to the same site. Later on this may be made more configurable.

  The runtime for the jobs is specified in the transformation catalog by associating the **pegasus profile key runtime** with the entries.

  The number of processors in a site is picked up from the attribute **idle-nodes** associated with the vanilla jobmanager of the site in the site catalog.

- **NonJavaCallout**

  Pegasus will callout to an external site selector.In this mode a temporary file is prepared containing the job information that is passed to the site selector as an argument while invoking it. The path to the site selector is specified by setting the property pegasus.site.selector.path. The environment variables that need to be set to run the site selector can be specified using the properties with a pegasus.site.selector.env. prefix. The temporary file contains information about the job that needs to be scheduled. It contains key value pairs with each key value pair being on a new line and separated by a =.

  The following pairs are currently generated for the site selector temporary file that is generated in the NonJava-Callout.

Table 5.1. Key Value Pairs that are currently generated for the site selector temporary file that is generated in the NonJavaCallout.

Key	Value

version	is the version of the site selector api,currently 2.0.
transformation	is the fully-qualified definition identifier for the transformation (TR) namespace::name:version.
derivation	is the fully qualified definition identifier for the derivation (DV), namespace::name:version.
job.level	is the job's depth in the tree of the workflow DAG.
job.id	is the job's ID, as used in the DAX file.
resource.id	is a pool handle, followed by whitespace, followed by a gridftp server. Typically, each gridftp server is enumerated once, so you may have multiple occurances of the same site. There can be multiple occurances of this key.
input.lfn	is an input LFN, optionally followed by a whitespace and file size. There can be multiple occurances of this key,one for each input LFN required by the job.
wf.name	label of the dax, as found in the DAX's root element. wf.index is the DAX index, that is incremented for each partition in case of deferred planning.
wf.time	is the mtime of the workflow.
wf.manager	is the name of the workflow manager being used .e.g condor
vo.name	is the name of the virtual organization that is running this workflow. It is currently set to NONE
vo.group	unused at present and is set to NONE.

Tip

The site selector to use for site selection can be specified by setting the property **pegasus.selector.site**

Figure 5.3. Workflow Site Selection
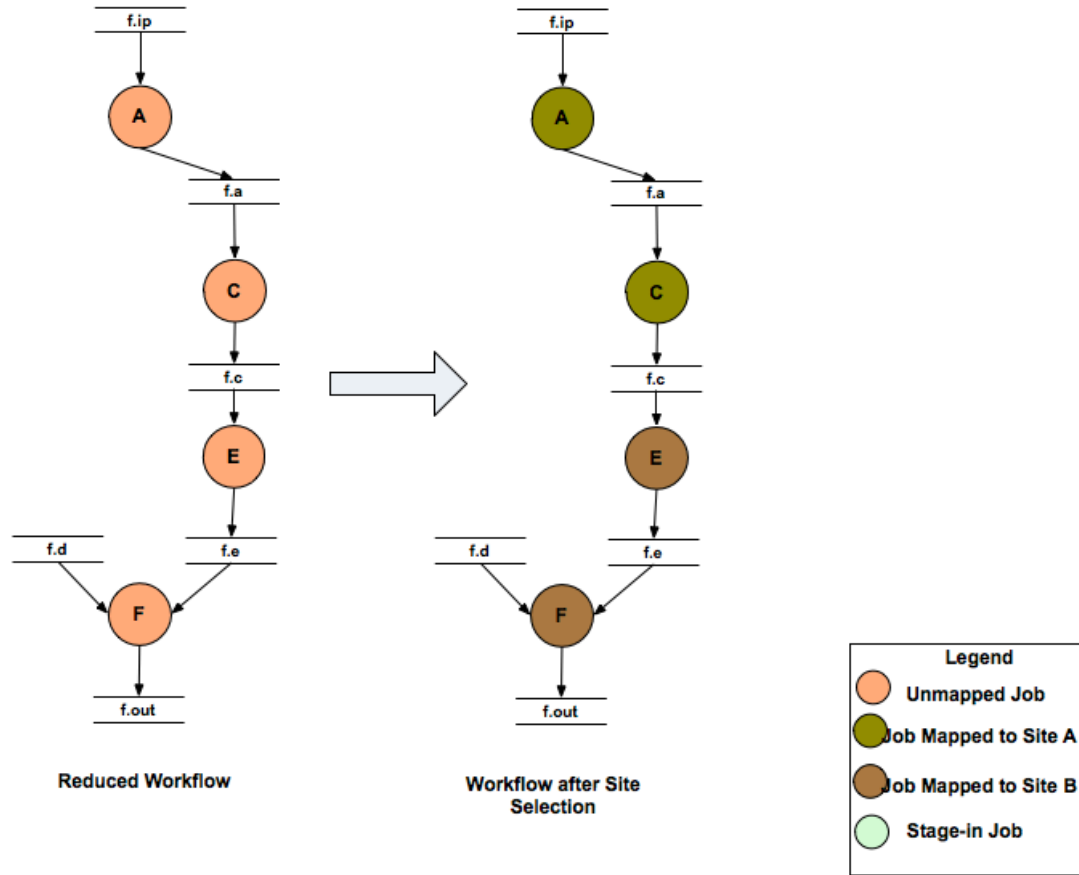
Job Clustering

After site selection, the workflow is optionally handed for to the job clustering module, which clusters jobs that are scheduled to the same site. Clustering is usually done on short running jobs in order to reduce the remote execution overheads associated with a job. Clustering is described in detail in the optimization chapter.

Tip

The job clustering is turned on by passing the **--cluster** option to pegasus-plan.

Addition of Data Transfer and Registration Nodes

After job clustering, the workflow is handed to the Data Transfer module that adds data stage-in , inter site and stage-out nodes to the workflow. Data Stage-in Nodes transfer input data required by the workflow from the locations specified in the Replica Catalog to a directory on the staging site associated with the job. The staging site for a job is the execution site if running in a sharedfs mode, else it is the one specified by **--staging-site** option to the planner. In case, multiple locations are specified for the same input file, the location from where to stage the data is selected using a **Replica Selector** . Replica Selection is described in detail in the Replica Selection section of the Data Management chapter. More details about staging site can be found in the data staging configuration chapter.

The process of adding the data stage-in and data stage-out nodes is handled by Transfer Refiners. All data transfer jobs in Pegasus are executed using **pegasus-transfer** . The pegasus-transfer client is a python based wrapper around various transfer clients like globus-url-copy, s3cmd, irods-transfer, scp, wget, cp, ln . It looks at source and destination

url and figures out automatically which underlying client to use. pegasus-transfer is distributed with the PEGASUS and can be found in the bin subdirectory . Pegasus Transfer Refiners are are described in the detail in the Transfers section of the Data Management chapter. The default transfer refiner that is used in Pegasus is the **BalancedCluster** Transfer Refiner, that clusters data stage-in nodes and data stage-out nodes per level of the workflow, on the basis of certain pegasus profile keys associated with the workflow.

Figure 5.4. Addition of Data Transfer Nodes to the Workflow
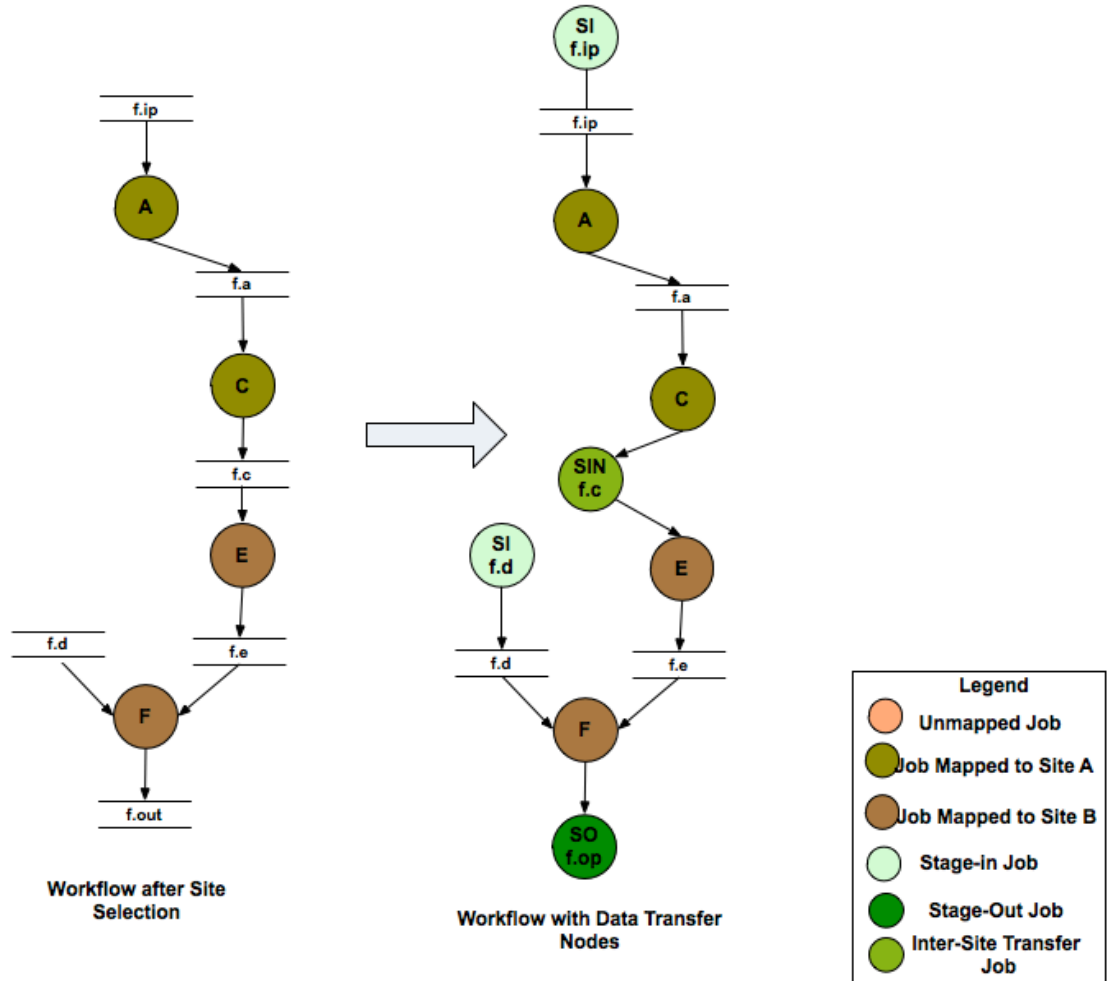
Data Registration Nodes may also be added to the final executable workflow to register the location of the output files on the final output site back in the Replica Catalog . An output file is registered in the Replica Catalog if the register flag for the file is set to true in the DAX.

Figure 5.5. Addition of Data Registration Nodes to the Workflow
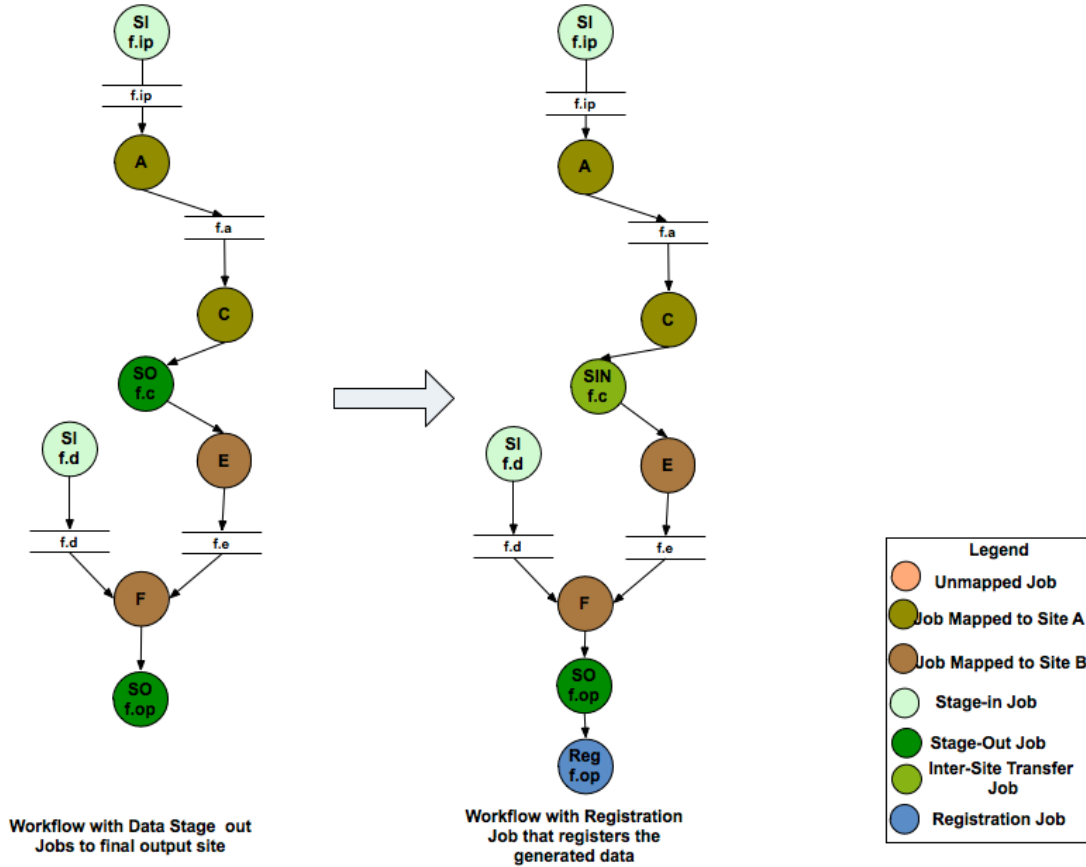
The data staged-in and staged-out from a directory that is created on the head node by a create dir job in the workflow. In the vanilla case, the directory is visible to all the worker nodes and compute jobs are launched in this directory on the shared filesystem. In the case where there is no shared filesystem, users can turn on worker node execution, where the data is staged from the head node directory to a directory on the worker node filesystem. This feature will be refined further for Pegasus 3.1. To use it with Pegasus 3.0 send email to **pegasus-support at isi.edu**.

Tip

The replica selector to use for replica selection can be specified by setting the property **pegasus.selector.replica**

Addition of Create Dir and Cleanup Jobs

After the data transfer nodes have been added to the workflow, Pegasus adds a create dir jobs to the workflow. Pegasus usually , creates one workflow specific directory per compute site , that is on the staging site associated with the job. In the case of shared shared filesystem setup, it is a directory on the shared filesystem of the compute site. In case of shared filesystem setup, this directory is visible to all the worker nodes and that is where the data is staged-in by the data stage-in jobs.

The staging site for a job is the execution site if running in a sharedfs mode, else it is the one specified by **--staging-site** option to the planner. More details about staging site can be found in the data staging configuration chapter.

After addition of the create dir jobs, the workflow is optionally handed to the cleanup module. The cleanup module adds cleanup nodes to the workflow that remove data from the directory on the shared filesystem when it is no longer required by the workflow. This is useful in reducing the peak storage requirements of the workflow.

Tip

The addition of the cleanup nodes to the workflow can be disabled by passing the **--nocleanup** option to pegasus-plan.

Figure 5.6. Addition of Directory Creation and File Removal Jobs
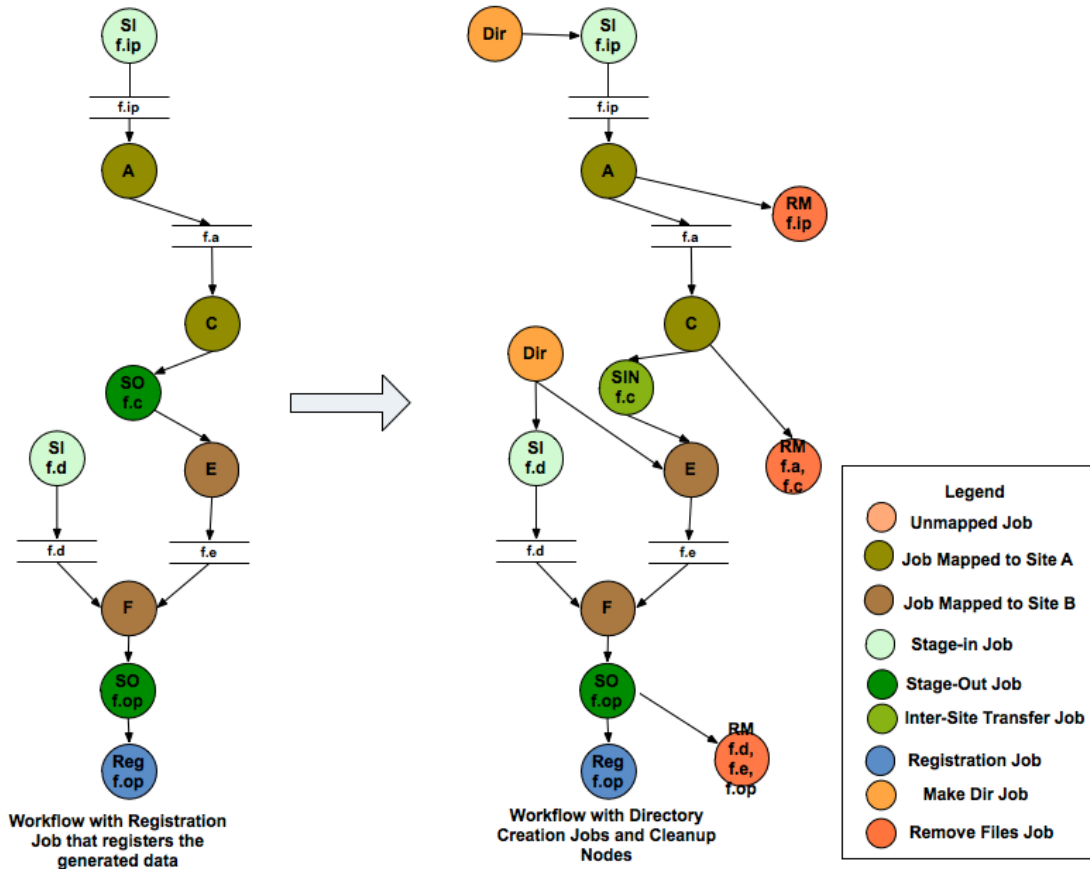
Tip

Users can specify the maximum number of cleanup jobs added per level by specifying the property **pegasus.file.cleanup.clusters.num** in the properties.

Code Generation

The last step of refinement process, is the code generation where Pegasus writes out the executable workflow in a form understandable by the underlying workflow executor. At present Pegasus supports the following code generators

1. **Condor**

   This is the default code generator for Pegasus . This generator generates the executable workflow as a Condor DAG file and associated job submit files. The Condor DAG file is passed as input to Condor DAGMan for job execution.

2. **Shell**

   This Code Generator generates the executable workflow as a shell script that can be executed on the submit host. While using this code generator, all the jobs should be mapped to site local i.e specify **--sites local** to pegasus-plan.

Tip

To use the Shell code Generator set the property **pegasus.code.generator** Shell

3. **PMC**

This Code Generator generates the executable workflow as a PMC task workflow. This is useful to run on platforms where it not feasible to run Condor such as the new XSEDE machines such as Blue Waters. In this mode, Pegasus will generate the executable workflow as a PMC task workflow and a sample PBS submit script that submits this workflow. Note that the generated PBS file needs to be manually updated before it can be submitted.

Tip

To use the Shell code Generator set the property **pegasus.code.generator** PMC

Figure 5.7. Final Executable Workflow

Data Staging Configuration

Pegasus can be broadly setup to run workflows in the following configurations

- **Shared File System**

  This setup applies to where the head node and the worker nodes of a cluster share a filesystem. Compute jobs in the workflow run in a directory on the shared filesystem.

- **NonShared FileSystem**

This setup applies to where the head node and the worker nodes of a cluster don't share a filesystem. Compute jobs in the workflow run in a local directory on the worker node

- **Condor Pool Without a shared filesystem**

This setup applies to a condor pool where the worker nodes making up a condor pool don't share a filesystem. All data IO is achieved using Condor File IO. This is a special case of the non shared filesystem setup, where instead of using pegasus-transfer to transfer input and output data, Condor File IO is used.

For the purposes of data configuration various sites, and directories are defined below.

1. **Submit Host**

   The host from where the workflows are submitted . This is where Pegasus and Condor DAGMan are installed. This is referred to as the **"local"** site in the site catalog .

2. **Compute Site**

   The site where the jobs mentioned in the DAX are executed. There needs to be an entry in the Site Catalog for every compute site. The compute site is passed to pegasus-plan using **--sites** option

3. **Staging Site**

   A site to which the separate transfer jobs in the executable workflow (jobs with stage_in , stage_out and stage_inter prefixes that Pegasus adds using the transfer refiners) stage the input data to and the output data from to transfer to the final output site. Currently, the staging site is always the compute site where the jobs execute.

4. **Output Site**

   The output site is the final storage site where the users want the output data from jobs to go to. The output site is passed to pegasus-plan using the **--output** option. The stageout jobs in the workflow stage the data from the staging site to the final storage site.

5. **Input Site**

   The site where the input data is stored. The locations of the input data are catalogued in the Replica Catalog, and the pool attribute of the locations gives us the site handle for the input site.

6. **Workflow Execution Directory**

   This is the directory created by the create dir jobs in the executable workflow on the Staging Site. This is a directory per workflow per staging site. Currently, the Staging site is always the Compute Site.

7. **Worker Node Directory**

   This is the directory created on the worker nodes per job usually by the job wrapper that launches the job.

You can specifiy the data configuration to use either in

1. properties - Specify the global property pegasus.data.configuration .

2. site catalog - Starting 4.5.0 release, you can specify pegasus profile key named data.configuration and associate that with your compute sites in the site catalog.

Shared File System

By default Pegasus is setup to run workflows in the shared file system setup, where the worker nodes and the head node of a cluster share a filesystem.

Figure 5.8. Shared File System Setup

DATA FLOW TO COMPUTE JOBS ON THE WORKER NODES RELYING ON A
SHARED FILESYSTEM

COMPUTE AND STAGING SITE ARE SAME

The data flow is as follows in this case

1. Stagein Job executes (either on Submit Host or Head Node) to stage in input data from Input Sites (1---n) to a workflow specific execution directory on the shared filesystem.

2. Compute Job starts on a worker node in the workflow execution directory. Accesses the input data using Posix IO

3. Compute Job executes on the worker node and writes out output data to workflow execution directory using Posix IO

4. Stageout Job executes (either on Submit Host or Head Node) to stage out output data from the workflow specific execution directory to a directory on the final output site.

Tip

Set **pegasus.data.configuration** to **sharedfs** to run in this configuration.

Non Shared Filesystem

In this setup , Pegasus runs workflows on local file-systems of worker nodes with the the worker nodes not sharing a filesystem. The data transfers happen between the worker node and a staging / data coordination site. The staging site server can be a file server on the head node of a cluster or can be on a separate machine.

Setup

- compute and staging site are the different

- head node and worker nodes of compute site don't share a filesystem

- Input Data is staged from remote sites.

- Remote Output Site i.e site other than compute site. Can be submit host.

Figure 5.9. Non Shared Filesystem Setup

The data flow is as follows in this case

1. Stagein Job executes (either on Submit Host or on staging site) to stage in input data from Input Sites (1---n) to a workflow specific execution directory on the staging site.

2. Compute Job starts on a worker node in a local execution directory. Accesses the input data using pegasus transfer to transfer the data from the staging site to a local directory on the worker node

3. The compute job executes in the worker node, and executes on the worker node.

4. The compute Job writes out output data to the local directory on the worker node using Posix IO

5. Output Data is pushed out to the staging site from the worker node using pegasus-transfer.

6. Stageout Job executes (either on Submit Host or staging site) to stage out output data from the workflow specific execution directory to a directory on the final output site.

In this case, the compute jobs are wrapped as PegasusLite instances.

This mode is especially useful for running in the cloud environments where you don't want to setup a shared filesystem between the worker nodes. Running in that mode is explained in detail here.

Tip

Set p**egasus.data.configuration** to **nonsharedfs** to run in this configuration. The staging site can be specified using the **--staging-site** option to pegasus-plan.

Condor Pool Without a Shared Filesystem

This setup applies to a condor pool where the worker nodes making up a condor pool don't share a filesystem. All data IO is achieved using Condor File IO. This is a special case of the non shared filesystem setup, where instead of using pegasus-transfer to transfer input and output data, Condor File IO is used.

Setup

- Submit Host and staging site are same

- head node and worker nodes of compute site don't share a filesystem

- Input Data is staged from remote sites.

- Remote Output Site i.e site other than compute site. Can be submit host.

Figure 5.10. Condor Pool Without a Shared Filesystem

The data flow is as follows in this case

1. Stagein Job executes on the submit host to stage in input data from Input Sites (1---n) to a workflow specific execution directory on the submit host

2. Compute Job starts on a worker node in a local execution directory. Before the compute job starts, Condor transfers the input data for the job from the workflow execution directory on the submit host to the local execution directory on the worker node.

3. The compute job executes in the worker node, and executes on the worker node.

4. The compute Job writes out output data to the local directory on the worker node using Posix IO

5. When the compute job finishes, Condor transfers the output data for the job from the local execution directory on the worker node to the workflow execution directory on the submit host.

6. Stageout Job executes (either on Submit Host or staging site) to stage out output data from the workflow specific execution directory to a directory on the final output site.

In this case, the compute jobs are wrapped as PegasusLite instances.

This mode is especially useful for running in the cloud environments where you don't want to setup a shared filesystem between the worker nodes. Running in that mode is explained in detail here.

Tip

Set p**egasus.data.configuration** to **condorio** to run in this configuration. In this mode, the staging site is automatically set to site **local**

PegasusLite

Starting Pegasus 4.0 , all compute jobs (single or clustered jobs) that are executed in a non shared filesystem setup, are executed using lightweight job wrapper called PegasusLite.

Figure 5.11. Workflow Running in NonShared Filesystem Setup with PegasusLite launching compute jobs
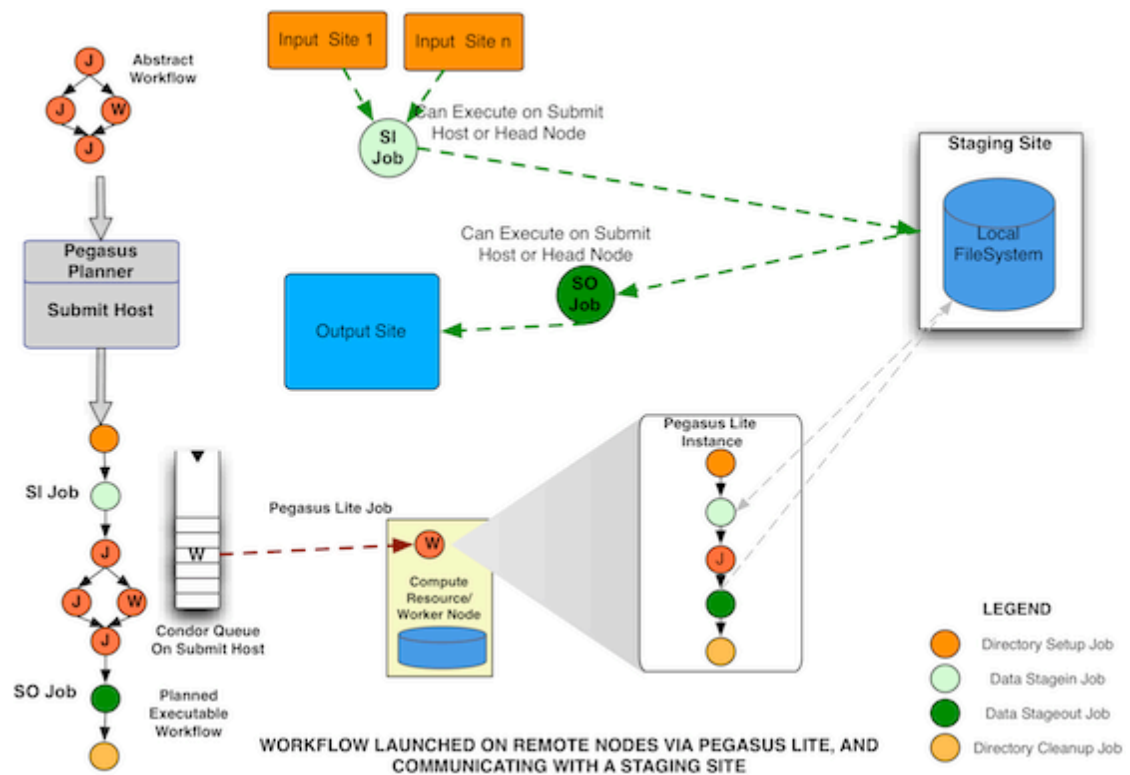
When PegasusLite starts on a remote worker node to run a compute job , it performs the following actions:

1. Discovers the best run-time directory based on space requirements and create the directory on the local filesystem of the worker node to execute the job.

2. Prepare the node for executing the unit of work. This involves discovering whether the pegasus worker tools are already installed on the node or need to be brought in.

3. Use pegasus-transfer to stage in the input data to the runtime directory (created in step 1) on the remote worker node.

4. Launch the compute job.

5. Use pegasus-transfer to stage out the output data to the data coordination site.

6. Remove the directory created in Step 1.

Pegasus-Plan

pegasus-plan is the main executable that takes in the abstract workflow (DAX) and generates an executable workflow (usually a Condor DAG) by querying various catalogs and performing several refinement steps. Before users can run pegasus plan the following needs to be done:

1. Populate the various catalogs

    a. **Replica Catalog**

    The Replica Catalog needs to be catalogued with the locations of the input files required by the workflows. This can be done by using pegasus-rc-client (See the Replica section of Creating Workflows).

    b. **Transformation Catalog**

    The Transformation Catalog needs to be catalogued with the locations of the executables that the workflows will use. This can be done by using pegasus-tc-client (See the Transformation section of Creating Workflows).

    c. **Site Catalog**

    The Site Catalog needs to be catalogued with the site layout of the various sites that the workflows can execute on. A site catalog can be generated for OSG by using the client pegasus-sc-client (See the Site section of the Creating Workflows).

2. Configure Properties

    After the catalogs have been configured, the user properties file need to be updated with the types and locations of the catalogs to use. These properties are described in the **basic.properties** files in the **etc** sub directory (see the Properties section of the Configuration chapter.

    The basic properties that need to be set usually are listed below:

    Table 5.2. Basic Properties that need to be set

pegasus.catalog.replica
pegasus.catalog.replica.file | pegasus.catalog.replica.url
pegasus.catalog.transformation
pegasus.catalog.transformation.file
pegasus.catalog.site.file

To execute pegasus-plan user usually requires to specify the following options:

1. **--dax** the path to the DAX file that needs to be mapped.

2. **--dir** the base directory where the executable workflow is generated

3. **--sites** comma separated list of execution sites.

4. **--output** the output site where to transfer the materialized output files.

5. **--submit** boolean value whether to submit the planned workflow for execution after planning is done.

Basic Properties

Properties are primarily used to configure the behavior of the Pegasus Workflow Planner at a global level. The properties file is actually a java properties file and follows the same conventions as that to specify the properties.

Please note that the values rely on proper capitalization, unless explicitly noted otherwise.

Some properties rely with their default on the value of other properties. As a notation, the curly braces refer to the value of the named property. For instance, ${pegasus.home} means that the value depends on the value of the pegasus.home property plus any noted additions. You can use this notation to refer to other properties, though the extent of the subsitutions are limited. Usually, you want to refer to a set of the standard system properties. Nesting is not allowed. Substitutions will only be done once.

There is a priority to the order of reading and evaluating properties. Usually one does not need to worry about the priorities. However, it is good to know the details of when which property applies, and how one property is able to overwrite another. The following is a mutually exclusive list (highest priority first) of property file locations.

1. --conf option to the tools. Almost all of the clients that use properties have a --conf option to specify the property file to pick up.

2. submit-dir/pegasus.xxxxxxx.properties file. All tools that work on the submit directory (i.e after pegasus has planned a workflow) pick up the pegasus.xxxxx.properties file from the submit directory. The location for the pegasus.xxxxxxx.propertiesis picked up from the braindump file.

3. The properties defined in the user property file *${user.home}/.pegasusrc* have lowest priority.

Commandline properties have the highest priority. These override any property loaded from a property file. Each commandline property is introduced by a -D argument. Note that these arguments are parsed by the shell wrapper, and thus the -D arguments must be the first arguments to any command. Commandline properties are useful for debugging purposes.

From Pegasus 3.1 release onwards, support has been dropped for the following properties that were used to signify the location of the properties file

- pegasus.properties

- pegasus.user.properties

The following example provides a sensible set of properties to be set by the user property file. These properties use mostly non-default settings. It is an example only, and will not work for you:

```
pegasus.catalog.replica              File
pegasus.catalog.replica.file         ${pegasus.home}/etc/sample.rc.data
pegasus.catalog.transformation       Text
pegasus.catalog.transformation.file  ${pegasus.home}/etc/sample.tc.text
pegasus.catalog.site.file            ${pegasus.home}/etc/sample.sites.xml
```

If you are in doubt which properties are actually visible, pegasus during the planning of the workflow dumps all properties after reading and prioritizing in the submit directory in a file with the suffix properties.

pegasus.home

Systems:	all
Type:	directory location string
Default:	"$PEGASUS_HOME"

The property pegasus.home cannot be set in the property file. This property is automatically set up by the pegasus clients internally by determining the installation directory of pegasus. Knowledge about this property is important for developers who want to invoke PEGASUS JAVA classes without the shell wrappers.

Catalog Related Properties

Table 5.3. Replica Catalog Properties

Key Attributes	Description
Property Key: pegasus.catalog.replica **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Default** : File	Pegasus queries a Replica Catalog to discover the physical filenames (PFN) for input files specified in the DAX. Pegasus can interface with various types of Replica Catalogs. This property specifies which type of Replica Catalog to use during the planning process.
	JDBCRC In this mode, Pegasus queries a SQL based replica catalog that is accessed via JDBC. The sql schema's for this catalog can be found at $PEGASUS_HOME/sql directory. To use JDBCRC, the user additionally needs to set the following properties 1. pegasus.catalog.replica.db.driver = mysql 2. pegasus.catalog.replica.db.url = jdbc url to database e.g jdbc:mysql://database-host.isi.edu/database-name 3. pegasus.catalog.replica.db.user = database-user 4. pegasus.catalog.replica.db.password = database-password
	File In this mode, Pegasus queries a file based replica catalog. It is neither transactionally safe, nor advised to use for production purposes in any way. Multiple concurrent instances *will clobber* each other!. The site attribute should be specified whenever possible. The attribute key for the site attribute is "site". The LFN may or may not be quoted. If it contains linear whitespace, quotes, backslash or an equality sign, it must be quoted and escaped. Ditto for the PFN. The attribute key-value pairs are separated by an equality sign without any whitespaces. The value may be in quoted. The LFN sentiments about quoting apply. ``` LFN PFN LFN PFN a=b [..] LFN PFN a="b" [..] "LFN w/LWS" "PFN w/LWS" [..] ``` To use File, the user additionally needs to specify pegasus.catalog.replica.file property to specify the path to the file based RC.
	Regex In this mode, Pegasus queries a file based replica catalog. It is neither transactionally safe, nor advised to use for production

purposes in any way. Multiple concurrent access to the File will end up clobbering the contents of the file. The site attribute should be specified whenever possible. The attribute key for the site attribute is "site".

The LFN may or may not be quoted. If it contains linear whitespace, quotes, backslash or an equality sign, it must be quoted and escaped. Ditto for the PFN. The attribute key-value pairs are separated by an equality sign without any whitespaces. The value may be in quoted. The LFN sentiments about quoting apply.

In addition users can specifiy regular expression based LFN's. A regular expression based entry should be qualified with an attribute named 'regex'. The attribute regex when set to true identifies the catalog entry as a regular expression based entry. Regular expressions should follow Java regular expression syntax.

For example, consider a replica catalog as shown below.

Entry 1 refers to an entry which does not use a resular expressions. This entry would only match a file named 'f.a', and nothing else. Entry 2 referes to an entry which uses a regular expression. In this entry f.a referes to files having name as f[any-character]a i.e. faa, f.a, f0a, etc.

```
f.a file:///Vol/input/f.a
 site="local"
f.a file:///Vol/input/f.a
 site="local" regex="true"
```

Regular expression based entries also support substitutions. For example, consider the regular expression based entry shown below.

Entry 3 will match files with name alpha.csv, alpha.txt, alpha.xml. In addition, values matched in the expression can be used to generate a PFN.

For the entry below if the file being looked up is alpha.csv, the PFN for the file would be generated as file:///Volumes/data/input/csv/alpha.csv. Similary if the file being lookedup was alpha.csv, the PFN for the file would be generated as file:///Volumes/data/input/xml/alpha.xml i.e. The section [0], [1] will be replaced. Section [0] refers to the entire string i.e. alpha.csv. Section [1]

refers to a partial match in the input i.e. csv, or txt, or xml. Users can utilize as many sections as they wish.

```
alpha\.(csv|txt|xml) file:///
Vol/input/[1]/[0] site="local"
 regex="true"
```

To use File, the user additionally needs to specify pegasus.catalog.replica.file property to specify the path to the file based RC.

Directory

In this mode, Pegasus does a directory listing on an input directory to create the LFN to PFN mappings. The directory listing is performed recursively, resulting in deep LFN mappings. For example, if an input directory $input is specified with the following structure

```
$input
$input/f.1
$input/f.2
$input/D1
$input/D1/f.3
```

Pegasus will create the mappings the following LFN PFN mappings internally

```
f.1 file://$input/f.1  site="local"
f.2 file://$input/f.2  site="local"
D1/f.3 file://$input/D2/f.3
 site="local"
```

If you don't want the deep lfn's to be created then, you can set pegasus.catalog.replica.directory.flat.lfn to true In that case, for the previous example, Pegasus will create the following LFN PFN mappings internally.

```
f.1 file://$input/f.1  site="local"
f.2 file://$input/f.2  site="local"
f.3 file://$input/D2/f.3
 site="local"
```

pegasus-plan has --input-dir option that can be used to specify an input directory.

Users can optionally specify additional properties to configure the behvavior of this implementation.

pegasus.catalog.replica.directory.site   to specify a site attribute other than local to associate with the mappings.

pegasus.catalog.replica.directory.url.prefix to associate a URL prefix for the PFN's constructed. If not specified, the URL defaults to file://

	MRC In this mode, Pegasus queries multiple replica catalogs to discover the file locations on the grid. To use it set
	
	`pegasus.catalog.replica MRC`
	
	Each associated replica catalog can be configured via properties as follows.
	
	The user associates a variable name referred to as [value] for each of the catalogs, where [value] is any legal identifier (concretely [A-Za-z][_A-Za-z0-9]*) For each associated replica catalogs the user specifies the following properties.
	
	`pegasus.catalog.replica.mrc.[value]`
	`        specifies the type of \`
	
	`    replica catalog.`
	`pegasus.catalog.replica.mrc.`
	`[value].key   specifies a property`
	` name\`
	
	`        key for a particular catalog`
	
	`pegasus.catalog.replica.mrc.directory1`
	` LRC`
	`pegasus.catalog.replica.mrc.directory1.url /`
	`input/dir1`
	`pegasus.catalog.replica.mrc.directory2`
	` LRC`
	`pegasus.catalog.replica.mrc.directory2.url /`
	`input/dir2`
	
	In the above example, directory1, directory2 are any valid identifier names and url is the property key that needed to be specified.

Property Key: pegasus.catalog.replica.url **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Default** : (no default)	When using the modern RLS replica catalog, the URI to the Replica catalog must be provided to Pegasus to enable it to look up filenames. There is no default.

Table 5.4. Site Catalog Properties

Key Attributes	Description
Property Key: pegasus.catalog.site **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Default** : XML	Pegasus supports two different types of site catalogs in XML format conforming to sc-3.0.xsd and sc-4.0.xsd. Pegasus is able to auto-detect what schema a user site catalog refers to. Hence, this property may no longer be set.
Property Key: pegasus.catalog.site.file **Profile Key :** N/A **Scope** : Properties **Since** : 2.0 **Default** : ${pegasus.home.sysconfdir}/sites.xml	The path to the site catalog file, that describes the various sites and their layouts to Pegasus.

Table 5.5. Transformation Catalog Properties

Key Attributes	Description

Property Key: pegasus.catalog.transformation
Profile Key: N/A
Scope　　: Properties
Since　　: 2.0
Default　　: Text
	The only recommended and supported version of Transformation Catalog for Pegasus is Text. For the old File based formats, users should use pegasus-tc-converter to convert File format to Text Format.
	
	Text　　In this mode, a multiline file based format is understood. The file is read and cached in memory. Any modifications, as adding or deleting, causes an update of the memory and hence to the file underneath. All queries are done against the memory representation.
	
	　　The file sample.tc.text in the etc directory contains an example
	
	　　Here is a sample textual format for transfomation catalog containing one transformation on two sites
	
	<pre>tr example::keg:1.0 {
	#specify profiles that apply for all the
	 sites for the transformation
	#in each site entry the profile can be
	 overriden
	profile env "APP_HOME" "/tmp/karan"
	profile env "JAVA_HOME" "/bin/app"
	site isi {
	profile env "me" "with"
	profile condor "more" "test"
	profile env "JAVA_HOME" "/bin/java.1.6"
	pfn "/path/to/keg"
	arch "x86"
	os "linux"
	osrelease "fc"
	osversion "4"
	type "INSTALLED"
	site wind {
	profile env "me" "with"
	profile condor "more" "test"
	pfn "/path/to/keg"
	arch "x86"
	os "linux"
	osrelease "fc"
	osversion "4"
	type "STAGEABLE"</pre>

Property Key: pegasus.catalog.transformation **Profile Key :** N/A **Scope**　　: Properties **Since**　　: 2.0 **Default**　　: ${pegasus.home.sysconfdir}/tc.text	The path to the transformation catalog file, that describes the locations of the executables.

Data Staging Configuration Properties

Table 5.6. Data Configuration Properties

Key Attributes	Description
Property Key: pegasus.data.configuration **Profile Key:** N/A **Scope**　　: Properties **Since**　　: 4.0.0 **Values**　　: sharedfs|nonsharedfs|condorio	This property sets up Pegasus to run in different environments. sharedfs　　　　If this is set, Pegasus will be setup to execute jobs on the shared filesystem

		on the execution site. This assumes, that the head node of a cluster and the worker nodes share a filesystem. The staging site in this case is the same as the execution site. Pegasus adds a create dir job to the executable work-flow that creates a workflow specif-ic directory on the shared filesystem . The data transfer jobs in the executable workflow (stage_in_ , stage_inter_ , stage_out_) transfer the data to this di-rectory.The compute jobs in the exe-cutable workflow are launched in the directory on the shared filesystem. In-ternally, if this is set the following properties are set.
Default : sharedfs **See Also** : pegasus.transfer.bypass.input.staging		

```
pegasus.execute.*.filesystem.local
     false
```

	condorio	If this is set, Pegasus will be setup to run jobs in a pure condor pool, with the nodes not sharing a filesystem. Data is staged to the compute nodes from the submit host using Condor File IO. The planner is automatically setup to use the submit host (site local) as the stag-ing site. All the auxillary jobs added by the planner to the executable workflow (create dir, data stagein and stage-out, cleanup) jobs refer to the workflow specific directory on the local site. The data transfer jobs in the executable workflow (stage_in_ , stage_inter_ , stage_out_) transfer the data to this di-rectory. When the compute jobs start, the input data for each job is shipped from the workflow specific directory on the submit host to compute/worker node using Condor file IO. The output data for each job is similarly shipped back to the submit host from the com-pute/worker node. This setup is par-ticularly helpful when running work-flows in the cloud environment where setting up a shared filesystem across the VM's may be tricky. On loading this property, internally the following properies are set

```
pegasus.transfer.lite.*.impl
     Condor
pegasus.execute.*.filesystem.local
   true
pegasus.gridstart
 PegasusLite
pegasus.transfer.worker.package
    true
```

	nonsharedfs	If this is set, Pegasus will be setup to execute jobs on an execution site with-

out relying on a shared filesystem between the head node and the worker nodes. You can specify staging site (using --staging-site option to pegasus-plan) to indicate the site to use as a central storage location for a workflow. The staging site is independant of the execution sites on which a workflow executes. All the auxillary jobs added by the planner to the executable workflow (create dir, data stagein and stage-out, cleanup) jobs refer to the workflow specific directory on the staging site. The data transfer jobs in the executable workflow (stage_in_ , stage_inter_ , stage_out_) transfer the data to this directory. When the compute jobs start, the input data for each job is shipped from the workflow specific directory on the submit host to compute/worker node using pegasus-transfer. The output data for each job is similarly shipped back to the submit host from the compute/worker node. The protocols supported are at this time SRM, GridFTP, iRods, S3. This setup is particularly helpful when running workflows on OSG where most of the execution sites don't have enough data storage. Only a few sites have large amounts of data storage exposed that can be used to place data during a workflow run. This setup is also helpful when running workflows in the cloud environment where setting up a shared filesystem across the VM's may be tricky. On loading this property, internally the following properies are set

```
pegasus.execute.*.filesystem.local
   true
pegasus.gridstart
 PegasusLite
pegasus.transfer.worker.package
    true
```

Chapter 6. Monitoring, Debugging and Statistics

Pegasus comes bundled with useful tools that help users debug workflows and generate useful statistics and plots about their workflow runs. Most of the tools query a runtime workflow database (usually a sqllite in the workflow submit directory) populated at runtime by pegasus-monitord. With the exception of pegasus-monitord (see below), all tools take in the submit directory as an argument. Users can invoke the tools listed in this chapter as follows:

```
$ pegasus-[toolname]   <path to the submit directory>
```

Workflow Status

As the number of jobs and tasks in workflows increase, the ability to track the progress and quickly debug a workflow becomes more and more important. Pegasus comes with a series of utilities that can be used to monitor and debug workflows both in real-time as well as after execution is already completed.

pegasus-status

To monitor the execution of the workflow run the **pegasus-status** command as suggested by the output of the **pegasus-run** command. **pegasus-status** shows the current status of the Condor Q as pertaining to the master workflow from the workflow directory you are pointing it to. In a second section, it will show a summary of the state of all jobs in the workflow and all of its sub-workflows.

The details of **pegasus-status** are described in its respective manual page. There are many options to help you gather the most out of this tool, including a watch-mode to repeatedly draw information, various modes to add more information, and legends if you are new to it, or need to present it.

```
$ pegasus-status /Workflow/dags/directory
STAT   IN_STATE   JOB
Run       05:08   level-3-0
Run       04:32    |-sleep_ID000005
Run       04:27    \_subdax_level-2_ID000004
Run       03:51        |-sleep_ID000003
Run       03:46        \_subdax_level-1_ID000002
Run       03:10            \_sleep_ID000001
Summary: 6 Condor jobs total (R:6)

UNREADY    READY      PRE   QUEUED    POST SUCCESS FAILURE %DONE
     0        0        0        6       0       3       0  33.3
Summary: 3 DAGs total (Running:3)
```

Without the $-l$ option, the only a summary of the workflow statistics is shown under the current queue status. However, with the $-l$ option, it will show each sub-workflow separately:

```
$ pegasus-status -l /Workflow/dags/directory
STAT   IN_STATE   JOB
Run       07:01   level-3-0
Run       06:25    |-sleep_ID000005
Run       06:20    \_subdax_level-2_ID000004
Run       05:44        |-sleep_ID000003
Run       05:39        \_subdax_level-1_ID000002
Run       05:03            \_sleep_ID000001
Summary: 6 Condor jobs total (R:6)

UNRDY READY     PRE   IN_Q   POST   DONE   FAIL %DONE STATE   DAGNAME
    0     0       0      1      0      1      0  50.0 Running level-2_ID000004/level-1_ID000002/
level-1-0.dag
    0     0       0      2      0      1      0  33.3 Running level-2_ID000004/level-2-0.dag
    0     0       0      3      0      1      0  25.0 Running *level-3-0.dag
    0     0       0      6      0      3      0  33.3         TOTALS (9 jobs)
Summary: 3 DAGs total (Running:3)
```

The following output shows a successful workflow of workflow summary after it has finished.

```
$ pegasus-status work/2011080514
```

```
(no matching jobs found in Condor Q)
UNREADY    READY     PRE   QUEUED     POST SUCCESS FAILURE %DONE
     0        0       0       0        0   7,137       0 100.0
Summary: 44 DAGs total (Success:44)
```

Warning

For large workflows with many jobs, please note that **pegasus-status** will take time to compile state from all workflow files. This typically affects the initial run, and sub-sequent runs are faster due to the file system's buffer cache. However, on a low-RAM machine, thrashing is a possibility.

The following output show a failed workflow after no more jobs from it exist. Please note how no active jobs are shown, and the failure status of the total workflow.

```
$ pegasus-status work/submit
(no matching jobs found in Condor Q)
UNREADY    READY     PRE   QUEUED     POST SUCCESS FAILURE %DONE
    20        0       0       0        0       0       2   0.0
Summary: 1 DAG total (Failure:1)
```

pegasus-analyzer

Pegasus-analyzer is a command-line utility for parsing several files in the workflow directory and summarizing useful information to the user. It should be used after the workflow has already finished execution. pegasus-analyzer quickly goes through the jobstate.log file, and isolates jobs that did not complete successfully. It then parses their submit, and kickstart output files, printing to the user detailed information for helping the user debug what happened to his/ her workflow.

The simplest way to invoke pegasus-analyzer is to simply give it a workflow run directory, like in the example below:

```
$ pegasus-analyzer  /home/user/run0004
pegasus-analyzer: initializing...

**********************************Summary**************************************

 Total jobs         :     26 (100.00%)
 # jobs succeeded   :     25 (96.15%)
 # jobs failed      :      1 (3.84%)
 # jobs held        :      1 (3.84%)
 # jobs unsubmitted :      0 (0.00%)

******************************Held jobs' details*******************************

==============================sleep_ID0000001==============================

      submit file         : sleep_ID0000001.sub
      last_job_instance_id : 7
      reason               :  Error from slot1@corbusier.isi.edu:
                               STARTER at 128.9.64.188 failed to
                               send file(s) to
                               <128.9.64.188:62639>: error reading from
                               /opt/condor/8.4.8/local.corbusier/execute/dir_76205/f.out:
                               (errno 2) No such file or directory;
                               SHADOW failed to receive file(s) from <128.9.64.188:62653>

*****************************Failed jobs' details******************************

===========================register_viz_glidein_7_0===========================

 last state: POST_SCRIPT_FAILURE
       site: local
submit file: /home/user/run0004/register_viz_glidein_7_0.sub
output file: /home/user/run0004/register_viz_glidein_7_0.out.002
 error file: /home/user/run0004/register_viz_glidein_7_0.err.002

------------------------------Task #1 - Summary-------------------------------

site       : local
executable : /lfs1/software/install/pegasus/default/bin/rc-client
arguments  : -Dpegasus.user.properties=/lfs1/work/pegasus/run0004/pegasus.15181.properties \
-Dpegasus.catalog.replica.url=rlsn://smarty.isi.edu --insert register_viz_glidein_7_0.in
exitcode   : 1
```

```
working dir : /lfs1/work/pegasus/run0004

---------Task #1 - pegasus::rc-client - pegasus::rc-client:1.0 - stdout---------

2009-02-20 16:25:13.467 ERROR [root] You need to specify the pegasus.catalog.replica property
2009-02-20 16:25:13.468 WARN  [root] non-zero exit-code 1
```

In the case above, pegasus-analyzer's output contains a brief summary section, showing how many jobs have succeeded and how many have failed. If there are any held jobs, pegasus-analyzer will report the name of the job that was held, and the reason why , as determined from the dagman.out file for the workflow. The last_job_instance_id is the database id for the job in the job instance table of the monitoring database. After that, pegasus-analyzer will print information about each job that failed, showing its last known state, along with the location of its submit, output, and error files. pegasus-analyzer will also display any stdout and stderr from the job, as recorded in its kickstart record. Please consult pegasus-analyzer's man page for more examples and a detailed description of its various command-line options.

Note

Starting with 4.0 release, by default pegasus analyzer queries the database to debug the workflow. If you want it to use files in the submit directory , use the **--files** option.

pegasus-remove

If you want to abort your workflow for any reason you can use the pegasus-remove command listed in the output of pegasus-run invocation or by specifying the Dag directory for the workflow you want to terminate.

$ pegasus-remove /PATH/To/WORKFLOW DIRECTORY

Resubmitting failed workflows

Pegasus will remove the DAGMan and all the jobs related to the DAGMan from the condor queue. A rescue DAG will be generated in case you want to resubmit the same workflow and continue execution from where it last stopped. A rescue DAG only skips jobs that have completely finished. It does not continue a partially running job unless the executable supports checkpointing.

To resubmit an aborted or failed workflow with the same submit files and rescue Dag just rerun the pegasus-run command

$ pegasus-run /Path/To/Workflow/Directory

Plotting and Statistics

Pegasus plotting and statistics tools queries the Stampede database created by pegasus-monitord for generating the output.The stampede scheme can be found here.

The statistics and plotting tools use the following terminology for defining tasks, jobs etc. Pegasus takes in a DAX which is composed of tasks. Pegasus plans it into a Condor DAG / Executable workflow that consists of Jobs. In case of Clustering, multiple tasks in the DAX can be captured into a single job in the Executable workflow. When DAGMan executes a job, a job instance is populated . Job instances capture information as seen by DAGMan. In case DAGMan retires a job on detecting a failure , a new job instance is populated. When DAGMan finds a job instance has finished , an invocation is associated with job instance. In case of clustered job, multiple invocations will be associated with a single job instance. If a Pre script or Post Script is associated with a job instance, then invocations are populated in the database for the corresponding job instance.

pegasus-statistics

Pegasus statistics can compute statistics over one or more than one workflow run.

Command to generate statistics over a single run is as shown below.

$ **pegasus-statistics /scratch/grid-setup/run0001/ -s all**

```
#
# Pegasus Workflow Management System - http://pegasus.isi.edu
#
# Workflow summary:
#   Summary of the workflow execution. It shows total
#   tasks/jobs/sub workflows run, how many succeeded/failed etc.
#   In case of hierarchical workflow the calculation shows the
#   statistics across all the sub workflows.It shows the following
#   statistics about tasks, jobs and sub workflows.
#     * Succeeded - total count of succeeded tasks/jobs/sub workflows.
#     * Failed - total count of failed tasks/jobs/sub workflows.
#     * Incomplete - total count of tasks/jobs/sub workflows that are
#       not in succeeded or failed state. This includes all the jobs
#       that are not submitted, submitted but not completed etc. This
#       is calculated as  difference between 'total' count and sum of
#       'succeeded' and 'failed' count.
#     * Total - total count of tasks/jobs/sub workflows.
#     * Retries - total retry count of tasks/jobs/sub workflows.
#     * Total+Retries - total count of tasks/jobs/sub workflows executed
#       during workflow run. This is the cumulative of retries,
#       succeeded and failed count.
# Workflow wall time:
#   The wall time from the start of the workflow execution to the end as
#   reported by the DAGMAN.In case of rescue dag the value is the
#   cumulative of all retries.
# Workflow cumulative job wall time:
#   The sum of the wall time of all jobs as reported by kickstart.
#   In case of job retries the value is the cumulative of all retries.
#   For workflows having sub workflow jobs (i.e SUBDAG and SUBDAX jobs),
#   the wall time value includes jobs from the sub workflows as well.
# Cumulative job wall time as seen from submit side:
#   The sum of the wall time of all jobs as reported by DAGMan.
#   This is similar to the regular cumulative job wall time, but includes
#   job management overhead and delays. In case of job retries the value
#   is the cumulative of all retries. For workflows having sub workflow
#   jobs (i.e SUBDAG and SUBDAX jobs), the wall time value includes jobs
#   from the sub workflows as well.
# Cumulative job wall time as seen from submit side:
#   The sum of the wall time of all jobs as reported by DAGMan.
#   This is similar to the regular cumulative job wall time, but includes
#   job management overhead and delays. In case of job retries the value
#   is the cumulative of all retries. For workflows having sub workflow
#   jobs (i.e SUBDAG and SUBDAX jobs), the wall time value includes jobs
#   from the sub workflows as well.
# Workflow cumulative job badput wall time:
#   The sum of the wall time of all failed jobs as reported by kickstart.
#   In case of job retries the value is the cumulative of all retries.
#   For workflows having sub workflow jobs (i.e SUBDAG and SUBDAX jobs),
#   the wall time value includes jobs from the sub workflows as well.
# Cumulative job badput wall time as seen from submit side:
#   The sum of the wall time of all failed jobs as reported by DAGMan.
#   This is similar to the regular cumulative job badput wall time, but includes
#   job management overhead and delays. In case of job retries the value
#   is the cumulative of all retries. For workflows having sub workflow
#   jobs (i.e SUBDAG and SUBDAX jobs), the wall time value includes jobs
#   from the sub workflows as well.

------------------------------------------------------------------------
Type            Succeeded Failed  Incomplete  Total    Retries  Total+Retries
Tasks           4         0       0           4        0        4
Jobs            17        0       0           17       0        17
Sub-Workflows   0         0       0           0        0        0
------------------------------------------------------------------------

Workflow wall time                                    : 5 mins, 18 secs
Workflow cumulative job wall time                     : 4 mins, 2 secs
Cumulative job wall time as seen from submit side     : 4 mins, 10 secs
Workflow cumulative job badput wall time              : 0
Cumulative job badput wall time as seen from submit side : 0
```

By default the output gets generated to a statistics folder inside the submit directory. The output that is generated by pegasus-statistics is based on the value set for command line option 's'(statistics_level). In the sample run the command line option 's' is set to 'all' to generate all the statistics information for the workflow run. Please consult the pegasus-statistics man page to find a detailed description of various command line options.

Note

In case of hierarchal workflows, the metrics that are displayed on stdout take into account all the jobs/tasks/ sub workflows that make up the workflow by recursively iterating through each sub workflow.

Command to generate statistics over all workflow runs populated in a single database is as shown below.

```
$ pegasus-statistics -Dpegasus.monitord.output='mysql://s_user:s_user123@127.0.0.1:3306/stampede' -
o /scratch/workflow_1_2/statistics -s all --multiple-wf
```

```
#
# Pegasus Workflow Management System - http://pegasus.isi.edu
#
# Workflow summary:
#   Summary of the workflow execution. It shows total
#   tasks/jobs/sub workflows run, how many succeeded/failed etc.
#   In case of hierarchical workflow the calculation shows the
#   statistics across all the sub workflows.It shows the following
#   statistics about tasks, jobs and sub workflows.
#     * Succeeded - total count of succeeded tasks/jobs/sub workflows.
#     * Failed - total count of failed tasks/jobs/sub workflows.
#     * Incomplete - total count of tasks/jobs/sub workflows that are
#       not in succeeded or failed state. This includes all the jobs
#       that are not submitted, submitted but not completed etc. This
#       is calculated as  difference between 'total' count and sum of
#       'succeeded' and 'failed' count.
#     * Total - total count of tasks/jobs/sub workflows.
#     * Retries - total retry count of tasks/jobs/sub workflows.
#     * Total+Retries - total count of tasks/jobs/sub workflows executed
#       during workflow run. This is the cumulative of retries,
#       succeeded and failed count.
# Workflow wall time:
#   The wall time from the start of the workflow execution to the end as
#   reported by the DAGMAN.In case of rescue dag the value is the
#   cumulative of all retries.
# Workflow cumulative job wall time:
#   The sum of the wall time of all jobs as reported by kickstart.
#   In case of job retries the value is the cumulative of all retries.
#   For workflows having sub workflow jobs (i.e SUBDAG and SUBDAX jobs),
#   the wall time value includes jobs from the sub workflows as well.
# Cumulative job wall time as seen from submit side:
#   The sum of the wall time of all jobs as reported by DAGMan.
#   This is similar to the regular cumulative job wall time, but includes
#   job management overhead and delays. In case of job retries the value
#   is the cumulative of all retries. For workflows having sub workflow
#   jobs (i.e SUBDAG and SUBDAX jobs), the wall time value includes jobs
#   from the sub workflows as well.
# Workflow cumulative job badput wall time:
#   The sum of the wall time of all failed jobs as reported by kickstart.
#   In case of job retries the value is the cumulative of all retries.
#   For workflows having sub workflow jobs (i.e SUBDAG and SUBDAX jobs),
#   the wall time value includes jobs from the sub workflows as well.
# Cumulative job badput wall time as seen from submit side:
#   The sum of the wall time of all failed jobs as reported by DAGMan.
#   This is similar to the regular cumulative job badput wall time, but includes
#   job management overhead and delays. In case of job retries the value
#   is the cumulative of all retries. For workflows having sub workflow
#   jobs (i.e SUBDAG and SUBDAX jobs), the wall time value includes jobs
#   from the sub workflows as well.


------------------------------------------------------------------------
Type          Succeeded Failed  Incomplete  Total    Retries  Total+Retries
Tasks         8         0       0           8        0        8
Jobs          34        0       0           34       0        34
Sub-Workflows 0         0       0           0        0        0
------------------------------------------------------------------------

Workflow cumulative job wall time                     : 8 mins, 5 secs
Cumulative job wall time as seen from submit side     : 8 mins, 35 secs
Workflow cumulative job badput wall time              : 0
Cumulative job badput wall time as seen from submit side : 0
```

Note

When computing statistics over multiple workflows, please note,

1. All workflow run information should be populated in a single STAMPEDE database.

2. The --output argument must be specified.

3. Job statistics information is not computed.

4. Workflow wall time information is not computed.

Pegasus statistics can also compute statistics over a few specified workflow runs, by specifying the either the submit directories, or the workflow UUIDs.

```
pegasus-statistics -Dpegasus.monitord.output='<DB_URL>' -o <OUTPUT_DIR> <SUBMIT_DIR_1>
 <SUBMIT_DIR_2> .. <SUBMIT_DIR_n>

OR

pegasus-statistics -Dpegasus.monitord.output='<DB_URL>' -o <OUTPUT_DIR> --isuuid <UUID_1>
 <UUID_2> .. <UUID_n>
```

pegasus-statistics summary which is printed on the stdout contains the following information.

- **Workflow summary** - Summary of the workflow execution. In case of hierarchical workflow the calculation shows the statistics across all the sub workflows.It shows the following statistics about tasks, jobs and sub workflows.

  - **Succeeded** - total count of succeeded tasks/jobs/sub workflows.

  - **Failed** - total count of failed tasks/jobs/sub workflows.

  - **Incomplete** - total count of tasks/jobs/sub workflows that are not in succeeded or failed state. This includes all the jobs that are not submitted, submitted but not completed etc. This is calculated as difference between 'total' count and sum of 'succeeded' and 'failed' count.

  - **Total** - total count of tasks/jobs/sub workflows.

  - **Retries** - total retry count of tasks/jobs/sub workflows.

  - **Total Run** - total count of tasks/jobs/sub workflows executed during workflow run. This is the cumulative of total retries, succeeded and failed count.

- **Workflow wall time** - The wall time from the start of the workflow execution to the end as reported by the DAG-MAN.In case of rescue dag the value is the cumulative of all retries.

- **Workflow cummulate job wall time** - The sum of the wall time of all jobs as reported by kickstart. In case of job retries the value is the cumulative of all retries. For workflows having sub workflow jobs (i.e SUBDAG and SUBDAX jobs), the wall time value includes jobs from the sub workflows as well. This value is multiplied by the multiplier_factor in the job instance table.

- **Cumulative job wall time as seen from submit side** - The sum of the wall time of all jobs as reported by DAGMan. This is similar to the regular cumulative job wall time, but includes job management overhead and delays. In case of job retries the value is the cumulative of all retries. For workflows having sub workflow jobs (i.e SUBDAG and SUBDAX jobs), the wall time value includes jobs from the sub workflows. This value is multiplied by the multiplier_factor in the job instance table.

pegasus-statistics generates the following statistics files based on the command line options set.

Workflow statistics file per workflow [workflow.txt]

Workflow statistics file per workflow contains the following information about each workflow run. In case of hierarchal workflows, the file contains a table for each sub workflow. The file also contains a 'Total' table at the bottom which is the cumulative of all the individual statistics details.

A sample table is shown below. It shows the following statistics about tasks, jobs and sub workflows.

- **Workflow retries** - number of times a workflow was retried.

- **Succeeded** - total count of succeeded tasks/jobs/sub workflows.

- **Failed** - total count of failed tasks/jobs/sub workflows.

- **Incomplete** - total count of tasks/jobs/sub workflows that are not in succeeded or failed state. This includes all the jobs that are not submitted, submitted but not completed etc. This is calculated as difference between 'total' count and sum of 'succeeded' and 'failed' count.

- **Total** - total count of tasks/jobs/sub workflows.

- **Retries** - total retry count of tasks/jobs/sub workflows.

- **Total Run** - total count of tasks/jobs/sub workflows executed during workflow run. This is the cumulative of total retries, succeeded and failed count.

Table 6.1. Workflow Statistics

#	Type	Succeeded	Failed	Incom-plete	Total	Retries	Total Run	Workflow Retries
2a6d-f11b-9972-4ba0-b4ba-4fd39c357af4								0
	Tasks	4	0	0	4	0	4	
	Jobs	13	0	0	13	0	13	
	Sub Work-flows	0	0	0	0	0	0	

Job statistics file per workflow [jobs.txt]

Job statistics file per workflow contains the following details about the job instances in each workflow. A sample file is shown below.

- **Job** - the name of the job instance

- **Try** - the number representing the job instance run count.

- **Site** - the site where the job instance ran.

- **Kickstart(sec.)** - the actual duration of the job instance in seconds on the remote compute node.

- **Mult** - multiplier factor from the job instance table for the job.

- **Kickstart_Mult** - value of the Kickstart column multiplied by Mult.

- **CPU-Time** - remote CPU time computed as the stime + utime (when Kickstart is not used, this is empty).

- **Post(sec.)** - the postscript time as reported by DAGMan.

- **CondorQTime(sec.)** - the time between submission by DAGMan and the remote Grid submission. It is an estimate of the time spent in the condor q on the submit node .

- **Resource(sec.)** - the time between the remote Grid submission and start of remote execution . It is an estimate of the time job instance spent in the remote queue .

- **Runtime(sec.)** - the time spent on the resource as seen by Condor DAGMan . Is always >=kickstart .

- **Seqexec(sec.)** - the time taken for the completion of a clustered job instance .

- **Seqexec-Delay(sec.)** - the time difference between the time for the completion of a clustered job instance and sum of all the individual tasks kickstart time .

Table 6.2. Job statistics

Job	Try	Site	Kick-start	Mult	Kick-start_Mult	CPU-Time	Post	Con-dorQ-Time	Re-source	Run-time	Se-qexec	Seqex-ec-De-lay
ana-lyze_ID0000004	1	local	60.002	1	60.002	59.843	5.0	0.0	-	62.0	-	-
cre-ate_dir_di-a-mond_0_lo-cal	1	local	0.027	1	0.027	0.003	5.0	5.0	-	0.0	-	-
find-range_ID0000002	1	local	60.001	10	600.01	59.921	5.0	0.0	-	60.0	-	-
find-range_ID0000003	1	local	60.002	10	600.02	59.912	5.0	10.0	-	61.0	-	-
pre-process_ID0000001	1	local	60.002	1	60.002	59.898	5.0	5.0	-	60.0	-	-
regis-ter_lo-cal_1_0	1	local	0.459	1	0.459	0.432	6.0	5.0	-	0.0	-	-
regis-ter_lo-cal_1_1	1	local	0.338	1	0.338	0.331	5.0	5.0	-	0.0	-	-
regis-ter_lo-cal_2_0	1	local	0.348	1	0.348	0.342	5.0	5.0	-	0.0	-	-
stage_in_lo-cal_lo-cal_0	1	local	0.39	1	0.39	0.032	5.0	5.0	-	0.0	-	-
stage_out_lo-cal_lo-cal_0_0	1	local	0.165	1	0.165	0.108	5.0	10.0	-	0.0	-	-
stage_out_lo-cal_lo-cal_1_0	1	local	0.147	1	0.147	0.098	7.0	5.0	-	0.0	-	-
stage_out_lo-cal_lo-cal_1_1	1	local	0.139	1	0.139	0.089	5.0	6.0	-	0.0	-	-
stage_out_lo-cal_lo-cal_2_0	1	local	0.145	1	0.145	0.101	5.0	5.0	-	0.0	-	-

Transformation statistics file per workflow [breakdown.txt]

Transformation statistics file per workflow contains information about the invocations in each workflow grouped by transformation name. A sample file is shown below.

- **Transformation** - name of the transformation.

- **Count** - the number of times invocations with a given transformation name was executed.

- **Succeeded** - the count of succeeded invocations with a given logical transformation name .

- **Failed** - the count of failed invocations with a given logical transformation name .

- **Min (sec.)** - the minimum runtime value of invocations with a given logical transformation name times the multiplier_factor.

- **Max (sec.)** - the minimum runtime value of invocations with a given logical transformation name times the multiplier_factor.

- **Mean (sec.)** - the mean of the invocation runtimes with a given logical transformation name times the multiplier_factor.

- **Total (sec.)** - the cumulative of runtime value of invocations with a given logical transformation name times the multiplier_factor.

Table 6.3. Transformation Statistics

Transfor-mation	Count	Succeeded	Failed	Min	Max	Mean	Total
dag-man::post	13	13	0	5.0	7.0	5.231	68.0
diamond::an-alyze	1	1	0	60.002	60.002	60.002	60.002
dia-mond::find-range	2	2	0	600.01	600.02	600.02	1200.03
dia-mond::pre-process	1	1	0	60.002	60.002	60.002	60.002
pega-sus::dirman-ager	1	1	0	0.027	0.027	0.027	0.027
pega-sus::pega-sus-transfer	5	5	0	0.139	0.39	0.197	0.986
pega-sus::rc-client	3	3	0	0.338	0.459	0.382	1.145

Time statistics file [time.txt]

Time statistics file contains job instance and invocation statistics information grouped by time and host. The time grouping can be on day/hour. The file contains the following tables Job instance statistics per day/hour, Invocation statistics per day/hour, Job instance statistics by host per day/hour and Invocation by host per day/hour. A sample Invocation statistics by host per day table is shown below.

- **Job instance statistics per day/hour** - the number of job instances run, total runtime sorted by day/hour.

- **Invocation statistics per day/hour** - the number of invocations , total runtime sorted by day/hour.

- **Job instance statistics by host per day/hour** - the number of job instances run, total runtime on each host sorted by day/hour.

- **Invocation statistics by host per day/hour** - the number of invocations , total runtime on each host sorted by day/hour.

Table 6.4. Invocation statistics by host per day

Date [YYYY-MM-DD]	Host	Count	Runtime (Sec.)
2011-07-15	butterfly.isi.edu	54	625.094

pegasus-plots

Pegasus-plots generates graphs and charts to visualize workflow execution. To generate graphs and charts run the command as shown below.

```
$ pegasus-plots  -p all  /scratch/grid-setup/run0001/


...

********************************************* SUMMARY *********************************************

Graphs and charts generated by pegasus-plots can be viewed by opening the generated html file in the
 web browser  :
/scratch/grid-setup/run0001/plots/index.html


**************************************************************************************************
```

By default the output gets generated to plots folder inside the submit directory. The output that is generated by pegasus-plots is based on the value set for command line option 'p'(plotting_level).In the sample run the command line option 'p' is set to 'all' to generate all the charts and graphs for the workflow run. Please consult the pegasus-plots man page to find a detailed description of various command line options. pegasus-plots generates an index.html file which provides links to all the generated charts and plots. A sample index.html page is shown below.

Figure 6.1. pegasus-plot index page
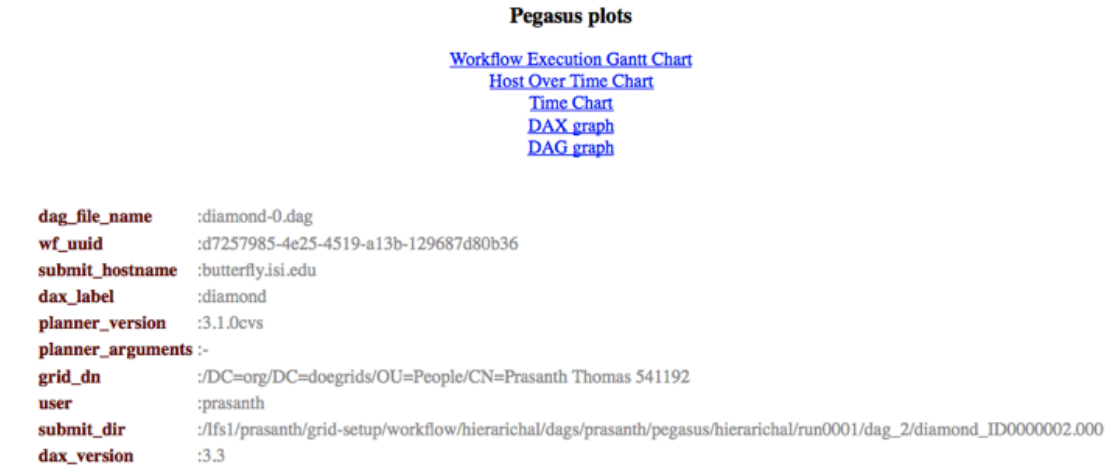
pegasus-plots generates the following plots and charts.

Dax Graph

Graph representation of the DAX file. A sample page is shown below.

Figure 6.2. DAX Graph

DAX Graph

Top level workflow (ad180edc-222b-49d5-bcf3-7c49b8969422)

wf_uuid :ad180edc-222b-49d5-bcf3-7c49b8969422
dax label :hierarichal

Sub workflow's of workflow (ad180edc-222b-49d5-bcf3-7c49b8969422)

wf_uuid :d1ada867-5499-436d-b480-52d73845d27e
dax label :diamond

wf_uuid :812ac7e2-11a0-4fff-ada5-85ecfe4b4fa0
dax label :diamond
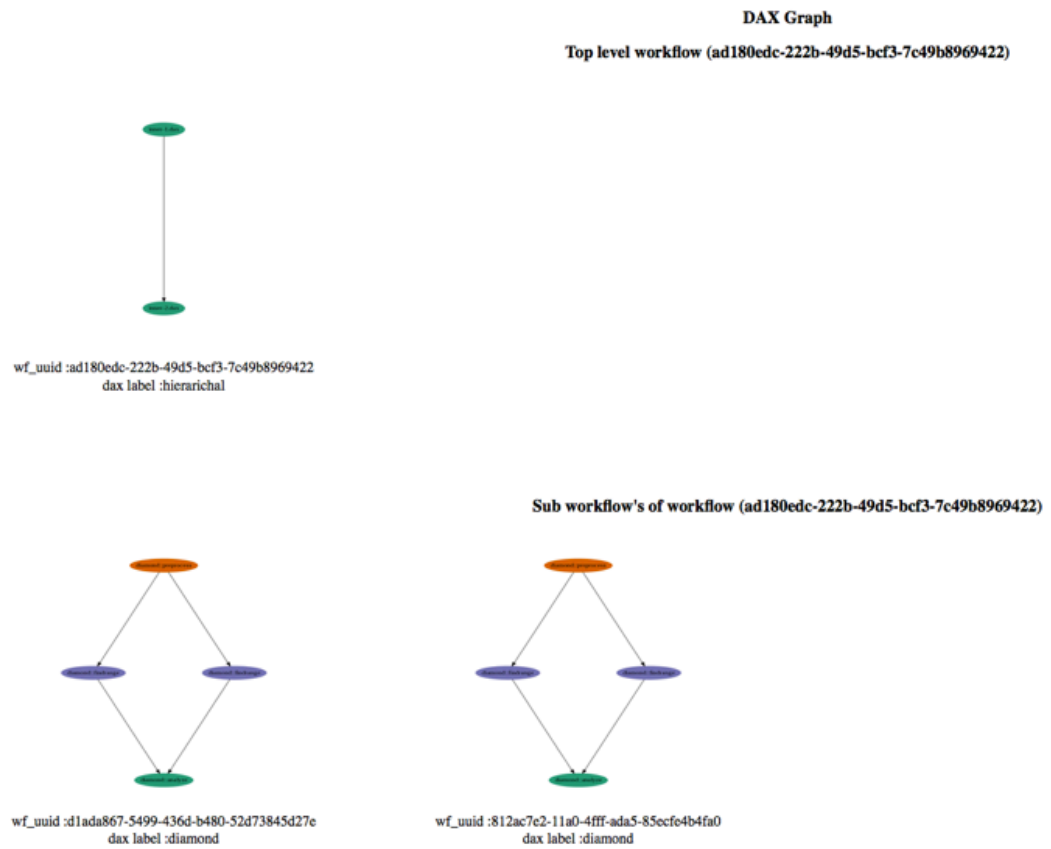
Dag Graph

Graph representation of the DAG file. A sample page is shown below.

Figure 6.3. DAG Graph

wf_uuid :ad180edc-222b-49d5-bcf3-7c49b8969422
dag label :hierarichal-0

wf_uuid :d1ada867-5499-436d-b480-52d73845d27e
dag label :diamond-0

wf_uuid :812ac7e2-11a0-4fff-ada5-85ecfe4b4fa0
dag label :diamond-0
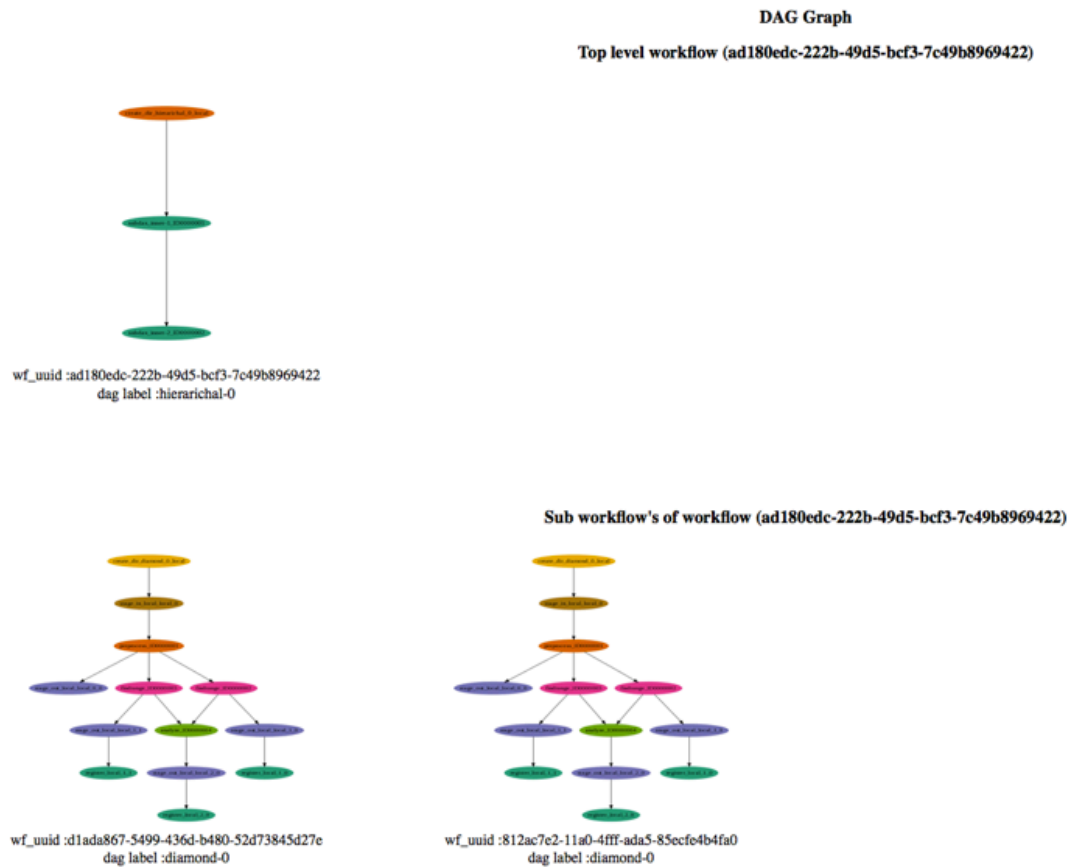
Gantt workflow execution chart

Gantt chart of the workflow execution run. A sample page is shown below.

Figure 6.4. Gantt Chart

The toolbar at the top provides zoom in/out , pan left/right/top/bottom and show/hide job name functionality.The toolbar at the bottom can be used to show/hide job states. Failed job instances are shown in red border in the chart. Clicking on a sub workflow job instance will take you to the corresponding sub workflow chart.

Host over time chart

Host over time chart of the workflow execution run. A sample page is shown below.

Figure 6.5. Host over time chart
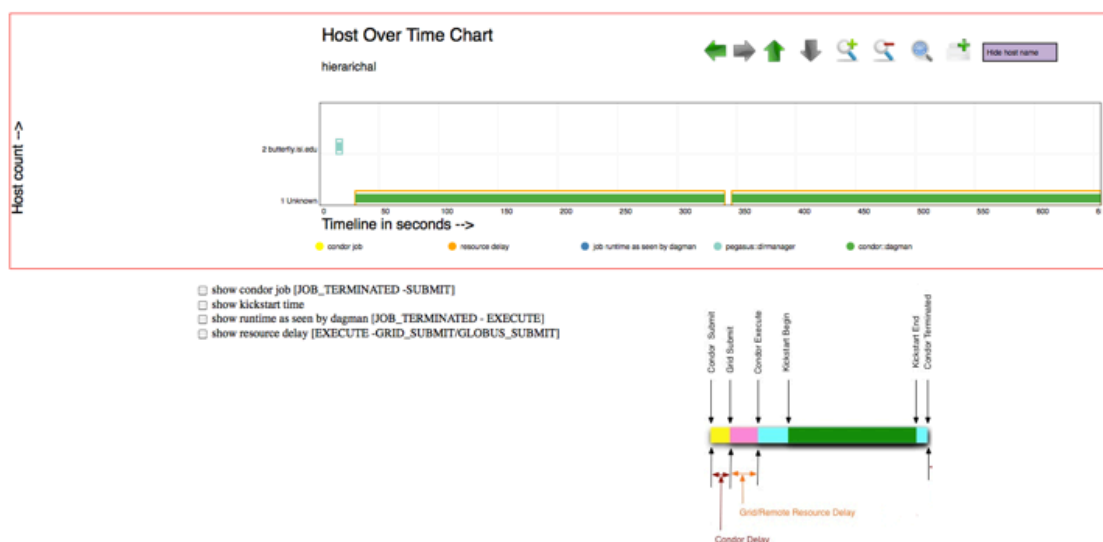
The toolbar at the top provides zoom in/out , pan left/right/top/bottom and show/hide host name functionality.The toolbar at the bottom can be used to show/hide job states. Failed job instances are shown in red border in the chart. Clicking on a sub workflow job instance will take you to the corresponding sub workflow chart.

Time chart

Time chart shows job instance/invocation count and runtime of the workflow run over time. A sample page is shown below.

Figure 6.6. Time chart
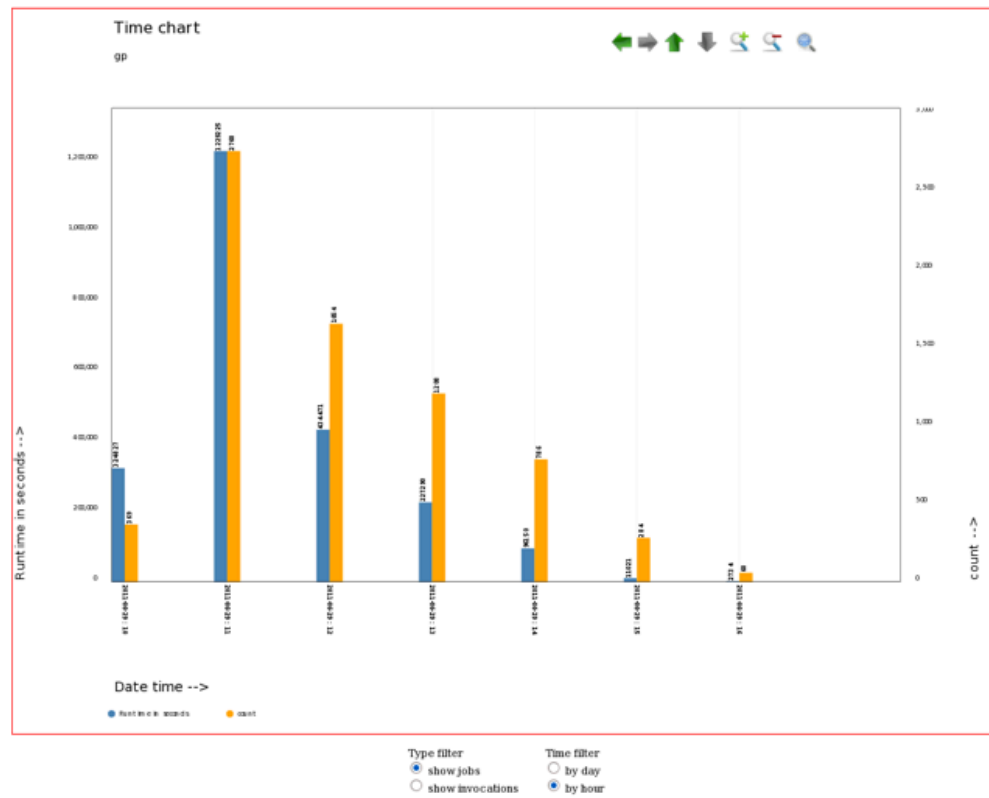
The toolbar at the top provides zoom in/out and pan left/right/top/bottom functionality. The toolbar at the bottom can be used to switch between job instances/ invocations and day/hour filtering.

Breakdown chart

Breakdown chart shows invocation count and runtime of the workflow run grouped by transformation name. A sample page is shown below.

Figure 6.7. Breakdown chart
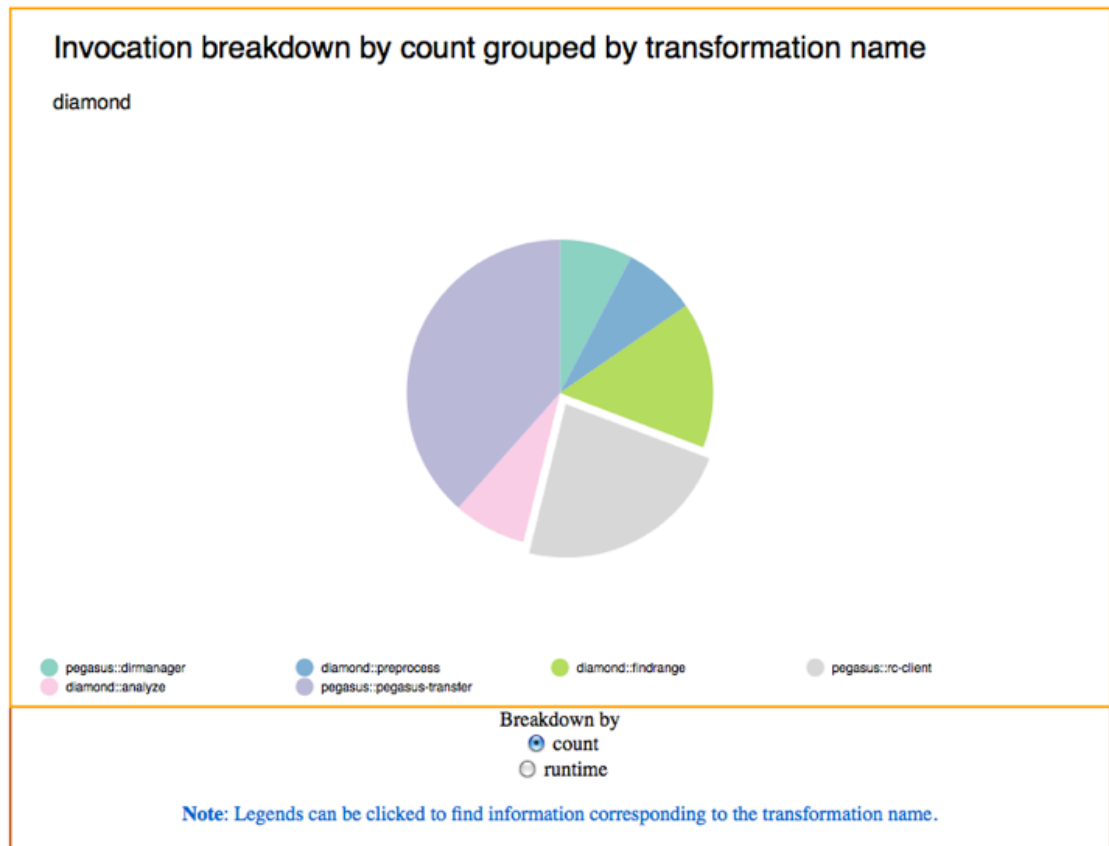
The toolbar at the bottom can be used to switch between invocation count and runtime filtering. Legends can be clicked to get more details.

Dashboard

As the number of jobs and tasks in workflows increase, the ability to track the progress and quickly debug a workflow becomes more and more important. The dashboard provides users with a tool to monitor and debug workflows both in real-time as well as after execution is already completed, through a browser.

Workflow Dashboard

Pegasus Workflow Dashboard is bundled with Pegasus. The pegasus-service is developed in Python and uses the Flask framework to implement the web interface.The users can then connect to this server using a browser to monitor/debug workflows.

Note

the workflow dashboard can only monitor workflows which have been executed using Pegasus 4.2.0 and above.

To start the Pegasus Dashboard execute the following command

```
$ pegasus-service --host 127.0.0.1 --port 5000

SSL is not configured: Using self-signed certificate
2015-04-13 16:14:23,074:Pegasus.service.server:79: WARNING: SSL is not configured: Using self-signed
 certificate
```

```
Service not running as root: Will not be able to switch users
2015-04-13 16:14:23,074:Pegasus.service.server:86: WARNING: Service not running as root: Will not be
 able to switch users
```

By default, the server is configured to listen only on localhost/127.0.0.1 on port 5000. A user can view the dashboard on **https://localhost:5000/**

To make the Pegasus Dashboard listen on all network interfaces OR on a different port, users can pass different values to the --host and/or --port options.

By default, the dashboard server can only monitor workflows run by the current user i.e. the user who is running the pegasus-service.

The Dashboard's home page lists all workflows, which have been run by the current-user. The home page shows the status of each of the workflow i.e. Running/Successful/Failed/Failing. The home page lists only the top level workflows (Pegasus supports hierarchical workflows i.e. workflows within a workflow). The rows in the table are color coded

- **Green**: indicates workflow finished successfully.

- **Red**: indicates workflow finished with a failure.

- **Blue**: indicates a workflow is currently running.

- **Gray**: indicates a workflow that was archived.

Figure 6.8. Dashboard Home Page
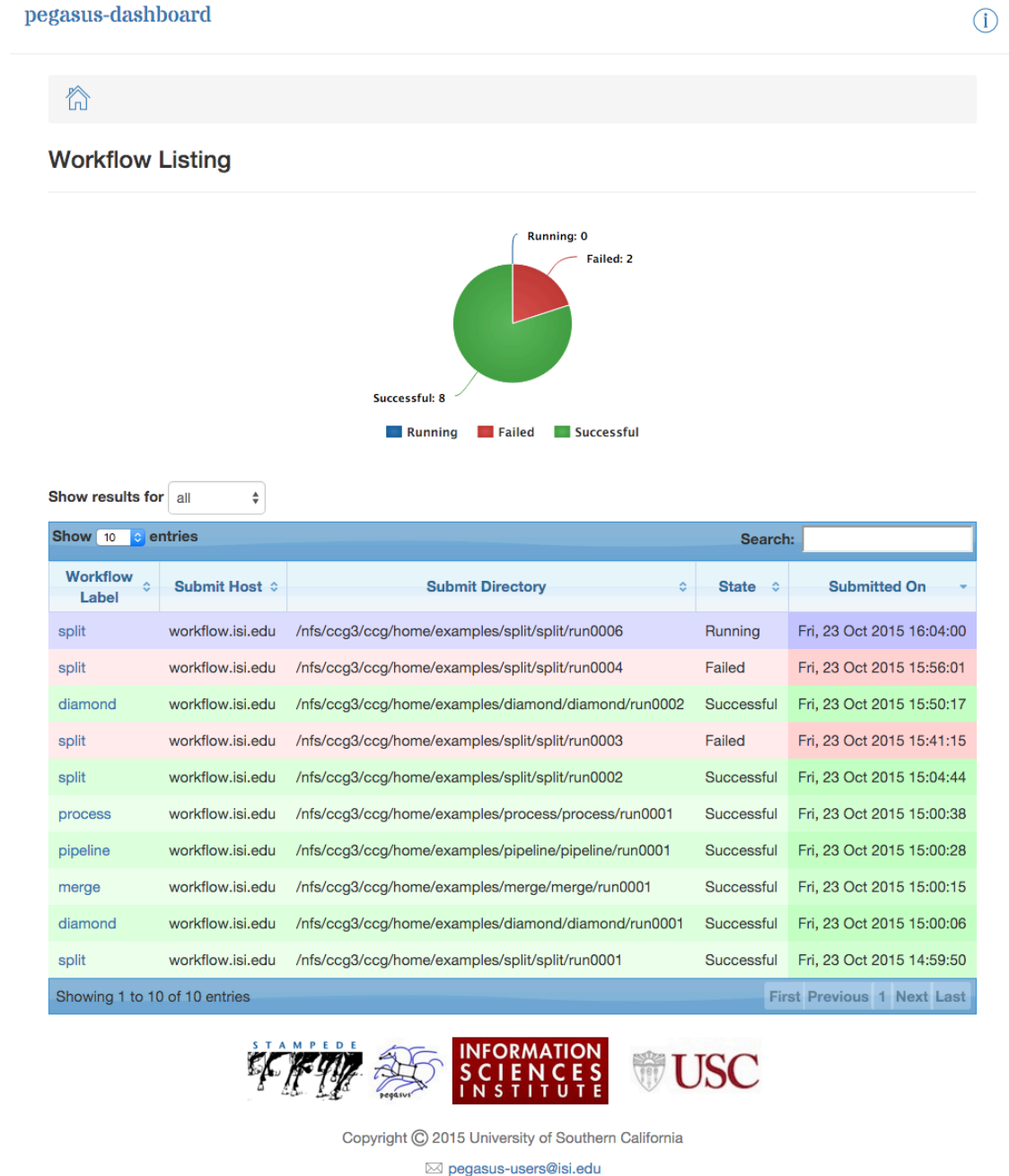
To view details specific to a workflow, the user can click on corresponding workflow label. The workflow details page lists workflow specific information like workflow label, workflow status, location of the submit directory, files, and metadata associated with the workflow etc. The details page also displays pie charts showing the distribution of jobs based on status.

In addition, the details page displays a tab listing all sub-workflows and their statuses. Additional tabs exist which list information for all running, failed, successful, and failing jobs.

Note

Failing jobs are currently running jobs (visible in Running tab), which have failed in previous attempts to execute them.

The information displayed for a job depends on it's status. For example, the failed jobs tab displays the job name, exit code, links to available standard output, and standard error contents.

Workflow Details 1145e2d5-ad2f-45d6-a3ce-4bd68499d8af

Summary | Files **6** | Metadata **2**

Label	diamond
Type	root-wf
Progress	Successful
Submit Host	cartman
User	bamboo
Submit Directory	/lfs1/software/bamboo/data/xml-data/build-dir/PEGASUS-WT-T39A/test/core/039-bl...
DAGMan Out File	diamond-0.dag.dagman.out
Wall Time	5 mins 9 secs
Cumulative Wall Time	5 mins 52 secs

Job Status (Entire Workflow)

Unsubmitted: 0
Failed: 0
Successful: 26

■ Unsubmitted ■ Failed ■ Successful

Job Status (Per Workflow)

Jobs: 0
Workflows: 0
Total: 0

Jobs: 0
Workflows: 0
Total: 0

Jobs: 26
Workflows: 0
Total: 26

■ Running ■ Failed ■ Successful

Charts | **Statistics**

Sub Workflows | **Failed** | **Running** | **Successful** | **Failing**

Show 10 entries   Search: [   ]

Job Name	Time Taken
analyze_ID0000004	1 min
clean_up_local_level_3_0	5 secs
clean_up_local_level_4_0	5 secs
clean_up_local_level_4_1	3 secs
clean_up_local_level_5_0	7 secs
clean_up_local_level_6_0	3 secs
cleanup_diamond_0_local	3 secs
create_dir_diamond_0_local	2 secs
findrange_ID0000002	1 min 1 sec
findrange_ID0000003	1 min

Showing 1 to 10 of 26 entries   First Previous 1 **2** 3 Next Last

Figure 6.10. Dashboard Workflow Metadata

Figure 6.11. Dashboard Workflow Files

To view details specific to a job the user can click on the corresponding job's job label. The job details page lists information relevant to a specific job. For example, the page lists information like job name, exit code, run time, etc.

The job instance section of the job details page lists all attempts made to run the job i.e. if a job failed in its first attempt due to transient errors, but ran successfully when retried, the job instance section shows two entries; one for each attempt to run the job.

The job details page also shows tab's for failed, and successful task invocations (Pegasus allows users to group multiple smaller task's into a single job i.e. a job may consist of one or more tasks)

🏠 / Workflow / Job

Job Details

Label	**ls_ID0000001**
Type	Compute
Exit Code	0
Working Directory	🗋 /private/var/condor/execute/dir_12968
Application Stdour/Stderr	Preview
Kickstart Output	🔗 00/00/ls_ID0000001.out.000
Condor Stderr/Pegasus Lite Log	🔗 00/00/ls_ID0000001.err.000
Condor Submit File	🔗 ls_ID0000001.sub
Site	condorpool
Host	128.9.72.154 > isis.isi.edu

Job States

Submit	Thu Mar 23, 2017 01:25:53 PM (0 secs)
Execute	Thu Mar 23, 2017 01:26:08 PM (15 secs)
Image Size	Thu Mar 23, 2017 01:26:08 PM (0 secs)
Job Terminated	Thu Mar 23, 2017 01:26:08 PM (0 secs)
Job Success	Thu Mar 23, 2017 01:26:08 PM (0 secs)
Post Script Started	Thu Mar 23, 2017 01:26:08 PM (0 secs)
Post Script Terminated	Thu Mar 23, 2017 01:26:13 PM (5 secs)
Post Script Success	Thu Mar 23, 2017 01:26:13 PM (0 secs)

Job Instances

Show 10 ⇕ entries

Try ▼	Job Instance ID ⇕	Exitcode ⇕	Stdout ⇕	Stderr ⇕
1	2	0	Preview	Preview

Showing 1 to 1 of 1 entries

First Previous 1 Next Last

Job Invocations

Failed	Successful

No failed invocations.

The task invocation details page provides task specific information like task name, exit code, duration, metadata associated with the task, etc. Task details differ from job details, as they are more granular in nature.

Figure 6.13. Dashboard Invocation Page

The dashboard also has web pages for workflow statistics and workflow charts, which graphically renders information provided by the pegasus-statistics and pegasus-plots command respectively.

The Statistics page shows the following statistics.

1. Workflow level statistics

2. Job breakdown statistics

3. Job specific statistics

Figure 6.14. Dashboard Statistics Page
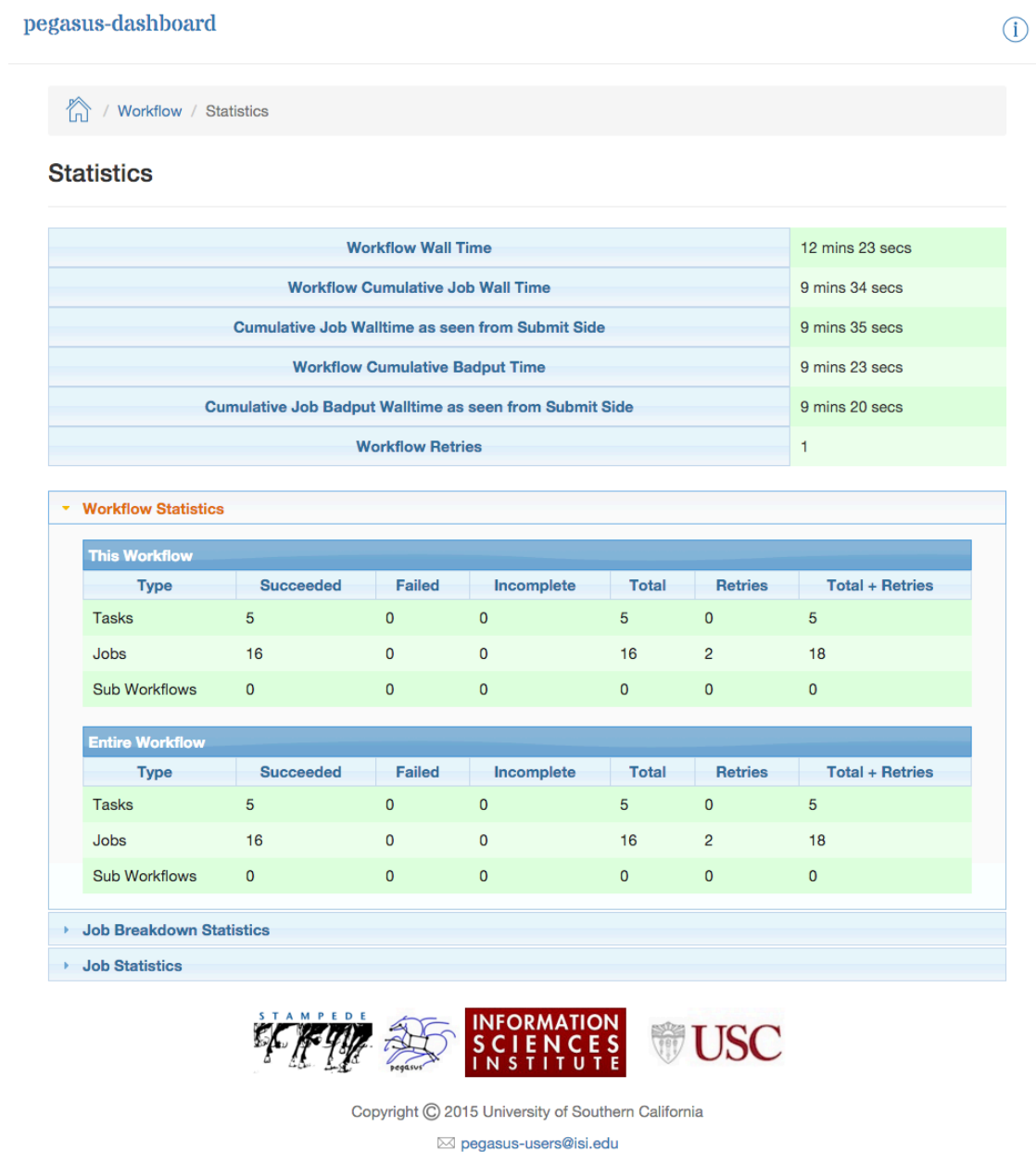
The Charts page shows the following charts.

1. Job Distribution by Count/Time

2. Time Chart by Job/Invocation

3. Workflow Execution Gantt Chart

The chart below shows the invocation distribution by count or time.

Figure 6.15. Dashboard Plots - Job Distribution
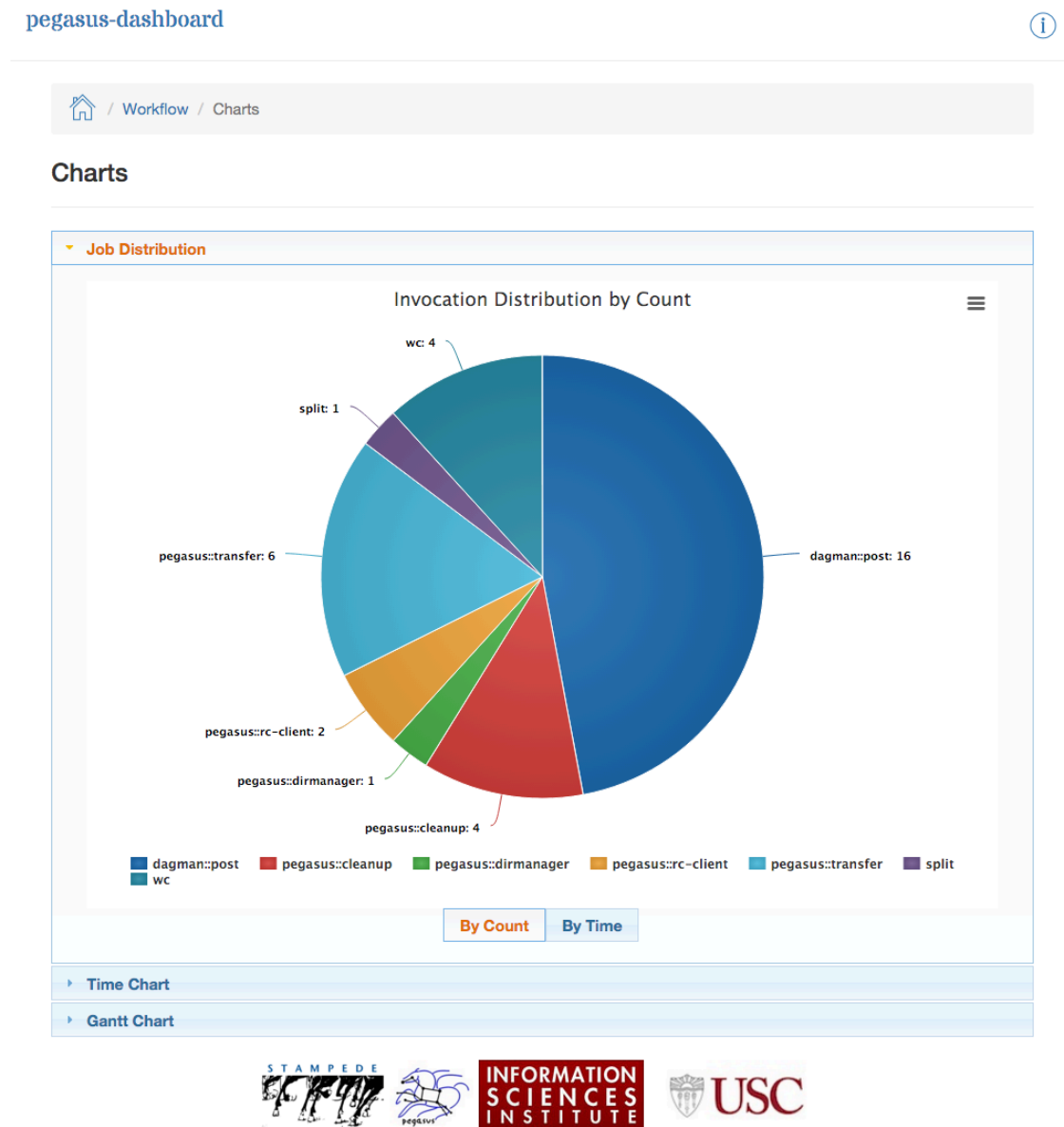
The time chart shown below shows the number of jobs/invocations in the workflow and their total runtime

Figure 6.16. Dashboard Plots - Time Chart
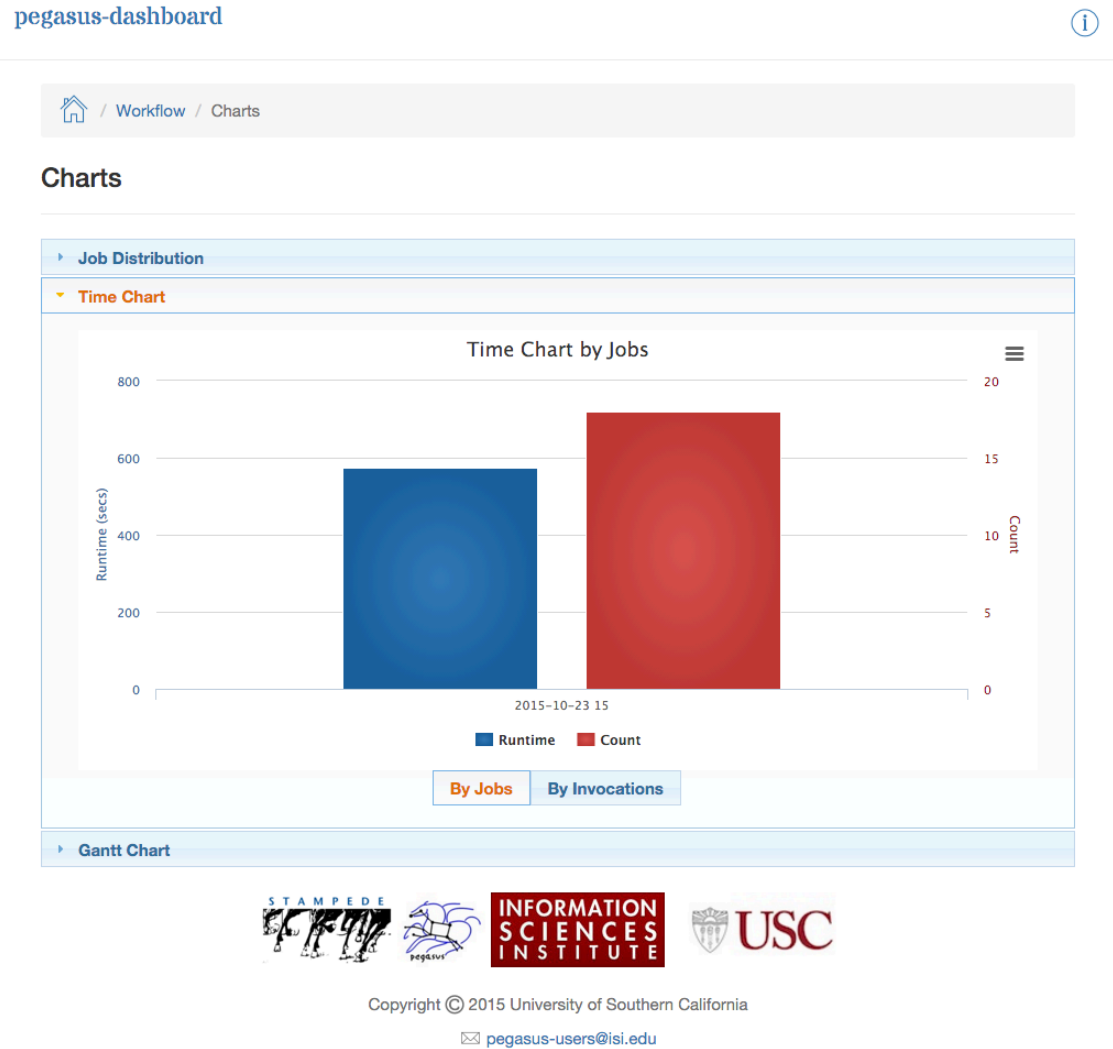
The workflow gantt chart lays out the execution of the jobs in the workflow over time.

Figure 6.17. Dashboard Plots - Workflow Gantt Chart
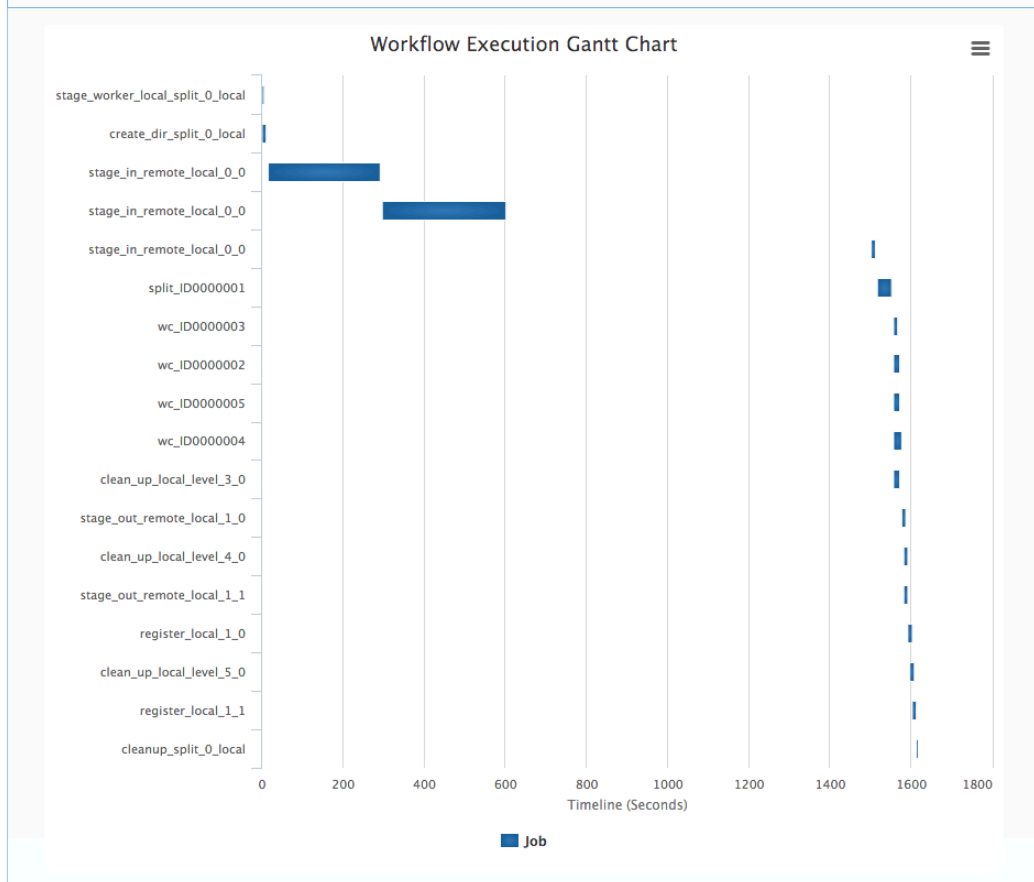
Notifications

The Pegasus Workflow Mapper now supports job and workflow level notifications. You can specify in the DAX with the job or the workflow

- the event when the notification needs to be sent

- the executable that needs to be invoked.

The notifications are issued from the submit host by the pegasus-monitord daemon that monitors the Condor logs for the workflow. When a notification is issued, pegasus-monitord while invoking the notifying executable sets certain environment variables that contain information about the job and workflow state.

The Pegasus release comes with default notification clients that send notifications via email or jabber.

Specifying Notifications in the DAX

Currently, you can specify notifications for the jobs and the workflow by the use of invoke elements.

Invoke elements can be sub elements for the following elements in the DAX schema.

- job - to associate notifications with a compute job in the DAX.

- dax - to associate notifications with a dax job in the DAX.

- dag - to associate notifications with a dag job in the DAX.

- executable - to associate notifications with a job that uses a particular notification

The invoke element can be specified at the root element level of the DAX to indicate workflow level notifications.

The invoke element may be specified multiple times, as needed. It has a mandatory **when** attribute with the following value set

Table 6.5. Invoke Element attributes and meaning.

Enumeration of Values for when attribute	Meaning
never	(default). Never notify of anything. This is useful to temporarily disable an existing notifications.
start	create a notification when the job is submitted.
on_error	after a job finishes with failure (exitcode != 0).
on_success	after a job finishes with success (exitcode == 0).
at_end	after a job finishes, regardless of exitcode.
all	like start and at_end combined.

You can specify multiple invoke elements corresponding to same when attribute value in the DAX. This will allow you to have multiple notifications for the same event.

Here is an example that illustrates that.

```
<job id="ID000001" namespace="example" name="mDiffFit" version="1.0"
      node-label="preprocess" >
    <argument>-a top -T 6  -i <file name="f.a"/>  -o <file name="f.b1"/></argument>

    <!-- profiles are optional -->
    <profile namespace="execution" key="site">isi_viz</profile>
    <profile namespace="condor" key="getenv">true</profile>

    <uses name="f.a" link="input"  register="false" transfer="true" type="data" />
    <uses name="f.b" link="output" register="false" transfer="true" type="data" />

    <!-- 'WHEN' enumeration: never, start, on_error, on_success, at_end, all -->
    <invoke when="start">/path/to/notify1 arg1 arg2</invoke>
    <invoke when="start">/path/to/notify1 arg3 arg4</invoke>
    <invoke when="on_success">/path/to/notify2 arg3 arg4</invoke>
  </job>
```

In the above example the executable notify1 will be invoked twice when a job is submitted (when="start"), once with arguments arg1 and arg2 and second time with arguments arg3 and arg4.

The DAX Generator API chapter has information about how to add notifications to the DAX using the DAX api's.

Notify File created by Pegasus in the submit directory

Pegasus while planning a workflow writes out a notify file in the submit directory that contains all the notifications that need to be sent for the workflow. pegasus-monitord picks up this notifications file to determine what notifications need to be sent and when.

1. ENTITY_TYPE ID NOTIFICATION_CONDITION ACTION

   - ENTITY_TYPE can be either of the following keywords

     - WORKFLOW - indicates workflow level notification

     - JOB - indicates notifications for a job in the executable workflow

     - DAXJOB - indicates notifications for a DAX Job in the executable workflow

     - DAGJOB - indicates notifications for a DAG Job in the executable workflow

   - ID indicates the identifier for the entity. It has different meaning depending on the entity type - -

     - workflow - ID is wf_uuid

     - JOB|DAXJOB|DAGJOB - ID is the job identifier in the executable workflow (DAG).

   - NOTIFICATION_CONDITION is the condition when the notification needs to be sent. The notification conditions are enumerated in this table

   - ACTION is what needs to happen when condition is satisfied. It is executable + arguments

2. INVOCATION JOB_IDENTIFIER INV.ID NOTIFICATION_CONDITION ACTION

   The INVOCATION lines are only generated for clustered jobs, to specifiy the finer grained notifications for each constitutent job/invocation .

   - JOB IDENTIFIER is the job identifier in the executable workflow (DAG).

   - INV.ID indicates the index of the task in the clustered job for which the notification needs to be sent.

   - NOTIFICATION_CONDITION is the condition when the notification needs to be sent. The notification conditions are enumerated in Table 1

   - ACTION is what needs to happen when condition is satisfied. It is executable + arguments

A sample notifications file generated is listed below.

```
WORKFLOW d2c4f79c-8d5b-4577-8c46-5031f4d704e8 on_error /bin/date1

INVOCATION merge_vahi-preprocess-1.0_PID1_ID1 1 on_success /bin/date_executable
INVOCATION merge_vahi-preprocess-1.0_PID1_ID1 1 on_success /bin/date_executable
INVOCATION merge_vahi-preprocess-1.0_PID1_ID1 1 on_error /bin/date_executable

INVOCATION merge_vahi-preprocess-1.0_PID1_ID1 2 on_success /bin/date_executable
INVOCATION merge_vahi-preprocess-1.0_PID1_ID1 2 on_error /bin/date_executable

DAXJOB subdax_black_ID000003 on_error /bin/date13
JOB    analyze_ID00004    on_success /bin/date
```

Configuring pegasus-monitord for notifications

Whenever pegasus-monitord enters a workflow (or sub-workflow) directory, it will read the notifications file generated by Pegasus. Pegasus-monitord will match events in the running workflow against the notifications specified in the notifications file and will initiate the script specified in a notification when that notification matches an event in the workflow. It is important to note that there will be a delay between a certain event happening in the workflow, and pegasus-monitord processing the log file and executing the corresponding notification script.

The following command line options (and properties) can change how pegasus-monitord handles notifications:

- --no-notifications (pegasus.monitord.notifications=False): Will disable notifications completely.

- --notifications-max=nn (pegasus.monitord.notifications.max=nn): Will limit the number of concurrent notification scripts to nn. Once pegasus-monitord reaches this number, it will wait until one notification script finishes before starting a new one. Notifications happening during this time will be queued by the system. The default number of concurrent notification scripts for pegasus-monitord is 10.

- --notifications-timeout=nn (pegasus.monitord.notifications.timeout=nn): This setting is used to change how long will pegasus-monitord wait for a notification script to finish. By default pegasus-monitord will wait for as long as it takes (possibly indefinitely) until a notification script ends. With this option, pegasus-monitord will wait for at most nn seconds before killing the notification script.

It is also important to understand that pegasus-monitord will not issue any notifications when it is executed in replay mode.

Environment set for the notification scripts

Whenever a notification in the notifications file matches an event in the running workflow, pegasus-monitord will run the corresponding script specified in the ACTION field of the notifications file. Pegasus-monitord will set the following environment variables for each notification script is starts:

- PEGASUS_EVENT: The NOTIFICATION_CONDITION that caused the notification. In the case of the "all" condition, pegasus-monitord will substitute it for the actual event that caused the match (e.g. "start" or "at_end").

- PEGASUS_EVENT_TIMESTAMP: Timestamp in EPOCH format for the event (better for automated processing).

- PEGASUS_EVENT_TIMESTAMP_ISO: Same as above, but in ISO format (better for human readability).

- PEGASUS_SUBMIT_DIR: The submit directory for the workflow (usually the value from "submit_dir" in the braindump.txt file)

- PEGASUS_STDOUT: For workflow notifications, this will correspond to the dagman.out file for that workflow. For job and invocation notifications, this field will contain the output file (stdout) for that particular job instance.

- PEGASUS_STDERR: For job and invocation notifications, this field will contain the error file (stderr) for the particular executable job instance. This field does not exist in case of workflow notifications.

- PEGASUS_WFID: Contains the workflow id for this notification in the form of DAX_LABEL + DAX_INDEX (from the braindump.txt file).

- PEGASUS_JOBID: For workflow notifications, this contains the worfkflow wf_uuid (from the braindump.txt file). For job and invocation notifications, this field contains the job identifier in the executable workflow (DAG) for the particular notification.

- PEGASUS_INVID: Contains the index of the task in the clustered job for the notification.

- PEGASUS_STATUS: For workflow notifications, this contains DAGMan's exit code. For job and invocation notifications, this field contains the exit code for the particular job/task. Please note that this field is not present for 'start' notification events.

Default Notification Scripts

Pegasus ships with two reference notification scripts. These can be used as starting point when creating your own notification scripts, or if the default one is all you need, you can use them directly in your workflows. The scripts are:

- **libexec/notification/email** - sends email, including the output from **pegasus-status** (default) or **pegasus-analyzer**.

```
$ ./libexec/notification/email --help
Usage: email [options]

Options:
  -h, --help            show this help message and exit
```

```
             -t TO_ADDRESS, --to=TO_ADDRESS
                               The To: email address. Defines the recipient for the
                               notification.
             -f FROM_ADDRESS, --from=FROM_ADDRESS
                               The From: email address. Defaults to the required To:
                               address.
             -r REPORT, --report=REPORT
                               Include workflow report. Valid values are: none
                               pegasus-analyzer pegasus-status (default)
```

- **libexec/notification/jabber** - sends simple notifications to Jabber/GTalk. This can be useful for job failures.

```
$ ./libexec/notification/jabber --help
Usage: jabber [options]

Options:
  -h, --help              show this help message and exit
  -i JABBER_ID, --jabberid=JABBER_ID
                          Your jabber id. Example: user@jabberhost.com
  -p PASSWORD, --password=PASSWORD
                          Your jabber password
  -s HOST, --host=HOST    Jabber host, if different from the host in your jabber
                          id. For Google talk, set this to talk.google.com
  -r RECIPIENT, --recipient=RECIPIENT
                          Jabber id of the recipient. Not necessary if you want
                          to send to your own jabber id
```

For example, if the DAX generator is written in Python and you want notifications on 'at_end' events (successful or failed):

```
# job level notifications - in this case for at_end events
job.invoke('at_end', pegasus_home + "/libexec/notifications/email --to me@somewhere.edu")
```

Please see the notifications example to see a full workflow using notifications.

Monitoring Database

Pegasus launches a monitoring daemon called pegasus-monitord per workflow (a single daemon is launched if a user submits a hierarchal workflow) . pegasus-monitord parses the workflow and job logs in the submit directory and populates to a database. This chapter gives an overview of the pegasus-monitord and describes the schema of the runtime database.

pegasus-monitord

Pegasus-monitord is used to follow workflows, parsing the output of DAGMan's dagman.out file. In addition to generating the jobstate.log file, which contains the various states that a job goes through during the workflow execution, **pegasus-monitord** can also be used to mine information from jobs' submit and output files, and either populate a database, or write a file with NetLogger events containing this information. **Pegasus-monitord** can also send notifications to users in real-time as it parses the workflow execution logs.

Pegasus-monitord is automatically invoked by **pegasus-run**, and tracks workflows in real-time. By default, it produces the jobstate.log file, and a SQLite database, which contains all the information listed in the Stampede schema. When a workflow fails, and is re-submitted with a rescue DAG, **pegasus-monitord** will automatically pick up from where it left previously and continue to write the jobstate.log file and populate the database.

If, after the workflow has already finished, users need to re-create the jobstate.log file, or re-populate the database from scratch, **pegasus-monitord**'s **--replay** option should be used when running it manually.

Populating to different backend databases

In addition to SQLite, **pegasus-monitord** supports other types of databases, such as MySQL and Postgres. Users will need to install the low-level database drivers, and can use the **--dest** command-line option, or the **pegasus.monitord.output** property to select where the logs should go.

As an example, the command:

```
$ pegasus-monitord -r diamond-0.dag.dagman.out
```

will launch **pegasus-monitord** in replay mode. In this case, if a jobstate.log file already exists, it will be rotated and a new file will be created. It will also create/use a SQLite database in the workflow's run directory, with the name of diamond-0.stampede.db. If the database already exists, it will make sure to remove any references to the current workflow before it populates the database. In this case, **pegasus-monitord** will process the workflow information from start to finish, including any restarts that may have happened.

Users can specify an alternative database for the events, as illustrated by the following examples:

```
$ pegasus-monitord -r -d mysql://username:userpass@hostname/database_name diamond-0.dag.dagman.out
```

```
$ pegasus-monitord -r -d sqlite:////tmp/diamond-0.db diamond-0.dag.dagman.out
```

In the first example, **pegasus-monitord** will send the data to the **database_name** database located at server **hostname**, using the **username** and **userpass** provided. In the second example, **pegasus-monitord** will store the data in the /tmp/diamond-0.db SQLite database.

Note

> For absolute paths four slashes are required when specifying an alternative database path in SQLite.

Users should also be aware that in all cases, with the exception of SQLite, the database should exist before **pegasus-monitord** is run (as it creates all needed tables but does not create the database itself).

Finally, the following example:

```
$ pegasus-monitord -r --dest diamond-0.bp diamond-0.dag.dagman.out
```

sends events to the diamond-0.bp file. (please note that in replay mode, any data on the file will be overwritten).

One important detail is that while processing a workflow, **pegasus-monitord** will automatically detect if/when sub-workflows are initiated, and will automatically track those sub-workflows as well. In this case, although **pegasus-monitord** will create a separate jobstate.log file in each workflow directory, the database at the top-level workflow will contain the information from not only the main workflow, but also from all sub-workflows.

Monitoring related files in the workflow directory

Pegasus-monitord generates a number of files in each workflow directory:

- **jobstate.log**: contains a summary of workflow and job execution.

- **monitord.log**: contains any log messages generated by **pegasus-monitord**. It is not overwritten when it restarts. This file is not generated in replay mode, as all log messages from **pegasus-monitord** are output to the console. Also, when sub-workflows are involved, only the top-level workflow will have this log file. Starting with release 4.0 and 3.1.1, monitord.log file is rotated if it exists already.

- **monitord.started**: contains a timestamp indicating when **pegasus-monitord** was started. This file get overwritten every time **pegasus-monitord** starts.

- **monitord.done**: contains a timestamp indicating when **pegasus-monitord** finished. This file is overwritten every time **pegasus-monitord** starts.

- **monitord.info**: contains **pegasus-monitord** state information, which allows it to resume processing if a workflow does not finish properly and a rescue dag is submitted. This file is erased when **pegasus-monitord** is executed in replay mode.

- **monitord.recover**: contains **pegasus-monitord** state information that allows it to detect that a previous instance of **pegasus-monitord** failed (or was killed) midway through parsing a workflow's execution logs. This file is only present while **pegasus-monitord** is running, as it is deleted when it ends and the **monitord.info** file is generated.

- **monitord.subwf.db**: contains information that aids **pegasus-monitord** to track when sub-workflows fail and are re-planned/re-tried. It is overwritten when **pegasus-monitord** is started in replay mode.

- **monitord-notifications.log**: contains the log file for notification-related messages. Normally, this file only includes logs for failed notifications, but can be populated with all notification information when **pegasus-monitord** is run in verbose mode via the **-v** command-line option.

Overview of the Workflow Database Schema.

Pegasus takes in a DAX which is composed of tasks. Pegasus plans it into a Condor DAG / Executable workflow that consists of Jobs. In case of Clustering, multiple tasks in the DAX can be captured into a single job in the Executable workflow. When DAGMan executes a job, a job instance is populated . Job instances capture information as seen by DAGMan. In case DAGMan retires a job on detecting a failure , a new job instance is populated. When DAGMan finds a job instance has finished , an invocation is associated with job instance. In case of clustered job, multiple invocations will be associated with a single job instance. If a Pre script or Post Script is associated with a job instance, then invocations are populated in the database for the corresponding job instance.

The current schema version is **4.0** that is stored in the schema_info table.

Figure 6.18. Workflow Database Schema

Stampede Schema Upgrade Tool

Starting Pegasus 4.x the monitoring and statistics database schema has changed. If you want to use the pegasus-statistics, pegasus-analyzer and pegasus-plots against a 3.x database you will need to upgrade the schema first using the schema upgrade tool /usr/share/pegasus/sql/schema_tool.py or /path/to/pegasus-4.x/share/pegasus/sql/schema_tool.py

Upgrading the schema is required for people using the MySQL database for storing their monitoring information if it was setup with 3.x monitoring tools.

If your setup uses the default SQLite database then the new databases run with Pegasus 4.x are automatically created with the correct schema. In this case you only need to upgrade the SQLite database from older runs if you wish to query them with the newer clients.

To upgrade the database

```
For SQLite Database
```

cd /to/the/workflow/directory/with/3.x.monitord.db

```
Check the db version
```

/usr/share/pegasus/sql/schema_tool.py -c connString=sqlite:////to/the/workflow/directory/with/workflow.stampede.db
```
2012-02-29T01:29:43.330476Z INFO    netlogger.analysis.schema.schema_check.SchemaCheck.init |
2012-02-29T01:29:43.330708Z INFO
 netlogger.analysis.schema.schema_check.SchemaCheck.check_schema.start |
2012-02-29T01:29:43.348995Z INFO    netlogger.analysis.schema.schema_check.SchemaCheck.check_schema
                                    | Current version set to: 3.1.
2012-02-29T01:29:43.349133Z ERROR   netlogger.analysis.schema.schema_check.SchemaCheck.check_schema
                                    | Schema version 3.1 found - expecting 4.0 - database admin will
 need to run upgrade tool.
```

```
Convert the Database to be version 4.x compliant
```

**/usr/share/pegasus/sql/schema_tool.py -u connString=sqlite:////to/the/workflow/directory/with/
workflow.stampede.db**
```
2012-02-29T01:35:35.046317Z INFO   netlogger.analysis.schema.schema_check.SchemaCheck.init |
2012-02-29T01:35:35.046554Z INFO
 netlogger.analysis.schema.schema_check.SchemaCheck.check_schema.start |
2012-02-29T01:35:35.064762Z INFO   netlogger.analysis.schema.schema_check.SchemaCheck.check_schema
                                 | Current version set to: 3.1.
2012-02-29T01:35:35.064902Z ERROR  netlogger.analysis.schema.schema_check.SchemaCheck.check_schema
                                 | Schema version 3.1 found - expecting 4.0 - database admin will
 need to run upgrade tool.
2012-02-29T01:35:35.065001Z INFO   netlogger.analysis.schema.schema_check.SchemaCheck.upgrade_to_4_0
                                 | Upgrading to schema version 4.0.
```
```
Verify if the database has been converted to Version 4.x
```

**/usr/share/pegasus/sql/schema_tool.py -c connString=sqlite:////to/the/workflow/directory/with/
workflow.stampede.db**
```
2012-02-29T01:39:17.218902Z INFO   netlogger.analysis.schema.schema_check.SchemaCheck.init |
2012-02-29T01:39:17.219141Z INFO
 netlogger.analysis.schema.schema_check.SchemaCheck.check_schema.start |
2012-02-29T01:39:17.237492Z INFO   netlogger.analysis.schema.schema_check.SchemaCheck.check_schema |
 Current version set to: 4.0.
2012-02-29T01:39:17.237624Z INFO   netlogger.analysis.schema.schema_check.SchemaCheck.check_schema |
 Schema up to date.
```
```
For upgrading a MySQL database the steps remain the same. The only thing that changes is the
 connection String to the database
E.g.
```

/usr/share/pegasus/sql/schema_tool.py -u connString=mysql://username:password@server:port/dbname

After the database has been upgraded you can use either 3.x or 4.x clients to query the database with **pegasus-statistics**, as well as **pegasus-plots** and **pegasus-analyzer.**

Storing of Exitcode in the database

Kickstart records capture raw status in addition to the exitcode . The exitcode is derived from the raw status. Starting with Pegasus 4.0 release, all exitcode columns (i.e invocation and job instance table columns) are stored with the raw status by pegasus-monitord. If an exitcode is encountered while parsing the dagman log files , the value is converted to the corresponding raw status before it is stored. All user tools, pegasus-analyzer and pegasus-statistics then convert the raw status to exitcode when retrieving from the database.

Multiplier Factor

Starting with the 4.0 release, there is a multiplier factor associated with the jobs in the job_instance table. It defaults to one, unless the user associates a Pegasus profile key named **cores** with the job in the DAX. The factor can be used for getting more accurate statistics for jobs that run on multiple processors/cores or mpi jobs.

The multiplier factor is used for computing the following metrics by pegasus statistics.

• In the summary, the workflow cumulative job wall time

• In the summary, the cumulative job wall time as seen from the submit side

• In the jobs file, the multiplier factor is listed along-with the multiplied kickstart time.

• In the breakdown file, where statistics are listed per transformation the mean, min , max and average values take into account the multiplier factor.

Stampede Workflow Events

All the events generated by the system (Pegasus planner and monitoring daemon) are formatted as Netlogger BP events. The netlogger events that Pegasus generates are described in Yang schema file that can be found in the share/pegasus/schema/ directory. The stampede yang schema is described below.

Typedefs

The following typedefs are used in the yang schema to describe the certain event attributes.

- distinguished-name

```
typedef distinguished-name {
    type string;
}
```

- uuid

```
typedef uuid {
    type string {
        length "36";
        pattern
            '[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}';
    }
}
```

- intbool

```
typedef intbool {
    type uint8 {
        range "0 .. 1";
    }
}
```

- nl_ts

```
typedef nl_ts {
    type string {
        pattern
            '(\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}(\.\d+)?(Z|[\+\-]\d{2}:\d{2}))|(\d{1,9}(\.\d+)?)';
    }
}
```

- peg_inttype

```
typedef peg_inttype {
    type uint8 {
        range "0 .. 11";
    }
}
```

- peg_strtype

```
typedef peg_strtype {
    type enumeration {
        enum "unknown" {
          value 0;
        }
        enum "compute" {
          value 1;
        }
        enum "stage-in-tx" {
          value 2;
        }
        enum "stage-out-tx" {
          value 3;
        }
        enum "registration" {
          value 4;
        }
        enum "inter-site-tx" {
          value 5;
        }
        enum "create-dir" {
          value 6;
        }
        enum "staged-compute" {
          value 7;
        }
        enum "cleanup" {
          value 8;
        }
        enum "chmod" {
```

```
                value 9;
            }
            enum "dax" {
              value 10;
            }
            enum "dag" {
              value 11;
            }
        }
    }
```

- condor_jobstates

```
typedef condor_jobstates {
    type enumeration {
        enum "PRE_SCRIPT_STARTED" {
          value 0;
        }
        enum "PRE_SCRIPT_TERMINATED" {
          value 1;
        }
        enum "PRE_SCRIPT_SUCCESS" {
          value 2;
        }
        enum "PRE_SCRIPT_FAILED" {
          value 3;
        }
        enum "SUBMIT" {
          value 4;
        }
        enum "GRID_SUBMIT" {
          value 5;
        }
        enum "GLOBUS_SUBMIT" {
          value 6;
        }
        enum "SUBMIT_FAILED" {
          value 7;
        }
        enum "EXECUTE" {
          value 8;
        }
        enum "REMOTE_ERROR" {
          value 9;
        }
        enum "IMAGE_SIZE" {
          value 10;
        }
        enum "JOB_TERMINATED" {
          value 11;
        }
        enum "JOB_SUCCESS" {
          value 12;
        }
        enum "JOB_FAILURE" {
          value 13;
        }
        enum "JOB_HELD" {
          value 14;
        }
        enum "JOB_EVICTED" {
          value 15;
        }
        enum "JOB_RELEASED" {
          value 16;
        }
        enum "POST_SCRIPT_STARTED" {
          value 17;
        }
        enum "POST_SCRIPT_TERMINATED" {
          value 18;
        }
        enum "POST_SCRIPT_SUCCESS" {
          value 19;
        }
        enum "POST_SCRIPT_FAILED" {
          value 20;
        }
```

```
        }
    }
```

- condor_wfstates

```
typedef condor_wfstates {
    type enumeration {
        enum "WORKFLOW_STARTED" {
          value 0;
        }
        enum "WORKFLOW_TERMINATED" {
          value 1;
        }
    }
}
```

Groupings

Groupings are groups of common attributes that different type of events refer to. The following groupings are defined.

- **base-event** - Common components in all events
  - ts - Timestamp, ISO8601 or numeric seconds since 1/1/1970"
  - level - Severity level of event. Subset of NetLogger BP levels. For '*.end' events, if status is non-zero then level should be Error."
  - xwf.id - DAG workflow UUID

```
grouping base-event {
      description
        "Common components in all events";
      leaf ts {
        type nl_ts;
        mandatory true;
        description
          "Timestamp, ISO8601 or numeric seconds since 1/1/1970";
      }

      leaf level {
        type enumeration {
          enum "Info" {
            value 0;
          }
          enum "Error" {
            value 1;
          }
        }
        description
          "Severity level of event. "
            + "Subset of NetLogger BP levels. "
            + "For '*.end' events, if status is non-zero then level should be Error.";
      }

      leaf xwf.id {
        type uuid;
        description "DAG workflow id";
      }
  }  // grouping base-event
```

- **base-job-inst** - Common components for all job instance events
  - all attributes from base-event
  - job_inst.id - Job instance identifier i.e the submit sequence generated by monitord.
  - js.id - Jobstate identifier
  - job.id - Identifier for corresponding job in the DAG

```
grouping base-job-inst {
      description
        "Common components for all job instance events";
      uses base-event;

      leaf job_inst.id {
        type int32;
        mandatory true;
        description
          "Job instance identifier i.e the submit sequence generated by monitord";
      }
```

```
        leaf js.id {
          type int32;
          description "Jobstate identifier";
        }

        leaf job.id {
          type string;
          mandatory true;
          description
            "Identifier for corresponding job in the DAG";
        }
      }
```

- **sched-job-inst** - Scheduled job instance.
  - all attributes from base-job-inst
  - sched.id - Identifier for job in scheduler

```
grouping sched-job-inst {
      description "Scheduled job instance";
      uses base-job-inst;

      leaf sched.id {
        type string;
        mandatory true;
        description
          "Identifier for job in scheduler";
      }
}
```

- **base-metadata**
  - uses
  - key
  - value

```
grouping base-metadata {
      description
        "Common components for all metadata events that describe metadata for an entity.";
      uses base-event;

      leaf key {
        type string;
        mandatory true;
        description
          "Key for the metadata tuple";
      }

      leaf value {
        type string;
        description
          "Corresponding value of the key";
      }
}  // grouping base-metadata
```

Events

The system generates following types of events, that are described below.

- stampede.wf.plan
- stampede.static.start
- stampede.static.end
- stampede.xwf.start
- stampede.xwf.end
- stampede.task.info
- stampede.task.edge
- stampede.wf.map.task_job
- stampede.xwf.map.subwf_job
- stampede.job.info
- stampede.job.edge
- stampede.job_inst.pre.start

- stampede.job_inst.pre.term
- stampede.job_inst.pre.end
- stampede.job_inst.submit.start
- stampede.job_inst.submit.end
- stampede.job_inst.held.start
- stampede.job_inst.held.end
- stampede.job_inst.main.start
- stampede.job_inst.main.term
- stampede.job_inst.main.end
- stampede.job_inst.post.start
- stampede.job_inst.post.term
- stampede.job_inst.post.end
- stampede.job_inst.host.info
- stampede.job_inst.image.info
- stampede.inv.start
- stampede.inv.end
- stampede.static.meta.start
- stampede.static.meta.end
- stampede.xwf.meta
- stampede.task.meta
- stampede.rc.meta
- stampede.wf.map.file

The events are described in detail below

- **stampede.wf.plan**

```
container stampede.wf.plan {
        uses base-event;

        leaf submit.hostname {
          type inet:host;
          mandatory true;
          description
            "The hostname of the Pegasus submit host";
        }

        leaf dax.label {
          type string;
          default "workflow";
          description
            "Label for abstract workflow specification";
        }

        leaf dax.index {
          type string;
          default "workflow";
          description
            "Index for the DAX";
        }

        leaf dax.version {
          type string;
          mandatory true;
          description
            "Version number for DAX";
        }

        leaf dax.file {
          type string;
          mandatory true;
          description
            "Filename for for the DAX";
        }

        leaf dag.file.name {
          type string;
```

```
                              mandatory true;
                              description
                                "Filename for the DAG";
                            }

                            leaf planner.version {
                              type string;
                              mandatory true;
                              description
                                "Version string for Pegasus planner, e.g. 3.0.0cvs";
                            }

                            leaf grid_dn {
                              type distinguished-name;
                              description
                                "Grid DN of submitter";
                            }

                            leaf user {
                              type string;
                              description
                                "User name of submitter";
                            }

                            leaf submit.dir {
                              type string;
                              mandatory true;
                              description
                                "Directory path from which workflow was submitted";
                            }

                            leaf argv {
                              type string;
                              description
                                "All arguments given to planner on command-line";
                            }

                            leaf parent.xwf.id {
                              type uuid;
                              description
                                "Parent workflow in DAG, if any";
                            }

                            leaf root.xwf.id {
                              type string;
                              mandatory true;
                              description
                                "Root of workflow hierarchy, in DAG. "
                                  + "Use this workflow's UUID if it is the root";
                            }
              }  // container stampede.wf.plan
```

- **stampede.static.start**

```
container stampede.static.start {
      uses base-event;
}
```

- **stampede.static.end**

```
container stampede.static.end {
    uses base-event;
}  //
```

- **stampede.xwf.start**

```
container stampede.xwf.start {
              uses base-event;

              leaf restart_count {
                type uint32;
                mandatory true;
                description
                  "Number of times workflow was restarted (due to failures)";
              }
}  // container stampede.xwf.start
```

- **stampede.xwf.end**

```
container stampede.xwf.end {
```

```
                 uses base-event;

                 leaf restart_count {
                   type uint32;
                   mandatory true;
                   description
                     "Number of times workflow was restarted (due to failures)";
                 }

                 leaf status {
                   type int16;
                   mandatory true;
                   description
                     "Status of workflow. 0=success, -1=failure";
                 }
    }  // container stampede.xwf.end
```

- **stampede.task.info**

```
container stampede.task.info {
             description
               "Information about task in DAX";
             uses base-event;

             leaf transformation {
               type string;
               mandatory true;
               description
                 "Logical name of the underlying executable";
             }

             leaf argv {
               type string;
               description
                 "All arguments given to transformation on command-line";
             }

             leaf type {
               type peg_inttype;
               mandatory true;
               description "Type of task";
             }

             leaf type_desc {
               type peg_strtype;
               mandatory true;
               description
                 "String description of task type";
             }

             leaf task.id {
               type string;
               mandatory true;
               description
                 "Identifier for this task in the DAX";
             }
           }  // container stampede.task.info
```

- **stampede.task.edge**

```
container stampede.task.edge {
             description
               "Represents child/parent relationship between two tasks in DAX";
             uses base-event;

             leaf parent.task.id {
               type string;
               mandatory true;
               description "Parent task";
             }

             leaf child.task.id {
               type string;
               mandatory true;
               description "Child task";
             }
   }  // container stampede.task.edge
```

- **stampede.wf.map.task_job**

```
container stampede.wf.map.task_job {

        description
          "Relates a DAX task to a DAG job.";
        uses base-event;

        leaf task.id {
          type string;
          mandatory true;
          description
            "Identifier for the task in the DAX";
        }

        leaf job.id {
          type string;
          mandatory true;
          description
            "Identifier for corresponding job in the DAG";
        }
} // container stampede.wf.map.task_job
```

- **stampede.xwf.map.subwf_job**

```
container stampede.xwf.map.subwf_job {

        description
          "Relates a sub workflow to the corresponding job instance";
        uses base-event;

        leaf subwf.id {
          type string;
          mandatory true;
          description
            "Sub Workflow Identified / UUID";
        }

        leaf job.id {
          type string;
          mandatory true;
          description
            "Identifier for corresponding job in the DAG";
        }

        leaf job_inst.id {
          type int32;
          mandatory true;
          description
            "Job instance identifier i.e the submit sequence generated by monitord";
        }
} // container stampede.xwf.map.subwf_job
```

- **stampede.job.info**

```
container stampede.job.info {

        description
          "A description of a job in the DAG";
        uses base-event;

        leaf job.id {
          type string;
          mandatory true;
          description
            "Identifier for job in the DAG";
        }

        leaf submit_file {
          type string;
          mandatory true;
          description
            "Name of file being submitted to the scheduler";
        }

        leaf type {
          type peg_inttype;
          mandatory true;
          description "Type of task";
        }
```

```
                leaf type_desc {
                  type peg_strtype;
                  mandatory true;
                  description
                    "String description of task type";
                }

                leaf clustered {
                  type intbool;
                  mandatory true;
                  description
                    "Whether job is clustered or not";
                }

                leaf max_retries {
                  type uint32;
                  mandatory true;
                  description
                    "How many retries are allowed for this job before giving up";
                }

                leaf task_count {
                  type uint32;
                  mandatory true;
                  description
                    "Number of DAX tasks for this job. "
                      + "Auxiliary jobs without a task in the DAX will have the value '0'";
                }

                leaf executable {
                  type string;
                  mandatory true;
                  description
                    "Program to execute";
                }

                leaf argv {
                  type string;
                  description
                    "All arguments given to executable (on command-line)";
                }
        }  // container stampede.job.info
```

- **stampede.job.edge**

```
container stampede.job.edge {

                description
                  "Parent/child relationship between two jobs in the DAG";
                uses base-event;

                leaf parent.job.id {
                  type string;
                  mandatory true;
                  description "Parent job";
                }

                leaf child.job.id {
                  type string;
                  mandatory true;
                  description "Child job";
                }
        }  // container stampede.job.edge
```

- **stampede.job_inst.pre.start**

```
container stampede.job_inst.pre.start {

                description
                  "Start of a prescript for a job instance";
                uses base-job-inst;
        }  // container stampede.job_inst.pre.start
```

- **stampede.job_inst.pre.term**

```
container stampede.job_inst.pre.term {
                description
                  "Job prescript is terminated (success or failure not yet known)";
```

```
}   // container stampede.job_inst.pre.term
```

- **stampede.job_inst.pre.end**

```
container stampede.job_inst.pre.end {
            description
              "End of a prescript for a job instance";
            uses base-job-inst;

            leaf status {
              type int32;
              mandatory true;
              description
                "Status of prescript. 0 is success, -1 is error";
            }

            leaf exitcode {
              type int32;
              mandatory true;
              description
                "the exitcode with which the prescript exited";
            }
}   // container stampede.job_inst.pre.end
```

- **stampede.job_inst.submit.start**

```
container stampede.job_inst.submit.start {
            description
              "When job instance is going to be submitted. "
                + "Scheduler job id is not yet known";
            uses sched-job-inst;
}   // container stampede.job_inst.submit.start
```

- **stampede.job_inst.submit.end**

```
container stampede.job_inst.submit.end {
            description
              "When executable job is submitted";
            uses sched-job-inst;

            leaf status {
              type int16;
              mandatory true;
              description
                "Status of workflow. 0=success, -1=failure";
            }
}   // container stampede.job_inst.submit.end
```

- **stampede.job_inst.held.start**

```
container stampede.job_inst.held.start {
            description
              "When Condor holds the jobs";
            uses sched-job-inst;
}   // container stampede.job_inst.held.start
```

- **stampede.job_inst.held.end**

```
container stampede.job_inst.held.end {
            description
              "When the job is released after being held";
            uses sched-job-inst;

            leaf status {
              type int16;
              mandatory true;
              description
                "Status of workflow. 0=success, -1=failure";
            }
}   // container stampede.job_inst.held.end
```

- **stampede.job_inst.main.start**

```
container stampede.job_inst.main.start {
            description
              "Start of execution of a scheduler job";
            uses sched-job-inst;

            leaf stdin.file {
              type string;
              description
                "Path to file containing standard input of job";
```

```
                  }

                  leaf stdout.file {
                    type string;
                    mandatory true;
                    description
                      "Path to file containing standard output of job";
                  }

                  leaf stderr.file {
                    type string;
                    mandatory true;
                    description
                      "Path to file containing standard error of job";
                  }
     }  // container stampede.job_inst.main.start
```

- **stampede.job_inst.main.term**

```
  container stampede.job_inst.main.term {
                  description
                    "Job is terminated (success or failure not yet known)";
                  uses sched-job-inst;

                  leaf status {
                    type int32;
                    mandatory true;
                    description
                      "Execution status. 0=means job terminated, -1=job was evicted, not terminated";
                  }
     }  // container stampede.job_inst.main.term
```

- **stampede.job_inst.main.end**

```
  container stampede.job_inst.main.end {
                  description
                    "End of main part of scheduler job";
                  uses sched-job-inst;

                  leaf stdin.file {
                    type string;
                    description
                      "Path to file containing standard input of job";
                  }

                  leaf stdout.file {
                    type string;
                    mandatory true;
                    description
                      "Path to file containing standard output of job";
                  }

                  leaf stdout.text {
                    type string;
                    description
                      "Text containing output of job";
                  }

                  leaf stderr.file {
                    type string;
                    mandatory true;
                    description
                      "Path to file containing standard error of job";
                  }

                  leaf stderr.text {
                    type string;
                    description
                      "Text containing standard error of job";
                  }

                  leaf user {
                    type string;
                    description
                      "Scheduler's name for user";
                  }

                  leaf site {
```

```
                         type string;
                         mandatory true;
                         description
                           "DAX name for the site at which the job ran";
                       }

                       leaf work_dir {
                         type string;
                         description
                           "Path to working directory";
                       }

                       leaf local.dur {
                         type decimal64 {
                           fraction-digits 6;
                         }
                         units "seconds";
                         description
                           "Duration as seen at the local node";
                       }

                       leaf status {
                         type int32;
                         mandatory true;
                         description
                           "Execution status. 0=success, -1=failure";
                       }

                       leaf exitcode {
                         type int32;
                         mandatory true;
                         description
                           "the exitcode with which the executable exited";
                       }

                       leaf multiplier_factor {
                         type int32;
                         mandatory true;
                         description
                           "the multiplier factor for use in statistics";
                       }

                       leaf cluster.start {
                         type nl_ts;
                         description
                           "When the enclosing cluster started";
                       }

                       leaf cluster.dur {
                         type decimal64 {
                           fraction-digits 6;
                         }
                         units "seconds";
                         description
                           "Duration of enclosing cluster";
                       }
    }  // container stampede.job_inst.main.end
```

- **stampede.job_inst.post.start**

```
container stampede.job_inst.post.start {
           description
             "Start of a postscript for a job instance";
           uses sched-job-inst;
    }  // container stampede.job_inst.post.start
```

- **stampede.job_inst.post.term**

```
container stampede.job_inst.post.term {
           description
             "Job postscript is terminated (success or failure not yet known)";
           uses sched-job-inst;
    }  // container stampede.job_inst.post.term
```

- **stampede.job_inst.post.end**

```
container stampede.job_inst.post.end {
           description
             "End of a postscript for a job instance";
```

```
                uses sched-job-inst;

                leaf status {
                  type int32;
                  mandatory true;
                  description
                    "Status of postscript. 0 is success, -1=failure";
                }

                leaf exitcode {
                  type int32;
                  mandatory true;
                  description
                    "the exitcode with which the postscript exited";
                }
   }  // container stampede.job_inst.post.end
```

- **stampede.job_inst.host.info**

```
   container stampede.job_inst.host.info {
                description
                  "Host information associated with a job instance";
                uses base-job-inst;

                leaf site {
                  type string;
                  mandatory true;
                  description "Site name";
                }

                leaf hostname {
                  type inet:host;
                  mandatory true;
                  description "Host name";
                }

                leaf ip {
                  type inet:ip-address;
                  mandatory true;
                  description "IP address";
                }

                leaf total_memory {
                  type uint64;
                  description
                    "Total RAM on host";
                }

                leaf uname {
                  type string;
                  description
                    "Operating system name";
                }
   }  // container stampede.job_inst.host.info
```

- **stampede.job_inst.image.info**

```
   container stampede.job_inst.image.info {
                description
                  "Image size associated with a job instance";
                uses base-job-inst;

                leaf size {
                  type uint64;
                  description "Image size";
                }

                leaf sched.id {
                  type string;
                  mandatory true;
                  description
                    "Identifier for job in scheduler";
                }
   }  // container stampede.job_inst.image.info
```

- **stampede.inv.start**

```
   container stampede.inv.start {
                description
```

```
                "Start of an invocation";
              uses base-event;

              leaf job_inst.id {
                type int32;
                mandatory true;
                description
                  "Job instance identifier i.e the submit sequence generated by monitord";
              }

              leaf job.id {
                type string;
                mandatory true;
                description
                  "Identifier for corresponding job in the DAG";
              }

              leaf inv.id {
                type int32;
                mandatory true;
                description
                  "Identifier for invocation. "
                    + "Sequence number, with -1=prescript and -2=postscript";
              }
    } // container stampede.inv.start
```

- **stampede.inv.end**

```
  container stampede.inv.end {
              description
                "End of an invocation";
              uses base-event;

              leaf job_inst.id {
                type int32;
                mandatory true;
                description
                  "Job instance identifier i.e the submit sequence generated by monitord";
              }

              leaf inv.id {
                type int32;
                mandatory true;
                description
                  "Identifier for invocation. "
                    + "Sequence number, with -1=prescript and -2=postscript";
              }

              leaf job.id {
                type string;
                mandatory true;
                description
                  "Identifier for corresponding job in the DAG";
              }

              leaf start_time {
                type nl_ts;
                description
                  "The start time of the event";
              }

              leaf dur {
                type decimal64 {
                  fraction-digits 6;
                }
                units "seconds";
                description
                  "Duration of invocation";
              }

              leaf remote_cpu_time {
                type decimal64 {
                  fraction-digits 6;
                }
                units "seconds";
                description
                  "remote CPU time computed as the stime  + utime";
              }
```

```
                leaf exitcode {
                  type int32;
                  description
                    "the exitcode with which the              executable exited";
                }

                leaf transformation {
                  type string;
                  mandatory true;
                  description
                    "Transformation associated with this invocation";
                }

                leaf executable {
                  type string;
                  mandatory true;
                  description
                    "Program executed for this invocation";
                }

                leaf argv {
                  type string;
                  description
                    "All arguments given to executable on command-line";
                }

                leaf task.id {
                  type string;
                  description
                    "Identifier for related task in the DAX";
                }
    }  // container stampede.inv.end
```

- **stampede.static.meta.start**

```
container stampede.static.meta.start {
            uses base-event;
}  // container stampede.static.meta.start
```

- **stampede.static.meta.end**

```
container stampede.static.meta.end {
            uses base-event;
} // container stampede.static.meta.end
```

- **stampede.xwf.meta**

```
container stampede.xwf.meta {
            description
              "Metadata associated with a workflow";
            uses base-metadata;
}  // container stampede.xwf.meta
```

- **stampede.task.meta**

```
container stampede.task.meta {
            description
              "Metadata associated with a task";
            uses base-metadata;

            leaf task.id {
              type string;
              description
                "Identifier for related task in the DAX";
            }
}  // container stampede.task.meta
```

- **stampede.rc.meta**

```
container stampede.rc.meta {
            description
              "Metadata associated with a file in the replica catalog";
            uses base-metadata;

            leaf lfn.id {
              type string;
              description
                "Logical File Identifier for the file";
            }
}  // container stampede.rc.meta
```

- **stampede.wf.map.file**

```
container stampede.wf.map.file {
            description
              "Event that captures what task generates or consumes a particular file";
            uses base-event;

            leaf lfn.id {
              type string;
              description
                "Logical File Identifier for the file";
            }

            leaf task.id {
              type string;
              description
                "Identifier for related task in the DAX";
            }
}  // container stampede.wf.map.file
```

Chapter 7. Execution Environments

Pegasus supports a number of execution environments. An execution environment is a setup where jobs from a workflow are running.

Localhost

In this configuration, Pegasus schedules the jobs to run locally on the submit host. Running locally is a good approach for smaller workflows, testing workflows, and for demonstations such as the Pegasus tutorial. Pegasus supports two methods of local execution: local HTCondor pool, and shell planner. The former is preferred as the latter does not support all Pegasus' features (such as notifications).

Running on a local HTCondor pool is achieved by executing the workflow on site local (**--sites local** option to pegasus-plan). The site "local" is a reserved site in Pegasus and results in the jobs to run on the submit host in HTCondor universe local. The site catalog can be left very simple in this case:

```
<?xml version="1.0" encoding="UTF-8"?>
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog http://pegasus.isi.edu/
schema/sc-4.0.xsd"
             version="4.0">

    <site  handle="local" arch="x86_64" os="LINUX">
        <directory type="shared-scratch" path="/tmp/wf/work">
            <file-server operation="all" url="file:///tmp/wf/work"/>
        </directory>
        <directory type="local-storage" path="/tmp/wf/storage">
            <file-server operation="all" url="file:///tmp/wf/storage"/>
        </directory>
    </site>

</sitecatalog>
```

The simplest execution environment does not involve HTCondor. Pegasus is capable of planning small workflows for local execution using a shell planner. Please refer to the `share/pegasus/examples` directory in your Pegasus installation, the shell planner's documentation section, or the tutorials, for details.

Condor Pool

A HTCondor pool is a set of machines that use HTCondor for resource management. A HTCondor pool can be a cluster of dedicated machines or a set of distributively owned machines. Pegasus can generate concrete workflows that can be executed on a HTCondor pool.

Figure 7.1. The distributed resources appear to be part of a HTCondor pool.
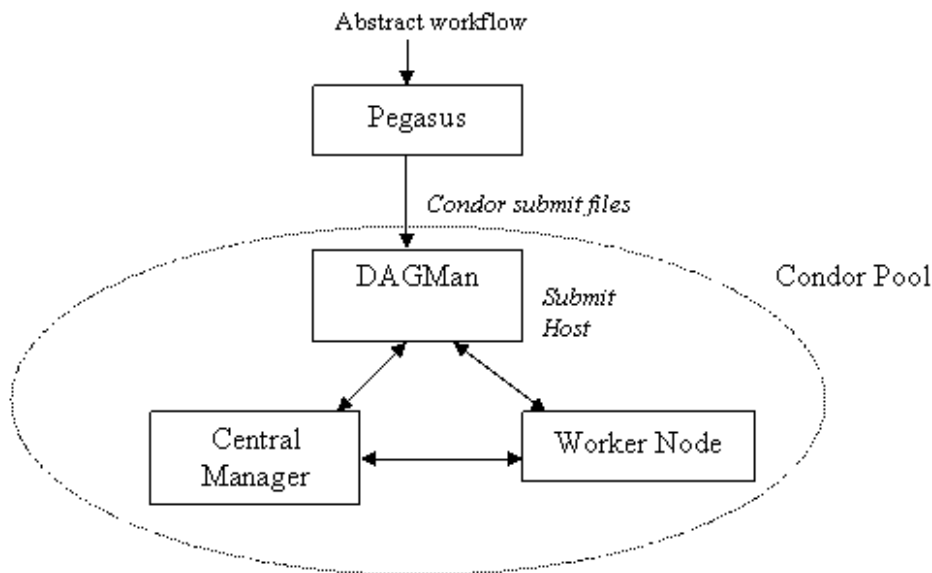
The workflow is submitted using DAGMan from one of the job submission machines in the HTCondor pool. It is the responsibility of the Central Manager of the pool to match the task in the workflow submitted by DAGMan to the execution machines in the pool. This matching process can be guided by including HTCondor specific attributes in the submit files of the tasks. If the user wants to execute the workflow on the execution machines (worker nodes) in a HTCondor pool, there should be a resource defined in the site catalog which represents these execution machines. The universe attribute of the resource should be vanilla. There can be multiple resources associated with a single HTCondor pool, where each resource identifies a subset of machine (worker nodes) in the pool.

When running on a HTCondor pool, the user has to decide how Pegasus should transfer data. Please see the Data Staging Configuration for the options. The easiest is to use **condorio** as that mode does not require any extra setup - HTCondor will do the transfers using the existing HTCondor daemons. For an example of this mode see the example workflow in `share/pegasus/examples/condor-blackdiamond-condorio/` . In HTCondorio mode, the site catalog for the execution site is very simple as storage is provided by HTCondor:

```
<?xml version="1.0" encoding="UTF-8"?>
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog http://pegasus.isi.edu/
schema/sc-4.0.xsd"
             version="4.0">

    <site  handle="local" arch="x86_64" os="LINUX">
        <directory type="shared-scratch" path="/tmp/wf/work">
            <file-server operation="all" url="file:///tmp/wf/work"/>
        </directory>
        <directory type="local-storage" path="/tmp/wf/storage">
            <file-server operation="all" url="file:///tmp/wf/storage"/>
        </directory>
    </site>

    <site  handle="condorpool" arch="x86_64" os="LINUX">
        <profile namespace="pegasus" key="style" >condor</profile>
        <profile namespace="condor" key="universe" >vanilla</profile>
    </site>

</sitecatalog>
```

There is a set of HTCondor profiles which are used commonly when running Pegasus workflows. You may have to set some or all of these depending on the setup of the HTCondor pool:

```
<!-- Change the style to HTCondor for jobs to be executed in the HTCondor Pool.
     By default, Pegasus creates jobs suitable for grid execution. -->
<profile namespace="pegasus" key="style">condor</profile>

<!-- Change the universe to vanilla to make the jobs go to remote compute
     nodes. The default is local which will only run jobs on the submit host -->
<profile namespace="condor" key="universe" >vanilla</profhile>

<!-- The requirements expression allows you to limit where your jobs go -->
<profile namespace="condor" key="requirements">(Target.FileSystemDomain !=
&quot;yggdrasil.isi.edu&quot;)</profile>

<!-- The following two profiles forces HTCondor to always transfer files. This
     has to be used if the pool does not have a shared filesystem -->
<profile namespace="condor" key="should_transfer_files">True</profile>
<profile namespace="condor" key="when_to_transfer_output">ON_EXIT</profile>
```

Glideins

In this section we describe how machines from different administrative domains and supercomputing centers can be dynamically added to a HTCondor pool for certain timeframe. These machines join the HTCondor pool temporarily and can be used to execute jobs in a non preemptive manner. This functionality is achieved using a HTCondor feature called **glideins** (see http://cs.wisc.edu/condor/glidein [http://cs.wisc.edu/condor/glidein]) . The startd daemon is the HTCondor daemon which provides the compute slots and runs the jobs. In the glidein case, the submit machine is usually a static machine and the glideins are told confiugued to report to that submit machine. The glideins can be submitted to any type of resource: a GRAM enabled cluster, a campus cluster, a cloud environment such as Amazon AWS, or even another HTCondor cluster.

Tip

As glideins are usually coming from different compute resource, and/or the glideins are running in an administrative domain different from the submit node, there is usually no shared filesystem available. Thus the most common data staging modes are **condorio** and **nonsharedfs** .

There are many useful tools which submits and manages glideins for you:

- GlideinWMS [http://www.uscms.org/SoftwareComputing/Grid/WMS/glideinWMS/] is a tool and host environment used mostly on the Open Science Grid [http://www.opensciencegrid.org/].

- CorralWMS [http://pegasus.isi.edu/projects/corralwms] is a personal frontend for GlideinWMS. CorralWMS was developed by the Pegasus team and works very well for high throughput workflows.

- condor_glidein [http://research.cs.wisc.edu/condor/manual/v7.6/condor_glidein.html] is a simple glidein tool for Globus GRAM clusters. condor_glidein is shipped with HTCondor.

- Glideins can also be created by hand or scripts. This is a useful solution for example for cluster which have no external job submit mechanisms or do not allow outside networking.

CondorC

Using HTCondorC users can submit workflows to remote HTCondor pools. HTCondorC is a HTCondor specific solution for remote submission that does not involve the setting up a GRAM on the headnode. To enable HTCondorC submission to a site, user needs to associate pegasus profile key named style with value as HTCondorc. In case, the remote HTCondor pool does not have a shared filesytem between the nodes making up the pool, users should use pegasus in the HTCondorio data configuration. In this mode, all the data is staged to the remote node in the HTCondor pool using HTCondor File transfers and is executed using PegasusLite.

A sample site catalog for submission to a HTCondorC enabled site is listed below

```
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog http://pegasus.isi.edu/
schema/sc-4.0.xsd"
```

```
          version="4.0">

    <site  handle="local" arch="x86_64" os="LINUX">
        <directory type="shared-scratch" path="/tmp/wf/work">
            <file-server operation="all" url="file:///tmp/wf/work"/>
        </directory>
        <directory type="local-storage" path="/tmp/wf/storage">
            <file-server operation="all" url="file:///tmp/wf/storage"/>
        </directory>
    </site>

    <site  handle="condorcpool" arch="x86_86" os="LINUX">
        <!-- the grid gateway entries are used to designate
             the remote schedd for the HTCondorC pool -->
        <grid type="condor" contact="ccg-condorctest.isi.edu" scheduler="Condor"
 jobtype="compute" />
        <grid type="condor" contact="ccg-condorctest.isi.edu" scheduler="Condor"
 jobtype="auxillary" />

        <!-- enable submission using HTCondorc -->
        <profile namespace="pegasus" key="style">condorc</profile>

        <!-- specify which HTCondor collector to use.
             If not specified defaults to remote schedd specified in grid gateway -->
        <profile namespace="condor" key="condor_collector">condorc-collector.isi.edu</profile>

        <profile namespace="condor" key="should_transfer_files">Yes</profile>
        <profile namespace="condor" key="when_to_transfer_output">ON_EXIT</profile>
        <profile namespace="env" key="PEGASUS_HOME" >/usr</profile>
        <profile namespace="condor" key="universe">vanilla</profile>

    </site>

</sitecatalog>
```

To enable PegasusLite in HTCondorIO mode, users should set the following in their properties

```
# pegasus properties
pegasus.data.configuration     condorio
```

Cloud (Amazon EC2/S3, Google Cloud, ...)

Figure 7.2. Cloud Sample Site Layout
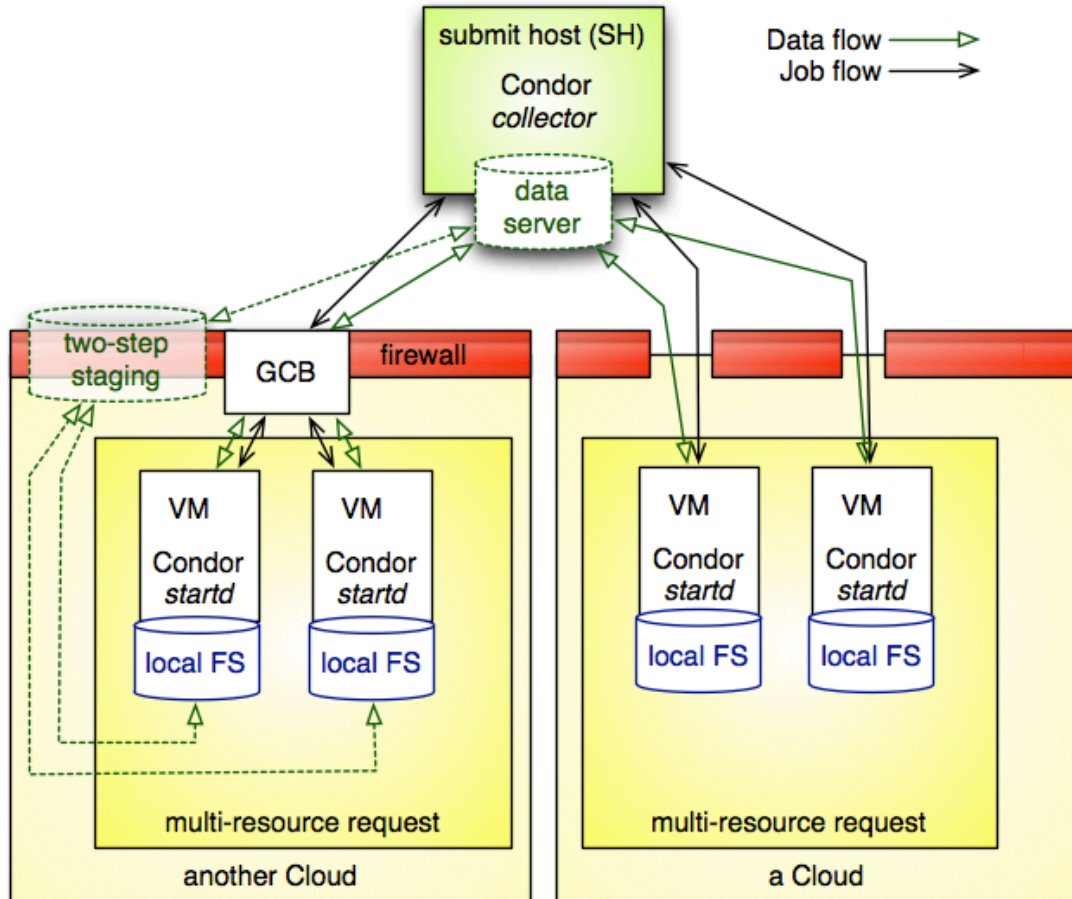
This figure shows a sample environment for executing Pegasus across multiple clouds. At this point, it is up to the user to provision the remote resources with a proper VM image that includes a HTCondor worker that is configured to report back to a HTCondor master, which can be located inside one of the clouds, or outside the cloud.

The submit host is the point where a user submits Pegasus workflows for execution. This site typically runs a HTCondor collector to gather resource announcements, or is part of a larger HTCondor pool that collects these announcements. HTCondor makes the remote resources available to the submit host's HTCondor installation.

The figure above shows the way Pegasus WMS is deployed in cloud computing resources, ignoring how these resources were provisioned. The provisioning request shows multiple resources per provisioning request.

The initial stage-in and final stage-out of application data into and out of the node set is part of any Pegasus-planned workflow. Several configuration options exist in Pegasus to deal with the dynamics of push and pull of data, and when to stage data. In many use-cases, some form of external access to or from the shared file system that is visible to the application workflow is required to facilitate successful data staging. However, Pegasus is prepared to deal with a set of boundary cases.

The data server in the figure is shown at the submit host. This is not a strict requirement. The data server for consumed data and data products may both be different and external to the submit host, or one of the object storage solution offered by the cloud providers

Once resources begin appearing in the pool managed by the submit machine's HTCondor collector, the application workflow can be submitted to HTCondor. A HTCondor DAGMan will manage the application workflow execution. Pegasus run-time tools obtain timing-, performance and provenance information as the application workflow is executed. At this point, it is the user's responsibility to de-provision the allocated resources.

In the figure, the cloud resources on the right side are assumed to have uninhibited outside connectivity. This enables the HTCondor I/O to communicate with the resources. The right side includes a setup where the worker nodes use all private IP, but have out-going connectivity and a NAT router to talk to the internet. The *Condor connection broker* (CCB) facilitates this setup almost effortlessly.

The left side shows a more difficult setup where the connectivity is fully firewalled without any connectivity except to in-site nodes. In this case, a proxy server process, the *generic connection broker* (GCB), needs to be set up in the DMZ of the cloud site to facilitate HTCondor I/O between the submit host and worker nodes.

If the cloud supports data storage servers, Pegasus is starting to support workflows that require staging in two steps: Consumed data is first staged to a data server in the remote site's DMZ, and then a second staging task moves the data from the data server to the worker node where the job runs. For staging out, data needs to be first staged from the job's worker node to the site's data server, and possibly from there to another data server external to the site. Pegasus is capable to plan both steps: Normal staging to the site's data server, and the worker-node staging from and to the site's data server as part of the job.

Amazon EC2

There are many different ways to set up an execution environment in Amazon EC2. The easiest way is to use a submit machine outside the cloud, and to provision several worker nodes and a file server node in the cloud as shown here:

Figure 7.3. Amazon EC2
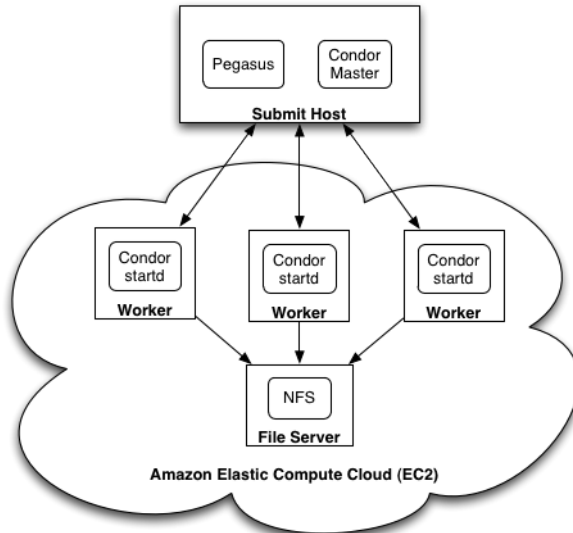
The submit machine runs Pegasus and a HTCondor master (collector, schedd, negotiator). The workers run a HTCondor startd. And the file server node exports an NFS file system. The startd on the workers is configured to connect to the master running outside the cloud, and the workers also mount the NFS file system. More information on setting up HTCondor for this environment can be found at http://www.isi.edu/~gideon/condor-ec2 [http://www.isi.edu/~gideon/condor-ec2/].

The site catalog entry for this configuration is similar to what you would create for running on a local Condor pool with a shared file system.

Google Cloud Platform

Using the Google Cloud Platform is just like any other cloud platform. You can choose to host the central manager / submit host inside the cloud or outside. The compute VMs will have HTCondor installed and configured to join the pool managed by the central manager.

Google Storage is supported using gsutil. First, create a .boto file by running:

```
gsutil config
```

Then, use a site catalog which specifies which .boto file to use. You can then use gs:// URLs in your workflow. Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog
                 http://pegasus.isi.edu/schema/sc-4.0.xsd" version="4.0">

    <site  handle="local" arch="x86_64" os="LINUX">
        <directory type="shared-scratch" path="/tmp">
            <file-server operation="all" url="file:///tmp"/>
        </directory>
        <profile namespace="env" key="PATH">/opt/gsutil:/usr/bin:/bin</profile>

    </site>



    <!-- compute site -->
    <site  handle="condorpool" arch="x86_86" os="LINUX">
        <profile namespace="pegasus" key="style" >condor</profile>
        <profile namespace="condor" key="universe" >vanilla</profile>
    </site>

    <!-- storage sites have to be in the site catalog, just liek a compute site -->
    <site  handle="google_storage" arch="x86_64" os="LINUX">
        <directory type="shared-scratch" path="/my-bucket/scratch">
            <file-server operation="all" url="gs://my-bucket/scratch"/>
        </directory>
        <directory type="local-storage" path="/my-bucket/outputs">
            <file-server operation="all" url="gs://my-bucket/outputs"/>
        </directory>
        <profile namespace="pegasus" key="BOTO_CONFIG">/home/myuser/.boto</profile>
    </site>

</sitecatalog>
```

Remote Cluster using PyGlidein

Glideins (HTCondor pilot jobs) provide an efficient solution for high-throughput workflows. The glideins are submitted to the remote cluster scheduler, and once started up, makes it appear like your HTCondor pool extends into the remote cluster. HTCondor can then schedule the jobs to the remote compute node in the same way it would schedule jobs to local compute nodes.

Some infrastructures, such as Open Science Grid, provide infrastructure level glidein solutions, such as GlideinWMS. Another solution is BOSCO. For some more custom setups, pyglidein [https://github.com/WIPACrepo/pyglidein] from the IceCube [http://icecube.wisc.edu/] project provides a nice framework. The architecture consists on a server on the submit host, which job it is to determining the demand. On the remote resource, the client can be invoked for example via cron, and submits directly to HTCondor, SLURM and PBS schedulers. This makes pyglidein very flexible and works well for example if the resource requires two-factor authentication.

Figure 7.4. pyglidein overview
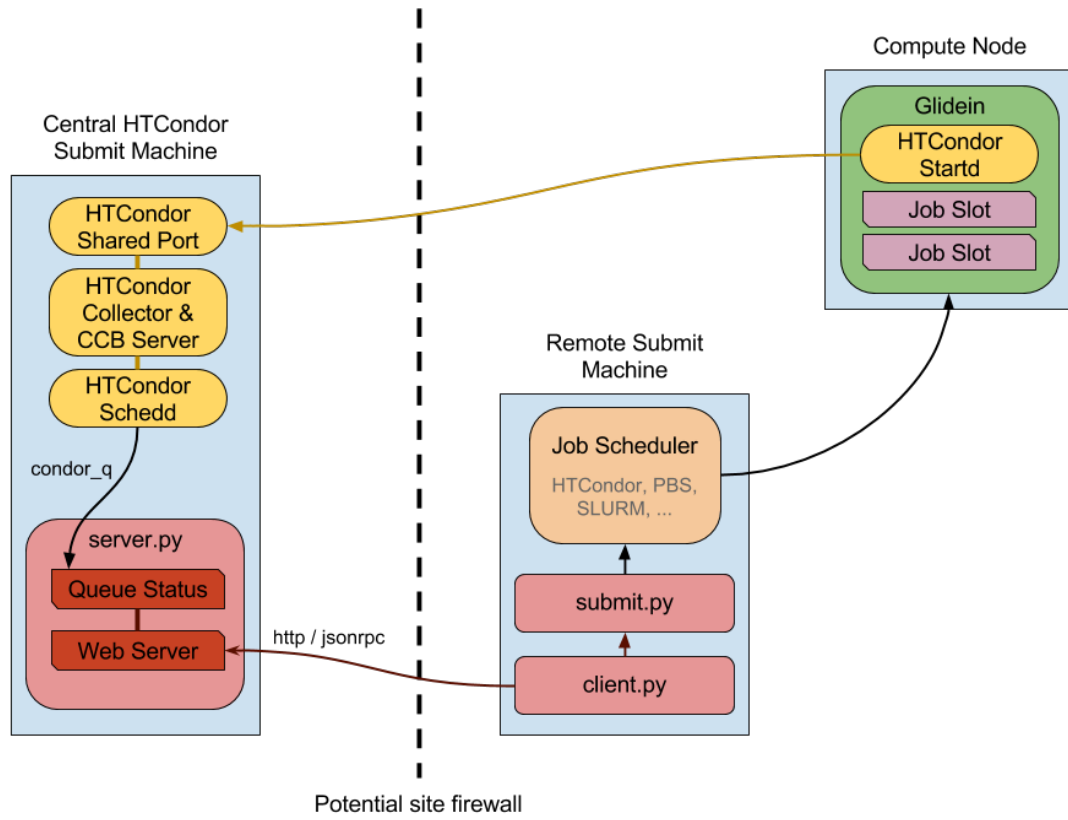
To get started with pyglidein, check out a copy of the Git repository on both your submit host as well as the cluster you want to glidein to. Starting with the submit host, first make sure you have HTCondor configured for PASS-WORD [http://research.cs.wisc.edu/htcondor/manual/current/3_8Security.html#SECTION00483400000000000000] authentication. Make a copy of the HTCondor pool password file. You will need it on the cluster, and it is a binary file, so make sure you cp instead of a copy-and-paste of the file contents. To get the server started:

```
./server.py --port 11001
```

By default, the pyglidein server will use all jobs in the system to determine if glideins are needed. If you want user jobs to explicitly let us know they want glideins, you can pass a constraint for the server to use. For example, jobs could have the +*WantStampede2 = True* attribute, and then we could start the server with:

```
./server.py --port 11001 --constraint "'WantStampede2 == True'"
```

One the server is running, you can check status by pointing a web browser to it.

Next step is to create a *glidein.tar.gz* file containing the HTCondor binaries, our pool password file, and a modified job wrapper script. This can be accomplished by building HTCondor with the *create_glidein_tarball.py* script, but first we need to modify *glidein_template/*. Start by copying your pool password file over the existing *passwdfile* file.

Edit *user_job_wrapper.sh*. We don't need most of it, so edit it to read:

```
#!/bin/bash

# This script is started just before the user job
# It is referenced by the USER_JOB_WRAPPER

export HOME=$PWD
```

```
# fix PATH and LD_LIBRARY_PATH
export PATH=$PATH:/usr/bin:/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib64:/usr/local/lib:/usr/lib64:/usr/lib:/usr/
lib/x86_64-linux-gnu:/lib64:/lib:/lib/x86_64-linux-gnu

GLIDEIN_DIR=$GLIDEIN_LOCAL_TMP_DIR
if [ ! -d $GLIDEIN_DIR ]; then
    GLIDEIN_DIR=$PWD
fi
JOB_WRAPPER="${GLIDEIN_DIR}/job_wrapper.sh"

# fall through to next/default job wrapper
if [ ! -e $JOB_WRAPPER ]; then
    exec "$@"
else
    exec ${JOB_WRAPPER} "$@"
fi
```

Create the glidein.tar.gz by running:

```
python create_glidein_tarball.py
```

Once you have the glidein.tar.gz file, copy it to the Git checkout you have on the remote cluster. Then move over there for the remaining steps. Create a configuration file for your glidein under *configs/*. Here is an example for TACC Stampede2:

```
[Mode]
debug = True

[Glidein]
address = http://workflow.isi.edu:11001/jsonrpc
site = TACC-Stampede2
tarball = /home1/00384/rynge/git/pyglidein/glidein.tar.gz

[Cluster]
user = rynge
os = RHEL7
scheduler = slurm
submit_command = sbatch
walltime_hrs = 48
max_total_jobs = 10
max_idle_jobs = 1
limit_per_submit = 1

gpu_only = False
whole_node = True
whole_node_cpus = 1
whole_node_memory = 96000
whole_node_disk = 30000
whole_node_gpus = 0
group_jobs = False
partition = normal
running_cmd = squeue -u $USER -t RUNNING -p normal -h | wc -l
idle_cmd = squeue -u $USER -t PENDING -p normal -h | wc -l

[SubmitFile]
filename = submit.slurm
local_dir = /tmp/$SLURM_JOB_ID
custom_header = #SBATCH -A TG-ABC00001
cvmfs_job_wrapper = False

[CustomEnv]
CLUSTER = workflow.isi.edu
```

This configuration will obviously look different for different clusters. *configs/* has a bunch of example configs, but a few things to note:

- **address** is the location of the server we started earlier

- **tarball** is the full path to our custom glidein.tar.gz file we created above.

- **CLUSTER** is the location of your HTCondor central manager. In many cases this is the same host you started the server on. Please note that if you do not set this variable, the glideins will try to register into the IceCube infrastructure.

At this point we can try our first glidein:

```
./client.py --config=$HOME/git/pyglidein/configs/stampede2.config
```

Once we have a seen a successful glidein, we can add the client to the crontab:

```
# m  h  dom mon dow    command
*/10 *   *   *   *     (cd ~/git/pyglidein/ && ./client.py --config=$HOME/git/pyglidein/configs/
stampede2.config) >~/cron-pyglidein.log 2>&1
```

With this setup, glideins will now appear automatically based on the demand in the local HTCondor queue.

Remote Cluster using Globus GRAM

Figure 7.5. Grid Sample Site Layout
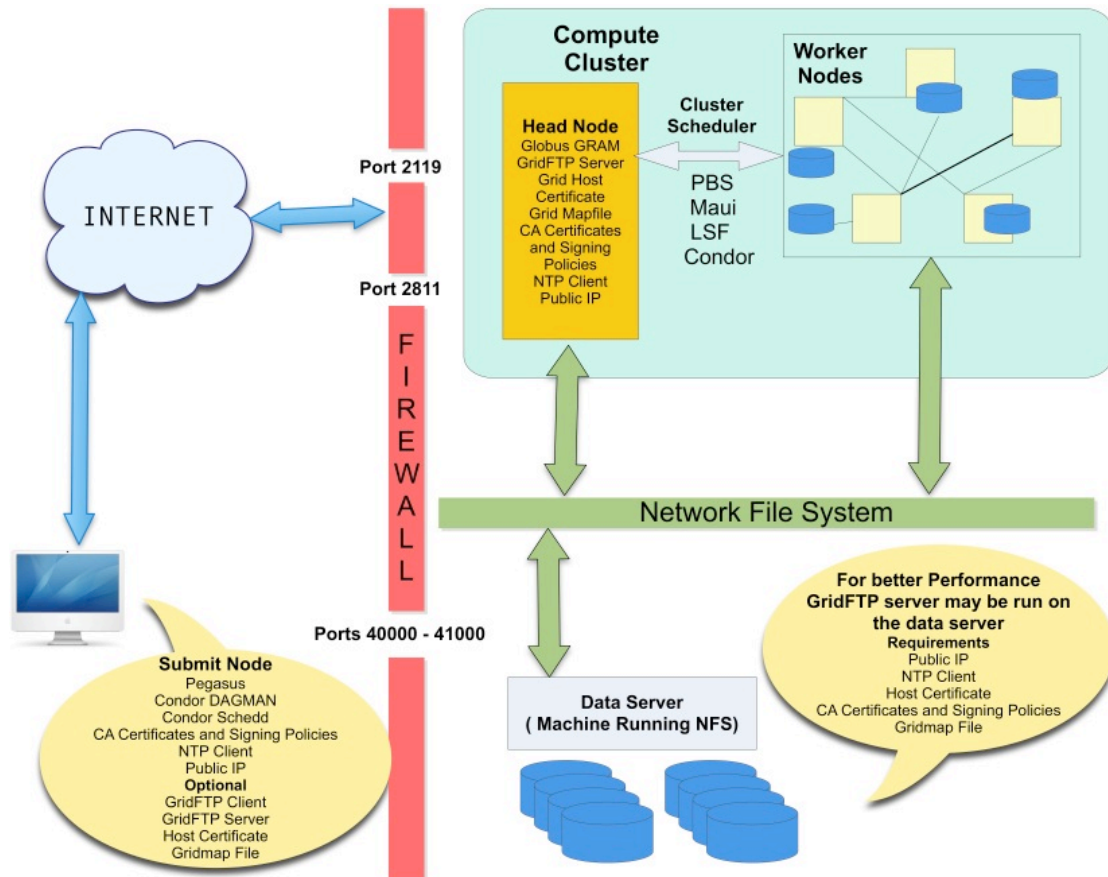
A generic grid environment shown in the figure above. We will work from the left to the right top, then the right bottom.

On the left side, you have a submit machine where Pegasus runs, HTCondor schedules jobs, and workflows are executed. We call it the *submit host* (SH), though its functionality can be assumed by a virtual machine image. In order to properly communicate over secured channels, it is important that the submit machine has a proper notion of time, i.e. runs an NTP daemon to keep accurate time. To be able to connect to remote clusters and receive connections from the remote clusters, the submit host has a public IP address to facilitate this communication.

In order to send a job request to the remote cluster, HTCondor wraps the job into Globus calls via HTCondor-G. Globus uses GRAM to manage jobs on remote sites. In terms of a software stack, Pegasus wraps the job into HTCondor. HTCondor wraps the job into Globus. Globus transports the job to the remote site, and unwraps the Globus component, sending it to the remote site's *resource manager* (RM).

To be able to communicate using the Globus security infrastructure (GSI), the submit machine needs to have the certificate authority (CA) certificates configured, requires a host certificate in certain circumstances, and the user a user certificate that is enabled on the remote site. On the remote end, the remote gatekeeper node requires a host certificate, all signing CA certificate chains and policy files, and a goot time source.

In a grid environment, there are one or more clusters accessible via grid middleware like the Globus Toolkit [http://www.globus.org/]. In case of Globus, there is the Globus gatekeeper listening on TCP port 2119 of the remote cluster. The port is opened to a single machine called *head node* (HN).The head-node is typically located in a de-militarized zone (DMZ) of the firewall setup, as it requires limited outside connectivity and a public IP address so that it can be contacted. Additionally, once the gatekeeper accepted a job, it passes it on to a jobmanager. Often, these jobmanagers require a limited port range, in the example TCP ports 40000-41000, to call back to the submit machine.

For the user to be able to run jobs on the remote site, the user must have some form of an account on the remtoe site. The user's grid identity is passed from the submit host. An entity called *grid mapfile* on the gatekeeper maps the user's grid identity into a remote account. While most sites do not permit account sharing, it is possible to map multiple user certificates to the same account.

The gatekeeper is the interface through which jobs are submitted to the remote cluster's resource manager. A resource manager is a scheduling system like PBS, Maui, LSF, FBSNG or HTCondor that queues tasks and allocates worker nodes. The *worker nodes* (WN) in the remote cluster might not have outside connectivity and often use all private IP addresses. The Globus toolkit requires a shared filesystem to properly stage files between the head node and worker nodes.

Note

The shared filesystem requirement is imposed by Globus. Pegasus is capable of supporting advanced site layouts that do not require a shared filesystem. Please contact us for details, should you require such a setup.

To stage data between external sites for the job, it is recommended to enable a GridFTP server. If a shared networked filesystem is involved, the GridFTP server should be located as close to the file-server as possible. The GridFTP server requires TCP port 2811 for the control channel, and a limited port range for data channels, here as an example the TPC ports from 40000 to 41000. The GridFTP server requires a host certificate, the signing CA chain and policy files, a stable time source, and a gridmap file that maps between a user's grid identify and the user's account on the remote site.

The GridFTP server is often installed on the head node, the same as the gatekeeper, so that they can share the grid map-file, CA certificate chains and other setups. However, for performance purposes it is recommended that the GridFTP server has its own machine.

An example site catalog entry for a GRAM enabled site looks as follow in the site catalog

```
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog http://pegasus.isi.edu/
schema/sc-4.0.xsd"
             version="4.0">

    <site handle="Trestles" arch="x86_64" os="LINUX">
        <grid type="gt5" contact="trestles.sdsc.edu/jobmanager-fork" scheduler="Fork"
 jobtype="auxillary"/>
        <grid type="gt5" contact="trestles.sdsc.edu/jobmanager-pbs" scheduler="unknown"
 jobtype="compute"/>

        <directory type="shared-scratch" path="/oasis/projects/nsf/USERNAME">
            <file-server operation="all" url="gsiftp://trestles-dm1.sdsc.edu/oasis/projects/nsf/
USERNAME"/>
        </directory>

        <!-- specify the path to a PEGASUS WORKER INSTALL on the site -->
        <profile namespace="env" key="PEGASUS_HOME" >/path/to/PEGASUS/INSTALL</profile>
    </site>
```

```
    </sitecatalog>
```

Remote Cluster using CREAMCE

CREAM [https://wiki.italiangrid.it/twiki/bin/view/CREAM/FunctionalDescription] is a webservices based job submission front end for remote compute clusters. It can be viewed as a replaced for Globus GRAM and is mainly popular in Europe. It widely used in the Italian Grid.

In order to submit a workflow to compute site using the CREAMCE front end, the user needs to specify the following for the site in their site catalog

1. **pegasus** profile **style** with value set to **cream**

2. **grid gateway**defined for the site with **contact** attribute set to CREAMCE frontend and **scheduler** attribute to remote scheduler.

3. a remote queue can be optionally specified using **globus** profile **queue** with value set to **queue-name**

An example site catalog entry for a creamce site looks as follow in the site catalog

```
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog http://pegasus.isi.edu/
schema/sc-4.0.xsd"
             version="4.0">

    <site  handle="creamce" arch="x86" os="LINUX">
        <grid type="cream" contact="https://ce01-lcg.cr.cnaf.infn.it:8443/ce-cream/services/CREAM2"
 scheduler="LSF" jobtype="compute" />
        <grid type="cream" contact="https://ce01-lcg.cr.cnaf.infn.it:8443/ce-cream/services/CREAM2"
 scheduler="LSF" jobtype="auxillary" />

         <!-- Scratch directory on the cluster -->
        <directory type="shared-scratch" path="/home/virgo034">
            <file-server operation="all" url="gsiftp://ce01-lcg.cr.cnaf.infn.it/home/virgo034"/>
        </directory>

        <!-- cream is the style to use for CREAMCE submits -->


        <profile namespace="pegasus" key="style">cream</profile>

        <!-- the remote queue is picked up from globus profile -->
        <profile namespace="globus" key="queue">virgo</profile>

        <!-- Staring HTCondor 8.0 additional cream attributes
             can be passed by setting cream_attributes -->
        <profile namespace="condor" key="cream_attributes">key1=value1;key2=value2</profile>
    </site>

</sitecatalog>
```

The pegasus distribution comes with creamce examples in the examples directory. They can be used as a starting point to configure your setup.

Tip

Usually , the CREAMCE frontends accept VOMS generated user proxies using the command voms-proxy-init . Steps on generating a VOMS proxy are listed in the CREAM User Guide here [https://wiki.italian-grid.it/twiki/bin/view/CREAM/UserGuide#1_1_Before_starting_get_your_use] .

Local Campus Cluster Using Glite

This section describes the configuration required for Pegasus to generate an executable workflow that uses glite to submit to a Slurm, PBS, or SGE batch system on a local cluster. This environment is referred to as the local campus cluster, as the workflow submit node (Pegasus + HTCondor) need to be installed on a login node (or a node where the local batch scheduler commands can be executed) of the cluster.

Note

Glite is the old name for BLAH (or BLAHP). BLAH binaries are distributed with HTCondor as the "batch_gahp". For historical reasons, we often use the term "glite", and you will see "glite" and "batch_gahp" references in HTCondor, but all of them refer to the same thing, which has been renamed BLAH.

This guide covers Slurm, PBS, Moab, and SGE, but glite also works with other PBS-like batch systems, including LSF, Cobalt and others. If you need help configuring Pegasus and HTCondor to work with one of these systems, please contact pegasus-support@isi.edu. For the sake of brevity, the text below will say "PBS", but you should read that as "PBS or PBS-like system such as SGE, Moab, LSF, and others".

This is because the glite layer communicates with the batch system running on the cluster using `squeue/qsub/...` or equivalent commands. If you can submit jobs to the local scheduler from the workflow submit host, then the local HTCondor can be used to submit jobs via glite (with some modifications described below). If you need to SSH to a different cluster head node in order to submit jobs to the scheduler, then you can use BOSCO, which is documented in another section.

Tip

There is also a way to do remote job submission via glite even if you cannot SSH to the head node. This might be the case, for example, if the head node requires 2-factor authentication (e.g. RSA tokens). This approach is called the "Reverse GAHP" and you can find out more information on the GitHub page [https://github.com/juve/rvgahp]. All it requires is SSH from the cluster head node back to the workflow submit host.

In either case, you need to modifiy the HTCondor glite installation that will be used to submit jobs to the local scheduler. To do this, run the `pegasus-configure-glite` command. This command will install all the required scripts to map Pegasus profiles to batch-system specific job attributes, and add support for Moab. You may need to run it as root depending on how you installed HTCondor.

Tip

HTCondor has an issue for the Slurm configuration when running on Ubuntu systems. Since in Ubuntu, `/bin/sh` does not link to `bash`, the Slurm script will fail when trying to run the `source` command. A quick fix to this issue is to force the script to use `bash`. In the `bls_set_up_local_and_extra_args` function of the `blah_common_submit_functions.sh` script, which is located in the same folder as the installation above, only add `bash` before `$bls_opt_tmp_req_file >> $bls_tmp_file 2> /dev/null` command line.

In order to configure a workflow to use glite you need to create an entry in your site catalog for the cluster and set the following profiles:

1. **pegasus** profile **style** with value set to **glite**.

2. **condor** profile **grid_resource** with value set to **batch slurm**, **batch pbs**, **batch sge** or **batch moab**.

An example site catalog entry for a local glite PBS site looks like this:

```
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog http://pegasus.isi.edu/
schema/sc-4.0.xsd"
             version="4.0">

    <site  handle="local" arch="x86" os="LINUX">
        <directory type="shared-scratch" path="/lfs/shared-scratch/glite-sharedfs-example/work">
            <file-server operation="all" url="file:///lfs/local-scratch/glite-sharedfs-example/
work"/>
        </directory>
        <directory type="local-storage" path="/shared-scratch//glite-sharedfs-example/outputs">
            <file-server operation="all" url="file:///lfs/local-scratch/glite-sharedfs-example/
outputs"/>
        </directory>
    </site>

    <site  handle="local-slurm" arch="x86" os="LINUX">
```

```
        <!-- the following is a shared directory shared amongst all the nodes in the cluster -->
        <directory type="shared-scratch" path="/lfs/glite-sharedfs-example/local-slurm/shared-
scratch">
            <file-server operation="all" url="file:///lfs/glite-sharedfs-example/local-slurm/shared-
scratch"/>
        </directory>

        <profile namespace="env" key="PEGASUS_HOME">/lfs/software/pegasus</profile>

        <profile namespace="pegasus" key="style" >glite</profile>

        <profile namespace="condor" key="grid_resource">batch slurm</profile>
        <profile namespace="pegasus" key="queue">normal</profile>
        <profile namespace="pegasus" key="runtime">30000</profile>
    </site>

</sitecatalog>
```

Tip

Starting 4.2.1, in the examples directory you can find a glite shared filesystem example that you can use to test out this configuration.

You probably don't need to know this, but Pegasus generates a `+remote_cerequirements` expression for an HTCondor glite job based on the Pegasus profiles associated with the job. This expression is passed to glite and used by the `*_local_submit_attributes.sh` scripts installed by `pegasus-configure-glite` to generate the correct batch submit script. An example `+remote_cerequirements` classad expression in the HTCondor submit file looks like this:

```
+remote_cerequirements = JOBNAME=="preprocessj1" && PASSENV==1 && WALLTIME=="01:00:00" && \
 EXTRA_ARGUMENTS=="-N testjob -l walltime=01:23:45 -l nodes=2" && \
 MYENV=="CONDOR_JOBID=$(cluster).$(process),PEGASUS_DAG_JOB_ID=preprocess_j1,PEGASUS_HOME=/
usr,PEGASUS_WF_UUID=aae14bc4-b2d1-4189-89ca-ccd99e30464f"
```

The job name and environment variables are automatically passed through to the remote job.

The following sections document the mapping of Pegasus profiles to batch system job requirements as implemented by Pegasus, HTCondor, and glite.

Setting job requirements

The job requirements are constructed based on the following profiles:

Table 7.1. Mapping of Pegasus Profiles to Job Requirements

Profile Key	Key in +remote_cerequirements	PBS Parameter	SGE Parameter	SLURM parameter	Moab Parameter	Cobalt Parameter	Description
pegasus.cores	CORES	n/a	-pe ompi	--ntasks cores	n/a	--proccount cores	Pegasus uses cores to calculate either nodes or ppn. If cores and ppn are specified, then nodes is computed. If cores and nodes is specified, then ppn is computed. If both nodes and ppn are specified,

Profile Key	Key in +remote_cerequirements	PBS Parameter	SGE Parameter	SLURM parameter	Moab Parameter	Cobalt Parameter	Description
							then cores is ignored. The resulting values for nodes and ppn are used to set the job requirements for PBS and Moab. If neither nodes nor ppn is specified, then no requirements are set in the PBS or Moab submit script. For SGE, how the processes are distributed over nodes depends on how the parallel environment has been configured; it is set to 'ompi' by default.
pegasus.nodes	NODES	-l nodes	n/a	--nodes nodes	-l nodes	-n nodes	This specifies the number of nodes that the job should use. This is not used for SGE.
pegasus.ppn	PROCS	-l ppn	n/a	n/a	-l ppn	--mode c[ppn]	This specifies the number of processors per node that the job should use. This is not used for SGE.
pegasus.runtime	WALLTIME	-l walltime	-l h_rt	--time walltime	-l walltime	-t walltime	This specifies the maximum runtime for

Profile Key	Key in +remote_cerequirements	PBS Parameter	SGE Parameter	SLURM parameter	Moab Parameter	Cobalt Parameter	Description
							the job in seconds. It should be an integer value. Pegasus converts it to the "hh:mm:ss" format required by the batch system. The value is rounded up to the next whole minute.
pegasus.memory	PER_PROCESS_MEMORY	-l pmem	-l h_vmem	--mem memory	--mem-per-cpu pmem	n/a	This specifies the maximum amount of physical memory used by any process in the job. For example, if the job runs four processes and each requires up to 2 GB (gigabytes) of memory, then this value should be set to "2gb" for PBS and Moab, and "2G" for SGE. The corresponding PBS directive would be "#PBS -l pmem=2gb".
pegasus.project	PROJECT	-A project_name	n/a	n/a	-A project_name	-A project_name	Causes the job time to be charged to or associated with a particular project/ account. This is not used for SGE.
pegasus.queue	n/a	-q	-q	n/a	-q		This specifies the

Profile Key	Key in +remote_cerequirements	PBS Parameter	SGE Parameter	SLURM parameter	Moab Parameter	Cobalt Parameter	Description
							queue for the job. This profile does not have a corresponding value in +remote_cerequirements. Instead, Pegasus sets the batch_queue key in the Condor submit file, which gLite/blahp translates into the appropriate batch system requirement.
globus.totalmemory	TOTAL_MEMORY	-l mem	n/a	--mem memory	-l mem	n/a	The total memory that your job requires. It is usually better to just specify the pegasus.memory profile. This is not mapped for SGE.
pegasus.glite.arguments	EXTRA_ARGUMENTS	prefixed by "#PBS"	prefixed by "#?"	prefixed by "#SBATCH"	prefixed by "#MSUB"	n/a	This specifies the extra arguments that must appear in the generated submit script for a job. The value of this profile is added to the submit script prefixed by the batch system-specific value. These requirements override any requirements specified us-

Profile Key	Key in +re-mote_cerequire-ments	PBS Para-meter	SGE Para-meter	SLURM pa-rameter	Moab Para-meter	Cobalt Pa-rameter	Description
							ing other pro-files. This is useful when you want to pass through spe-cial options to the under-lying batch system. For example, on the USC cluster we use resource properties to specify the network type. If you want to use the Myrinet network, you must spec-ify some-thing like "-l nodes=8:pp-n=2:myri". For infini-band, you would use something like "-l nodes=8:pp-n=2:IB". In that case, both the nodes and ppn profiles would be ef-fectively ig-nored.

Specifying a remote directory for the job

gLite/blahp does not follow the `remote_initialdir` or `initialdir` classad directives. Therefore, all the jobs that have the `glite` style applied don't have a remote directory specified in the submit script. Instead, Pegasus uses Kickstart to change to the working directory when the job is launched on the remote system.

SDSC Comet with BOSCO glideins

BOSCO documentation: https://twiki.opensciencegrid.org/bin/view/CampusGrids/BoSCO

BOSCO is part of the HTCondor system which allows you to set up a personal pool of resources brought in from a remote cluster. In this section, we describe how to use BOSCO to run glideins (pilot jobs) dynamically on the SDSC Comet cluster. The glideins are submitted based on the demand of the user jobs in the pool.

As your regular user, on the host you want to use as a workflow submit host, download the latest version of HTCondor from the HTCondor Download page [https://research.cs.wisc.edu/htcondor/downloads/>]. At this point the latest version was 8.5.2 and we downloaded condor-8.5.2-x86_64_RedHat6-stripped.tar.gz. Untar, and run the installer:

```
$ tar xzf condor-8.5.2-x86_64_RedHat6-stripped.tar.gz
$ cd condor-8.5.2-x86_64_RedHat6-stripped
$ ./bosco_install
...
Created a script you can source to setup your Condor environment
variables. This command must be run each time you log in or may
be placed in your login scripts:
    source /home/$USER/bosco/bosco_setenv
```

Source the setup file as instructed, run *bosco_start*, and then test that *condor_q* and *condor_status* works.

```
$ source /home/$USER/bosco/bosco_setenv
$ condor_q

-- Schedd: workflow.iu.xsede.org : 127.0.0.1:11000?...
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD

0 jobs; 0 completed, 0 removed, 0 idle, 0 running, 0 held, 0 suspended
$ condor_status
```

Let's tell BOSCO about our SDSC Comet account:

```
$ bosco_cluster -a YOUR_SDSC_USERNAME@comet-ln2.sdsc.edu pbs
```

BOSCO needs a little bit more information to be able to submit the glideins to Comet. Log in to your Comet account via ssh (important - this step has to take place on Comet) and create the *~/bosco/glite/bin/pbs_local_submit_attributes.sh* file with the following content. You can find your allocation by running *show_accounts* and looking at the project column.

```
echo "#PBS -q compute"
echo "#PBS -l nodes=1:ppn=24"
echo "#PBS -l walltime=24:00:00"
echo "#PBS -A [YOUR_COMET_ALLOCATION]"
```

Also chmod the file:

```
$ chmod 755 ~/bosco/glite/bin/pbs_local_submit_attributes.sh
```

Log out of Comet, and get back into the host and user BOSCO was installed into. We also need to edit a few files on that host. *~/bosco/libexec/campus_factory/share/glidein_jobs/glidein_wrapper.sh* has a bug in some versions of HTCondor. Open up the file and make sure the eval line in the beginning is below the unset/export HOME section. If that is not the case, edit the file to look like:

```
#!/bin/sh

starting_dir="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

# BLAHP does weird things with home directory
unset HOME
export HOME

eval campus_factory_dir=$_campusfactory_CAMPUSFACTORY_LOCATION
```

If the order of the HOME and eval statements are reversed in your file, change them to look like the above. At the end of *~/bosco/libexec/campus_factory/share/glidein_jobs/glidein_condor_config* add:

```
# dynamic slots
SLOT_TYPE_1 = cpus=100%,disk=100%,swap=100%
SLOT_TYPE_1_PARTITIONABLE = TRUE
NUM_SLOTS = 1
NUM_SLOTS_TYPE_1 = 1
```

In the file *~/bosco/libexec/campus_factory/share/glidein_jobs/job.submit.template* find the line reading:

```
_condor_NUM_CPUS=1; \
```

You should now have a functioning BOSCO setup. Submit a Pegasus workflow.

Remote PBS Cluster using BOSCO and SSH

BOSCO [http://bosco.opensciencegrid.org/about/] enables HTCondor to submit jobs to remote PBS clusters using SSH. This section describes how to specify a site catalog entry for a site that has been configured for BOSCO job submissions.

First, the site needs to be setup for BOSCO according to the BOSCO documentation [https://twiki.opensciencegrid.org/bin/view/CampusGrids/BoSCO]. BOSCO uses glite to submit jobs to the PBS scheduler on the remote cluster. You will also need to configure the glite installed for BOSCO on the remote system according to the documentation in the glite section in order for the mapping of Pegasus profiles to PBS job requirements to work. In particular, you will need to install the `pbs_local_submit_attributes.sh` and `sge_local_submit_attributes.sh` scripts in the correct place in the glite `bin` directory on the remote cluster, usually in the directory *~/bosco/glite/bin/* .

Second, to tag a site for SSH submission, the following profiles need to be specified for the site in the site catalog:

1. **pegasus** profile **style** with value set to **ssh**

2. Specify the service information as grid gateways. This should match what BOSCO provided when the cluster was set up.

An example site catalog entry for a BOSCO site looks like this:

```
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog http://pegasus.isi.edu/
schema/sc-4.0.xsd"
             version="4.0">

    <site  handle="USC_HPCC_Bosco" arch="x86_64" os="LINUX">

        <!-- Specify the service information. This should match what Bosco provided when the cluster
             was set up. -->
        <grid type="batch" contact="vahi@hpc-pegasus.usc.edu" scheduler="PBS" jobtype="compute"/>
        <grid type="batch" contact="vahi@hpc-pegasus.usc.edu" scheduler="PBS" jobtype="auxillary"/>

        <!-- Scratch directory on the cluster -->
        <directory type="shared-scratch" path="/home/rcf-40/vahi/tmp">
            <file-server operation="all" url="scp://vahi@hpc-pegasus.usc.edu/home/rcf-40/vahi/tmp"/>
        </directory>

        <!-- SSH is the style to use for Bosco SSH submits -->
        <profile namespace="pegasus" key="style">ssh</profile>

        <!--  works around bug in the HTCondor GAHP, that does not
              set the remote directory -->
        <profile namespace="pegasus" key="change.dir">true</profile>

        <!-- Job requirements should be specified using Pegasus profiles -->
        <profile namespace="pegasus" key="queue">default</profile>
        <profile namespace="pegasus" key="runtime">30</profile>

    </site>


</sitecatalog>
```

Note

It is recommended to have a submit node configured either as a BOSCO submit node or a vanilla HTCondor node. You cannot have HTCondor configured both as a BOSCO install and a traditional HTCondor submit node at the same time as BOSCO will override the traditional HTCondor pool in the user environment.

There is a bosco-shared-fs example in the examples directory of the distribution.

Job Requirements for the jobs can be set using the same profiles as listed here .

Campus Cluster

There are almost as many different configurations of campus clusters as there are campus clusters, and because of that it can be hard to determine what the best way to run Pegasus workflows. Below is a ordered checklist with some ideas we have collected from working with users in the past:

1. If the cluster scheduler is HTCondor, please see the HTCondor Pool section.

2. If the cluster is Globus GRAM enabled, see the Globus GRAM section. If you have have a lot of short jobs, also read the Glidein section.

3. For clusters without GRAM, you might be able to do glideins. If outbound network connectivity is allowed, your submit host can be anywhere. If the cluster is setup to not allow any network connections to the outside, you will probably have to run the submit host inside the cluster as well.

If the cluster you are trying to use is not fitting any of the above scenarios, please post to the Pegasus users mailing list [http://pegasus.isi.edu/support] and we will help you find a solution.

XSEDE

The Extreme Science and Engineering Discovery Environment (XSEDE) [https://www.xsede.org/] provides a set of High Performance Computing (HPC) and High Throughput Computing (HTC) resources.

For the HPC resources, it is recommended to run using Globus GRAM or glideins. Most of these resources have fast parallel file systesm, so running with sharedfs data staging is recommended. Below is example site catalog and pegasusrc to run on SDSC Trestles [http://www.sdsc.edu/us/resources/trestles/]:

```
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog http://pegasus.isi.edu/
schema/sc-4.0.xsd"
             version="4.0">

    <site  handle="local" arch="x86_64" os="LINUX">
        <directory type="shared-scratch" path="/tmp/wf/work">
            <file-server operation="all" url="file:///tmp/wf/work"/>
        </directory>
        <directory type="local-storage" path="/tmp/wf/storage">
            <file-server operation="all" url="file:///tmp/wf/storage"/>
        </directory>
    </site>

    <site handle="Trestles" arch="x86_64" os="LINUX">
        <grid type="gt5" contact="trestles.sdsc.edu:2119/jobmanager-fork" scheduler="PBS"
 jobtype="auxillary"/>
        <grid type="gt5" contact="trestles.sdsc.edu:2119/jobmanager-pbs" scheduler="PBS"
 jobtype="compute"/>
        <directory type="shared-scratch" path="/phase1/USERNAME">
            <file-server operation="all" url="gsiftp://trestles-dm1.sdsc.edu/phase1/USERNAME"/>
        </directory>
    </site>

</sitecatalog>
```

pegasusrc:

```
pegasus.catalog.replica=SimpleFile
pegasus.catalog.replica.file=rc

pegasus.catalog.site.file=sites.xml

pegasus.catalog.transformation=Text
pegasus.catalog.transformation.file=tc

pegasus.data.configuration = sharedfs

# Pegasus might not be installed, or be of a different version
# so stage the worker package
pegasus.transfer.worker.package = true
```

The HTC resources available on XSEDE are all HTCondor based, so standard HTCondor Pool setup will work fine.

If you need to run high throughput workloads on the HPC machines (for example, post processing after a large parallel job), glideins can be useful as it is a more efficient method for small jobs on these systems.

Open Science Grid Using glideinWMS

glideinWMS [http://www.uscms.org/SoftwareComputing/Grid/WMS/glideinWMS/] is a glidein system widely used on Open Science Grid. Running on top of glideinWMS is like running on a Condor Pool without a shared filesystem.

Chapter 8. Containers

Overview

Application containers provides a solution to package software with complex dependencies to be used during workflow execution. Starting with Pegasus 4.8.0, Pegasus has support for application containers in the non-shared filesystem or condorio data configurations using PegasusLite. Users can specify with their transformations in the Transformation Catalog the container in which the the transformation should be executed. Pegasus currently has support for the following container technologies:

1. Docker

2. Singularity

The worker package is not required to be pre-installed in images. If a matching worker package is not installed, Pegasus will try to determine which package is required and download it.

Configuring Workflows To Use Containers

Containers currently can only be specified in the Transformation Catalog. Users have the option of either using a different container for each executable or same container for all executables. In the case, where you wants to use a container that does not have your executable pre-installed, you can mark the executable as STAGEABLE and Pegasus will stage the executable into the container, as part of executable staging.

The DAX API extensions don't support references for containers.

Containerized Applications in the Transformation Catalog

Users can specify what container they want to use for running their application in the Transformation Catalog using the multi line text based format described in this section. Users can specify an optional attribute named container that refers to the container to be used for the application.

```
tr example::keg:1.0 {

  #specify profiles that apply for all the sites for the transformation
  #in each site entry the profile can be overriden

  profile env "APP_HOME" "/tmp/myscratch"
  profile env "JAVA_HOME" "/opt/java/1.6"

  site isi {
    # environment to be set when the job is run in the container
    # overrides env profiles specified in the container
    profile env "HELLo" "WORLD"
    profile env "JAVA_HOME" "/bin/java.1.6"

    profile condor "FOO" "bar"

    pfn "/path/to/keg
    arch "x86"
    os "linux"
    osrelease "fc"
    osversion "4"

    # INSTALLED means pfn refers to path in the container.
    # STAGEABLE means the executable can be staged into the container
    type "INSTALLED"

    #optional attribute to specify the container to use
    container "centos-pegasus"
  }
}

cont centos-pegasus{
      # can be either docker or singularity
```

```
        type "docker"

        # URL to image in a docker|singularity hub OR
        # URL to an existing docker image exported as a tar file or singularity image
        image "docker:///rynge/montage:latest"

        # optional site attribute to tell pegasus which site tar file
        # exists. useful for handling file URL's correctly
        image_site "optional site"

        # environment to be set when the job is run in the container
        # only env profiles are supported
        profile env "JAVA_HOME" "/opt/java/1.6"
}
```

The container itself is defined using the cont entry. Multiple transformations can refer to the same container.

1. **cont** cont - A container identifier.

2. **image** - URL to image in a docker|singularity hub or URL to an existing docker image exported as a tar file or singularity image. Example of a docker hub URL is docker:///rynge/montage:latest, while for singularity shub://pegasus-isi/fedora-montage

3. **image_site** - The site identifier for the site where the container is available

4. **Profiles** - One or many profiles can be attached to a transformation for all sites or to a transformation on a particular site. For containers, only env profiles are supported.

Note

Containerized Applications can only be specified in the transformation catalog, not via the DAX API.

Container Execution Model

User's containerized applications are launched as part of PegasusLite jobs. PegasusLite job when starting on a remote worker node.

1. Sets up a directory to run a user job in.

2. Pulls in all the relevant input data, executables and the container image to that directory

3. Optionally, loads the container from the container image file and sets up the user to run as in the container (only applicable for Docker containers)

4. Mounts the job directory into the container as /scratch for Docker containers, while as /srv for Singularity containers.

5. Container will run a job specific script that figures out the appropriate Pegasus worker to use in the container if not already installed, and sets up the job environment to use it, before launching the user application using *pegasus-kickstart*.

6. Optionally, shuts down the container (only applicable for Docker containers)

7. Ships out the output data to the staging site

8. Cleans up the directory on the worker node.

The above model, allows for all credential handling required for data transfers to be handled outside the container within the PegasusLite job.

Staging of Application Containers

Pegasus treats containers as other files in terms of data management. Container to be used for a job is tracked as an input dependency that needs to be staged if it is not already there. Similar to executables, you specify the location for your container image in the Transformation Catalog. You can specify the source URL's for containers as the following.

1. URL to a container hosted on a central hub repository

   Example of a docker hub URL is docker:///rynge/montage:latest, while for singularity shub://pegasus-isi/fedora-montage

2. URL to a container image file on a file server.

   - **Docker -** Docker supports loading of containers from a tar file, Hence, containers images can only be specified as tar files and the extension for the filename is not important.

   - **Singularity -** Singularity supports container images in various forms and relies on the extension in the filename to determine what format the file is in. Pegasus supports the following extensions for singularity container images

     - .img

     - .tar

     - .tar.gz

     - .tar.bz2

     - .cpio

     - .cpio.gz

     Singularity will fail to run the container if you don't specify the right extension , when specify the source URL for the image.

In both the cases, Pegasus will place the container image on the staging site used for the workflow, as part of the data stage-in nodes, using pegasus-transfer. When pulling in an image from a container hub repository, pegasus-transfer will export the container as a tar file in case of Docker, and as .img file in case of Singularity

Symlinking

Since, Pegasus only mounts the job directory determined by PegasusLite into the application container, symlinking of input data sets does not work. This is because the symlink in PegasusLite directory points to a source directory on the worker node, that is not mounted in the container. Hence user's jobs would fail. Hence, Pegasus will automatically disable symlinking for jobs that use containers. The only exception being the application container itself. If you specify a URL for the container image, the image will be symlinked if Pegasus determines that it can be.

Enabling symlinking of containers is useful, when running large workflows on a single cluster. Pegasus can pull the image from the container repository once, and place it on the shared filesystem where it can then be symlinked from, when the PegasusLite jobs start on the worker nodes of that cluster. In order to do this, you need to be running the nonsharedfs data configuration mode with the staging site set to be the same as the compute site.

Container Example - Montage Workflow

Montage Using Containers

This section contains an example of a real workflow running inside Singularity containers. The application is Montage [http://montage.ipac.caltech.edu/] using the montage-v2 workflow [https://github.com/pegasus-isi/montage-workflow-v2]. Be aware that this workflow can be fairly data intensive, and when running with containers in *condorio* or *nonsharedfs* data management modes, the data staging of the application data and the container image to each job can result in a non-trivial amount of network traffic.

The software dependencies consists of the *Montage* software stack, and *AstroPy*. These are installed into the image (see the *Singularity* file in the GitHub repository). The image has been made available in Singularity Hub [https://singularity-hub.org/].

Now that we have an image, the next step is to check out the workflow from GitHub, and use it to create an abstract workflow description, a transformation catalog and a replica catalog. The *montage-workflow.py* command create all

this for us, but the command itself requires Montage to look up input data for the specified location in the sky. The provide the environment, run this command inside the same Singularity image. For example:

```
singularity exec \
            --bind $PWD:/srv --workdir /srv \
            shub://pegasus-isi/montage-workflow-v2 \
            /srv/montage-workflow.py \
                --tc-target container \
                --center "56.7 24.00" \
                --degrees 2.0 \
                --band dss:DSS2B:blue \
                --band dss:DSS2R:green \
                --band dss:DSS2IR:red
```

The command executes a data find for the 3 specified bands, 2.0 degrees around the location 56.7 24.00, and generates a workflow to combine the images into a single image. One extra flag is provided to let the command know we want to execute the workflow inside containers: *--tc-target container*. The result is a transformation catalog in *data/tc.txt* , with starts with:

```
cont montage {
   type "singularity"
   image "shub://pegasus-isi/montage-workflow-v2"
   profile env "MONTAGE_HOME" "/opt/Montage"
}

tr mDiffFit {
  site condor_pool {
    type "INSTALLED"
    container "montage"
    pfn "file:///opt/Montage/bin/mDiffFit"
    profile pegasus "clusters.size" "5"
  }
}
...
```

The first entry describes the container, where the image can be found (Singularity Hub in this example), and a special environment variable we want to be set for the jobs.

The second entry, of which there are many more similar ones in the file, describes the application. Note how it refers back to the *"montage"* container, specifying that we want the job to be wrapped in the container.

In the *data/* directory. we can also find the abstract workflow (*montage.dax*), and replica catalog (*rc.dax*). Note that this are the same as if the workflow was running in a non-container environment. To plan the workflow:

```
pegasus-plan \
        --dir work \
        --relative-dir `date +'%s'` \
        --dax data/montage.dax \
        --sites condor_pool \
        --output-site local \
        --cluster horizontal
```

Chapter 9. Example Workflows

These examples are included in the Pegasus distribution and can be found under `share/pegasus/examples` in your Pegasus install (`/usr/share/pegasus/examples` for native packages)

Note

These examples are intended to be a starting point for when you want to create your own workflows and want to see how other workflows are set up. The example workflows will probably not work in your environment without modifications. Site and transformation catalogs contain site and user specifics such as paths to scratch directories and installed software, and at least minor modificiations are required to get the workflows to plan and run.

Grid Examples

These examples assumes you have access to a cluster with Globus installed. A pre-ws gatekeeper and gridftp server is required. You also need Globus and Pegasus installed, both on the machine you are submitting from, and the cluster.

Black Diamond

Pegasus is shipped with 3 different Black Diamond examples for the grid. This is to highlight the available DAX APIs which are Java, Perl and Python. The examples can be found under:

```
share/pegasus/examples/grid-blackdiamond-java/
share/pegasus/examples/grid-blackdiamond-perl/
share/pegasus/examples/grid-blackdiamond-python/
```

The workflow has 4 nodes, layed out in a diamond shape, with files being passed between them (f.*):
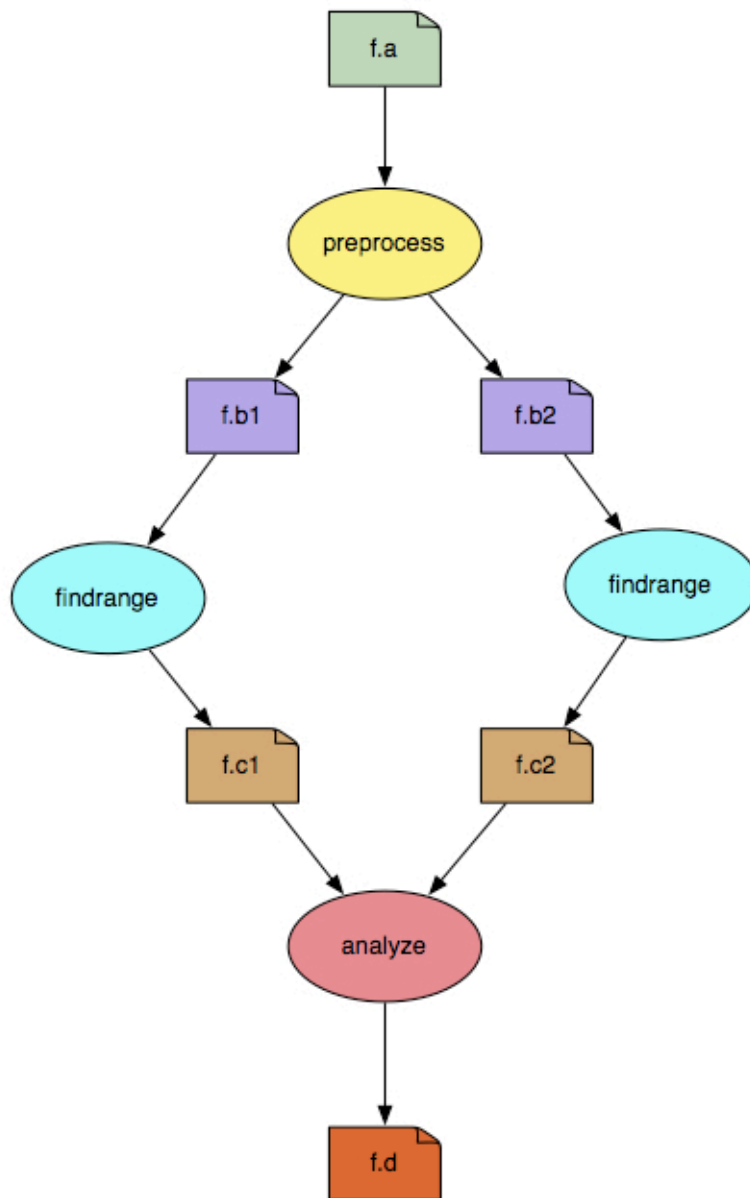
The binary for the nodes is a simple "mock application" name **keg** ("canonical example for the grid") which reads input files designated by arguments, writes them back onto output files, and produces on STDOUT a summary of where and when it was run. Keg ships with Pegasus in the bin directory.

This example ships with a "submit" script which will build the replica catalog, the transformation catalog, and the site catalog. When you create your own workflows, such a submit script is not needed if you want to maintain those catalogs manually.

Note

The use of `./submit` scripts in these examples are just to make it more easy to run the examples out of the box. For a production site, the catalogs (transformation, replica, site) may or may not be static or generated by other tooling.

To test the examples, edit the **submit** script and change the cluster config to the setup and install locations for your cluster. Then run:

```
$ ./submit
```

The workflow should now be submitted and in the output you should see a work dir location for the instance. With that directory you can monitor the workflow with:

```
$ pegasus-status [workdir]
```

Once the workflow is done, you can make sure it was sucessful with:

```
$ pegasus-analyzer -d [workdir]
```

NASA/IPAC Montage

This example can be found under

```
share/pegasus/examples/grid-montage/
```

The NASA IPAC Montage (http://montage.ipac.caltech.edu/) workflow projects/montages a set of input images from telescopes like Hubble and end up with images like http://montage.ipac.caltech.edu/images/m104.jpg . The test workflow is for a 1 by 1 degrees tile. It has about 45 input images which all have to be projected, background modeled and adjusted to come out as one seamless image.

Just like the Black Diamond above, this example uses a ./submit script.

The Montage DAX is generated with a tool called mDAG shipped with Montage which generates the workflow.

Rosetta

This example can be found under

```
share/pegasus/examples/grid-rosetta/
```

Rosetta (http://www.rosettacommons.org/) is a high resolution protein prediction and design software. Highlights in this example are:

- Using the Pegasus Java API to generate the DAX

- The DAX generator loops over the input PDBs and creates a job for each input

- The jobs all have a dependency on a flatfile database. For simplicity, each job depends on all the files in the database directory.

- Job clustering is turned on to make each grid job run longer and better utilize the compute cluster

Just like the Black Diamond above, this example uses a ./submit script.

Condor Examples

Black Diamond - condorio

There are a set of Condor examples available, highlighting different data staging configurations.The most basic one is condorio, and the example can be found under:

```
share/pegasus/examples/condor-blackdiamond-condorio/
```

This example is using the same abstract workflow as the Black Diamond grid example above, and can be executed either on the submit machine (universe="local") or on a local Condor pool (universe="vanilla").

You can run this example with the ./submit script. Example:

```
$ ./submit
```

Container Examples

Montage Using Containers

This section contains an example of a real workflow running inside Singularity containers. The application is Montage [http://montage.ipac.caltech.edu/] using the montage-v2 workflow [https://github.com/pegasus-isi/montage-work-flow-v2]. Be aware that this workflow can be fairly data intensive, and when running with containers in *condorio* or *nonsharedfs* data management modes, the data staging of the application data and the container image to each job can result in a non-trivial amount of network traffic.

The software dependencies consists of the *Montage* software stack, and *AstroPy*. These are installed into the image (see the *Singularity* file in the GitHub repository). The image has been made available in Singularity Hub [https://singularity-hub.org/].

Now that we have an image, the next step is to check out the workflow from GitHub, and use it to create an abstract workflow description, a transformation catalog and a replica catalog. The *montage-workflow.py* command create all this for us, but the command itself requires Montage to look up input data for the specified location in the sky. The provide the environment, run this command inside the same Singularity image. For example:

```
singularity exec \
            --bind $PWD:/srv --workdir /srv \
            shub://pegasus-isi/montage-workflow-v2 \
            /srv/montage-workflow.py \
                --tc-target container \
                --center "56.7 24.00" \
                --degrees 2.0 \
                --band dss:DSS2B:blue \
                --band dss:DSS2R:green \
                --band dss:DSS2IR:red
```

The command executes a data find for the 3 specified bands, 2.0 degrees around the location 56.7 24.00, and generates a workflow to combine the images into a single image. One extra flag is provided to let the command know we want to execute the workflow inside containers: *--tc-target container*. The result is a transformation catalog in *data/tc.txt*, with starts with:

```
cont montage {
   type "singularity"
   image "shub://pegasus-isi/montage-workflow-v2"
   profile env "MONTAGE_HOME" "/opt/Montage"
}

tr mDiffFit {
  site condor_pool {
    type "INSTALLED"
    container "montage"
    pfn "file:///opt/Montage/bin/mDiffFit"
    profile pegasus "clusters.size" "5"
  }
}
...
```

The first entry describes the container, where the image can be found (Singularity Hub in this example), and a special environment variable we want to be set for the jobs.

The second entry, of which there are many more similar ones in the file, describes the application. Note how it refers back to the *"montage"* container, specifying that we want the job to be wrapped in the container.

In the *data/* directory. we can also find the abstract workflow (*montage.dax*), and replica catalog (*rc.dax*). Note that this are the same as if the workflow was running in a non-container environment. To plan the workflow:

```
pegasus-plan \
        --dir work \
        --relative-dir `date +'%s'` \
        --dax data/montage.dax \
        --sites condor_pool \
```

```
                    --output-site local \
                    --cluster horizontal
```

Local Shell Examples

Black Diamond

To aid in workflow development and debugging, Pegasus can now map a workflow to a local shell script. One advantage is that you do not need a remote compute resource.

This example is using the same abstract workflow as the Black Diamond grid example above. The difference is that a property is set in pegasusrc to force shell execution:

```
# tell pegasus to generate shell version of
# the workflow
pegasus.code.generator = Shell
```

You can run this example with the `./submit` script.

Notifications Example

A new feature in Pegasus 3.1. is notifications. While the workflow is running, a monitoring tool is running side by side to the workflow, and issues user defined notifications when certain events takes place, such as job completion or failure. See notifications section for detailed information. A workflow example with notifications can be found under examples/notifications. This workflow is based on the Black Diamond, with the changes being notifications added to the DAX generator. For example, notifications are added at the workflow level:

```
# Create a abstract dag
diamond = ADAG("diamond")
# dax level notifications
diamond.invoke('all', os.getcwd() + "/my-notify.sh")
```

The DAX generator also contains job level notifications:

```
# job level notifications - in this case for at_end events
frr.invoke('at_end', os.getcwd() + "/my-notify.sh")
```

These invoke lines specify that the **my-notify.sh** script will be invoked for events generated (**all** in the first case, **at_end** in the second). The **my-notify.sh** script contains callouts sample notification tools shipped with Pegasus, one for email and for Jabber/GTalk (commented out by default):

```
#!/bin/bash

# Pegasus ships with a couple of basic notification tools. Below
# we show how to notify via email and gtalk.

# all notifications will be sent to email
# change $USER to your full email addess
$PEGASUS_HOME/libexec/notification/email -t $USER

# this sends notifications about failed jobs to gtalk.
# note that you can also set which events to trigger on in your DAX.
# set jabberid to your gmail address, and put in yout
# password
# uncomment to enable
if [ "x$PEGASUS_STATUS" != "x" -a "$PEGASUS_STATUS" != "0" ]; then
    $PEGASUS_HOME/libexec/notification/jabber --jabberid FIXME@gmail.com \
                                              --password FIXME \
                                              --host talk.google.com
fi
```

Workflow of Workflows

Galactic Plane

The Galactic Plane [http://en.wikipedia.org/wiki/Galactic_plane] workflow is a workflow of many Montage workflows. The output is a set of tiles which can be used in software which takes the tiles and produces a seamless image

which can be scrolled and zoomed into. As this is more of a production workflow than an example one, it can be a little bit harder to get running in your environment.

Highlights of the example are:

- The subworkflow DAXes are generated as jobs in the parent workflow - this is an example on how to make more dynamic workflows. For example, if you need a job in your workflow to determine the number of jobs in the next level, you can have the first job create a subworkflow with the right number of jobs.

- DAGMan job categories are used to limit the number of concurrant jobs in certain places. This is used to limit the number of concurrant connections to the data find service, as well limit the number of concurrant subworkflows to manage disk usage on the compute cluster.

- Job priorities are used to make sure we overlap staging and computation. Pegasus sets default priorities, which for most jobs are fine, but the priority of the data find job is set explicitly to a higher priority.

- A specific output site is defined the the site catalog and specified with the --output option of subworkflows.

The DAX API has support for sub workflows:

```
    remote_tile_setup = Job(namespace="gp", name="remote_tile_setup", version="1.0")
    remote_tile_setup.addArguments("%05d" % (tile_id))
    remote_tile_setup.addProfile(Profile("dagman", "CATEGORY", "remote_tile_setup"))
    remote_tile_setup.uses(params, link=Link.INPUT, register=False)
    remote_tile_setup.uses(mdagtar, link=Link.OUTPUT, register=False, transfer=True)
    uberdax.addJob(remote_tile_setup)
...
    subwf = DAX("%05d.dax" % (tile_id), "ID%05d" % (tile_id))
    subwf.addArguments("-Dpegasus.schema.dax=%s/etc/dax-2.1.xsd" %(os.environ["PEGASUS_HOME"]),
                       "-Dpegasus.catalog.replica.file=%s/rc.data" % (tile_work_dir),
                       "-Dpegasus.catalog.site.file=%s/sites.xml" % (work_dir),
                       "-Dpegasus.transfer.links=true",
                       "--sites", cluster_name,
                       "--cluster", "horizontal",
                       "--basename", "tile-%05d" % (tile_id),
                       "--force",
                       "--output", output_name)
    subwf.addProfile(Profile("dagman", "CATEGORY", "subworkflow"))
    subwf.uses(subdax_file, link=Link.INPUT, register=False)
    uberdax.addDAX(subwf)
```

Chapter 10. Data Management

Replica Selection

Each job in the DAX maybe associated with input LFN's denoting the files that are required for the job to run. To determine the physical replica (PFN) for a LFN, Pegasus queries the Replica catalog to get all the PFN's (replicas) associated with a LFN. The Replica Catalog may return multiple PFN's for each of the LFN's queried. Hence, Pegasus needs to select a single PFN amongst the various PFN's returned for each LFN. This process is known as replica selection in Pegasus. Users can specify the replica selector to use in the properties file.

This document describes the various Replica Selection Strategies in Pegasus.

Configuration

The user properties determine what replica selector Pegasus Workflow Mapper uses. The property **pegasus.selector.replica** is used to specify the replica selection strategy. Currently supported Replica Selection strategies are

1. Default

2. Regex

3. Restricted

4. Local

The values are case sensitive. For example the following property setting will throw a Factory Exception .

```
pegasus.selector.replica  default
```

The correct way to specify is

```
pegasus.selector.replica  Default
```

Supported Replica Selectors

The various Replica Selectors supported in Pegasus Workflow Mapper are explained below.

Note

Starting 4.6.0 release the Default and Regex Replica Selectors return an ordered list with priorities set. pegasus-transfer at runtime will failover to alternate url's specified, if a higher priority source URL is inaccessible.

Default

This is the default replica selector used in the Pegasus Workflow Mapper. If the property pegasus.selector.replica is not defined in properties, then Pegasus uses this selector.

The selector orders the various candidate replica's according to the following rules

1. valid file URL's . That is URL's that have the site attribute matching the site where the executable *pegasus-transfer* is executed.

2. all URL's from preferred site (usually the compute site)

3. all other remotely accessible (non file) URL's

To use this replica selector set the following property

```
pegasus.selector.replica                    Default
```

Regex

This replica selector allows the user to specific regular expressions that can be used to rank various PFN's returned from the Replica Catalog for a particular LFN. This replica selector orders the replicas based on the rank. Lower the rank higher the preference.

The regular expressions are assigned different rank, that determine the order in which the expressions are employed. The rank values for the regex can expressed in user properties using the property.

```
pegasus.selector.replica.regex.rank.[value]              regex-expression
```

The **[value]** in the above property is an integer value that denotes the rank of an expression with a rank value of 1 being the highest rank.

For example, a user can specify the following regex expressions that will ask Pegasus to prefer file URL's over gsiftp url's from example.isi.edu

```
pegasus.selector.replica.regex.rank.1                    file://.*
pegasus.selector.replica.regex.rank.2                    gsiftp://example\.isi\.edu.*
```

User can specify as many regex expressions as they want.

Since Pegasus is in Java , the regex expression support is what Java supports. It is pretty close to what is supported by Perl. More details can be found at http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html

Before applying any regular expressions on the PFN's for a particular LFN that has to be staged to a site X, the file URL's that don't match the site X are explicitly filtered out.

To use this replica selector set the following property

```
pegasus.selector.replica                  Regex
```

Restricted

This replica selector, allows the user to specify good sites and bad sites for staging in data to a particular compute site. A good site for a compute site X, is a preferred site from which replicas should be staged to site X. If there are more than one good sites having a particular replica, then a random site is selected amongst these preferred sites.

A bad site for a compute site X, is a site from which replicas should not be staged. The reason of not accessing replica from a bad site can vary from the link being down, to the user not having permissions on that site's data.

The good | bad sites are specified by the following properties

```
pegasus.replica.*.prefer.stagein.sites
pegasus.replica.*.ignore.stagein.sites
```

where the * in the property name denotes the name of the compute site. A * in the property key is taken to mean all sites. The value to these properties is a comma separated list of sites.

For example the following settings

```
pegasus.selector.replica.*.prefer.stagein.sites          usc
pegasus.replica.uwm.prefer.stagein.sites                 isi,cit
```

means that prefer all replicas from site usc for staging in to any compute site. However, for uwm use a tighter constraint and prefer only replicas from site isi or cit. The pool attribute associated with the PFN's tells the replica selector to what site a replica/PFN is associated with.

The pegasus.replica.*.prefer.stagein.sites property takes precedence over pegasus.replica.*.ignore.stagein.sites property i.e. if for a site X, a site Y is specified both in the ignored and the preferred set, then site Y is taken to mean as only a preferred site for a site X.

To use this replica selector set the following property

```
pegasus.selector.replica          Restricted
```

Local

This replica selector always prefers replicas from the local host (pool attribute set to local) and that start with a file: URL scheme. It is useful, when users want to stagein files to a remote site from the submit host using the Condor file transfer mechanism.

To use this replica selector set the following property

```
pegasus.selector.replica          Local
```

Data Transfers

As part of the Workflow Mapping Process, Pegasus does data management for the executable workflow . It queries a Replica Catalog to discover the locations of the input datasets and adds data movement and registration nodes in the workflow to

1. stage-in input data to the staging sites (a site associated with the compute job to be used for staging. In the shared filesystem setup, staging site is the same as the execution sites where the jobs in the workflow are executed)

2. stage-out output data generated by the workflow to the final storage site.

3. stage-in intermediate data between compute sites if required.

4. data registration nodes to catalog the locations of the output data on the final storage site into the replica catalog.

The separate data movement jobs that are added to the executable workflow are responsible for staging data to a workflow specific directory accessible to the staging server on a staging site associated with the compute sites. Depending on the data staging configuration, the staging site for a compute site is the compute site itself. In the default case, the staging server is usually on the headnode of the compute site and has access to the shared filesystem between the worker nodes and the head node. Pegasus adds a directory creation job in the executable workflow that creates the workflow specific directory on the staging server.

In addition to data, Pegasus does transfer user executables to the compute sites if the executables are not installed on the remote sites before hand. This chapter gives an overview of how transfers of data and executables is managed in Pegasus.

Data Staging Configuration

Pegasus can be broadly setup to run workflows in the following configurations

- **Shared File System**

  This setup applies to where the head node and the worker nodes of a cluster share a filesystem. Compute jobs in the workflow run in a directory on the shared filesystem.

- **NonShared FileSystem**

  This setup applies to where the head node and the worker nodes of a cluster don't share a filesystem. Compute jobs in the workflow run in a local directory on the worker node

- **Condor Pool Without a shared filesystem**

  This setup applies to a condor pool where the worker nodes making up a condor pool don't share a filesystem. All data IO is achieved using Condor File IO. This is a special case of the non shared filesystem setup, where instead of using pegasus-transfer to transfer input and output data, Condor File IO is used.

For the purposes of data configuration various sites, and directories are defined below.

1. **Submit Host**

   The host from where the workflows are submitted . This is where Pegasus and Condor DAGMan are installed. This is referred to as the **"local"** site in the site catalog .

2. **Compute Site**

   The site where the jobs mentioned in the DAX are executed. There needs to be an entry in the Site Catalog for every compute site. The compute site is passed to pegasus-plan using **--sites** option

3. **Staging Site**

   A site to which the separate transfer jobs in the executable workflow (jobs with stage_in , stage_out and stage_inter prefixes that Pegasus adds using the transfer refiners) stage the input data to and the output data from to transfer to the final output site. Currently, the staging site is always the compute site where the jobs execute.

4. **Output Site**

   The output site is the final storage site where the users want the output data from jobs to go to. The output site is passed to pegasus-plan using the **--output** option. The stageout jobs in the workflow stage the data from the staging site to the final storage site.

5. **Input Site**

   The site where the input data is stored. The locations of the input data are catalogued in the Replica Catalog, and the *"site"* attribute of the locations gives us the site handle for the input site.

6. **Workflow Execution Directory**

   This is the directory created by the create dir jobs in the executable workflow on the Staging Site. This is a directory per workflow per staging site. Currently, the Staging site is always the Compute Site.

7. **Worker Node Directory**

   This is the directory created on the worker nodes per job usually by the job wrapper that launches the job.

Shared File System

By default Pegasus is setup to run workflows in the shared file system setup, where the worker nodes and the head node of a cluster share a filesystem.

Figure 10.1. Shared File System Setup

DATA FLOW TO COMPUTE JOBS ON THE WORKER NODES RELYING ON A
SHARED FILESYSTEM

COMPUTE AND STAGING SITE ARE SAME

The data flow is as follows in this case

1. Stagein Job executes (either on Submit Host or Head Node) to stage in input data from Input Sites (1---n) to a workflow specific execution directory on the shared filesystem.

2. Compute Job starts on a worker node in the workflow execution directory. Accesses the input data using Posix IO

3. Compute Job executes on the worker node and writes out output data to workflow execution directory using Posix IO

4. Stageout Job executes (either on Submit Host or Head Node) to stage out output data from the workflow specific execution directory to a directory on the final output site.

Tip

Set **pegasus.data.configuration** to **sharedfs** to run in this configuration.

Non Shared Filesystem

In this setup , Pegasus runs workflows on local file-systems of worker nodes with the the worker nodes not sharing a filesystem. The data transfers happen between the worker node and a staging / data coordination site. The staging site server can be a file server on the head node of a cluster or can be on a separate machine.

Setup

• compute and staging site are the different

• head node and worker nodes of compute site don't share a filesystem

• Input Data is staged from remote sites.

• Remote Output Site i.e site other than compute site. Can be submit host.

Figure 10.2. Non Shared Filesystem Setup

The data flow is as follows in this case

1. Stagein Job executes (either on Submit Host or on staging site) to stage in input data from Input Sites (1---n) to a workflow specific execution directory on the staging site.

2. Compute Job starts on a worker node in a local execution directory. Accesses the input data using pegasus transfer to transfer the data from the staging site to a local directory on the worker node

3. The compute job executes in the worker node, and executes on the worker node.

4. The compute Job writes out output data to the local directory on the worker node using Posix IO

5. Output Data is pushed out to the staging site from the worker node using pegasus-transfer.

6. Stageout Job executes (either on Submit Host or staging site) to stage out output data from the workflow specific execution directory to a directory on the final output site.

In this case, the compute jobs are wrapped as PegasusLite instances.

This mode is especially useful for running in the cloud environments where you don't want to setup a shared filesystem between the worker nodes. Running in that mode is explained in detail here.

Tip

Set p **egasus.data.configuration** to **nonsharedfs** to run in this configuration. The staging site can be specified using the **--staging-site** option to pegasus-plan.

In this setup, Pegasus always stages the input files through the staging site i.e the stage-in job stages in data from the input site to the staging site. The PegasusLite jobs that start up on the worker nodes, then pull the input data from the staging site for each job. In some cases, it might be useful to setup the PegasusLite jobs to pull input data directly from the input site without going through the staging server. This is based on the assumption that the worker nodes can access the input site. Starting 4.3 release, users can enable this. However, you should be aware that the access to

the input site is no longer throttled (as in case of stage in jobs). If large number of compute jobs start at the same time in a workflow, the input server will see a connection from each job.

Tip

Set **pegasus.transfer.bypass.input.staging** to **true**to enable the bypass of staging of input files via the staging server.

Condor Pool Without a Shared Filesystem

This setup applies to a condor pool where the worker nodes making up a condor pool don't share a filesystem. All data IO is achieved using Condor File IO. This is a special case of the non shared filesystem setup, where instead of using pegasus-transfer to transfer input and output data, Condor File IO is used.

Setup

- Submit Host and staging site are same

- head node and worker nodes of compute site don't share a filesystem

- Input Data is staged from remote sites.

- Remote Output Site i.e site other than compute site. Can be submit host.

Figure 10.3. Condor Pool Without a Shared Filesystem

The data flow is as follows in this case

1. Stagein Job executeson the submit host to stage in input data from Input Sites (1---n) to a workflow specific execution directory on the submit host

2. Compute Job starts on a worker node in a local execution directory. Before the compute job starts, Condor transfers the input data for the job from the workflow execution directory on thesubmit host to the local execution directory on the worker node.

3. The compute job executes in the worker node, and executes on the worker node.

4. The compute Job writes out output data to the local directory on the worker node using Posix IO

5. When the compute job finishes, Condor transfers the output data for the job from the local execution directory on the worker node to the workflow execution directory on the submit host.

6. Stageout Job executes (either on Submit Host or staging site) to stage out output data from the workflow specific execution directory to a directory on the final output site.

In this case, the compute jobs are wrapped as PegasusLite instances.

This mode is especially useful for running in the cloud environments where you don't want to setup a shared filesystem between the worker nodes. Running in that mode is explained in detail here.

Tip

Set p **egasus.data.configuration** to **condorio** to run in this configuration. In this mode, the staging site is automatically set to site **local**

In this setup, Pegasus always stages the input files through the submit host i.e the stage-in job stages in data from the input site to the submit host (local site). The input data is then transferred to remote worker nodes from the submit host using Condor file transfers. In the case, where the input data is locally accessible at the submit host i.e the input site and the submit host are the same, then it is possible to bypass the creation of separate stage in jobs that copy the data to the workflow specific directory on the submit host. Instead, Condor file transfers can be setup to transfer the input files directly from the locally accessible input locations (file URL's with "*site*" attribute set to local) specified in the replica catalog. Starting 4.3 release, users can enable this.

Tip

Set **pegasus.transfer.bypass.input.staging** to **true**to bypass the creation of separate stage in jobs.

Local versus Remote Transfers

As far as possible, Pegasus will ensure that the transfer jobs added to the executable workflow are executed on the submit host. By default, Pegasus will schedule a transfer to be executed on the remote staging site only if there is no way to execute it on the submit host. Some scenarios where transfer jobs are executed on remote sites are as follows:

- the file server specified for the staging site/compute site is a file server. In that case, Pegasus will schedule all the stage in data movement jobs on the compute site to stage-in the input data for the workflow.

- a user has symlinking turned on. In that case, the transfer jobs that symlink against the input data on the compute site, will be executed remotely (on the compute site).

In certain execution environments, such a local campus cluster the compute site and the local share a filesystem (i.e. compute site has file servers specified for the staging/compute site, and the scratch and storage directories mentioned for the compute site are locally mounted on the submit host), it is beneficial to have the remote transfer jobs run locally and hence bypass going through the local scheduler queue. In that case, users can set a boolean profile auxillary.local in pegasus namespace in the site catalog for the compute/staging site to true.

Users can specify the property **pegasus.transfer.*.remote.sites** to change the default behaviour of Pegasus and force pegasus to run different types of transfer jobs for the sites specified on the remote site. The value of the property is a comma separated list of compute sites for which you want the transfer jobs to run remotely.

The table below illustrates all the possible variations of the property.

Table 10.1. Property Variations for pegasus.transfer.*.remote.sites

Property Name	Applies to
pegasus.transfer.stagein.remote.sites	the stage in transfer jobs

Property Name	Applies to
pegasus.transfer.stageout.remote.sites	the stage out transfer jobs
pegasus.transfer.inter.remote.sites	the inter site transfer jobs
pegasus.transfer.*.remote.sites	all types of transfer jobs

The prefix for the transfer job name indicates whether the transfer job is to be executed locallly (on the submit host) or remotely (on the compute site). For example stage_in_local_ in a transfer job name stage_in_local_isi_viz_0 indicates that the transfer job is a stage in transfer job that is executed locally and is used to transfer input data to compute site isi_viz. The prefix naming scheme for the transfer jobs is **[stage_in|stage_out|inter]_[local|remote]_** .

Controlling Transfer Parallelism

When it comes to data transfers, Pegasus ships with a default configuration which is trying to strike a balance between performance and aggressiveness. We obviously want data transfers to be as quick as possibly, but we also do not want our transfers to overwhelm data services and systems.

Starting 4.8.0 release, the default configuration of Pegasus now adds transfer jobs and cleanup jobs based on the number of jobs at a particular level of the workflow. For example, for every 10 compute jobs on a level of a workflow, one data transfer job(stage-in and stage-out) is created. The default configuration also sets how many threads such a pegasus-transfer job can spawn. Cleanup jobs are similarly constructed with an internal ratio of 5.

Information on how to control the number of stagein and stageout jobs can be found in the Data Movement Nodes section.

How to control the number of threads pegasus-transfer can use depends on if you want to control standard transfer jobs, or PegasusLite. For the former, see the pegasus.transfer.threads property, and for the latter the pegasus.transfer.lite.threads property.

Symlinking Against Input Data

If input data for a job already exists on a compute site, then it is possible for Pegasus to symlink against that data. In this case, the remote stage in transfer jobs that Pegasus adds to the executable workflow will symlink instead of doing a copy of the data.

Pegasus determines whether a file is on the same site as the compute site, by inspecting the *"site"* attribute associated with the URL in the Replica Catalog. If the *"site"* attribute of an input file location matches the compute site where the job is scheduled, then that particular input file is a candidate for symlinking.

For Pegasus to symlink against existing input data on a compute site, following must be true

1. Property **pegasus.transfer.links** is set to **true**

2. The input file location in the Replica Catalog has the *"site"* attribute matching the compute site.

> ## Tip
>
> To confirm if a particular input file is symlinked instead of being copied, look for the destination URL for that file in stage_in_remote*.in file. The destination URL will start with symlink:// .

In the symlinking case, Pegasus strips out URL prefix from a URL and replaces it with a file URL.

For example if a user has the following URL catalogued in the Replica Catalog for an input file f.input

```
f.input   gsiftp://server.isi.edu/shared/storage/input/data/f.input site="isi"
```

and the compute job that requires this file executes on a compute site named isi , then if symlinking is turned on the data stage in job (stage_in_remote_viz_0) will have the following source and destination specified for the file

```
#viz viz
file:///shared/storage/input/data/f.input  symlink://shared-scratch/workflow-exec-dir/f.input
```

Addition of Separate Data Movement Nodes to Executable Workflow

Pegasus relies on a Transfer Refiner that comes up with the strategy on how many data movement nodes are added to the executable workflow. All the compute jobs scheduled to a site share the same workflow specific directory. The transfer refiners ensure that only one copy of the input data is transferred to the workflow execution directory. This is to prevent data clobbering . Data clobbering can occur when compute jobs of a workflow share some input files, and have different stage in transfer jobs associated with them that are staging the shared files to the same destination workflow execution directory.

Pegasus supports three different transfer refiners that dictate how the stagein and stageout jobs are added for the workflow.The default Transfer Refiner used in Pegasus is the BalancedCluster Refiner. Starting 4.8.0 release, the default configuration of Pegasus now adds transfer jobs and cleanup jobs based on the number of jobs at a particular level of the workflow. For example, for every 10 compute jobs on a level of a workflow, one data transfer job(stage-in and stage-out) is created.

The transfer refiners also allow the user to specify how many local|remote stagein|stageout jobs are created per execution site.

The behavior of the refiners (BalancedCluster and Cluster) are controlled by specifying certain pegasus profiles

1. either with the execution sites in the site catalog

2. OR globally in the properties file

Table 10.2. Pegasus Profile Keys For the Cluster Transfer Refiner

Profile Key	Description
stagein.clusters	This key determines the maximum number of stage-in jobs that are can executed locally or remotely per compute site per workflow.
stagein.local.clusters	This key provides finer grained control in determining the number of stage-in jobs that are executed locally and are responsible for staging data to a particular remote site.
stagein.remote.clusters	This key provides finer grained control in determining the number of stage-in jobs that are executed remotely on the remote site and are responsible for staging data to it.
stageout.clusters	This key determines the maximum number of stage-out jobs that are can executed locally or remotely per compute site per workflow.
stageout.local.clusters	This key provides finer grained control in determining the number of stage-out jobs that are executed locally and are responsible for staging data from a particular remote site.
stageout.remote.clusters	This key provides finer grained control in determining the number of stage-out jobs that are executed remotely on the remote site and are responsible for staging data from it.

Tip

Which transfer refiner to use is controlled by property pegasus.transfer.refiner

BalancedCluster

This is a new transfer refiner that was introduced in Pegasus 4.4.0 and is the default one used in Pegasus. It does a round robin distribution of the files amongst the stagein and stageout jobs per level of the workflow. The figure below illustrates the behavior of this transfer refiner.

Figure 10.4. BalancedCluster Transfer Refiner : Input Data To Workflow Specific Directory on Shared File System

Addition of Data Stage-In and Stage-Out Nodes by the BalancedCluster Transfer Refiner

1. Original Workflow

2. Workflow After Adding the Stage-In and Stage-Out Nodes
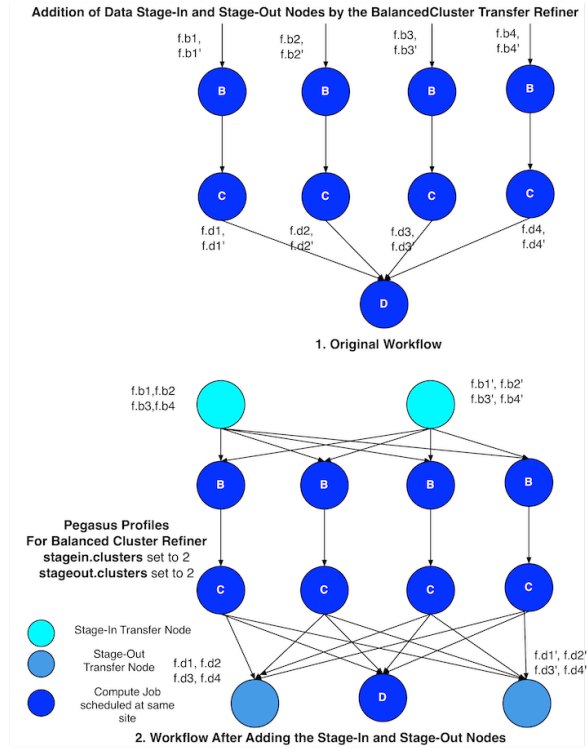
Cluster

This transfer refiner is similar to BalancedCluster but differs in the way how distribution of files happen across stagein and stageout jobs per level of the workflow. In this refiner, all the input files for a job get associated with a single transfer job. As illustrated in the figure below each compute usually gets associated with one stagein transfer job. In contrast, for the BalancedCluster a compute job maybe associated with multiple data stagein jobs.

Figure 10.5. Cluster Transfer Refiner : Input Data To Workflow Specific Directory on Shared File System
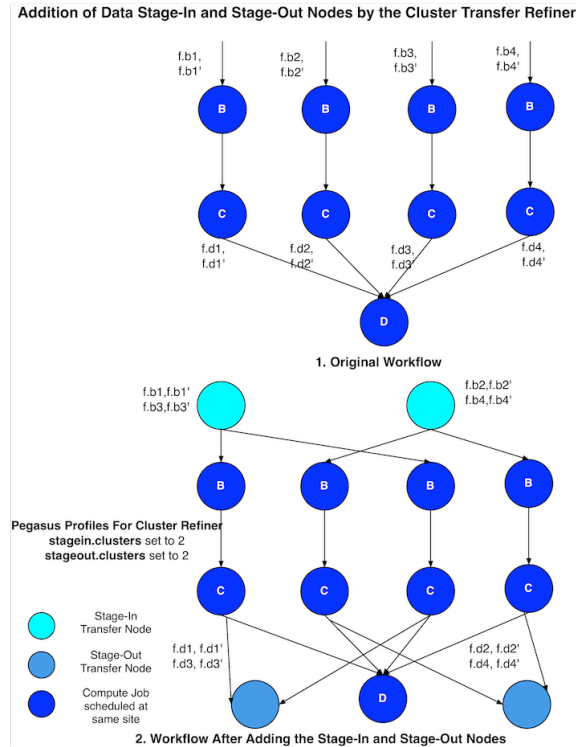
Basic

Pegasus also supports a basic Transfer Refiner that adds one stagein and stageout job per compute job of the workflow. This is not recommended to be used for large workflows as the number of data transfer nodes in the worst case are 2n where n is the number of compute jobs in the workflow.

Executable Used for Transfer and Cleanup Jobs

Pegasus refers to a python script called **pegasus-transfer** as the executable in the transfer jobs to transfer the data. pegasus-transfer looks at source and destination url and figures out automatically which underlying client to use. pegasus-transfer is distributed with the PEGASUS and can be found at $PEGASUS_HOME/bin/pegasus-transfer.

Currently, pegasus-transfer interfaces with the following transfer clients

Table 10.3. Transfer Clients interfaced to by pegasus-transfer

Transfer Client	Used For
gfal-copy	staging file to and from GridFTP servers
globus-url-copy	staging files to and from GridFTP servers, only if gfal is not detected in the path.
gfal-copy	staging files to and from SRM or XRootD servers
wget	staging files from a HTTP server
cp	copying files from a POSIX filesystem
ln	symlinking against input files
pegasus-s3	staging files to and from S3 buckets in Amazon Web Services

Transfer Client	Used For
gsutil	staging files to and from Google Storage buckets
scp	staging files using scp
gsiscp	staging files using gsiscp and X509
iget	staging files to and from iRODS servers

For remote sites, Pegasus constructs the default path to pegasus-transfer on the basis of PEGASUS_HOME env profile specified in the site catalog. To specify a different path to the pegasus-transfer client , users can add an entry into the transformation catalog with fully qualified logical name as **pegasus::pegasus-transfer**

Preference of GFAL over GUC

JGlobus is no longer actively supported and is not in compliance with RFC 2818 [https://docs.globus.org/security-bulletins/2015-12-strict-mode] . As a result cleanup jobs using pegasus-gridftp client would fail against the servers supporting the strict mode. We have removed the pegasus-gridftp client and now use gfal clients as globus-url-copy does not support removes. If gfal is not available, globus-url-copy is used for cleanup by writing out zero bytes files instead of removing them.

If you want to force globus-url-copy to be preferred over GFAL, set the **PEGASUS_FORCE_GUC=1** environment variable in the site catalog for the sites you want the preference to be enforced. Please note that we expect globus-url-copy support to be completely removed in future releases of Pegasus due to the end of life of Globus Toolkit (see announcement [https://www.globus.org/blog/support-open-source-globus-toolkit-ends-january-2018]).

Staging of Executables

Users can get Pegasus to stage the user executables (executables that the jobs in the DAX refer to) as part of the transfer jobs to the workflow specific execution directory on the compute site. The URL locations of the executables need to be specified in the transformation catalog as the PFN and the type of executable needs to be set to **STAGEABLE** .

The location of a transformation can be specified either in

- DAX in the executables section. More details here .

- Transformation Catalog. More details here .

A particular transformation catalog entry of type STAGEABLE is compatible with a compute site only if all the System Information attributes associated with the entry match with the System Information attributes for the compute site in the Site Catalog. The following attributes make up the System Information attributes

1. arch

2. os

3. osrelease

4. osversion

Transformation Mappers

Pegasus has a notion of transformation mappers that determines what type of executables are picked up when a job is executed on a remote compute site. For transfer of executables, Pegasus constructs a soft state map that resides on top of the transformation catalog, that helps in determining the locations from where an executable can be staged to the remote site.

Users can specify the following property to pick up a specific transformation mapper

```
pegasus.catalog.transformation.mapper
```

Currently, the following transformation mappers are supported.

Table 10.4. Transformation Mappers Supported in Pegasus

Transformation Mapper	Description
Installed	This mapper only relies on transformation catalog entries that are of type INSTALLED to construct the soft state map. This results in Pegasus never doing any transfer of executables as part of the workflow. It always prefers the installed executables at the remote sites
Staged	This mapper only relies on matching transformation catalog entries that are of type STAGEABLE to construct the soft state map. This results in the executable workflow referring only to the staged executables, irrespective of the fact that the executables are already installed at the remote end
All	This mapper relies on all matching transformation catalog entries of type STAGEABLE or INSTALLED for a particular transformation as valid sources for the transfer of executables. This the most general mode, and results in the constructing the map as a result of the cartesian product of the matches.
Submit	This mapper only on matching transformation catalog entries that are of type STAGEABLE and reside at the submit host (site local), are used while constructing the soft state map. This is especially helpful, when the user wants to use the latest compute code for his computations on the grid and that relies on his submit host.

Staging of Worker Package

The worker package contains runtime tools such as *pegasus-kickstart* and *pegasus-transfer*, and is required to be available for most jobs.

How the package is made available to the jobs depends on multiple factors. For example, a pre-installed Pegasus can be used if the location is set using the environment profile PEGASUS_HOME for the site in the Site Catalog.

If Pegasus is not already available on the execution site, the worker package can be staged by setting the following property:

```
pegasus.transfer.worker.package          true
```

Note that how the package is transferred and accessed differs based on the configured data management mode:

- *sharedfs* mode: the package is staged in to the shared filesystem once, and reused for all the jobs

- *nonsharedfs* or *condorio* mode: each job carries a worker package. This is obviously less efficient, but the size of the worker package is kept small to minimize the impact of these extra transfers.

Which worker package is used is determined in the following order:

- There is an entry for pegasus::worker executable in the transformation catalog. Information on how to construct that entry is provided below.

- The planner at runtime creates a worker package out of the binary installation, and puts it in the submit directory. This worker package is used if the OS and architecture of the created worker package match with remote site, or there is an exact match with (osrelease and osversion) if specified by the user in the site catalog for the remote site.

- The worker package compatible with the remote site is available as a binary from the Pegasus download site.

- At runtime, in the *nonsharedfs* or *condorio* modes, extra checks are made to make sure the worker package matches the Pegasus version and the OS and architecture. The reason is that these workflows might be running in an heterogeneous environment, and thus there is no way to know before the job starts what worker package is required. If the runtime check fails, a worker package matching the Pegasus version, OS and architecture will be downloaded from

the Pegasus download site. This behavior can be controlled with the pegasus.transfer.worker.package.autodownload and pegasus.transfer.worker.package.strict properties.

If you want to specify a particular worker package to use, you can specify the transformation **pegasus::worker** in the transformation catalog with:

- type set to STAGEABLE

- System Information attributes of the transformation catalog entry match the System Information attributes of the compute site.

- the PFN specified should be a remote URL that can be pulled to the compute site.

```
# example of specifying a worker package in the transformation catalog
tr pegasus::worker {
site corbusier {
    pfn "https://download.pegasus.isi.edu/pegasus/4.8.0dev/pegasus-worker-4.8.0dev-
x86_64_macos_10.tar.gz"
    arch "x86_64"
    os "MACOSX"
    type "INSTALLED"
  }
}
```

Staging of Application Containers

Pegasus treats containers as other files in terms of data management. Container to be used for a job is tracked as an input dependency that needs to be staged if it is not already there. Similar to executables, you specify the location for your container image in the Transformation Catalog. You can specify the source URL's for containers as the following.

1. URL to a container hosted on a central hub repository

   Example of a docker hub URL is docker:///rynge/montage:latest, while for singularity shub://pegasus-isi/fedora-montage

2. URL to a container image file on a file server.

   - **Docker -** Docker supports loading of containers from a tar file, Hence, containers images can only be specified as tar files and the extension for the filename is not important.

   - **Singularity -** Singularity supports container images in various forms and relies on the extension in the filename to determine what format the file is in. Pegasus supports the following extensions for singularity container images

     - .img

     - .tar

     - .tar.gz

     - .tar.bz2

     - .cpio

     - .cpio.gz

     Singularity will fail to run the container if you don't specify the right extension , when specify the source URL for the image.

In both the cases, Pegasus will place the container image on the staging site used for the workflow, as part of the data stage-in nodes, using pegasus-transfer. When pulling in an image from a container hub repository, pegasus-transfer will export the container as a tar file in case of Docker, and as .img file in case of Singularity

Symlinking

Since, Pegasus only mounts the job directory determined by PegasusLite into the application container, symlinking of input data sets does not work. This is because the symlink in PegasusLite directory points to a source directory on

the worker node, that is not mounted in the container. Hence user's jobs would fail. Hence, Pegasus will automatically disable symlinking for jobs that use containers. The only exception being the application container itself. If you specify a URL for the container image, the image will be symlinked if Pegasus determines that it can be.

Enabling symlinking of containers is useful, when running large workflows on a single cluster. Pegasus can pull the image from the container repository once, and place it on the shared filesystem where it can then be symlinked from, when the PegasusLite jobs start on the worker nodes of that cluster. In order to do this, you need to be running the nonsharedfs data configuration mode with the staging site set to be the same as the compute site.

Staging of Job Checkpoint Files

Pegasus has support for transferring job checkpoint files back to the staging site, when a job exceeds it's advertised running time. In order to use this feature, you need to

1. Associate a job checkpoint file (that the job creates) with the job in the DAX. A checkpoint file is specified by setting the link attribute to checkpoint for the uses tag.

2. Associate a Pegasus profile key named **checkpoint.time** is the time in minutes after which a job is sent the TERM signal by pegasus-kickstart, telling it to create the checkpoint file.

3. Associate a Pegasus profile key named **maxwalltime** with the job that specifies the max runtime in minutes before the job will be killed by the local resource manager (such as PBS) deployed on the site. Usually, this value should be associated with the execution site in the site catalog.

Pegasus planner uses the above mentioned profile keys to setup pegasus-kickstart such that the job is sent a TERM signal when the checkpoint time of job is reached. A KILL signal is sent at (checkpoint.time + (maxwalltime-checkpoint.time)/2) minutes. This ensures that there is enough time for pegasus-lite to transfer the checkpoint file before the job is killed by the underlying scheduler.

Using Amazon S3 as a Staging Site

Pegasus can be configured to use Amazon S3 as a staging site. In this mode, Pegasus transfers workflow inputs from the input site to S3. When a job runs, the inputs for that job are fetched from S3 to the worker node, the job is executed, then the output files are transferred from the worker node back to S3. When the jobs are complete, Pegasus transfers the output data from S3 to the output site.

In order to use S3, it is necessary to create a config file for the S3 transfer client, pegasus-s3. See the man page for details on how to create the config file. You also need to specify S3 as a staging site.

Next, you need to modify your site catalog to tell the location of your s3cfg file. See the section on credential staging.

The following site catalog shows how to specify the location of the s3cfg file on the local site and how to specify an Amazon S3 staging site:

```
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog
             http://pegasus.isi.edu/schema/sc-3.0.xsd" version="3.0">
    <site handle="local" arch="x86_64" os="LINUX">
        <head-fs>
            <scratch>
                <shared>
                    <file-server protocol="file" url="file://" mount-point="/tmp/wf/work"/>
                    <internal-mount-point mount-point="/tmp/wf/work"/>
                </shared>
            </scratch>
            <storage>
                <shared>
                    <file-server protocol="file" url="file://" mount-point="/tmp/wf/storage"/>
                    <internal-mount-point mount-point="/tmp/wf/storage"/>
                </shared>
            </storage>
        </head-fs>
        <profile namespace="env" key="S3CFG">/home/username/.s3cfg</profile>
    </site>
```

```
            <site handle="s3" arch="x86_64" os="LINUX">
                <head-fs>
                    <scratch>
                        <shared>
                            <!-- wf-scratch is the name of the S3 bucket that will be used -->
                            <file-server protocol="s3" url="s3://user@amazon" mount-point="/wf-scratch"/>
                            <internal-mount-point mount-point="/wf-scratch"/>
                        </shared>
                    </scratch>
                </head-fs>
            </site>
        <site handle="condorpool" arch="x86_64" os="LINUX">
            <head-fs>
                <scratch/>
                <storage/>
            </head-fs>
            <profile namespace="pegasus" key="style">condor</profile>
            <profile namespace="condor" key="universe">vanilla</profile>
            <profile namespace="condor" key="requirements">(Target.Arch == "X86_64")</profile>
        </site>
</sitecatalog>
```

iRODS data access

iRODS can be used as a input data location, a storage site for intermediate data during workflow execution, or a location for final output data. Pegasus uses a URL notation to identify iRODS files. Example:

```
irods://some-host.org/path/to/file.txt
```

The path to the file is **relative** to the internal iRODS location. In the example above, the path used to refer to the file in iRODS is *path/to/file.txt* (no leading /).

See the section on credential staging for information on how to set up an irodsEnv file to be used by Pegasus.

GridFTP over SSH (sshftp)

Instead of using X.509 based security, newer version of Globus GridFTP can be configured to set up transfers over SSH. See the Globus Documentation [http://toolkit.globus.org/toolkit/docs/6.0/gridftp/admin/#gridftp-admin-config-security-sshftp] for details on installing and setting up this feature.

Pegasus requires the ability to specify which SSH key to be used at runtime, and thus a small modification is necessary to the default Globus configuration. On the hosts where Pegasus initiates transfers (which depends on the data configuration of the workflow), please replace *gridftp-ssh*, usually located under */usr/share/globus/gridftp-ssh*, with:

```
#!/bin/bash

url_string=$1
remote_host=$2
port=$3
user=$4

port_str=""
if  [ "X" = "X$port" ]; then
    port_str=""
else
    port_str=" -p $port "
fi

if  [ "X" != "X$user" ]; then
    remote_host="$user@$remote_host"
fi

remote_default1=.globus/sshftp
remote_default2=/etc/grid-security/sshftp
remote_fail="echo -e 500 Server is not configured for SSHFTP connections.\\\r\\\n"
remote_program=$GLOBUS_REMOTE_SSHFTP
if [ "X" = "X$remote_program" ]; then
    remote_program="(( test -f $remote_default1 && $remote_default1 ) || ( test -f $remote_default2
 && $remote_default2 ) || $remote_fail )"
fi
```

```
if [ "X" != "X$GLOBUS_SSHFTP_PRINT_ON_CONNECT" ]; then
    echo "Connecting to $1 ..." >/dev/tty
fi

# for pegasus-transfer
extra_opts=" -o StrictHostKeyChecking=no"
if [ "x$SSH_PRIVATE_KEY" != "x" ]; then
    extra_opts="$extra_opts -i $SSH_PRIVATE_KEY"
fi

exec /usr/bin/ssh $extra_opts $port_str $remote_host $remote_program
```

Once configured, you should be able to use URLs such as *sshftp://username@host/foo/bar.txt* in your workflows.

Globus Online

Globus Online [http://globus.org] is a transfer service with features such as policy based connection management and automatic failure detection and recovery. Pegasus has limited the support for Globus Online transfers.

If you want to use Globus Online in your workflow, all data has to be accessible via a Globus Online endpoint. You can not mix Globus Online endpoints with other protocols. For most users, this means they will have to create an endpoint for their submit host and probably modify both the replica catalog and DAX generator so that all URLs in the workflow are for Globus Online endpoints.

There are two levels of credentials required. One is for the workflow to use the Globus Online API, which is handled by OAuth tokens, provided by Globus Auth service. The second level is for the endpoints, which the user will have to manage via the Globus Online web interface. The required steps are:

1. Using *pegasus-globus-online-init*, provide authorization to Pegasus and retrieve your transfer access tokens. By default Pegasus acquires temporary tokens that expire within a few days. Using --permanent option you can request refreshable tokens that last indefinetely (or until access is revoked).

2. In the Globus Online web interface, under Endpoints, find the endpoints you need for the workflow, and activate them. Note that you should activate them for the whole duration of the workflow or you will have to regularly log in and re-activate the endpoints during workflow execution.

URLs for Globus Online endpoint data follows the following scheme: *go://[endpoint]/[path]*. For example, for a user with the Globus Online private endpoint *bob#researchdata* and a file */home/bsmith/experiment/1.dat*, the URL would be: *go://bob#researchdata/home/bsmith/experiment/1.dat*

Credentials Management

Pegasus tries to do data staging from localhost by default, but some data scenarios makes some remote jobs do data staging. An example of such a case is when running in nonsharedfs mode. Depending on the transfer protocols used, the job may have to carry credentials to enable these data transfers. To specify where which credential to use and where Pegasus can find it, use environment variable profiles in your site catalog. The supported credential types are X.509 grid proxies, Amazon AWS S3 keys, Google Cloud Platform OAuth token (.boto file), iRods password and SSH keys.

Credentials are usually associated per site in the site catalog. Users can associate the credentials either as a Pegasus profile or an environment profile with the site.

1. A pegasus profile with the value pointing to the path to the credential on the local site or the submit host. If a pegasus credential profile associated with the site, then Pegasus automatically transfers it along with the remote jobs.

2. A env profile with the value pointing to the path to the credential on the remote site. If an env profile is specified, then no credential is transferred along with the job. Instead the job's environment is set to ensure that the job picks up the path to the credential on the remote site.

Tip

Specifying credentials as Pegasus profiles was introduced in 4.4.0 release.

In case of data transfer jobs, it is possible to associate different credentials for a single file transfer (one for the source server and the other for the destination server) . For example, when leveraging GridFTP transfers between two sides that accept different grid credentials such as XSEDE Stampede site and NCSA Bluewaters. In that case, Pegasus picks up the associated credentials from the site catalog entries for the source and the destination sites associated with the transfer.

X.509 Grid Proxies

If the grid proxy is required by transfer jobs, and the proxy is in the standard location, Pegasus will pick the proxy up automatically. For non-standard proxy locations, you can use the X509_USER_PROXY environment variable. Site catalog example:

```
<profile namespace="pegasus" key="X509_USER_PROXY" >/some/location/x509up</profile>
```

Amazon AWS S3

If a workflow is using s3 URLs, Pegasus has to be told where to find the .s3cfg file. This format of the file is described in the pegaus-s3 command line client's man page. For the file to be picked up by the workflow, set the S3CFG profile to the location of the file. Site catalog example:

```
<profile namespace="pegasus" pegasus="S3CFG" >/home/user/.s3cfg</profile>
```

Google Storage

If a workflow is using gs:// URLs, Pegasus needs access to a Google Storage service account. First generate the credential by following the instructions at:

https://cloud.google.com/storage/docs/authentication#service_accounts

Download the credential in PKCS12 format, and then use "gsutil config -e" to generate a .boto file. For example:

```
$ gsutil config -e
This command will create a boto config file at /home/username/.boto
containing your credentials, based on your responses to the following
questions.
What is your service account email address? some-identifier@developer.gserviceaccount.com
What is the full path to your private key file? /home/username/my-cred.p12
What is the password for your service key file [if you haven't set one
explicitly, leave this line blank]?

Please navigate your browser to https://cloud.google.com/console#/project,
then find the project you will use, and copy the Project ID string from the
second column. Older projects do not have Project ID strings. For such projects,
click the project and then copy the Project Number listed under that project.

What is your project-id? your-project-id

Boto config file "/home/username/.boto" created. If you need to use a
proxy to access the Internet please see the instructions in that file.
```

Pegasus has to be told where to find both the .boto file as well as the PKCS12 file. For the files to be picked up by the workflow, set the BOTO_CONFIG and GOOGLE_PKCS12 profiles for the storage site. Site catalog example:

```
<profile namespace="pegasus" key="BOTO_CONFIG" >/home/user/.boto</profile>
<profile namespace="pegasus" key="GOOGLE_PKCS12" >/home/user/.google-service-account.p12</profile>
```

iRods Password

If a workflow is using iRods URLs, Pegasus has to be given an irodsEnv file. It is a standard file, with the addtion of an password attribute. Example when using iRods 3.X:

```
# iRODS personal configuration file.
#
# iRODS server host name:
```

```
irodsHost 'some.host.edu'
# iRODS server port number:
irodsPort 1259

# Account name:
irodsUserName 'someuser'
# Zone:
irodsZone 'somezone'

# this is used with Pegasus
irodsPassword 'somesecretpassword'
```

iRods 4.0 switched to a JSON based configuration file. Pegasus can handle either config file. JSON Example:

```
{
    "irods_host": "some.host.edu",
    "irods_port": 1247,
    "irods_user_name": "someuser",
    "irods_zone_name": "somezone",
    "irodsPassword" : "somesecretpassword"
}
```

The location of the file can be given to the workflow using the `irodsEnvFile` environment profile. Site catalog example:

```
<profile namespace="pegasus" key="irodsEnvFile" >/home/user/.irods/.irodsEnv</profile>
```

SSH Keys

New in Pegasus 4.0 is the support for data staging with scp using ssh public/private key authentication. In this mode, Pegasus transports a private key with the jobs. The storage machines will have to have the public part of the key listed in ~/.ssh/authorized_keys.

Warning

SSH keys should be handled in a secure manner. In order to keep your personal ssh keys secure, It is recommended that a special set of keys are created for use with the workflow. Note that Pegasus will not pick up ssh keys automatically. The user will have to specify which key to use with `SSH_PRIVATE_KEY`.

The location of the ssh private key can be specified with the `SSH_PRIVATE_KEY` environment profile. Site catalog example:

```
<profile namespace="pegasus" key="SSH_PRIVATE_KEY" >/home/user/wf/wfsshkey</profile>
```

Staging Mappers

Starting 4.7 release, Pegasus has support for staging mappers in the **nonsharedfs** data configuration. The staging mappers determine what sub directory on the staging site a job will be associated with. Before, the introduction of staging mappers, all files associated with the jobs scheduled for a particular site landed in the same directory on the staging site. As a result, for large workflows this could degrade filesystem performance on the staging servers.

To configure the staging mapper, you need to specify the following property

```
pegasus.dir.staging.mapper  <name of the mapper to use>
```

The following mappers are supported currently, with Hashed being the default .

1. **Flat** : This mapper results in Pegasus placing all the job submit files in the staging site directory as determined from the Site Catalog and planner options. This can result in too many files in one directory for large workflows, and was the only option before Pegasus 4.7.0 release.

2. **Hashed** : This mapper results in the creation of a deep directory structure rooted at the staging site directory created by the create dir jobs. The binning is at the job level, and not at the file level i.e each job will push out it's outputs to the same directory on the staging site, independent of the number of output files. To control behavior of this mapper, users can specify the following properties

```
pegasus.dir.staging.mapper.hashed.levels     the number of directory levels used to accomodate
 the files. Defaults to 2.
pegasus.dir.staging.mapper.hashed.multiplier the number of files associated with a job in the
 submit directory. defaults to 5.
```

Note

The staging mappers are only triggered if pegasus.data.configuration is set to nonsharedfs

Output Mappers

Starting 4.3 release, Pegasus has support for output mappers, that allow users fine grained control over how the output files on the output site are laid out. By default, Pegasus stages output products to the storage directory specified in the site catalog for the output site. Output mappers allow users finer grained control over where the output files are placed on the output site.

To configure the output mapper, you need to specify the following property

```
pegasus.dir.storage.mapper  <name of the mapper to use>
```

The following mappers are supported currently

1. **Flat** : By default, Pegasus will place the output files in the storage directory specified in the site catalog for the output site.

2. **Fixed** : This mapper allows users to specify an externally accesible url to the storage directory in their properties file. To use this mapper, the following property needs to be set.

   • pegasus.dir.storage.mapper.fixed.url an externally accessible URL to the storage directory on the output site e.g. gsiftp://outputs.isi.edu/shared/outputs

   Note: For hierarchal workflows, the above property needs to be set separately for each dax job, if you want the sub workflow outputs to goto a different directory.

3. **Hashed** : This mapper results in the creation of a deep directory structure on the output site, while populating the results. The base directory on the remote end is determined from the site catalog. Depending on the number of files being staged to the remote site a Hashed File Structure is created that ensures that only 256 files reside in one directory. To create this directory structure on the storage site, Pegasus relies on the directory creation feature of the underlying file servers such as theGrid FTP server, which appeared in globus 4.0.x

4. **Replica:** This mapper determines the path for an output file on the output site by querying an output replica catalog. The output site is one that is passed on the command line. The output replica catalog can be configured by specifying the following properties.

   • pegasus.dir.storage.mapper.replica Regex|File

   • pegasus.dir.storage.mapper.replica.file the RC file at the backend to use

   Please note that the output replica catalog (even though the formats are the same) is logically different from the input replica catalog, where you specify the locations for the input files. You cannot specify the locations for the output files to be used by the mapper in the DAX. The format for the File based replica catalog is described here, while for the Regex it is here.

Effect of pegasus.dir.storage.deep

For Flat and Hashed output mappers, the base directory to which the add on component is added is determined by the property pegasus.dir.storage.deep . The output directory on the output site is determined from the site catalog.

If pegasus.dir.storage.deep is set to true, then to this base directory, a relative directory is appended i.e. $storage_base = $base + $relative_directory. The relative directory is computed on the basis of the --relative-dir option. If that is not set, then defaults to the relative submit directory for the workflow (usually $user/$vogroup/$label/runxxxx).This is the base directory that is passed to the storage mappers.

Data Cleanup

When executing large workflows, users often may run out of diskspace on the remote clusters / staging site. Pegasus provides a couple of ways of enabling automated data cleanup on the staging site (i.e the scratch space used by the workflows). This is achieved by adding data cleanup jobs to the executable workflow that the Pegasus Mapper generates. These cleanup jobs are responsible for removing files and directories during the workflow execution. To enable data cleanup you can pass the --cleanup option to pegasus-plan . The value passed decides the cleanup strategy implemented

1. **none** disables cleanup altogether. The planner does not add any cleanup jobs in the executable workflow whatsoever.

2. **leaf** the planner adds a leaf cleanup node per staging site that removes the directory created by the create dir job in the workflow

3. **inplace** the mapper adds cleanup nodes per level of the workflow in addition to leaf cleanup nodes. The nodes remove files no longer required during execution. For example, an added cleanup node will remove input files for a particular compute job after the job has finished successfully. Starting 4.8.0 release, the number of cleanup nodes created by this algorithm on a particular level, is dictated by the number of nodes it encounters on a level of the workflow.

4. **constraint** the mapper adds cleanup nodes to constraint the amount of storage space used by a workflow, in addition to leaf cleanup nodes. The nodes remove files no longer required during execution. The added cleanup node guarantees limits on disk usage. File sizes are read from the **size** flag in the DAX, or from a CSV file (*pegasus.file.cleanup.constraint.csv*).

   ## Note

   For large workflows with lots of files, the inplace strategy may take a long time as the algorithm works at a per file level to figure out when it is safe to remove a file.

Behaviour of the cleanup strategies implemented in the Pegasus Mapper can be controlled by properties described here.

Data Cleanup in Hierarchal Workflows

By default, for hierarchal workflows the inplace cleanup is always turned off. This is because the cleanup algorithm (InPlace) does not work across the sub workflows. For example, if you have two DAX jobs in your top level workflow and the child DAX job refers to a file generated during the execution of the parent DAX job, the InPlace cleanup algorithm when applied to the parent dax job will result in the file being deleted, when the sub workflow corresponding to parent DAX job is executed. This would result in failure of sub workflow corresponding to the child DAX job, as the file deleted is required to present during it's execution.

In case there are no data dependencies across the dax jobs, then yes you can enable the InPlace algorithm for the sub dax'es . To do this you can set the property

• pegasus.file.cleanup.scope deferred

This will result in cleanup option to be picked up from the arguments for the DAX job in the top level DAX .

Metadata

Pegasus allows users to associate metadata at

• Workflow Level in the DAX

• Task level in the DAX and the Transformation Catalog

• File level in the DAX and Replica Catalog

Metadata is specified as a key value tuple, where both key and values are of type String.

All the metadata (user specified and auto-generated) gets populated into the workflow database (usually in the work-flow submit directory) by pegasus-monitord. The metadata in this database can be be queried for using the **pega-sus-metadata** command line tool, or is also shown in the Pegasus Dashboard.

Metadata in the DAX

In the DAX, metadata can be associated with the workflow, tasks, files and executables. For details on how to associate metadata in the DAX using the DAX API refer to the DAX API chapter. Below is an example DAX that illustrates metadata associations at workflow, task and file level.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated on: 2016-01-21T10:36:39-08:00 -->
<!-- generated by: vahi [ ?? ] -->
<adag xmlns="http://pegasus.isi.edu/schema/DAX" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://pegasus.isi.edu/schema/DAX http://pegasus.isi.edu/schema/
dax-3.6.xsd" version="3.6" name="diamond" index="0" count="1">

<!-- Section 1: Metadata attributes for the workflow (can be empty)  -->

   <metadata key="name">diamond</metadata>
   <metadata key="createdBy">Karan Vahi</metadata>

<!-- Section 2: Invokes - Adds notifications for a workflow (can be empty) -->

   <invoke when="start">/pegasus/libexec/notification/email -t notify@example.com</invoke>
   <invoke when="at_end">/pegasus/libexec/notification/email -t notify@example.com</invoke>

<!-- Section 3: Files - Acts as a Replica Catalog (can be empty) -->

   <file name="f.a">
      <metadata key="size">1024</metadata>
      <pfn url="file:///Volumes/Work/lfs1/work/pegasus-features/PM-902/f.a" site="local"/>
   </file>

<!-- Section 4: Executables - Acts as a Transformaton Catalog (can be empty) -->

   <executable namespace="pegasus" name="preprocess" version="4.0" installed="true" arch="x86"
 os="linux">
      <metadata key="size">2048</metadata>
      <pfn url="file:///usr/bin/keg" site="TestCluster"/>
   </executable>
   <executable namespace="pegasus" name="findrange" version="4.0" installed="true" arch="x86"
 os="linux">
      <pfn url="file:///usr/bin/keg" site="TestCluster"/>
   </executable>
   <executable namespace="pegasus" name="analyze" version="4.0" installed="true" arch="x86"
 os="linux">
      <pfn url="file:///usr/bin/keg" site="TestCluster"/>
   </executable>

<!-- Section 5: Transformations - Aggregates executables and Files (can be empty) -->


<!-- Section 6: Job's, DAX's or Dag's - Defines a JOB or DAX or DAG (Atleast 1 required) -->

   <job id="j1" namespace="pegasus" name="preprocess" version="4.0">
      <metadata key="time">60</metadata>
      <argument>-a preprocess -T 60 -i  <file name="f.a"/> -o  <file name="f.b1"/>   <file
 name="f.b2"/></argument>
      <uses name="f.a" link="input">
         <metadata key="size">1024</metadata>
      </uses>
      <uses name="f.b1" link="output" transfer="true" register="true"/>
      <uses name="f.b2" link="output" transfer="true" register="true"/>
      <invoke when="start">/pegasus/libexec/notification/email -t notify@example.com</invoke>
      <invoke when="at_end">/pegasus/libexec/notification/email -t notify@example.com</invoke>
   </job>
   <job id="j2" namespace="pegasus" name="findrange" version="4.0">
      <metadata key="time">60</metadata>
      <argument>-a findrange -T 60 -i  <file name="f.b1"/> -o  <file name="f.c1"/></argument>
      <uses name="f.b1" link="input"/>
      <uses name="f.c1" link="output" transfer="true" register="true"/>
      <invoke when="start">/pegasus/libexec/notification/email -t notify@example.com</invoke>
```

```
         <invoke when="at_end">/pegasus/libexec/notification/email -t notify@example.com</invoke>
      </job>
      <job id="j3" namespace="pegasus" name="findrange" version="4.0">
         <metadata key="time">60</metadata>
         <argument>-a findrange -T 60 -i  <file name="f.b2"/> -o  <file name="f.c2"/></argument>
         <uses name="f.b2" link="input"/>
         <uses name="f.c2" link="output" transfer="true" register="true"/>
         <invoke when="start">/pegasus/libexec/notification/email -t notify@example.com</invoke>
         <invoke when="at_end">/pegasus/libexec/notification/email -t notify@example.com</invoke>
      </job>
      <job id="j4" namespace="pegasus" name="analyze" version="4.0">
         <metadata key="time">60</metadata>
         <argument>-a analyze -T 60 -i  <file name="f.c1"/>   <file name="f.c2"/> -o  <file name="f.d"/
></argument>
         <uses name="f.c1" link="input"/>
         <uses name="f.c2" link="input"/>
         <uses name="f.d" link="output" transfer="true" register="true"/>
         <invoke when="start">/pegasus/libexec/notification/email -t notify@example.com</invoke>
         <invoke when="at_end">/pegasus/libexec/notification/email -t notify@example.com</invoke>
      </job>

<!-- Section 7: Dependencies - Parent Child relationships (can be empty) -->

      <child ref="j2">
         <parent ref="j1"/>
      </child>
      <child ref="j3">
         <parent ref="j1"/>
      </child>
      <child ref="j4">
         <parent ref="j2"/>
         <parent ref="j3"/>
      </child>
</adag>
```

Workflow Level Metadata

Workflow level metadata can be associated only in the DAX under the root element adag. Below is a snippet that illustrates this

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated on: 2016-01-21T10:36:39-08:00 -->
<!-- generated by: vahi [ ?? ] -->
<adag xmlns="http://pegasus.isi.edu/schema/DAX" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://pegasus.isi.edu/schema/DAX http://pegasus.isi.edu/schema/
dax-3.6.xsd" version="3.6" name="diamond" index="0" count="1">

<!-- Section 1: Metadata attributes for the workflow (can be empty)  -->

   <metadata key="name">diamond</metadata>
   <metadata key="createdBy">Karan Vahi</metadata>

...

</adag>
```

Task Level Metadata

Metadata for the tasks is picked up from

- metadata associated with the job element in the DAX

- metadata associated with the corresponding transformation. The transformation for a task is picked up from either a matching executable entry in the DAX (if exists) or the Transformation Catalog.

Below is a snippet that illustrates metadata for a task specified in the job element in the DAX

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated on: 2016-01-21T10:36:39-08:00 -->
<!-- generated by: vahi [ ?? ] -->
<adag xmlns="http://pegasus.isi.edu/schema/DAX" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://pegasus.isi.edu/schema/DAX http://pegasus.isi.edu/schema/
dax-3.6.xsd" version="3.6" name="diamond" index="0" count="1">
```

```
...
    <job id="j2" namespace="pegasus" name="findrange" version="4.0">
      <metadata key="time">60</metadata>
      <argument>-a findrange -T 60 -i  <file name="f.b1"/> -o  <file name="f.c1"/></argument>
      <uses name="f.b1" link="input"/>
      <uses name="f.c1" link="output" transfer="true" register="true"/>
      <invoke when="start">/pegasus/libexec/notification/email -t notify@example.com</invoke>
      <invoke when="at_end">/pegasus/libexec/notification/email -t notify@example.com</invoke>
    </job>

...

</adag>
```

Below is a snippet that illustrates metadata for a task specified in the executable element in the DAX

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated on: 2016-01-21T10:36:39-08:00 -->
<!-- generated by: vahi [ ?? ] -->
<adag xmlns="http://pegasus.isi.edu/schema/DAX" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://pegasus.isi.edu/schema/DAX http://pegasus.isi.edu/schema/
dax-3.6.xsd" version="3.6" name="diamond" index="0" count="1">

...

    <!-- Section 4: Executables - Acts as a Transformaton Catalog (can be empty) -->

  <executable namespace="pegasus" name="findrange" version="4.0" installed="true" arch="x86"
 os="linux">
      <metadata key="size">2048</metadata>
      <pfn url="file:///usr/bin/keg" site="TestCluster"/>
  </executable>

...

</adag>
```

Metadata can be associated with the transformation in the transformation catalog. The metadata specified in the transformation catalog gets automatically associated with the task level metadata for the corresponding task (that uses that executable). This resolution is similar to how profiles associated in the Transformation Catalog get associated with the tasks. Below is an example Transformation Catalog that illustrates metadata associated with the executables.

```
tr pegasus::findrange:4.0 {
    site TestCluster {
        pfn "/usr/bin/pegasus-keg"
        arch "x86_64"
        os "linux"
        type "INSTALLED"
        profile pegasus "clusters.size" "20"
        metadata "key" "value"
        metadata "appmodel" "myxform.aspen"
        metadata "version" "3.0"
    }
}
```

File Level Metadata

Metadata for the files is picked up from

- metadata associated with the file element in the DAX. File elements are optionally used to record the locations of input files for the workflow in the DAX.

- metadata associated with the files in the uses section of the job element in the DAX

- metadata associated with the file in the Replica Catalog.

Below is a snippet that illustrates metadata for a file specified in the file element in the DAX

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated on: 2016-01-21T10:36:39-08:00 -->
<!-- generated by: vahi [ ?? ] -->
<adag xmlns="http://pegasus.isi.edu/schema/DAX" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://pegasus.isi.edu/schema/DAX http://pegasus.isi.edu/schema/
dax-3.6.xsd" version="3.6" name="diamond" index="0" count="1">
```

```
...
   <!-- Section 3: Files - Acts as a Replica Catalog (can be empty) -->

  <file name="f.a">
     <metadata key="size">1024</metadata>
     <pfn url="file:///Volumes/Work/lfs1/work/pegasus-features/PM-902/f.a" site="local"/>
  </file>


...

</adag>
```

Below is a snippet that illustrates metadata for a file in the uses section of the job element

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated on: 2016-01-21T10:36:39-08:00 -->
<!-- generated by: vahi [ ?? ] -->
<adag xmlns="http://pegasus.isi.edu/schema/DAX" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://pegasus.isi.edu/schema/DAX http://pegasus.isi.edu/schema/
dax-3.6.xsd" version="3.6" name="diamond" index="0" count="1">


...

   <job id="j1" namespace="pegasus" name="preprocess" version="4.0">
      <argument>-a preprocess -T 60 -i  <file name="f.a"/> -o  <file name="f.b1"/>   <file
name="f.b2"/></argument>
      <uses name="f.a" link="input">
         <metadata key="size">1024</metadata>
         <metadata key="source">DAX</metadata>
      </uses>
      <uses name="f.b1" link="output" transfer="true" register="true"/>
      <uses name="f.b2" link="output" transfer="true" register="true"/>
   </job>

...

</adag>
```

Below is a snippet that illustrates metadata for an input file in the Replica Catalog entry for the file

```
# File Based Replica Catalog
f.a file://$PWD/production_200.conf site="local" source="replica_catalog"
```

Automatically Generated Metadata attributes

Pegasus captures certain metadata attributes as output files are generated and associates them at the file level in the database. Currently, the following attributes for the output files are automatically captured from the kickstart record and stored in the workflow database.

- pfn - the physical file location

- ctime - creation time

- size - size of the file in bytes

- user - the linux user as who the process ran that generated the output file.

Note

The automatic collection of the metadata attributes for output files is only triggered if the output file is marked to be registered in the replica catalog, and --output-site option to pegasus-plan is specified.

Tracing Metadata for an output file

The command line client pegasus-metadata allows a user to trace all the metadata associated with the file. The client will display metadata for the output file, the task that generated the file, the workflow which contains the task, and the root workflow which contains the task. Below is a sample illustration of it.

```
$ pegasus-metadata file --file-name f.d --trace /path/to/submit-dir
```

```
Workflow 493dda63-c6d0-4e62-bc36-26e5629449ad
    createdby : Test user
    name      : diamond

Task ID0000004
    size           : 2048
    time           : 60
    transformation : analyze

File f.d
    ctime        : 2016-01-20T19:02:14-08:00
    final_output : true
    size         : 582
    user         : bamboo
```

Chapter 11. Optimizing Workflows for Efficiency and Scalability

By default, Pegasus generates workflows which targets the most common usecases and execution environments. For more specialized environments or workflows, the following sections can provide hints on how to optimize your workflow to scale better, and run more efficient. Below are some common issues and solutions.

Optimizing Short Jobs / Scheduling Delays

Issue: Even though HTCondor is a high throughput system, there are overheads when scheduling short jobs. Common overheads include scheduling, data transfers, state notifications, and task book keeping. These overheads can be very noticeable for short jobs, but not noticeable at all for longer jobs as the ration between the computation and the overhead is higher.

Solution: If you have many short tasks to run, the solution to minimize the overheads is to use task clustering. This instructs Pegasus to take a set of tasks, selected horizontally, by labels, or by runtime, and create jobs containing that whole set of tasks. The result is more efficient jobs, for wich the overheads are less noticeable.

Job Clustering

A large number of workflows executed through the Pegasus Workflow Management System, are composed of several jobs that run for only a few seconds or so. The overhead of running any job on the grid is usually 60 seconds or more. Hence, it makes sense to cluster small independent jobs into a larger job. This is done while mapping an abstract workflow to an executable workflow. Site specific or transformation specific criteria are taken into consideration while clustering smaller jobs into a larger job in the executable workflow. The user is allowed to control the granularity of this clustering on a per transformation per site basis.

Overview

The abstract workflow is mapped onto the various sites by the Site Selector. This semi executable workflow is then passed to the clustering module. The clustering of the workflow can be either be

- level based horizontal clustering - where you can denote how many jobs get clustered into a single clustered job per level, or how many clustered jobs should be created per level of the workflow

- level based runtime clustering - similar to horizontal clustering , but while creating the clusters per level take into account the job runtimes.

- label based (label clustering)

The clustering module clusters the jobs into larger/clustered jobs, that can then be executed on the remote sites. The execution can either be sequential on a single node or on multiple nodes using MPI. To specify which clustering technique to use the user has to pass the **--cluster** option to **pegasus-plan** .

Generating Clustered Executable Workflow

The clustering of a workflow is activated by passing the **--cluster|-C** option to **pegasus-plan**. The clustering granularity of a particular logical transformation on a particular site is dependant upon the clustering techniques being used. The executable that is used for running the clustered job on a particular site is determined as explained in section 7.

```
#Running pegasus-plan to generate clustered workflows

$ pegasus-plan --dax example.dax --dir ./dags -p siteX --output local
               --cluster [comma separated list of clustering techniques]  -verbose
```

```
Valid clustering techniques are horizontal and label.
```

The naming convention of submit files of the clustered jobs is **merge_NAME_IDX.sub** . The NAME is derived from the logical transformation name. The IDX is an integer number between 1 and the total number of jobs in a cluster. Each of the submit files has a corresponding input file, following the naming convention **merge_NAME_IDX.in** . The input file contains the respective execution targets and the arguments for each of the jobs that make up the clustered job.

Horizontal Clustering

In case of horizontal clustering, each job in the workflow is associated with a level. The levels of the workflow are determined by doing a modified Breadth First Traversal of the workflow starting from the root nodes. The level associated with a node, is the furthest distance of it from the root node instead of it being the shortest distance as in normal BFS. For each level the jobs are grouped by the site on which they have been scheduled by the Site Selector. Only jobs of same type (txnamespace, txname, txversion) can be clustered into a larger job. To use horizontal clustering the user needs to set the **--cluster** option of **pegasus-plan to horizontal** .

Controlling Clustering Granularity

The number of jobs that have to be clustered into a single large job, is determined by the value of two parameters associated with the smaller jobs. Both these parameters are specified by the use of a PEGASUS namespace profile keys. The keys can be specified at any of the placeholders for the profiles (abstract transformation in the DAX, site in the site catalog, transformation in the transformation catalog). The normal overloading semantics apply i.e. profile in transformation catalog overrides the one in the site catalog and that in turn overrides the one in the DAX. The two parameters are described below.

- **clusters.size factor**

  The clusters.size factor denotes how many jobs need to be merged into a single clustered job. It is specified via the use of a PEGASUS namespace profile key 'clusters.size'. for e.g. if at a particular level, say 4 jobs referring to logical transformation B have been scheduled to a siteX. The clusters.size factor associated with job B for siteX is say 3. This will result in 2 clustered jobs, one composed of 3 jobs and another of 2 jobs. The clusters.size factor can be specified in the transformation catalog as follows

```
# multiple line text-based transformation catalog: 2014-09-30T16:05:01.731-07:00
tr B {
        site siteX {
                profile pegasus "clusters.size" "3"
                pfn "/shared/PEGASUS/bin/jobB"
                arch "x86"
                os "LINUX"
                type "INSTALLED"
        }
}

tr C {
        site siteX {
                profile pegasus "clusters.size" "2"
                pfn "/shared/PEGASUS/bin/jobC"
                arch "x86"
                os "LINUX"
                type "INSTALLED"
        }
}
```

Figure 11.1. Clustering by clusters.size
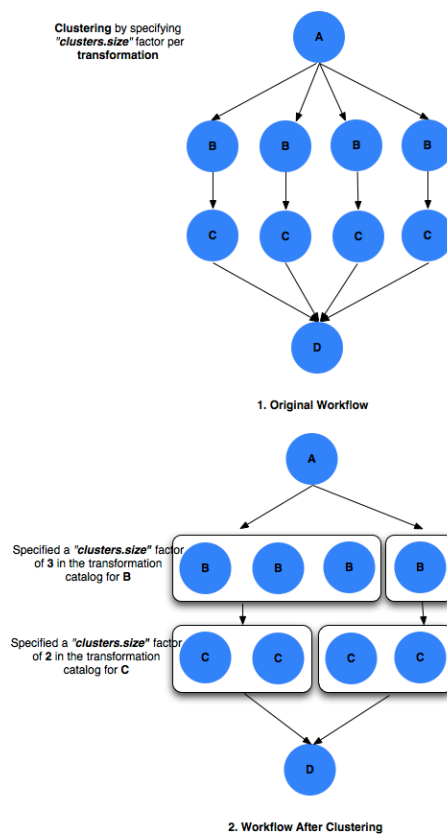
clusters.num factor

The clusters.num factor denotes how many clustered jobs does the user want to see per level per site. It is specified via the use of a PEGASUS namespace profile key 'clusters.num'. for e.g. if at a particular level, say 4 jobs referring to logical transformation B have been scheduled to a siteX. The 'clusters.num' factor associated with job B for siteX is say 3. This will result in 3 clustered jobs, one composed of 2 jobs and others of a single job each. The clusters.num factor in the transformation catalog can be specified as follows

```
# multiple line text-based transformation catalog: 2014-09-30T16:06:23.397-07:00
tr B {
        site siteX {
                profile pegasus "clusters.num" "3"
                pfn "/shared/PEGASUS/bin/jobB"
                arch "x86"
                os "LINUX"
                type "INSTALLED"
        }
}

tr C {
        site siteX {
                profile pegasus "clusters.num" "2"
                pfn "/shared/PEGASUS/bin/jobC"
                arch "x86"
                os "LINUX"
                type "INSTALLED"
        }
```

```
}
```

In the case, where both the factors are associated with the job, the clusters.num value supersedes the clusters.size value.

```
# multiple line text-based transformation catalog: 2014-09-30T16:08:01.537-07:00
tr B {
        site siteX {
                profile pegasus "clusters.num" "3"
                profile pegasus "clusters.size" "3"
                pfn "/shared/PEGASUS/bin/jobB"
                arch "x86"
                os "LINUX"
                type "INSTALLED"
        }
}
```

In the above case the jobs referring to logical transformation B scheduled on siteX will be clustered on the basis of 'clusters.num' value. Hence, if there are 4 jobs referring to logical transformation B scheduled to siteX, then 3 clustered jobs will be created.

Figure 11.2. Clustering by clusters.num
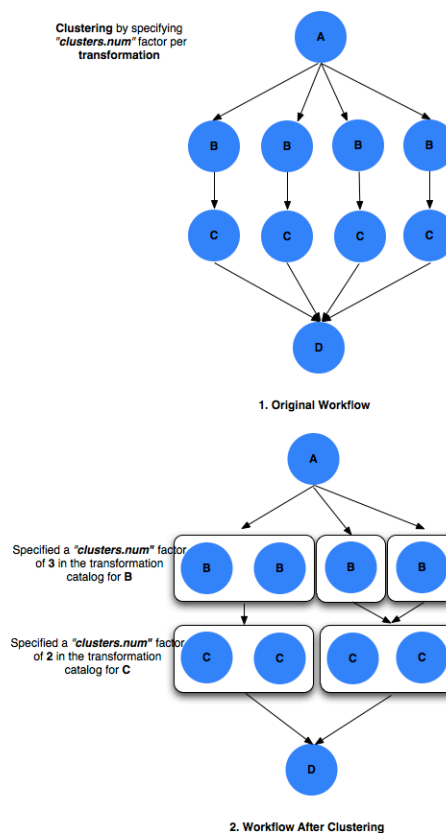
Runtime Clustering

Workflows often consist of jobs of same type, but have varying run times. Two or more instances of the same job, with varying inputs can differ significantly in their runtimes. A simple way to think about this is running the same program on two distinct input sets, where one input is smaller (1 MB) as compared to the other which is 10 GB in

size. In such a case the two jobs will having significantly differing run times. When such jobs are clustered using horizontal clustering, the benefits of job clustering may be lost if all smaller jobs get clustered together, while the larger jobs are clustered together. In such scenarios it would be beneficial to be able to cluster jobs together such that all clustered jobs have similar runtimes.

In case of runtime clustering, jobs in the workflow are associated with a level. The levels of the workflow are determined in the same manner as in horizontal clustering. For each level the jobs are grouped by the site on which they have been scheduled by the Site Selector. Only jobs of same type (txnamespace, txname, txversion) can be clustered into a larger job. To use runtime clustering the user needs to set the **--cluster** option of **pegasus-plan to horizontal**, and set the Pegasus property **pegasus.clusterer.preference** to **Runtime**.

Runtime clustering supports two modes of operation.

1. Clusters jobs together such that the clustered job's runtime does not exceed a user specified maxruntime.

   Basic Algorithm of grouping jobs into clusters is as follows

   ```
   // cluster.maxruntime - Is the maximum runtime for which the clustered job should run.
   // j.runtime - Is the runtime of the job j.
   1. Create a set of jobs of the same type (txnamespace, txname, txversion), and that run on the
    same site.
   2. Sort the jobs in decreasing order of their runtime.
   3. For each job j, repeat
     a. If j.runtime > cluster.maxruntime then
          ignore j.
     // Sum of runtime of jobs already in the bin + j.runtime <= cluster.maxruntime
     b. If j can be added to any existing bin (clustered job) then
          Add j to bin
       Else
          Add a new bin
          Add job j to newly added bin
   ```

   The runtime of a job, and the maximum runtime for which a clustered jobs should run is determined by the value of two parameters associated with the jobs.

   - **runtime**

     expected runtime for a job

   - **clusters.maxruntime**

     maxruntime for the clustered job i.e. Group as many jobs as possible into a cluster, as long as the clustered jobs' runtime does not exceed clusters.maxruntime.

2. Clusters all the into a fixed number of clusters (clusters.num), such that the runtimes of the clustered jobs are similar.

   Basic Algorithm of grouping jobs into clusters is as follows

   ```
   // cluster.num - Is the number of clustered jobs to create.
   // j.runtime - Is the runtime of the job j.
   1. Create a set of jobs of the same type (txnamespace, txname, txversion), and that run on the
    same site.
   2. Sort the jobs in decreasing order of their runtime.
   3. Create a heap containing clusters.num number of clustered jobs.
   4. For each job j, repeat
     a. Get cluster job cj, having the shortest runtime
     b. Add job j to clustered job cj
   ```

   The runtime of a job, and the number of clustered jobs to create is determined by the value of two parameters associated with the jobs.

   - **runtime**

     expected runtime for a job

   - **clusters.num**

     clusters.num factor denotes how many clustered jobs does the user want to see per level per site

Note

Users should either specify clusters.maxruntime or clusters.num. If both of them are specified, then cluster-s.num profile will be ignored by the clustering engine.

All of these parameters are specified by the use of a PEGASUS namespace profile keys. The keys can be specified at any of the placeholders for the profiles (abstract transformation in the DAX, site in the site catalog, transformation in the transformation catalog). The normal overloading semantics apply i.e. profile in transformation catalog overrides the one in the site catalog and that in turn overrides the one in the DAX. The two parameters are described below.

```
# multiple line text-based transformation catalog: 2014-09-30T16:09:40.610-07:00
#Cluster all jobs of type B at siteX, into 2 clusters such that the 2 clusters have similar runtimes
tr B {
        site siteX {
                profile pegasus "clusters.num" "2"
                profile pegasus "runtime" "100"
                pfn "/shared/PEGASUS/bin/jobB"
                arch "x86"
                os "LINUX"
                type "INSTALLED"
        }
}

#Cluster all jobs of type C at siteX, such that the duration of the clustered job does not exceed
 300.
tr C {
        site siteX {
                profile pegasus "clusters.maxruntime" "300"
                profile pegasus "runtime" "100"
                pfn "/shared/PEGASUS/bin/jobC"
                arch "x86"
                os "LINUX"
                type "INSTALLED"
        }
}
```

Figure 11.3. Clustering by runtime
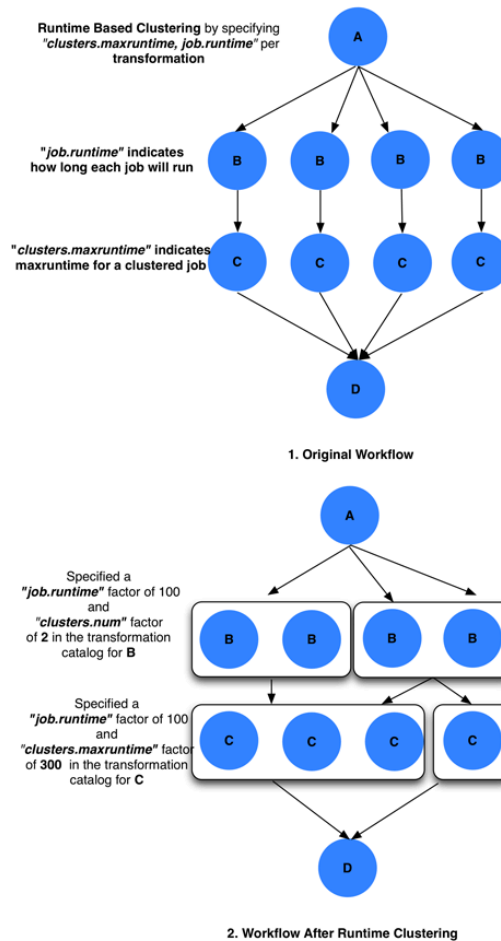
1. Original Workflow

2. Workflow After Runtime Clustering

In the above case the jobs referring to logical transformation B scheduled on siteX will be clustered such that all clustered jobs will run approximately for the same duration specified by the clusters.maxruntime property. In the above case we assume all jobs referring to transformation B run for 100 seconds. For jobs with significantly differing runtime, the runtime property will be associated with the jobs in the DAX.

In addition to the above two profiles, we need to inform pegasus-plan to use runtime clustering. This is done by setting the following property .

```
pegasus.clusterer.preference          Runtime
```

Label Clustering

In label based clustering, the user labels the workflow. All jobs having the same label value are clustered into a single clustered job. This allows the user to create clusters or use a clustering technique that is specific to his workflows. If there is no label associated with the job, the job is not clustered and is executed as is

Figure 11.4. Label-based clustering
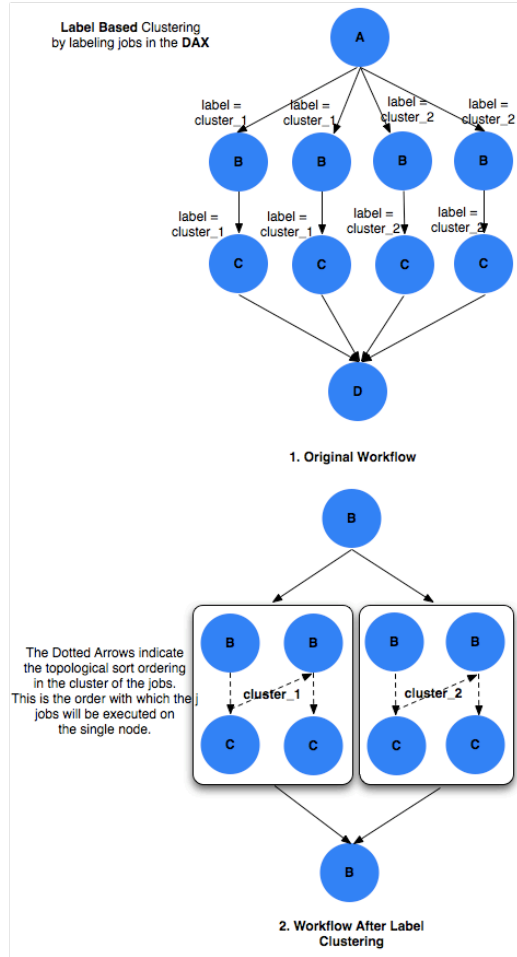
Since, the jobs in a cluster in this case are not independent, it is important the jobs are executed in the correct order. This is done by doing a topological sort on the jobs in each cluster. To use label based clustering the user needs to set the **--cluster** option of **pegasus-plan** to label.

Labelling the Workflow

The labels for the jobs in the workflow are specified by associated **pegasus** profile keys with the jobs during the DAX generation process. The user can choose which profile key to use for labeling the workflow. By default, it is assumed that the user is using the PEGASUS profile key label to associate the labels. To use another key, in the **pegasus** namespace the user needs to set the following property

- pegasus.clusterer.label.key

For example if the user sets **pegasus.clusterer.label.key** to **user_label** then the job description in the DAX looks as follows

```
<adag >
...
  <job id="ID000004" namespace="app" name="analyze" version="1.0" level="1" >
    <argument>-a bottom -T60  -i <filename file="user.f.c1"/>  -o <filename file="user.f.d"/></
argument>
    <profile namespace="pegasus" key="user_label">p1</profile>
    <uses file="user.f.c1" link="input" register="true" transfer="true"/>
    <uses file="user.f.c2" link="input" register="true" transfer="true"/>
    <uses file="user.f.d" link="output" register="true" transfer="true"/>
  </job>
```

```
...
</adag>
```

- The above states that the **pegasus** profiles with key as **user_label** are to be used for designating clusters.

- Each job with the same value for **pegasus** profile key **user_label** appears in the same cluster.

Recursive Clustering

In some cases, a user may want to use a combination of clustering techniques. For e.g. a user may want some jobs in the workflow to be horizontally clustered and some to be label clustered. This can be achieved by specifying a comma separated list of clustering techniques to the **--cluster** option of **pegasus-plan**. In this case the clustering techniques are applied one after the other on the workflow in the order specified on the command line.

For example

```
$ pegasus-plan --dax example.dax --dir ./dags --cluster label,horizontal -s siteX --output local --
verbose
```

Figure 11.5. Recursive clustering

1. Original Workflow

2. Workflow After Label Clustering

3. Workflow After Horizontal Clustering applied to Label based Clustered Workflow
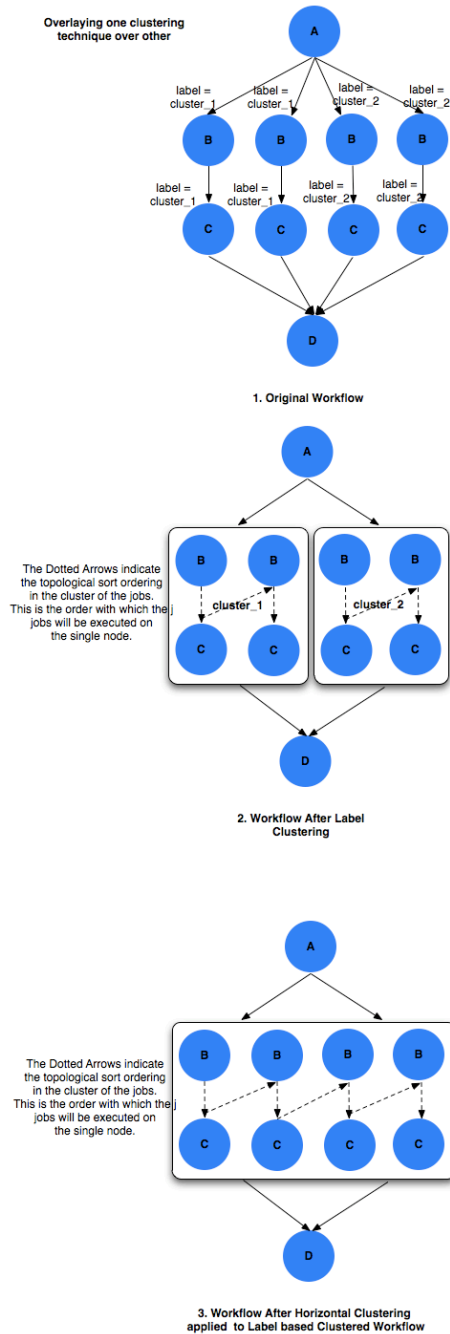
Execution of the Clustered Job

The execution of the clustered job on the remote site, involves the execution of the smaller constituent jobs either

- **sequentially on a single node of the remote site**

  The clustered job is executed using **pegasus-cluster**, a wrapper tool written in C that is distributed as part of the PEGASUS. It takes in the jobs passed to it, and ends up executing them sequentially on a single node. To use pegasus-cluster for executing any clustered job on a siteX, there needs to be an entry in the transformation catalog for an executable with the logical name seqexec and namespace as pegasus.

```
#site    transformation    pfn           type            architecture    profiles

siteX    pegasus::seqexec  /usr/pegasus/bin/pegasus-cluster INSTALLED      INTEL32::LINUX
 NULL
```

If the entry is not specified, Pegasus will attempt create a default path on the basis of the environment profile PEGASUS_HOME specified in the site catalog for the remote site.

- **On multiple nodes of the remote site using MPI based task management tool called Pegasus MPI Cluster (PMC)**

The clustered job is executed using **pegasus-mpi-cluster**, a wrapper MPI program written in C that is distributed as part of the PEGASUS. A PMC job consists of a single master process (this process is rank 0 in MPI parlance) and several worker processes. These processes follow the standard master-worker architecture. The master process manages the workflow and assigns workflow tasks to workers for execution. The workers execute the tasks and return the results to the master. Communication between the master and the workers is accomplished using a simple text-based protocol implemented using MPI_Send and MPI_Recv. PMC relies on a shared filesystem on the remote site to manage the individual tasks stdout and stderr and stage it back to the submit host as part of it's own stdout/stderr.

The input format for PMC is a DAG based format similar to Condor DAGMan's. PMC follows the dependencies specified in the DAG to release the jobs in the right order and executes parallel jobs via the workers when possible. The input file for PMC is automatically generated by the Pegasus Planner when generating the executable workflow. PMC allows for a finer grained control on how each task is executed. This can be enabled by associating the following pegasus profiles with the jobs in the DAX

Table 11.1. Pegasus Profiles that can be associated with jobs in the DAX for PMC

Key	Description
pmc_request_memory	This key is used to set the -m option for pegasus-mpi-cluster. It specifies the amount of memory in MB that a job requires. This profile is usually set in the DAX for each job.
pmc_request_cpus	This key is used to set the -c option for pegasus-mpi-cluster. It specifies the number of cpu's that a job requires. This profile is usually set in the DAX for each job.
pmc_priority	This key is used to set the -p option for pegasus-mpi-cluster. It specifies the priority for a job . This profile is usually set in the DAX for each job. Negative values are allowed for priorities.
pmc_task_arguments	The key is used to pass any extra arguments to the PMC task during the planning time. They are added to the very end of the argument string constructed for the task in the PMC file. Hence, allows for overriding of any argument constructed by the planner for any particular task in the PMC job.

Refer to the pegasus-mpi-cluster man page in the command line tools chapter to know more about PMC and how it schedules individual tasks.

It is recommended to have a pegasus::mpiexec entry in the transformation catalog to specify the path to PMC on the remote and specify the relevant globus profiles such as xcount, host_xcount and maxwalltime to control size of the MPI job.

```
# multiple line text-based transformation catalog: 2014-09-30T16:11:11.947-07:00
tr pegasus::mpiexec {
        site siteX {
                profile globus "host_xcount" "1"
                profile globus "xcount" "32"
                pfn "/usr/pegasus/bin/pegasus-mpi-cluster"
                arch "x86"
```

```
                    os "LINUX"
                    type "INSTALLED"
            }
    }
```

the entry is not specified, Pegasus will attempt create a default path on the basis of the environment profile PE-GASUS_HOME specified in the site catalog for the remote site.

Tip

Users are encouraged to use label based clustering in conjunction with PMC

Specification of Method of Execution for Clustered Jobs

The method execution of the clustered job(whether to launch via mpiexec or seqexec) can be specified

1. **globally in the properties file**

   The user can set a property in the properties file that results in all the clustered jobs of the workflow being executed by the same type of executable.

   ```
   #PEGASUS PROPERTIES FILE
   pegasus.clusterer.job.aggregator seqexec|mpiexec
   ```

   In the above example, all the clustered jobs on the remote sites are going to be launched via the property value, as long as the property value is not overridden in the site catalog.

2. **associating profile key job.aggregator with the site in the site catalog**

   ```
   <site handle="siteX" gridlaunch = "/shared/PEGASUS/bin/kickstart">
       <profile namespace="env" key="GLOBUS_LOCATION" >/home/shared/globus</profile>
       <profile namespace="env" key="LD_LIBRARY_PATH">/home/shared/globus/lib</profile>
       <profile namespace="pegasus" key="job.aggregator" >seqexec</profile>
       <lrc url="rls://siteX.edu" />
       <gridftp  url="gsiftp://siteX.edu/" storage="/home/shared/work" major="2" minor="4"
    patch="0" />
       <jobmanager universe="transfer" url="siteX.edu/jobmanager-fork" major="2" minor="4"
    patch="0" />
       <jobmanager universe="vanilla" url="siteX.edu/jobmanager-condor" major="2" minor="4"
    patch="0" />
       <workdirectory >/home/shared/storage</workdirectory>
    </site>
   ```

   In the above example, all the clustered jobs on a siteX are going to be executed via seqexec, as long as the value is not overridden in the transformation catalog.

3. **associating profile key job.aggregator with the transformation that is being clustered, in the transformation catalog**

   ```
   # multiple line text-based transformation catalog: 2014-09-30T16:11:52.230-07:00
   tr B {
           site siteX {
                   profile pegasus "clusters.size" "3"
                   profile pegasus "job.aggregator" "mpiexec"
                   pfn "/shared/PEGASUS/bin/jobB"
                   arch "x86"
                   os "LINUX"
                   type "INSTALLED"
           }
   }
   ```

   In the above example, all the clustered jobs that consist of transformation B on siteX will be executed via mpiexec.

   ## Note

   The clustering of jobs on a site only happens only if

   - there exists an entry in the transformation catalog for the clustering executable that has been determined by the above 3 rules

---

- the number of jobs being clustered on the site are more than 1

Outstanding Issues

1. **Label Clustering**

   More rigorous checks are required to ensure that the labeling scheme applied by the user is valid.

How to Scale Large Workflows

Issue: When planning and running large workflows, there are some scalability issues to be aware of. During the planning stage, Pegasus traverses the graphs multiple times, and some of the graph transforms can be slow depending on if the graph is large in the number of tasks, the number of files, or the number of dependencies. Once planned, large workflows can also see scalability limits when interacting with the operating system. A common problem is the number of files in a single directory, such as thousands or millons input or output files.

Solution: The most common solution to these problems is to use hierarchical workflows, which works really well if your workflow can be logically partitioned into smaller workflows. A hierarchical workflow still runs like a single workflow, with the difference being that some jobs in the workflow are actually sub-workflows.

For workflows with a large number of files, you can control the number of files in a single directory by reorganizing the files into a deep directory structure.

Hierarchical Workflows

Introduction

The Abstract Workflow in addition to containing compute jobs, can also contain jobs that refer to other workflows. This is useful for running large workflows or ensembles of workflows.

Users can embed two types of workflow jobs in the DAX

1. daxjob - refers to a sub workflow represented as a DAX. During the planning of a workflow, the DAX jobs are mapped to condor dagman jobs that have pegasus plan invocation on the dax (referred to in the DAX job) as the prescript.

   Figure 11.6. Planning of a DAX Job
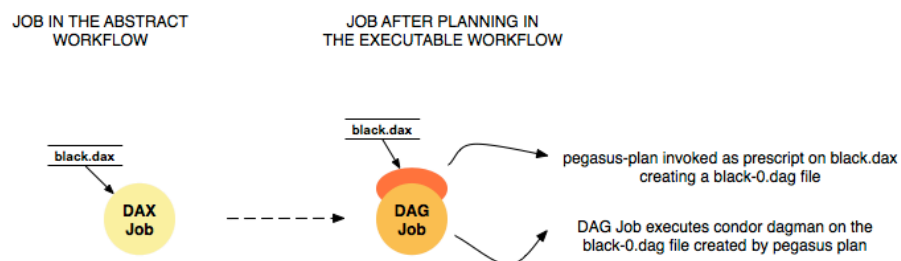
2. dagjob - refers to a sub workflow represented as a DAG. During the planning of a workflow, the DAG jobs are mapped to condor dagman and refer to the DAG file mentioned in the DAG job.

Figure 11.7. Planning of a DAG Job
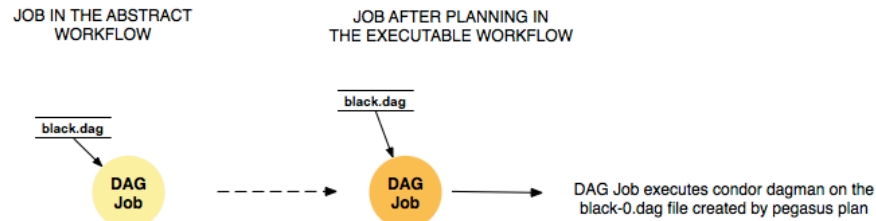
Specifying a DAX Job in the DAX

Specifying a DAXJob in a DAX is pretty similar to how normal compute jobs are specified. There are minor differences in terms of the xml element name (dax vs job) and the attributes specified. DAXJob XML specification is described in detail in the chapter on DAX API . An example DAX Job in a DAX is shown below

```
<dax id="ID000002" name="black.dax" node-label="bar" >
  <profile namespace="dagman" key="maxjobs">10</profile>
  <argument>-Xmx1024 -Xms512 -Dpegasus.dir.storage=storagedir  -Dpegasus.dir.exec=execdir -o local
-vvvvv --force -s dax_site </argument>
</dax>
```

DAX File Locations

The name attribute in the dax element refers to the LFN (Logical File Name) of the dax file. The location of the DAX file can be catalogued either in the

1. Replica Catalog

2. Replica Catalog Section in the DAX .

> ## Note
>
> Currently, only file url's on the local site (submit host) can be specified as DAX file locations.

Arguments for a DAX Job

Users can specify specific arguments to the DAX Jobs. The arguments specified for the DAX Jobs are passed to the pegasus-plan invocation in the prescript for the corresponding condor dagman job in the executable workflow.

The following options for pegasus-plan are inherited from the pegasus-plan invocation of the parent workflow. If an option is specified in the arguments section for the DAX Job then that overrides what is inherited.

Table 11.2. Options inherited from parent workflow

Option Name	Description
--sites	list of execution sites.

It is highly recommended that users **don't specify** directory related options in the arguments section for the DAX Jobs. Pegasus assigns values to these options for the sub workflows automatically.

1. --relative-dir

2. --dir

3. --relative-submit-dir

Profiles for DAX Job

Users can choose to specify dagman profiles with the DAX Job to control the behavior of the corresponding condor dagman instance in the executable workflow. In the example above maxjobs is set to 10 for the sub workflow.

Execution of the PRE script and Condor DAGMan instance

The pegasus plan that is invoked as part of the prescript to the condor dagman job is executed on the submit host. The log from the output of pegasus plan is redirected to a file (ending with suffix pre.log) in the submit directory of the workflow that contains the DAX Job. The path to pegasus-plan is automatically determined.

The DAX Job maps to a Condor DAGMan job. The path to condor dagman binary is determined according to the following rules -

1. entry in the transformation catalog for condor::dagman for site local, else

2. pick up the value of CONDOR_HOME from the environment if specified and set path to condor dagman as $CONDOR_HOME/bin/condor_dagman , else

3. pick up the value of CONDOR_LOCATION from the environment if specified and set path to condor dagman as $CONDOR_LOCATION/bin/condor_dagman , else

4. pick up the path to condor dagman from what is defined in the user's PATH

Tip

It is recommended that users specify dagman.maxpre in their properties file to control the maximum number of pegasus plan instances launched by each running dagman instance.

Specifying a DAG Job in the DAX

Specifying a DAGJob in a DAX is pretty similar to how normal compute jobs are specified. There are minor differences in terms of the xml element name (dag vs job) and the attributes specified. For DAGJob XML details,see the API Reference chapter . An example DAG Job in a DAX is shown below

```
<dag id="ID000003" name="black.dag" node-label="foo" >
  <profile namespace="dagman" key="maxjobs">10</profile>
  <profile namespace="dagman" key="DIR">/dag-dir/test</profile>
</dag>
```

DAG File Locations

The name attribute in the dag element refers to the LFN (Logical File Name) of the dax file. The location of the DAX file can be catalogued either in the

1. Replica Catalog

2. Replica Catalog Section in the DAX.

Note

Currently, only file url's on the local site (submit host) can be specified as DAG file locations.

Profiles for DAG Job

Users can choose to specify dagman profiles with the DAX Job to control the behavior of the corresponding condor dagman instance in the executable workflow. In the example above, maxjobs is set to 10 for the sub workflow.

The dagman profile DIR allows users to specify the directory in which they want the condor dagman instance to execute. In the example above black.dag is set to be executed in directory /dag-dir/test . The /dag-dir/test should be created beforehand.

File Dependencies Across DAX Jobs

In hierarchal workflows , if a sub workflow generates some output files required by another sub workflow then there should be an edge connecting the two dax jobs. Pegasus will ensure that the prescript for the child sub-workflow, has the path to the cache file generated during the planning of the parent sub workflow. The cache file in the submit directory for a workflow is a textual replica catalog that lists the locations of all the output files created in the remote workflow execution directory when the workflow executes.

This automatic passing of the cache file to a child sub-workflow ensures that the datasets from the same workflow run are used. However, the passing the locations in a cache file also ensures that Pegasus will prefer them over all other locations in the Replica Catalog. If you need the Replica Selection to consider locations in the Replica Catalog also, then set the following property.

```
pegasus.catalog.replica.cache.asrc   true
```

The above is useful in the case, where you are staging out the output files to a storage site, and you want the child sub workflow to stage these files from the storage output site instead of the workflow execution directory where the files were originally created.

Recursion in Hierarchal Workflows

It is possible for a user to add a dax jobs to a dax that already contain dax jobs in them. Pegasus does not place a limit on how many levels of recursion a user can have in their workflows. From Pegasus perspective recursion in hierarchal workflows ends when a DAX with only compute jobs is encountered . However, the levels of recursion are limited by the system resources consumed by the DAGMan processes that are running (each level of nesting produces another DAGMan process) .

The figure below illustrates an example with recursion 2 levels deep.

Figure 11.8. Recursion in Hierarchal Workflows
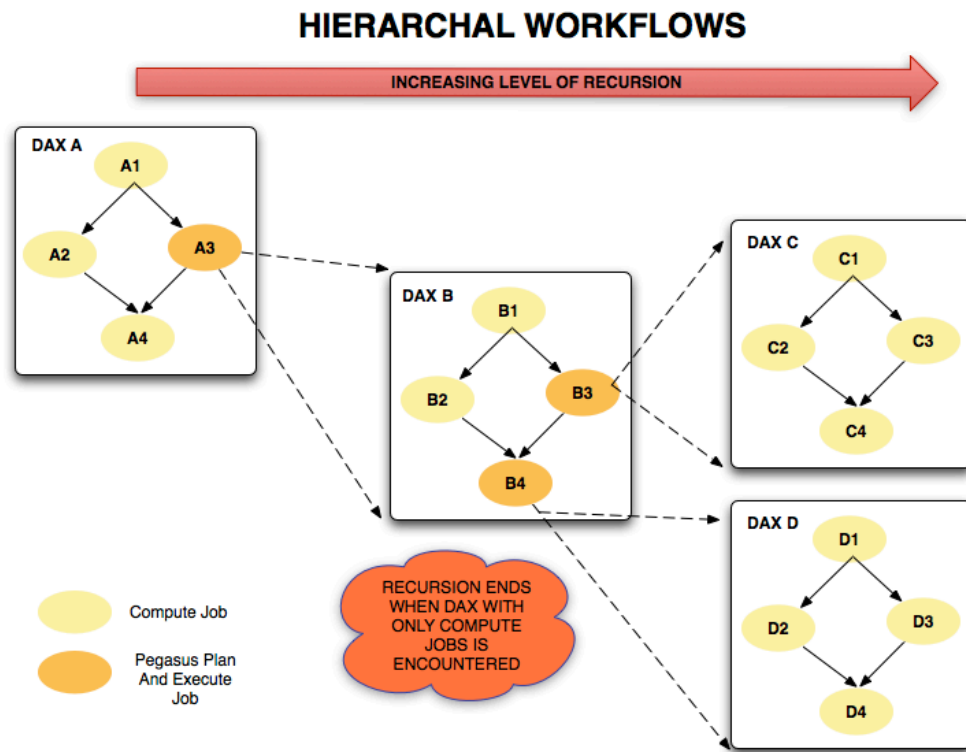
The execution time-line of the various jobs in the above figure is illustrated below.

Figure 11.9. Execution Time-line for Hierarchal Workflows
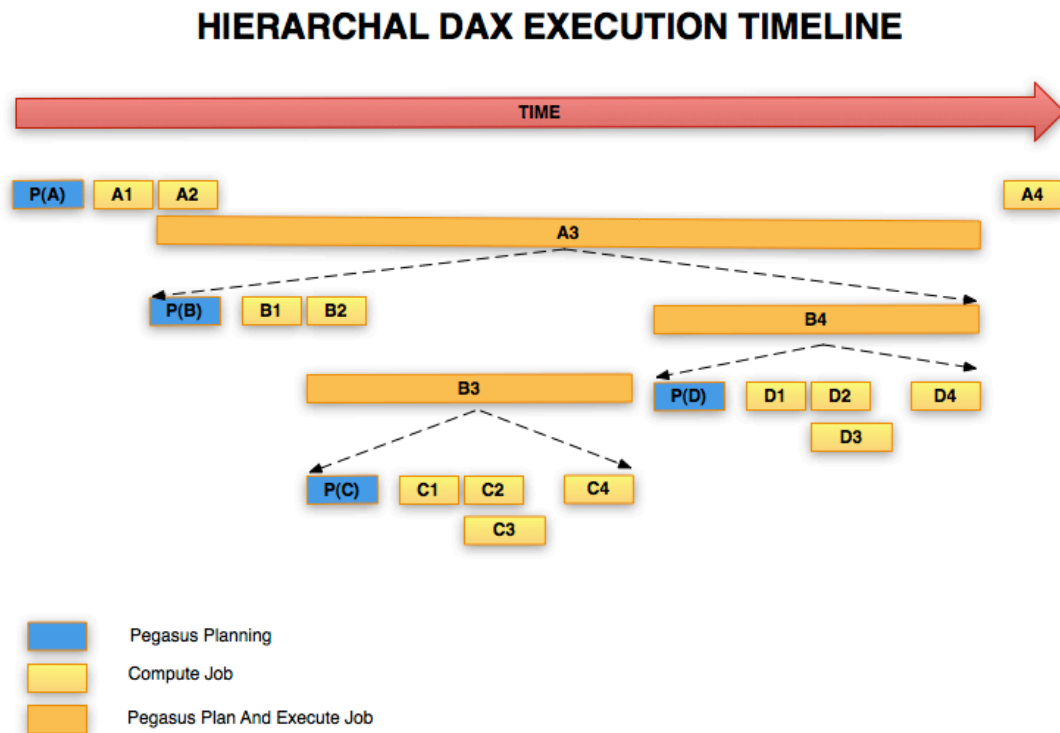
Example

The Galactic Plane workflow is a Hierarchical workflow of many Montage workflows. For details, see Workflow of Workflows.

Optimizing Data Transfers

Issue: When it comes to data transfers, Pegasus ships with a default configuration which is trying to strike a balance between performance and aggressiveness. We obviously want data transfers to be as quick as possibly, but we also do not want our transfers to overwhelm data services and systems.

Solution: Starting 4.8.0 release, the default configuration of Pegasus now adds transfer jobs and cleanup jobs based on the number of jobs at a particular level of the workflow. For example, for every 10 compute jobs on a level of a workflow, one data transfer job(stage-in and stage-out) is created. The default configuration also sets how many threads such a pegasus-transfer job can spawn. Cleanup jobs are similarly constructed with an internal ratio of 5.

Additionally, Pegasus makes use of DAGMan categories and associates the following default values with the transfer and cleanup jobs.

Table 11.3. Default Category names associated by Pegasus

DAGMan Category Name	Auxillary Job applied to.	Default Value Assigned in generated DAG file

stage-in	data stage-in jobs	10
stage-out	data stage-out jobs	10
stage-inter	inter site data transfer jobs	-
cleanup	data cleanup jobs	4
registration	registration jobs	1 (for file based RC)

Information on how to control manully the maxinum number of stagein and stageout jobs can be found in the Data Movement Nodes section.

How to control the number of threads pegasus-transfer can use depends on if you want to control standard transfer jobs, or PegasusLite. For the former, see the pegasus.transfer.threads property, and for the latter the pegasus.transfer.lite.threads property.

Job Throttling

Issue: For large workflows you may want to control the number of jobs released by DAGMan in local condor queue, or number of remote jobs submitted.

Solution: HTCondor DAGMan has knobs that can be tuned at a per workflow level to control it's behavior. These knobs control how it interacts with the local HTCondor Schedd to which it submits jobs that are ready to run in a particular DAG. These knobs are exposed as DAGMan profiles (maxidle, maxjobs, maxpre and maxpost) that you can set in your properties files.

Table 11.4. Useful dagman Commands that can be specified in the properties file.

Property Key	Description
Property Key: dagman.maxpre **Profile Key:** MAXPRE **Scope** : Properties **Since** : 2.0 **Type** : String	sets the maximum number of PRE scripts within the DAG that may be running at one time
Property Key: dagman.maxpost **Profile Key:** MAXPOST **Scope** : Properties **Since** : 2.0 **Type** : String	sets the maximum number of POST scripts within the DAG that may be running at one time
Property Key: dagman.maxjobs **Profile Key:** MAXJOBS **Scope** : Properties **Since** : 2.0 **Type** : String	sets the maximum number of jobs within the DAG that will be submitted to Condor at one time.
Property Key: dagman.maxidle **Profile Key:** MAXIDLE **Scope** : Properties **Since** : 2.0 **Type** : String	Sets the maximum number of idle jobs allowed before HTCondor DAGMan stops submitting more jobs. Once idle jobs start to run, HTCondor DAGMan will resume submitting jobs. If the option is omitted, the number of idle jobs is unlimited.
Property Key: dagman.[CATEGORY-NAME].maxjobs **Profile Key:** [CATEGORY-NAME].MAXJOBS **Scope** : Properties **Since** : 2.0 **Type** : String	is the value of maxjobs for a particular category. Users can associate different categories to the jobs at a per job basis. However, the value of a dagman knob for a category can only be specified at a per workflow basis in the properties.
Property Key: dagman.post.scope **Profile Key:** POST.SCOPE **Scope** : Properties	scope for the postscripts.

Since : 2.0 **Type** : String	1. If set to **all** , means each job in the workflow will have a postscript associated with it. 2. If set to **none** , means no job has postscript associated with it. None mode should be used if you are running vanilla / standard/ local universe jobs, as in those cases Condor traps the remote exitcode correctly. None scope is not recommended for grid universe jobs. 3. If set to **essential**, means only essential jobs have post scripts associated with them. At present the only non essential job is the replica registration job.

Within a single workflow, you can also control the number of jobs submitted per type (or category) of jobs. To associate categories, you needs to associate dagman profile key named category with the jobs and specify the property dagman.[CATEGORY-NAME].* in the properties file. More information about HTCondor DAGMan categories can be found in the HTCondor Documentation [http://research.cs.wisc.edu/htcondor/manual/v8.3.5/2_10DAGMan_Applications.html#SECTION003108400000000000000].

By default, pegasus associates default category names to following types of auxillary jobs

Table 11.5. Default Category names associated by Pegasus

DAGMan Category Name	Auxillary Job applied to.	Default Value Assigned in generated DAG file
stage-in	data stage-in jobs	10
stage-out	data stage-out jobs	10
stage-inter	inter site data transfer jobs	-
cleanup	data cleanup jobs	4
registration	registration jobs	1 (for file based RC)

Below is a sample properties file that illustrates how categories can be specified in the properties file

```
# pegasus properties file snippet illustrating
# how to specify dagman categories for different types of jobs

dagman.stage-in.maxjobs 4
dagman.stage-out.maxjobs 1
dagman.cleanup.maxjobs 2
```

HTCondor also exposes useful configuration parameters that can be specified in it's configuration file (condor_config_val -conf will list the condor configuration files), to control job submission across workflows. Some of the useful parameters that you may want to tune are

Table 11.6. Useful HTCondor Job Throttling Configuration Parameters

HTCondor Configuration Parameter	Description
Parameter Name: START_LOCAL_UNIVERSE **Sample Value :** TotalLocalJobsRunning < 20	Most of the pegauss added auxillary jobs (createdir, cleanup, registration and data cleanup) run in the local universe on the submit host. If you have a lot of workflows running, HTCondor may try to start too many local universe jobs, that may bring down your submit host. This global parameter is used to configure condor to not launch too many local universe jobs.
Parameter Name: GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE **Sample Value :** Integer	For grid jobs, limits the number of globus-job-manager processes that the condor_gridmanager lets run at a time on the remote head node. Allowing too many globus-job-managers to run causes severe load on

	the head note, possibly making it non-functional. Usually the default value in htcondor (as of version 8.3.5) is 10. This parameter is useful when you are doing remote job submissions using HTCondor-G.
Parameter Name: GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE **Sample Value :** Integer	An integer value the number of jobs that a condor_gridmanager daemon will submit to a resource. A comma-separated list of pairs that follows this integer limit will specify limits for specific remote resources. Each pair is a host name and the job limit for that host. Consider the example ``` GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE = 200, foo.edu, 50, bar.com, 100. ``` In this example, all resources have a job limit of 200, except foo.edu, which has a limit of 50, and bar.com, which has a limit of 100. Limits specific to grid types can be set by appending the name of the grid type to the configuration variable name, as the example GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE_CREAM = 300 In this example, the job limit for all CREAM resources is 300. Defaults to 1000 (as of version 8.3.5). This parameter is useful when you are doing remote job submissions using HTCondor-G.

Job Throttling Across Workflows

Issue: DAGMan throttling knobs are per workflow, and don't work across workflows. Is there any way to control different types of jobs run at a time across workflows?

Solution: While not possible in all cases, it is possible to throttle different types of jobs across workflows if you configure the jobs to run in vanilla universe by leverage HTCondor concurrency limits [http://research.cs.wisc.edu/htcondor/manual/v8.2/3_12Setting_Up.html#SECTION004121500000000000000]. Most of the Pegasus generated jobs (data transfer jobs and auxillary jobs such as create dir, cleanup and registration) execute in local universe where concurrency limits don't work. To use this you need to do the following

1. Get the local universe jobs to run locally in vanilla universe. You can do this by associating condor profiles universe and requirements in the site catalog for local site or individually in the transformation catalog for each pegasus executable. Here is an example local site catalog entry.

```
<site handle="local" arch="x86_64" os="LINUX">
    <directory type="shared-scratch" path="/shared-scratch/local">
      <file-server operation="all" url="file:///shared-scratch/local"/>
    </directory>
    <directory type="local-storage" path="/storage/local">
      <file-server operation="all" url="file:///storage/local"/>
    </directory>

    <!-- keys to make jobs scheduled to local site run on local site in vanilla universe -->
    <profile namespace="condor" key="universe">vanilla</profile>
    <profile namespace="condor" key="requirements">(Machine=="submit.example.com")</profile>
  </site>
```

Replace the Machine value in requirements with the hostname of your submit host.

2. Copy condor_config.pegasus file from share/pegasus/htcondor directory to your condor config.d directory.

Starting Pegasus 4.5.1 release, the following values for concurrency limits can be associated with different types of jobs Pegasus creates. To enable the generation of concurrency limits with the jobs set the following property in your properties file.

```
pegasus.condor.concurrency.limits   true
```

Table 11.7. Pegasus Job Types To Condor Concurrency Limits

Pegasus Job Type	HTCondor Concurrency Limit Compatible with distributed condor_config.pegasus
Data Stagein Job	pegasus_transfer.stagein
Data Stageout Job	pegasus_transfer.stageout
Inter Site Data Transfer Job	pegasus_transfer.inter
Worker Pacakge Staging Job	pegasus_transfer.worker
Create Directory Job	pegasus_auxillary.createdir
Data Cleanup Job	pegasus_auxillary.cleanup
Replica Registration Job	pegasus_auxillary.registration
Set XBit Job	pegasus_auxillary.chmod
User Compute Job	pegasus_compute

Note

It is not recommended to set limit for compute jobs unless you know what you are doing.

Increase Memory Requirements for Retries

Issue: Setting memory limits for codes with varying amounts of memory requirments can be challenging. Some codes do not use much RAM most of the time, but once in a while require more RAM due to for example initial condition and hitting a particular spot in the algorithm.

Solution: A common approach is to provide a smaller limit for the first try of a job, and if the job fails, increase the limit for subsequent tries. This can be accomplished with an expression for the **request_memory** attribute. For example, setting the attribute in the site catalog, setting the limit to 1 GB for the first try, and then 4 GB for remaining tries:

```
        <profile namespace="condor" key="request_memory"> ifthenelse(isundefined(DAGNodeRetry) ||
DAGNodeRetry == 0, 1024, 4096) </profile>
```

Chapter 12. Pegasus Service

Service Administration

Service Configuration

Create a file called service.py in $HOME/.pegasus/ OR modify the lib/pegasus/python/Pegasus/service/defaults.py file. The servuce can be configured using the properties described below.

Table 12.1. Pegasus Service Configuration Options

Property	Default Value	Description
SERVER_HOST	127.0.0.1	SERVER_HOST specifies the host-name/network interface on which the service listens for requests.
SERVER_PORT	5000	SERVER_PORT specifies the port number on which the service listens for requests.
CERTIFICATE	None	SSL certificate file used to encrypt sessions. If no certificate, key files are provided the service will generate and use self-signed certificates.
PRIVATE_KEY	None	SSL key file used to encrypt connections. If no certificate, key files are provided the service will generate and use self-signed certificates.
AUTHENTICATION	PAMAuthentication	By default the service uses PAM authentication i.e. When prompted for a username and password users can use the credentials that they use to login to the machine. Users can specify NoAuthentication to disable username/password prompt.
ADMIN_USERS	None	ADMIN_USERS can be used to specify which users have the ability to access other users workflow info. If ADMIN_USERS is None, False, or '' then users can only access their own workflow information. If ADMIN_USERS is '*' then all users are admin users and can access everyones workflow information. If ADMIN_USERS = {'u1', .., 'un'} OR ['u1', .., 'un'] then only users u1, .., un can access other users workflow information.
PROCESS_SWITCHING	True	File created by running Pegasus workflows have permissions as per user configuration. So one user migt not be able to view workflow information of other users. Setting PROCESS_SWITCHING to True makes the service change the process UID to the UID of the user whose information is being requested. pegasus-ser-

Property	Default Value	Description
		vice must be started as root for PROCESS_SWITCHING to work. PROCESS_SWITCHING can be set to False.
USERNAME	"	The username which pegasus-em client uses to connect to the pegasus-em server.
PASSWORD	"	The password which pegasus-em client uses to connect to the pegasus-em server.

All clients that connect to the web API will require the USERNAME and PASSWORD settings in the configuration file.

Running the Service

Pegasus Service can be started using the pegasus-service command as follows

```
$ pegasus-service
```

By default, the server will start on https://localhost:5000 [http://localhost:5000]. You can set the host and port in the configuration file OR pass it as a command line switch to pegasus-service as follows.

```
$ pegasus-service --host <SERVER_HOSTNAME> --port <SERVER_PORT>
```

Dashboard

The dashboard is automatically started when pegasus-service command is executed.

Running Pegasus Service under Apache HTTPD

Prerequisites Apache HTTPD, mod_ssl, and mod_wsgi to be installed.

To run pegasus-service under Apache HTTPD

1. Copy file share/pegasus/service/pegasus-service.wsgi to some other directory. We will refer to this directory as <WSGI_FILE_DIR>.

   Configure pegasus service by setting the AUTHENTICATION, PROCESS_SWITCHING, and/or ADMIN_USERS properties in the <WSGI_FILE_DIR>/pegasus-service.wsgi file as desired.

2. Copy file share/pegasus/service/pegasus-service-httpd.conf to your Apache conf directory.

   a. Replace PEGASUS_PYTHON_EXTERNALS with absolute path to pegasus python externals directory. Execute pegasus-config --python-externals to get this path

   b. Replace HOSTNAME with the hostname on which the server should listen for requests.

   c. Replace DOCUMENT_ROOT with <WSGI_FILE_DIR>

   d. Replace USER_NAME with the username as which the WSGIDaemonProcess should start

   e. Replace GROUP_NAME with the groupname as which the WSGIDaemonProcess should start

   f. Replace PATH_TO_PEGASUS_SERVICE_WSGI_FILE with <WSGI_FILE_DIR>/pegasus-service.wsgi

   g. Replace PATH_TO_SSL_CERT with absolute location of your SSL certificate file

   h. Replace PATH_TO_SSL_KEY with absolute location of your SSL private key file

For additional mod_wsgi configuration refer to https://code.google.com/p/modwsgi/wiki/ConfigurationDirectives

Ensemble Manager

The ensemble manager is a service that manages collections of workflows called ensembles. The ensemble manager is useful when you have a set of workflows you need to run over a long period of time. It can throttle the number of concurrent planning and running workflows, and plan and run workflows in priority order. A typical use-case is a user with 100 workflows to run, who needs no more than one to be planned at a time, and needs no more than two to be running concurrently.

The ensemble manager also allows workflows to be submitted and monitored programmatically through its RESTful interface, which makes it an ideal platform for integrating workflows into larger applications such as science gateways and portals.

To start the ensemble manager server, run:

```
$ pegasus-em server
```

Once the ensemble manager is running, you can create an ensemble with:

```
$ pegasus-em create myruns
```

where "myruns" is the name of the ensemble.

Then you can submit a workflow to the ensemble by running:

```
$ pegasus-em submit myruns.run1 ./plan.sh run1.dax
```

Where the name of the ensemble is "myruns", the name of the workflow is "run1", and "./plan.sh run1.dax" is the command for planning the workflow from the current working directory. The planning command should either be a direct invocation of pegasus-plan, or a shell script that calls pegasus-plan. If a shell script is used, then it should not redirect the output of pegasus-plan, because the ensemble manager reads the output to determine whether pegasus-plan succeeded and what is the submit directory of the workflow.

To check the status of your ensembles run:

```
$ pegasus-em ensembles
```

To check the status of your workflows run:

```
$ pegasus-em workflows myruns
```

To check the status of a specific workflow, run:

```
$ pegasus-em status myruns.run1
```

To help with debugging, the ensemble manager has an analyze command that emits diagnostic information about a workflow, including the output of pegasus-analyzer, if possible. To analyze a workflow, run:

```
$ pegasus-em analyze myruns.run1
```

Ensembles can be paused to prevent workflows from being planned and executed. Workflows in a paused ensemble will continue to run, but no new workflows will be planned or executed. To pause an ensemble, run:

```
$ pegasus-em pause myruns
```

Paused ensembles can be reactivated by running:

```
$ pegasus-em activate myruns
```

A workflow might fail during planning. In that case, run the analyze command to examine the planner output, make the necessary corrections to the workflow configuration, and replan the workflow by running:

```
$ pegasus-em replan myruns.run1
```

A workflow might also fail during execution. In that case, run the analyze command to identify the issue, correct the problem, and rerun the workflow by running:

```
$ pegasus-em rerun myruns.run1
```

Workflows in an ensemble can have different priorities. These priorities are used to determine the order in which workflows in the ensemble will be planned and executed. Priorities are specified using the '-p' option of the submit command. They can also be modified after a workflow has been submitted by running:

```
$ pegasus-em priority myruns.run1 -p 10
```

where 10 is the desired priority. Higher values have higher priority, the default is 0, and negative values are allowed.

Each ensemble has a pair of throttles that limit the number of workflows that are concurrently planning and executing. These throttles are called max_planning and max_running. Max planning limits the number of workflows in the ensemble that can be planned concurrently. Max running limits the number of workflows in the ensemble that can be running concurrently. These throttles are useful to limit the impact of planning on the memory usage of the submit host, and the load on the submit host and remote site caused by concurrently running workflows. The throttles can be specified with the '-R' and '-P' options of the create command. They can also be updated using the config command:

```
$ pegasus-em config myruns.run1 -P 1 -R 5
```

Chapter 13. Configuration

Pegasus has configuration options to configure

1. the behavior of an individual job via **profiles**

2. the behavior of the whole system via **properties**

For job level configuration (such as what environment a job is set with), the Pegasus Workflow Mapper uses the concept of profiles. Profiles encapsulate configurations for various aspects of dealing with the Grid infrastructure. They provide an abstract yet uniform interface to specify configuration options for various layers from planner/mapper behavior to remote environment settings. At various stages during the mapping process, profiles may be added associated with the job. The system supports five difffferent namespaces, with each namespace refers to a different aspect of a job's runtime settings. A profile's representation in the executable workflow (e.g. the Condor submit files) depends on its namespace. Pegasus supports the following Namespaces for profiles:

- **env** permits remote environment variables to be set.

- **globus** sets Globus RSL parameters.

- **condor** sets Condor configuration parameters for the submit file.

- **dagman** introduces Condor DAGMan configuration parameters.

- **pegasus** configures the behaviour of various planner/mapper components.

- **hints** allows to override site selection behavior of the planner. Can be specified only in the DAX.

Properties are primarily used to configure the behavior of the Pegasus WMS system at a global level. The properties file is actually a java properties file and follows the same conventions as that to specify the properties.

This chapter describes various types of profiles and properties, levels of priorities for intersecting profiles, and how to specify profiles in different contexts.

Differences between Profiles and Properties

The main difference between properties and profiles is that profiles eventually get associated at a per job level in the workflow. On the other hand, properties are a way of configuring and controlling the behavior of the whole system. While all profiles can be specified in the properties file, not all properties can be used as profiles. This section lists out the properties supported by Pegasus and if any can be used as a profile, it is clearly indicated.

Profiles

Profile Structure Heading

All profiles are triples comprised of a namespace, a name or key, and a value. The namespace is a simple identifier. The key has only meaning within its namespace, and it's yet another identifier. There are no constraints on the contents of a value

Profiles may be represented with different syntaxes in different context. However, each syntax will describe the underlying triple.

Sources for Profiles

Profiles may enter the job-processing stream at various stages. Depending on the requirements and scope a profile is to apply, profiles can be associated at

- as user property settings.

- dax level

- in the site catalog

- in the transformation catalog

Unfortunately, a different syntax applies to each level and context. This section shows the different profile sources and syntaxes. However, at the foundation of each profile lies the triple of namespace, key and value.

User Profiles in Properties

Users can specify all profiles in the properties files where the property name is **[namespace].key** and **value** of the property is the value of the profile.

Namespace can be env|condor|globus|dagman|pegasus

Any profile specified as a property applies to the whole workflow i.e (all jobs in the workflow) unless overridden at the DAX level , Site Catalog , Transformation Catalog Level.

Some profiles that they can be set in the properties file are listed below

```
env.JAVA_HOME "/software/bin/java"

condor.periodic_release 5
condor.periodic_remove  my_own_expression
condor.stream_error true
condor.stream_output fa

globus.maxwalltime  1000
globus.maxtime      900
globus.maxcputime   10
globus.project      test_project
globus.queue        main_queue

dagman.post.arguments --test arguments
dagman.retry  4
dagman.post simple_exitcode
dagman.post.path.simple_exitcode  /bin/exitcode/exitcode.sh
dagman.post.scope all
dagman.maxpre  12
dagman.priority 13

dagman.bigjobs.maxjobs 1


pegasus.clusters.size 5

pegasus.stagein.clusters 3
```

Profiles in DAX

The user can associate profiles with logical transformations in DAX. Environment settings required by a job's application, or a maximum estimate on the run-time are examples for profiles at this stage.

```
<job id="ID000001" namespace="asdf" name="preprocess" version="1.0"
 level="3" dv-namespace="voeckler" dv-name="top" dv-version="1.0">
  <argument>-a top -T10  -i <filename file="voeckler.f.a"/>
 -o <filename file="voeckler.f.b1"/>
 <filename file="voeckler.f.b2"/></argument>
  <profile namespace="pegasus" key="walltime">2</profile>
  <profile namespace="pegasus" key="diskspace">1</profile>
  &mldr;
</job>
```

Profiles in Site Catalog

If it becomes necessary to limit the scope of a profile to a single site, these profiles should go into the site catalog. A profile in the site catalog applies to all jobs and all application run at the site. Commonly, site catalog profiles set environment settings like the LD_LIBRARY_PATH, or globus rsl parameters like queue and project names.

Currently, there is no tool to manipulate the site catalog, e.g. by adding profiles. Modifying the site catalog requires that you load it into your editor.

The XML version of the site catalog uses the following syntax:

```
<profile namespace="namespace" key="key">value</profile>

<site  handle="CCG" arch="x86_64" os="LINUX">
     <grid  type="gt5" contact="obelix.isi.edu/jobmanager-fork" scheduler="Fork"
 jobtype="auxillary"/>

     <directory type="shared-scratch" path="/shared-scratch">
            <file-server operation="all" url="gsiftp://headnode.isi.edu/shared-scratch"/>
     </directory>
     <directory type="local-storage" path="/local-storage">
            <file-server operation="all" url="gsiftp://headnode.isi.edu/local-storage"/>
     </directory>
     <profile namespace="pegasus" key="clusters.num">1</profile>
     <profile namespace="env" key="PEGASUS_HOME">/usr</profile>
</site>
```

Profiles in Transformation Catalog

Some profiles require a narrower scope than the site catalog offers. Some profiles only apply to certain applications on certain sites, or change with each application and site. Transformation-specific and CPU-specific environment variables, or job clustering profiles are good candidates. Such profiles are best specified in the transformation catalog.

Profiles associate with a physical transformation and site in the transformation catalog. The Database version of the transformation catalog also permits the convenience of connecting a transformation with a profile.

The Pegasus tc-client tool is a convenient helper to associate profiles with transformation catalog entries. As benefit, the user does not have to worry about formats of profiles in the various transformation catalog instances.

```
tc-client -a -P -E -p /home/shared/executables/analyze -t INSTALLED -r isi_condor -e
 env::GLOBUS_LOCATION=&rdquor;/home/shared/globus&rdquor;
```

The above example adds an environment variable GLOBUS_LOCATION to the application /home/shared/executables/analyze on site isi_condor. The transformation catalog guide has more details on the usage of the tc-client.

```
tr example::keg:1.0 {

#specify profiles that apply for all the sites for the transformation
#in each site entry the profile can be overriden

  profile env "APP_HOME" "/tmp/myscratch"
  profile env "JAVA_HOME" "/opt/java/1.6"

  site isi {
    profile env "HELLo" "WORLD"
    profile condor "FOO" "bar"
    profile env "JAVA_HOME" "/bin/java.1.6"
    pfn "/path/to/keg"
    arch "x86"
    os "linux"
    osrelease "fc"
    osversion "4"
    type "INSTALLED"
  }

  site wind {
    profile env "CPATH" "/usr/cpath"
    profile condor "universe" "condor"
    pfn "file:///path/to/keg"
    arch "x86"
    os "linux"
    osrelease "fc"
    osversion "4"
    type "STAGEABLE"
  }
}
```

Most of the users prefer to edit the transformation catalog file directly in the editor.

Profiles Conflict Resolution

Irrespective of where the profiles are specified, eventually the profiles are associated with jobs. Multiple sources may specify the same profile for the same job. For instance, DAX may specify an environment variable X. The site catalog may also specify an environment variable X for the chosen site. The transformation catalog may specify an environment variable X for the chosen site and application. When the job is concretized, these three conflicts need to be resolved.

Pegasus defines a priority ordering of profiles. The higher priority takes precedence (overwrites) a profile of a lower priority.

1. Transformation Catalog Profiles

2. Site Catalog Profiles

3. DAX Profiles

4. Profiles in Properties

Details of Profile Handling

The previous sections omitted some of the finer details for the sake of clarity. To understand some of the constraints that Pegasus imposes, it is required to look at the way profiles affect jobs.

Details of env Profiles

Profiles in the env namespace are translated to a semicolon-separated list of key-value pairs. The list becomes the argument for the Condor environment command in the job's submit file.

```
######################################################################
# Pegasus WMS  SUBMIT FILE GENERATOR
# DAG : black-diamond, Index = 0, Count = 1
# SUBMIT FILE NAME : findrange_ID000002.sub
######################################################################
globusrsl = (jobtype=single)
environment=GLOBUS_LOCATION=/shared/globus;LD_LIBRARY_PATH=/shared/globus/lib;
executable = /shared/software/linux/pegasus/default/bin/kickstart
globusscheduler = columbus.isi.edu/jobmanager-condor
remote_initialdir = /shared/CONDOR/workdir/isi_hourglass
universe = globus
&mldr;
queue
######################################################################
# END OF SUBMIT FILE
```

Condor-G, in turn, will translate the *environment* command for any remote job into Globus RSL environment settings, and append them to any existing RSL syntax it generates. To permit proper mixing, all *environment* setting should solely use the env profiles, and none of the Condor nor Globus environment settings.

If *kickstart* starts a job, it may make use of environment variables in its executable and arguments setting.

Details of globus Profiles

Profiles in the *globus* Namespaces are translated into a list of paranthesis-enclosed equal-separated key-value pairs. The list becomes the value for the Condor *globusrsl* setting in the job's submit file:

```
######################################################################
# Pegasus WMS SUBMIT FILE GENERATOR
# DAG : black-diamond, Index = 0, Count = 1
# SUBMIT FILE NAME : findrange_ID000002.sub
######################################################################
globusrsl = (jobtype=single)(queue=fast)(project=nvo)
executable = /shared/software/linux/pegasus/default/bin/kickstart
globusscheduler = columbus.isi.edu/jobmanager-condor
remote_initialdir = /shared/CONDOR/workdir/isi_hourglass
universe = globus
&mldr;
queue
```

```
######################################################################
# END OF SUBMIT FILE
```

For this reason, Pegasus prohibits the use of the *globusrsl* key in the *condor* profile namespace.

The Env Profile Namespace

The *env* namespace allows users to specify environment variables of remote jobs. Globus transports the environment variables, and ensure that they are set before the job starts.

The key used in conjunction with an *env* profile denotes the name of the environment variable. The value of the profile becomes the value of the remote environment variable.

Grid jobs usually only set a minimum of environment variables by virtue of Globus. You cannot compare the environment variables visible from an interactive login with those visible to a grid job. Thus, it often becomes necessary to set environment variables like LD_LIBRARY_PATH for remote jobs.

If you use any of the Pegasus worker package tools like transfer or the rc-client, it becomes necessary to set PEGASUS_HOME and GLOBUS_LOCATION even for jobs that run locally

Table 13.1. Useful Environment Settings

Key Attributes	Description
Property Key: env.PEGASUS_HOME **Profile Key:** PEGASUS_HOME **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	Used by auxillary jobs created by Pegasus both on remote site and local site. Should be set usually set in the Site Catalog for the sites
Property Key: env.GLOBUS_LOCATION **Profile Key:** GLOBUS_LOCATION **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	Used by auxillary jobs created by Pegasus both on remote site and local site. Should be set usually set in the Site Catalog for the sites
Property Key: env.LD_LIBRARY_PATH **Profile Key:** LD_LIBRARY_PATH **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	Point this to $GLOBUS_LOCATION/lib, except you cannot use the dollar variable. You must use the full path. Applies to both, local and remote jobs that use Globus components and should be usually set in the site catalog for the sites

Even though Condor and Globus both permit environment variable settings through their profiles, all remote environment variables must be set through the means of *env* profiles.

The Globus Profile Namespace

The *globus* profile namespace encapsulates Globus resource specification language (RSL) instructions. The RSL configures settings and behavior of the remote scheduling system. Some systems require queue name to schedule jobs, a project name for accounting purposes, or a run-time estimate to schedule jobs. The Globus RSL addresses all these issues.

A key in the *globus* namespace denotes the command name of an RSL instruction. The profile value becomes the RSL value. Even though Globus RSL is typically shown using parentheses around the instruction, the out pair of parentheses is not necessary in globus profile specifications

The table below shows some commonly used RSL instructions. For an authoritative list of all possible RSL instructions refer to the Globus RSL specification.

Table 13.2. Useful Globus RSL Instructions

Property Key	Description
Property Key: globus.count	the number of times an executable is started.

Profile Key: count **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : Integer	
Property Key: globus.jobtype **Profile Key:** jobtype **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	specifies how the job manager should start the remote job. While Pegasus defaults to single, use mpi when running MPI jobs.
Property Key: globus.maxcputime **Profile Key:** maxcputime **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : Integer	the max CPU time in minutes for a single execution of a job.
Property Key: globus.maxmemory **Profile Key:** maxmemory **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : Integer	the maximum memory in MB required for the job
Property Key: globus.maxtime **Profile Key:** maxtime **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : Integer	the maximum time or walltime in minutes for a single execution of a job.
Property Key: globus.maxwalltime **Profile Key:** maxwalltime **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : Integer	the maximum walltime in minutes for a single execution of a job.
Property Key: globus.minmemory **Profile Key:** minmemory **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : Integer	the minumum amount of memory required for this job
Property Key: globus.project **Profile Key:** project **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	associates an account with a job at the remote end.
Property Key: globus.queue **Profile Key:** queue **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	the remote queue in which the job should be run. Used when remote scheduler is PBS that supports queues.

Pegasus prevents the user from specifying certain RSL instructions as globus profiles, because they are either automatically generated or can be overridden through some different means. For instance, if you need to specify remote environment settings, do not use the environment key in the globus profiles. Use one or more env profiles instead.

Table 13.3. RSL Instructions that are not permissible

Key	Reason for Prohibition
arguments	you specify arguments in the arguments section for a job in the DAX
directory	the site catalog and properties determine which directory a job will run in.

environment	use multiple env profiles instead
executable	the physical executable to be used is specified in the transformation catalog and is also dependant on the gridstart module being used. If you are launching jobs via kickstart then the executable created is the path to kickstart and the application executable path appears in the arguments for kickstart
stdin	you specify in the DAX for the job
stdout	you specify in the DAX for the job
stderr	you specify in the DAX for the job

The Condor Profile Namespace

The Condor submit file controls every detail how and where a job is run. The *condor* profiles permit to add or overwrite instructions in the Condor submit file.

The *condor* namespace directly sets commands in the Condor submit file for a job the profile applies to. Keys in the *condor* profile namespace denote the name of the Condor command. The profile value becomes the command's argument. All *condor* profiles are translated into key=value lines in the Condor submit file

Some of the common condor commands that a user may need to specify are listed below. For an authoritative list refer to the online condor documentation. Note: Pegasus Workflow Planner/Mapper by default specify a lot of condor commands in the submit files depending upon the job, and where it is being run.

Table 13.4. Useful Condor Commands

Property Key	Description
Property Key: condor.universe **Profile Key:** universe **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	Pegasus defaults to either globus or scheduler universes. Set to standard for compute jobs that require standard universe. Set to vanilla to run natively in a condor pool, or to run on resources grabbed via condor glidein.
Property Key: condor.periodic_release **Profile Key:** periodic_release **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	is the number of times job is released back to the queue if it goes to HOLD, e.g. due to Globus errors. Pegasus defaults to 3.
Property Key: condor.periodic_remove **Profile Key:** periodic_remove **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	is the number of times a job is allowed to get into HOLD state before being removed from the queue. Pegasus defaults to 3.
Property Key: condor.filesystemdomain **Profile Key:** filesystemdomain **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	Useful for Condor glide-ins to pin a job to a remote site.
Property Key: condor.stream_error **Profile Key:** stream_error **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : Boolean	boolean to turn on the streaming of the stderr of the remote job back to submit host.
Property Key: condor.stream_output **Profile Key:** stream_output **Scope** : TC, SC, DAX, Properties	boolean to turn on the streaming of the stdout of the remote job back to submit host.

Since : 2.0 **Type** : Boolean	
Property Key: condor.priority **Profile Key:** priority **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	integer value to assign the priority of a job. Higher value means higher priority. The priorities are only applied for vanilla / standard/ local universe jobs. Determines the order in which a users own jobs are executed.
Property Key: condor.request_cpus **Profile Key:** request_cpus **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	New in Condor 7.8.0 . Number of CPU's a job requires.
Property Key: condor.request_gpus **Profile Key:** request_cpus **Scope** : TC, SC, DAX, Properties **Since** : 4.6 **Type** : String	Number of GPU's a job requires.
Property Key: condor.request_memory **Profile Key:** request_memory **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	New in Condor 7.8.0 . Amount of memory a job requires.
Property Key: condor.request_disk **Profile Key:** request_disk **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	New in Condor 7.8.0 . Amount of disk a job requires.

Other useful condor keys, that advanced users may find useful and can be set by profiles are

1. should_transfer_files

2. transfer_output

3. transfer_error

4. whentotransferoutput

5. requirements

6. rank

Pegasus prevents the user from specifying certain Condor commands in condor profiles, because they are automatically generated or can be overridden through some different means. The table below shows prohibited Condor commands.

Table 13.5. Condor commands prohibited in condor profiles

Key	Reason for Prohibition
arguments	you specify arguments in the arguments section for a job in the DAX
environment	use multiple env profiles instead
executable	the physical executable to be used is specified in the transformation catalog and is also dependant on the gridstart module being used. If you are launching jobs via kickstart then the executable created is the path to kickstart and the application executable path appears in the arguments for kickstart

The Dagman Profile Namespace

DAGMan is Condor's workflow manager. While planners generate most of DAGMan's configuration, it is possible to tweak certain job-related characteristics using dagman profiles. A dagman profile can be used to specify a DAGMan pre- or post-script.

Pre- and post-scripts execute on the submit machine. Both inherit the environment settings from the submit host when pegasus-submit-dag or pegasus-run is invoked.

By default, kickstart launches all jobs except standard universe and MPI jobs. Kickstart tracks the execution of the job, and returns usage statistics for the job. A DAGMan post-script starts the Pegasus application exitcode to determine, if the job succeeded. DAGMan receives the success indication as exit status from exitcode.

If you need to run your own post-script, you have to take over the job success parsing. The planner is set up to pass the file name of the remote job's stdout, usually the output from kickstart, as sole argument to the post-script.

The table below shows the keys in the dagman profile domain that are understood by Pegasus and can be associated at a per job basis.

Table 13.6. Useful dagman Commands that can be associated at a per job basis

Property Key	Description
Property Key: dagman.pre **Profile Key:** PRE **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	is the path to the pre-script. DAGMan executes the pre-script before it runs the job.
Property Key: dagman.pre.arguments **Profile Key:** PRE.ARGUMENTS **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	are command-line arguments for the pre-script, if any.
Property Key: dagman.post **Profile Key:** POST **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	is the postscript type/mode that a user wants to associate with a job. 1. **pegasus-exitcode** - pegasus will by default associate this postscript with all jobs launched via kickstart, as long the POST.SCOPE value is not set to NONE. 2. **none** -means that no postscript is generated for the jobs. This is useful for MPI jobs that are not launched via kickstart currently. 3. **any legal identifier** - Any other identifier of the form ([_A-Za-z][_A-Za-z0-9]*), than one of the 2 reserved keywords above, signifies a user postscript. This allows the user to specify their own postscript for the jobs in the workflow. The path to the postscript can be specified by the dagman profile **POST.PATH.[value]** where [value] is this legal identifier specified. The user postscript is passed the name of the .out file of the job as the last argument on the command line. For e.g. if the following dagman profiles were associated with a job X a. POST with value user_script /bin/user_postscript b. POST.PATH.user_script with value /path/to/user/script

	c. POST.ARGUMENTS with value -verbose then the following postscript will be associated with the job X in the .dag file /path/to/user/script -verbose X.out where X.out contains the stdout of the job X
Property Key: dagman.post.path.[value of dagman.post] **Profile Key:** post.path.[value of dagman.post] **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	the path to the post script on the submit host.
Property Key: dagman.post.arguments **Profile Key:** POST.ARGUMENTS **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	are the command line arguments for the post script, if any.
Property Key: dagman.retry **Profile Key:** RETRY **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : Integer **Default** : 1	is the number of times DAGMan retries the full job cycle from pre-script through post-script, if failure was detected.
Property Key: dagman.category **Profile Key:** CATEGORY **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	the DAGMan category the job belongs to.
Property Key: dagman.priority **Profile Key:** PRIORITY **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : Integer	the priority to apply to a job. DAGMan uses this to select what jobs to release when MAXJOBS is enforced for the DAG.
Property Key: dagman.abort-dag-on **Profile Key:** ABORT-DAG-ON **Scope** : TC, DAX, **Since** : 4.5 **Type** : String	The ABORT-DAG-ON key word provides a way to abort the entire DAG if a given node returns a specific exit code (AbortExitValue). The syntax for the value of the key is AbortExitValue [RETURN DAGReturnValue] . When a DAG aborts, by default it exits with the node return value that caused the abort. This can be changed by using the optional RETURN key word along with specifying the desired DAGReturnValue

The table below shows the keys in the dagman profile domain that are understood by Pegasus and can be used to apply to the whole workflow. These are used to control DAGMan's behavior at the workflow level, and are recommended to be specified in the properties file.

Table 13.7. Useful dagman Commands that can be specified in the properties file.

Property Key	Description
Property Key: dagman.maxpre **Profile Key:** MAXPRE **Scope** : Properties **Since** : 2.0 **Type** : String	sets the maximum number of PRE scripts within the DAG that may be running at one time
Property Key: dagman.maxpost **Profile Key:** MAXPOST	sets the maximum number of POST scripts within the DAG that may be running at one time

Scope **:** Properties **Since** **:** 2.0 **Type** **:** String	
Property Key: dagman.maxjobs **Profile Key:** MAXJOBS **Scope** **:** Properties **Since** **:** 2.0 **Type** **:** String	sets the maximum number of jobs within the DAG that will be submitted to Condor at one time.
Property Key: dagman.maxidle **Profile Key:** MAXIDLE **Scope** **:** Properties **Since** **:** 2.0 **Type** **:** String	Sets the maximum number of idle jobs allowed before HTCondor DAGMan stops submitting more jobs. Once idle jobs start to run, HTCondor DAGMan will resume submitting jobs. If the option is omitted, the number of idle jobs is unlimited.
Property Key: dagman.[CATEGORY-NAME].maxjobs **Profile Key:** [CATEGORY-NAME].MAXJOBS **Scope** **:** Properties **Since** **:** 2.0 **Type** **:** String	is the value of maxjobs for a particular category. Users can associate different categories to the jobs at a per job basis. However, the value of a dagman knob for a category can only be specified at a per workflow basis in the properties.
Property Key: dagman.post.scope **Profile Key:** POST.SCOPE **Scope** **:** Properties **Since** **:** 2.0 **Type** **:** String	scope for the postscripts. 1. If set to **all** , means each job in the workflow will have a postscript associated with it. 2. If set to **none** , means no job has postscript associated with it. None mode should be used if you are running vanilla / standard/ local universe jobs, as in those cases Condor traps the remote exitcode correctly. None scope is not recommended for grid universe jobs. 3. If set to **essential**, means only essential jobs have post scripts associated with them. At present the only non essential job is the replica registration job.

The Pegasus Profile Namespace

The *pegasus* profiles allow users to configure extra options to the Pegasus Workflow Planner that can be applied selectively to a job or a group of jobs. Site selectors may use a sub-set of *pegasus* profiles for their decision-making.

The table below shows some of the useful configuration option Pegasus understands.

Table 13.8. Useful pegasus Profiles.

Property Key	Description
Property Key: pegasus.clusters.num **Profile Key:** clusters.num **Scope** **:** TC, SC, DAX, Properties **Since** **:** 3.0 **Type** **:** Integer	Please refer to the Pegasus Clustering Guide for detailed description. This option determines the total number of clusters per level. Jobs are evenly spread across clusters.
Property Key: pegasus.clusters.size **Profile Key:** clusters.size **Scope** **:** TC, SC, DAX, Properties **Since** **:** 3.0 **Type** **:** Integer	Please refer to the Pegasus Clustering Guide for detailed description. This profile determines the number of jobs in each cluster. The number of clusters depends on the total number of jobs on the level.
Property Key: pegasus.job.aggregator **Profile Key:** job.aggregator **Scope** **:** TC, SC, DAX, Properties **Since** **:** 2.0	Indicates the clustering executable that is used to run the clustered job on the remote site.

Type : Integer	
Property Key: pegasus.gridstart **Profile Key:** gridstart **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	Determines the executable for launching a job. This covers both tasks (jobs specified by the user in the DAX) and additional jobs added by Pegasus during the planning operation. Possible values are ***Kickstart | NoGridStart*** | PegasusLite | Distribute at the moment. ## Note This profile should only be set by users if you know what you are doing. Otherwise, let Pegasus do the right thing based on your configuration. Kickstart By default, all jobs executed are launched using a lightweight C executable called pegasus-kickstart. This generates valuable runtime provenance information for the job as it is executed on a remote node. This information serves as the basis for the monitoring and debugging capabilities provided by Pegasus. NoGridStart This explicity disables the wrapping of the jobs with pegasus-kickstart. This is internally used by the planner to launch dax jobs directly. If this is set, then the information populated in the monitording database is on the basis of what is recorded in the DAGMan out file. PegasusLite This value is automatically associated by the Planner whenever the job runs in either nonsharedfs or condorio mode. The property pegasus.data.configuration decides whether a job is launched via PegasusLite or not. PegasusLite is a lightweight Pegasus wrapper generated for each job that allows a job to run in a nonshared file system environment and is responsible for staging in the input data and staging out the output data back to a remote staging site for the job. Distribute This wrapper is a HubZero specfiic wrapper that allows compute jobs that are scheduled for a local PBS cluster to be run locally on the submit host. The jobs are wrapped with a distribute wrapper that is responsible for doing the qsub and tracking of the status of the jobs in the PBS cluster.
Property Key: pegasus.gridstart.path **Profile Key:** gridstart.path **Scope** : TC, SC, DAX, Properties **Since** : 2.0	Sets the path to the gridstart . This profile is best set in the Site Catalog.

Type : file path	
Property Key: pegasus.gridstart.arguments **Profile Key:** gridstart.arguments **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	Sets the arguments with which GridStart is used to launch a job on the remote site.
Property Key: pegasus.stagein.clusters **Profile Key:** stagein.clusters **Scope** : TC, SC, DAX, Properties **Since** : 4.0 **Type** : Integer	This key determines the maximum number of *stage-in* jobs that are can executed locally or remotely per compute site per workflow. This is used to configure the *BalancedCluster* Transfer Refiner, which is the Default Refiner used in Pegasus. This profile is best set in the Site Catalog or in the Properties file
Property Key: pegasus.stagein.local.clusters **Profile Key:** stagein.local.clusters **Scope** : TC, SC, DAX, Properties **Since** : 4.0 **Type** : Integer	This key provides finer grained control in determining the number of stage-in jobs that are executed locally and are responsible for staging data to a particular remote site. This profile is best set in the Site Catalog or in the Properties file
Property Key: pegasus.stagein.remote.clusters **Profile Key:** stagein.remote.clusters **Scope** : TC, SC, DAX, Properties **Since** : 4.0 **Type** : Integer	This key provides finer grained control in determining the number of stage-in jobs that are executed remotely on the remote site and are responsible for staging data to it. This profile is best set in the Site Catalog or in the Properties file
Property Key: pegasus.stageout.clusters **Profile Key:** stageout.clusters **Scope** : TC, SC, DAX, Properties **Since** : 4.0 **Type** : Integer	This key determines the maximum number of *stage-out* jobs that are can executed locally or remotely per compute site per workflow. This is used to configure the *BalancedCluster* Transfer Refiner, , which is the Default Refiner used in Pegasus.
Property Key: pegasus.stageout.local.clusters **Profile Key:** stageout.local.clusters **Scope** : TC, SC, DAX, Properties **Since** : 4.0 **Type** : Integer	This key provides finer grained control in determining the number of stage-out jobs that are executed locally and are responsible for staging data from a particular remote site. This profile is best set in the Site Catalog or in the Properties file
Property Key: pegasus.stageout.remote.clusters **Profile Key:** stageout.remote.clusters **Scope** : TC, SC, DAX, Properties **Since** : 4.0 **Type** : Integer	This key provides finer grained control in determining the number of stage-out jobs that are executed remotely on the remote site and are responsible for staging data from it. This profile is best set in the Site Catalog or in the Properties file
Property Key: pegasus.group **Profile Key:** group **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	Tags a job with an arbitrary group identifier. The group site selector makes use of the tag.
Property Key: pegasus.change.dir **Profile Key:** change.dir **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : Boolean	If true, tells *kickstart* to change into the remote working directory. Kickstart itself is executed in whichever directory the remote scheduling system chose for the job.
Property Key: pegasus.create.dir **Profile Key:** create.dir **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : Boolean	If true, tells *kickstart* to create the the remote working directory before changing into the remote working directory. Kickstart itself is executed in whichever directory the remote scheduling system chose for the job.
Property Key: pegasus.transfer.proxy **Profile Key:** transfer.proxy **Scope** : TC, SC, DAX, Properties	If true, tells Pegasus to explicitly transfer the proxy for transfer jobs to the remote site. This is useful, when you

Since : 2.0 **Type** : Boolean	want to use a full proxy at the remote end, instead of the limited proxy that is transferred by CondorG.
Property Key: pegasus.style **Profile Key:** style **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : String	Sets the condor submit file style. If set to globus, submit file generated refers to CondorG job submissions. If set to condor, submit file generated refers to direct Condor submission to the local Condor pool. It applies for glidein, where nodes from remote grid sites are glided into the local condor pool. The default style that is applied is globus.
Property Key: pegasus.pmc_request_memory **Profile Key:** pmc_request_memory **Scope** : TC, SC, DAX, Properties **Since** : 4.2 **Type** : Integer	This key is used to set the -m option for pegasus-mpi-cluster. It specifies the amount of memory in MB that a job requires. This profile is usually set in the DAX for each job.
Property Key: pegasus.pmc_request_cpus **Profile Key:** pmc_request_cpus **Scope** : TC, SC, DAX, Properties **Since** : 4.2 **Type** : Integer	This key is used to set the -c option for pegasus-mpi-cluster. It specifies the number of cpu's that a job requires. This profile is usually set in the DAX for each job.
Property Key: pegasus.pmc_priority **Profile Key:** pmc_priority **Scope** : TC, SC, DAX, Properties **Since** : 4.2 **Type** : Integer	This key is used to set the -p option for pegasus-mpi-cluster. It specifies the priority for a job . This profile is usually set in the DAX for each job. Negative values are allowed for priorities.
Property Key: pegasus.pmc_task_arguments **Profile Key:** pmc_task_arguments **Scope** : TC, SC, DAX, Properties **Since** : 4.2 **Type** : String	The key is used to pass any extra arguments to the PMC task during the planning time. They are added to the very end of the argument string constructed for the task in the PMC file. Hence, allows for overriding of any argument constructed by the planner for any particular task in the PMC job.
Property Key: pegasus.exitcode.failuremsg **Profile Key:** exitcode.failuremsg **Scope** : TC, SC, DAX, Properties **Since** : 4.4 **Type** : String	The message string that pegasus-exitcode searches for in the stdout and stderr of the job to flag failures.
Property Key: pegasus.exitcode.successmsg **Profile Key:** exitcode.successmsg **Scope** : TC, SC, DAX, Properties **Since** : 4.4 **Type** : String	The message string that pegasus-exitcode searches for in the stdout and stderr of the job to determine whether a job logged it's success message or not. Note this value is used to check for whether a job failed or not i.e if this profile is specified, and pegasus-exitcode DOES NOT find the string in the job stdout or stderr, the job is flagged as failed. The complete rules for determining failure are described in the man page for pegasus-exitcode.
Property Key: pegasus.checkpoint.time **Profile Key:** checkpoint_time **Scope** : TC, SC, DAX, Properties **Since** : 4.5 **Type** : Integer	the expected time in minutes for a job after which it should be sent a TERM signal to generate a job checkpoint file
Property Key: pegasus.maxwalltime **Profile Key:** maxwalltime **Scope** : TC, SC, DAX, Properties **Since** : 4.5 **Type** : Integer	the maximum walltime in minutes for a single execution of a job.
Property Key: pegasus.glite.arguments **Profile Key:** glite.arguments **Scope** : TC, SC, DAX, Properties	specifies the extra arguments that must appear in the local PBS generated script for a job, when running workflows on a local cluster with submissions through Glite. This is

Since : 4.5 **Type** : String	useful when you want to pass through special options to underlying LRMS such as PBS e.g. you can set value -l walltime=01:23:45 -l nodes=2 to specify your job's resource requirements.
Profile Key: auxillary.local **Scope** : SC **Since** : 4.6 **Type** : Boolean	indicates whether auxillary jobs associated with a compute site X, can be run on local site. This CAN ONLY be specified as a profile in the site catalog and should be set when the compute site filesystem is accessible locally on the submit host.
Property Key: pegasus.condor.arguments.quote **Profile Key:** condor.arguments.quote **Scope** : SC, Properties **Since** : 4.6 **Type** : Boolean	indicates whether condor quoting rules should be applied for writing out the arguments key in the condor submit file. By default it is true unless the job is schedule to a glite style site. The value is automatically set to false for glite style sites, as condor quoting is broken in batch_gahp.

Task Resource Requirements Profiles

Startng Pegasus 4.6.0 Release, users can specify pegasus profiles to describe resources requirements for their job. The planner will automatically translate them to appropriate execution environment specific directives. For example, the profiles are automatically translated to Globus RSL keys if submitting job via CondorG to remote GRAM instances, Condor Classad keys when running in a vanilla condor pool and to appropriate shell variables for Glite that can be picked up by the local attributes.sh. The profiles are described below.

Table 13.9. Task Resource Requirement Profiles.

Property Key	Description
Property Key: pegasus.runtime **Profile Key:** runtime **Scope** : TC, SC, DAX, Properties **Since** : 2.0 **Type** : Long	This profile specifies the expected runtime of a job in seconds. Refer to the Pegasus Clustering Guide for description on using it for runtime clustering.
Property Key: clusters.maxruntime **Profile Key:** pegasus.clusters.maxruntime **Scope** : TC, SC, DAX, Properties **Since** : 4.0 **Type** : Integer	Please refer to the Pegasus Clustering Guide for detailed description. This profile specifies the maximum runtime of a job.
Property Key: pegasus.cores **Profile Key:** cores **Scope** : TC, SC, DAX, Properties **Since** : 4.0 **Type** : Integer	The total number of cores, required for a job. This is also used for accounting purposes in the database while generating statistics. It corresponds to the multiplier_factor in the job_instance table described here.
Property Key: pegasus.nodes **Profile Key:** nodes **Scope** : TC, SC, DAX, Properties **Since** : 4.6 **Type** : Integer	Indicates the the number of nodes a job requires.
Property Key: pegasus.ppn **Profile Key:** ppn **Scope** : TC, SC, DAX, Properties **Since** : 4.6 **Type** : Integer	Indicates the number of processors per node . This profile is best set in the Site Catalog and usually set when running workflows with MPI jobs.
Property Key: pegasus.memory **Profile Key:** memory **Scope** : TC, SC, DAX, Properties **Since** : 4.6 **Type** : Long	Indicates the maximum memory a job requires in MB.

Property Key: pegasus.diskspace Profile Key: diskspace Scope : TC, SC, DAX, Properties Since : 4.6 Type : Long	Indicates the maximum diskspace a job requires in MB.

The automatic translation to various execution environment specific directives is explained below. It is important, to note that execution environment specific keys take precedence over the Pegasus profile keys. For example, Globus profile key maxruntime will be preferred over Pegasus profile key runtime when running jobs via HTCondorG.

Table 13.10. Table mapping translation of Pegasus Task Requirements to corresponding execution environment keys.

Pegasus Task Resource Requirement Profile Key	Corresponding Globus RSL Key	Corresponding Condor Classad Key	KEY in +remote_cerequirements classad for GLITE
runtime	maxruntime	-	WALLTIME
cores	count	request_cpus	CORES
nodes	hostcount	-	NODES
ppn	xcount	-	PROCS
memory	maxmemory	request_memory	PER_PROCESS_MEMORY
diskspace	-	request_diskspace	-

The Hints Profile Namespace

The *hints* namespace allows users to override the behavior of the Workflow Mapper during site selection. This gives you finer grained control over where a job executes and what executable it refers to. The hints namespace keys (execution.site and pfn) can only be specified in the DAX. It is important to note that these particular keys once specified in the DAX, cannot be overriden like other profiles.

Table 13.11. Useful Hints Profile Keys

Key Attributes	Description
Property Key: N/A Profile Key: execution.site Scope : DAX Since : 4.5 Type : String	the execution site where a job should be executed.
Property Key: N/A Profile Key: pfn Scope : TC, SC, DAX, Properties Since : 4.5 Type : String	the physical file name to the main executable that a job refers to. Overrides any entries specified in the transformation catalog.
Property Key: hints.grid.jobtype Profile Key: grid.jobtype Scope : TC, SC, DAX, Properties Since : 4.5 Type : String	applicable when submitting to remote sites via GRAM. The site catalog allows you to associate multiple job-managers with a GRAM site, for different type of jobs [compute, auxillary, transfer, register, cleanup] that Pegasus generates in the executable workflow. This profile is usually used to ensure that a compute job executes on another job manager. For example, if in site catalog you have headnode.example.com/jobmanager-condor for compute jobs, and headnode.example.com/jobmanager-fork for auxillary jobs. Associating this profile and setting value to auxillary for a compute job, will cause the

	compute job to run on the fork jobmanager instead of the condor jobmanager.

Properties

Properties are primarily used to configure the behavior of the Pegasus Workflow Planner at a global level. The properties file is actually a java properties file and follows the same conventions as that to specify the properties.

Please note that the values rely on proper capitalization, unless explicitly noted otherwise.

Some properties rely with their default on the value of other properties. As a notation, the curly braces refer to the value of the named property. For instance, ${pegasus.home} means that the value depends on the value of the pegasus.home property plus any noted additions. You can use this notation to refer to other properties, though the extent of the subsitutions are limited. Usually, you want to refer to a set of the standard system properties. Nesting is not allowed. Substitutions will only be done once.

There is a priority to the order of reading and evaluating properties. Usually one does not need to worry about the priorities. However, it is good to know the details of when which property applies, and how one property is able to overwrite another. The following is a mutually exclusive list (highest priority first) of property file locations.

1. --conf option to the tools. Almost all of the clients that use properties have a --conf option to specify the property file to pick up.

2. submit-dir/pegasus.xxxxxxx.properties file. All tools that work on the submit directory (i.e after pegasus has planned a workflow) pick up the pegasus.xxxxx.properties file from the submit directory. The location for the pegasus.xxxxxxx.propertiesis picked up from the braindump file.

3. The properties defined in the user property file *${user.home}/.pegasusrc* have lowest priority.

Commandline properties have the highest priority. These override any property loaded from a property file. Each commandline property is introduced by a -D argument. Note that these arguments are parsed by the shell wrapper, and thus the -D arguments must be the first arguments to any command. Commandline properties are useful for debugging purposes.

From Pegasus 3.1 release onwards, support has been dropped for the following properties that were used to signify the location of the properties file

• pegasus.properties

• pegasus.user.properties

The following example provides a sensible set of properties to be set by the user property file. These properties use mostly non-default settings. It is an example only, and will not work for you:

```
pegasus.catalog.replica              File
pegasus.catalog.replica.file         ${pegasus.home}/etc/sample.rc.data
pegasus.catalog.transformation       Text
pegasus.catalog.transformation.file  ${pegasus.home}/etc/sample.tc.text
pegasus.catalog.site.file            ${pegasus.home}/etc/sample.sites.xml
```

If you are in doubt which properties are actually visible, pegasus during the planning of the workflow dumps all properties after reading and prioritizing in the submit directory in a file with the suffix properties.

Local Directories Properties

This section describes the GNU directory structure conventions. GNU distinguishes between architecture independent and thus sharable directories, and directories with data specific to a platform, and thus often local. It also distinguishes between frequently modified data and rarely changing data. These two axis form a space of four distinct directories.

Table 13.12. Local Directories Related Properties

Key Attributes	Description

Property Key: pegasus.home.datadir Profile Key: N/A Scope : Properties Since : 2.0 Type : file path Default : ${pegasus.home}/share	The datadir directory contains broadly visible and possibly exported configuration files that rarely change. This directory is currently unused.
Property Key: pegasus.home.sysconfdir Profile Key: N/A Scope : Properties Since : 2.0 Type : file path Default : ${pegasus.home}/etc	The system configuration directory contains configuration files that are specific to the machine or installation, and that rarely change. This is the directory where the XML schema definition copies are stored, and where the base pool configuration file is stored.
Property Key: pegasus.home.sharedstatedir Profile Key: N/A Scope : Properties Since : 2.0 Type : file path Default : ${pegasus.home}/com	Frequently changing files that are broadly visible are stored in the shared state directory. This is currently unused.
Property Key: pegasus.home.localstatedir Profile Key: N/A Scope : Properties Since : 2.0 Type : file path Default : ${pegasus.home}/var	Frequently changing files that are specific to a machine and/or installation are stored in the local state directory. This is currently unused
Property Key: pegasus.dir.submit.logs Profile Key: N/A Scope : Properties Since : 2.0 Type : file path Default : (no default)	This property can be used to specify the directory where the condor logs for the workflow should go to. By default, starting 4.2.1 release, Pegasus will setup the log to be in the workflow submit directory. This can create problems, in case users submit directories are on NSF. This is done to ensure that the logs are created in a local directory even though the submit directory maybe on NFS

Site Directories Properties

The site directory properties modify the behavior of remotely run jobs. In rare occasions, it may also pertain to locally run compute jobs.

Table 13.13. Site Directories Related Properties

Key Attributes	Description
Property Key: pegasus.dir.useTimestamp Profile Key: N/A Scope : Properties Since : 2.1 Type : Boolean Default : false	While creating the submit directory, Pegasus employs a run numbering scheme. Users can use this Boolean property to use a timestamp based numbering scheme instead of the runxxxx scheme.
Property Key: pegasus.dir.exec Profile Key: N/A Scope : Properties Since : 2.0 Type : file path Default : (no default)	This property modifies the remote location work directory in which all your jobs will run. If the path is relative then it is appended to the work directory (associated with the site), as specified in the site catalog. If the path is absolute then it overrides the work directory specified in the site catalog.
Property Key: pegasus.dir.submit.mapper Profile Key: N/A	This property modifies determines how the directory for job submit files are mapped on the submit host.

Scope : Properties
Since : 4.7
Type : Enumeration
Values : Flat|Hashed
Default : Hashed
	Flat This mapper results in Pegasus placing all the job submit files in the submit directory as determined from the planner options. This can result in too many files in one directory for large workflows, and was the only option before Pegasus 4.7.0 release.
	
	Hashed This mapper results in the creation of a deep directory structure rooted at the submit directory. The base directory is the submit directory as determined from the planner options. By default, the directory structure created is two levels deep. To control behavior of this mapper, users can specify the following properties
	
	`pegasus.dir.submit.mapper.hashed.levels`
	`    the number of directory levels`
	`used`
	
	`      to accomodate the files. Defaults`
	`to 2.`
	`pegasus.dir.submit.mapper.hashed.multiplier`
	`  the number of files associated with a`
	`job`
	
	`      in the submit directory. defaults`
	`to 5.`

Property Key: pegasus.dir.staging.mapper
Profile Key: N/A
Scope : Properties
Since : 4.7
Type : Enumeration
Values : Flat|Hashed
Default : Hashed
	This property modifies determines how the job input and output files are mapped on the staging site. This only applies when the pegasus data configuration is set to non-sharedfs.
	
	Flat This mapper results in Pegasus placing all the job submit files in the staging site directory as determined from the Site Catalog and planner options. This can result in too many files in one directory for large workflows, and was the only option before Pegasus 4.7.0 release.
	
	Hashed This mapper results in the creation of a deep directory structure rooted at the staging site directory created by the create dir jobs. The binning is at the job level, and not at the file level i.e each job will push out it's outputs to the same directory on the staging site, independent of the number of output files. To control behavior of this mapper, users can specify the following properties
	
	`pegasus.dir.staging.mapper.hashed.levels`
	`    the number of directory levels`
	`used`
	
	`      to accomodate the files. Defaults`
	`to 2.`
	`pegasus.dir.staging.mapper.hashed.multiplier`
	`  the number of files associated with a`
	`job`
	
	`      in the submit directory. defaults`
	`to 5.`

Property Key: pegasus.dir.storage.mapper **Profile Key:** N/A **Scope** : Properties	This property modifies determines how the output files are mapped on the output site storage location.

Since : 4.3 **Type** : Enumeration **Values** : Flat|Fixed|Hashed|Replica **Default** : Flat	In order to preserve backward compatibility, setting the boolean property pegasus.dir.storage.deep results in the Hashed output mapper to be loaded, if no output mapper property is specified.

Flat	By default, Pegasus will place the output files in the storage directory specified in the site catalog for the output site.
| Fixed | Using this mapper, users can specify an externally accesible url to the storage directory in their properties file. The following property needs to be set.<br><br>```
pegasus.dir.storage.mapper.fixed.url
 an externally accessible URL to the
storage directory on the output site
e.g. gsiftp://outputs.isi.edu/shared/
outputs
```<br><br>Note: For hierarchal workflows, the above property needs to be set separately for each dax job, if you want the sub workflow outputs to goto a different directory. |
| Hashed | This mapper results in the creation of a deep directory structure on the output site, while populating the results. The base directory on the remote end is determined from the site catalog. Depending on the number of files being staged to the remote site a Hashed File Structure is created that ensures that only 256 files reside in one directory. To create this directory structure on the storage site, Pegasus relies on the directory creation feature of the Grid FTP server, which appeared in globus 4.0.x |
| Replica | This mapper determines the path for an output file on the output site by querying an output replica catalog. The output site is one that is passed on the command line. The output replica catalog can be configured by specifiing the properties with the prefix pegasus.dir.storage.replica. By default, a Regex File based backend is assumed unless overridden. For example<br><br>```
pegasus.dir.storage.mapper.replica
    Regex|File
pegasus.dir.storage.mapper.replica.file
   the RC file at the backend to use
      if using a file based RC
``` |

Property Key: pegasus.dir.storage.deep **Profile Key:** N/A **Scope** : Properties **Since** : 2.1 **Type** : Boolean **Default** : false	This Boolean property results in the creation of a deep directory structure on the output site, while populating the results. The base directory on the remote end is determined from the site catalog. To this base directory, the relative submit directory structure ($user/$vogroup/$label/runxxxx) is appended. $storage = $base + $relative_submit_directory

	This is the base directory that is passed to the storage mapper.
	Note: To preserve backward compatibilty, setting this property results in the Hashed mapper to be loaded unless pegasus.dir.storage.mapper is explicitly specified. Before 4.3, this property resulted in HashedDirectory structure.
Property Key: pegasus.dir.create.strategy **Profile Key:** N/A **Scope** : Properties **Since** : 2.2 **Type** : Enumeration **Values** : HourGlass|Tentacles|Minimal **Default** : Minimal	If the `--randomdir` option is given to the Planner at runtime, the Pegasus planner adds nodes that create the random directories at the remote pool sites, before any jobs are actually run. The two modes determine the placement of these nodes and their dependencies to the rest of the graph.
	HourGlass It adds a make directory node at the top level of the graph, and all these concat to a single dummy job before branching out to the root nodes of the original/ concrete dag so far. So we introduce a classic X shape at the top of the graph. Hence the name HourGlass.
	Tentacles This option places the jobs creating directories at the top of the graph. However instead of constricting it to an hour glass shape, this mode links the top node to all the relevant nodes for which the create dir job is necessary. It looks as if the node spreads its tentacleas all around. This puts more load on the DAGMan because of the added dependencies but removes the restriction of the plan progressing only when all the create directory jobs have progressed on the remote pools, as is the case in the HourGlass model.
	Minimal The strategy involves in walking the graph in a BFS order, and updating a bit set associated with each job based on the BitSet of the parent jobs. The BitSet indicates whether an edge exists from the create dir job to an ancestor of the node. For a node, the bit set is the union of all the parents BitSets. The BFS traversal ensures that the bitsets are of a node are only updated once the parents have been processed.

Schema File Location Properties

This section defines the location of XML schema files that are used to parse the various XML document instances in the PEGASUS. The schema backups in the installed file-system permit PEGASUS operations without being online.

Table 13.14. Schema File Location Properties

Key Attributes	Description

Property Key: pegasus.schema.dax **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Type** : file path **Default** : ${pegasus.home.sysconfdir}/dax-3.4.xsd	This file is a copy of the XML schema that describes abstract DAG files that are the result of the abstract planning process, and input into any concrete planning. Providing a copy of the schema enables the parser to use the local copy instead of reaching out to the Internet, and obtaining the latest version from the Pegasus website dynamically.
Property Key: pegasus.schema.sc **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Type** : file path **Default** : ${pegasus.home.sysconfdir}/sc-4.0.xsd	This file is a copy of the XML schema that describes the xml description of the site catalog. Providing a copy of the schema enables the parser to use the local copy instead of reaching out to the internet, and obtaining the latest version from the GriPhyN website dynamically.
Property Key: pegasus.schema.ivr **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Type** : file path **Default** : ${pegasus.home.sysconfdir}/iv-2.0.xsd	This file is a copy of the XML schema that describes invocation record files that are the result of the a grid launch in a remote or local site. Providing a copy of the schema enables the parser to use the local copy instead of reaching out to the Internet, and obtaining the latest version from the Pegasus website dynamically.

Database Drivers For All Relational Catalogs

Table 13.15. Database Driver Properties

Property Key	Description
Property Key: pegasus.catalog.*.db.driver **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Type** : Enumeration **Values** : MySQL|PostGres|SQLite **Default** : (no default)	The database driver class is dynamically loaded, as required by the schema. Currently, only MySQL 5.x, PostGreSQL >= 8.1 and SQlite are supported. Their respective JDBC3 driver is provided as part and parcel of the PEGASUS. The * in the property name can be replaced by a catalog name to apply the property only for that catalog. Valid catalog names are `replica`
Property Key: pegasus.catalog.*.db.url **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Type** : Database URL **Default** : (no default)	Each database has its own string to contact the database on a given host, port, and database. Although most driver URLs allow to pass arbitrary arguments, please use the pegasus.catalog.[catalog-name].db.* keys or pegasus.catalog.*.db.* to preload these arguments. THE URL IS A MANDATORY PROPERTY FOR ANY DBMS BACKEND.
Property Key: pegasus.catalog.*.db.user **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Type** : String **Default** :	In order to access a database, you must provide the name of your account on the DBMS. This property is database-independent. THIS IS A MANDATORY PROPERTY FOR MANY DBMS BACKENDS. The * in the property name can be replaced by a catalog name to apply the property only for that catalog. Valid catalog names are `replica`
Property Key: pegasus.catalog.*.db.password **Profile Key:** N/A **Scope** : Properties **Since** : 2.0	In order to access a database, you must provide an optional password of your account on the DBMS. This property is database-independent. THIS IS A MANDATORY

Type : String **Default** : (no default)	PROPERTY, IF YOUR DBMS BACKEND ACCOUNT REQUIRES A PASSWORD. The * in the property name can be replaced by a catalog name to apply the property only for that catalog. Valid catalog names are `replica`

Property Key: pegasus.catalog.*.db.*
Profile Key: N/A
Scope : Properties
Since : 2.0
Type : String
Default : (no default)
	Each database has a multitude of options to control in fine detail the further behaviour. You may want to check the JDBC3 documentation of the JDBC driver for your database for details. The keys will be passed as part of the connect properties by stripping the "pegasus.catalog.[catalog-name].db." prefix from them. The catalog-name can be replaced by the following values provenance for Provenance Catalog (PTC), replica for Replica Catalog (RC)
	
	Postgres >= 8.1 parses the following properties:
	
	`pegasus.catalog.*.db.user`
	`pegasus.catalog.*.db.password`
	`pegasus.catalog.*.db.PGHOST`
	`pegasus.catalog.*.db.PGPORT`
	`pegasus.catalog.*.db.charSet`
	`pegasus.catalog.*.db.compatible`
	
	MySQL 5.0 parses the following properties:
	
	`pegasus.catalog.*.db.user`
	`pegasus.catalog.*.db.password`
	`pegasus.catalog.*.db.databaseName`
	`pegasus.catalog.*.db.serverName`
	`pegasus.catalog.*.db.portNumber`
	`pegasus.catalog.*.db.socketFactory`
	`pegasus.catalog.*.db.strictUpdates`
	`pegasus.catalog.*.db.ignoreNonTxTables`
	`pegasus.catalog.*.db.secondsBeforeRetryMaster`
	`pegasus.catalog.*.db.queriesBeforeRetryMaster`
	`pegasus.catalog.*.db.allowLoadLocalInfile`
	`pegasus.catalog.*.db.continueBatchOnError`
	`pegasus.catalog.*.db.pedantic`
	`pegasus.catalog.*.db.useStreamLengthsInPrepStmts`
	`pegasus.catalog.*.db.useTimezone`
	`pegasus.catalog.*.db.relaxAutoCommit`
	`pegasus.catalog.*.db.paranoid`
	`pegasus.catalog.*.db.autoReconnect`
	`pegasus.catalog.*.db.capitalizeTypeNames`
	`pegasus.catalog.*.db.ultraDevHack`
	`pegasus.catalog.*.db.strictFloatingPoint`
	`pegasus.catalog.*.db.useSSL`
	`pegasus.catalog.*.db.useCompression`
	`pegasus.catalog.*.db.socketTimeout`
	`pegasus.catalog.*.db.maxReconnects`
	`pegasus.catalog.*.db.initialTimeout`
	`pegasus.catalog.*.db.maxRows`
	`pegasus.catalog.*.db.useHostsInPrivileges`
	`pegasus.catalog.*.db.interactiveClient`
	`pegasus.catalog.*.db.useUnicode`
	`pegasus.catalog.*.db.characterEncoding`
	
	MS SQL Server 2000 support the following properties (keys are case-insensitive, e.g. both "user" and "User" are valid):
	
	`pegasus.catalog.*.db.User`
	`pegasus.catalog.*.db.Password`
	`pegasus.catalog.*.db.DatabaseName`

```
pegasus.catalog.*.db.ServerName
pegasus.catalog.*.db.HostProcess
pegasus.catalog.*.db.NetAddress
pegasus.catalog.*.db.PortNumber
pegasus.catalog.*.db.ProgramName
pegasus.catalog.*.db.SendStringParametersAsUnicode
pegasus.catalog.*.db.SelectMethod
```

	The * in the property name can be replaced by a catalog name to apply the property only for that catalog. Valid catalog names are `replica`

Property Key: pegasus.catalog.*.timeout
  Profile Key: N/A
  Scope   : Properties
  Since   : 4.5.1
  Type   : Integer
  Default   : (no default)
	This property sets a busy handler that sleeps for a specified amount of time (in seconds) when a table is locked. This property has effect only in a sqlite database.
	
	The * in the property name can be replaced by a catalog name to apply the property only for that catalog. Valid catalog names are
	
	`master`
	`workflow`

Catalog Related Properties

Table 13.16. Replica Catalog Properties

Key Attributes	Description
Property Key: pegasus.catalog.replica **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Default** : File	Pegasus queries a Replica Catalog to discover the physical filenames (PFN) for input files specified in the DAX. Pegasus can interface with various types of Replica Catalogs. This property specifies which type of Replica Catalog to use during the planning process.
	JDBCRC In this mode, Pegasus queries a SQL based replica catalog that is accessed via JDBC. To use JDBCRC, the user additionally needs to set the following properties
	1. pegasus.catalog.replica.db.driver = mysql | postgres |sqlite
	2. pegasus.catalog.replica.db.url = <jdbc url to the database> e.g jdbc:mysql://database-host.isi.edu/database-name | jdbc:sqlite:/shared/jdbcrc.db
	3. pegasus.catalog.replica.db.user = database-user
	4. pegasus.catalog.replica.db.password = database-password
	File In this mode, Pegasus queries a file based replica catalog. It is neither transactionally safe, nor advised to use for production purposes in any way. Multiple concurrent instances *will clobber* each other!. The site attribute should be specified

whenever possible. The attribute key for the site attribute is "site".

The LFN may or may not be quoted. If it contains linear whitespace, quotes, backslash or an equality sign, it must be quoted and escaped. Ditto for the PFN. The attribute key-value pairs are separated by an equality sign without any whitespaces. The value may be in quoted. The LFN sentiments about quoting apply.

```
LFN PFN
LFN PFN a=b [..]
LFN PFN a="b" [..]
"LFN w/LWS" "PFN w/LWS" [..]
```

To use File, the user additionally needs to specify **pegasus.catalog.replica.file** property to specify the path to the file based RC. IF not specified , defaults to $PWD/rc.txt file.

	Regex	In this mode, Pegasus queries a file based replica catalog. It is neither transactionally safe, nor advised to use for production purposes in any way. Multiple concurrent access to the File will end up clobbering the contents of the file. The site attribute should be specified whenever possible. The attribute key for the site attribute is "site".

The LFN may or may not be quoted. If it contains linear whitespace, quotes, backslash or an equality sign, it must be quoted and escaped. Ditto for the PFN. The attribute key-value pairs are separated by an equality sign without any whitespaces. The value may be in quoted. The LFN sentiments about quoting apply.

In addition users can specifiy regular expression based LFN's. A regular expression based entry should be qualified with an attribute named 'regex'. The attribute regex when set to true identifies the catalog entry as a regular expression based entry. Regular expressions should follow Java regular expression syntax.

For example, consider a replica catalog as shown below.

Entry 1 refers to an entry which does not use a resular expressions. This entry would only match a file named 'f.a', and nothing else. Entry 2 referes to an entry which uses a regular expression. In this entry f.a referes to files having name as f[any-character]a i.e. faa, f.a, f0a, etc.

```
f.a file:///Vol/input/f.a
 site="local"
f.a file:///Vol/input/f.a
 site="local" regex="true"
```

Regular expression based entries also support substitutions. For example, consider the regular expression based entry shown below.

Entry 3 will match files with name alpha.csv, alpha.txt, alpha.xml. In addition, values matched in the expression can be used to generate a PFN.

For the entry below if the file being looked up is alpha.csv, the PFN for the file would be generated as file:/// Volumes/data/input/csv/alpha.csv. Similary if the file being lookedup was alpha.csv, the PFN for the file would be generated as file:///Volumes/data/input/xml/alpha.xml i.e. The section [0], [1] will be replaced. Section [0] refers to the entire string i.e. alpha.csv. Section [1] refers to a partial match in the input i.e. csv, or txt, or xml. Users can utilize as many sections as they wish.

```
alpha\.(csv|txt|xml) file:///
Vol/input/[1]/[0] site="local"
 regex="true"
```

To use File, the user additionally needs to specify pegasus.catalog.replica.file property to specify the path to the file based RC.

Directory	In this mode, Pegasus does a directory listing on an input directory to create the LFN to PFN mappings. The directory listing is performed recursively, resulting in deep LFN mappings. For example, if an input directory $input is specified with the following structure

```
$input
$input/f.1
$input/f.2
$input/D1
$input/D1/f.3
```

Pegasus will create the mappings the following LFN PFN mappings internally

```
f.1 file://$input/f.1  site="local"
f.2 file://$input/f.2  site="local"
D1/f.3 file://$input/D2/f.3
 site="local"
```

If you don't want the deep lfn's to be created then, you can set pegasus.catalog.replica.directory.flat.lfn to true In that case, for the previous example, Pegasus will create the following LFN PFN mappings internally.

```
f.1 file://$input/f.1  site="local"
f.2 file://$input/f.2  site="local"
f.3 file://$input/D2/f.3
 site="local"
```

pegasus-plan has --input-dir option that can be used to specify an input directory.

Users can optionally specify additional properties to configure the behvavior of this implementation.

pegasus.catalog.replica.directory to specify the path to the directory containing the files

pegasus.catalog.replica.directory.site to specify a site attribute other than local to associate with the mappings.

pegasus.catalog.replica.directory.url.prefix to associate a URL prefix for the PFN's constructed. If not specified, the URL defaults to file://

MRC In this mode, Pegasus queries multiple replica catalogs to discover the file locations on the grid. To use it set

```
pegasus.catalog.replica MRC
```

Each associated replica catalog can be configured via properties as follows.

The user associates a variable name referred to as [value] for each of the catalogs, where [value] is any legal identifier (concretely [A-Za-z][_A-Za-z0-9]*) For each associated replica catalogs the user specifies the following properties.

```
pegasus.catalog.replica.mrc.[value]
      specifies the type of \

      replica catalog.
pegasus.catalog.replica.mrc.
[value].key   specifies a property
 name\

      key for a particular catalog


pegasus.catalog.replica.mrc.directory1
 Directory
pegasus.catalog.replica.mrc.directory1.directory /
input/dir1
```

	<pre>pegasus.catalog.replica.mrc.directory1.directory.site
siteX
pegasus.catalog.replica.mrc.directory2
Directory
pegasus.catalog.replica.mrc.directory2.directory /
input/dir2
pegasus.catalog.replica.mrc.directory1.directory.site
siteY</pre> In the above example, directory1, directory2 are any valid identifier names and url is the property key that needed to be specified.
Property Key: pegasus.catalog.replica.chunk.size **Profile Key:** N/A **Scope** **:** Properties **Since** **:** 2.0 **Default** **:** 1000	The pegasus-rc-client takes in an input file containing the mappings upon which to work. This property determines, the number of lines that are read in at a time, and worked upon at together. This allows the various operations like insert, delete happen in bulk if the underlying replica implementation supports it.
Property Key: pegasus.catalog.replica.cache.asrc **Profile Key :** N/A **Scope** **:** Properties **Since** **:** 2.0 **Default** **:** false	This Boolean property determines whether to treat the cache file specified as a supplemental replica catalog or not. User can specify on the command line to pegasus-plan a comma separated list of cache files using the --cache option. By default, the LFN->PFN mappings contained in the cache file are treated as cache, i.e if an entry is found in a cache file the replica catalog is not queried. This results in only the entry specified in the cache file to be available for replica selection. Setting this property to true, results in the cache files to be treated as supplemental replica catalogs. This results in the mappings found in the replica catalog (as specified by pegasus.catalog.replica) to be merged with the ones found in the cache files. Thus, mappings for a particular LFN found in both the cache and the replica catalog are available for replica selection.
Property Key: pegasus.catalog.replica.dax.asrc **Profile Key :** N/A **Scope** **:** Properties **Since** **:** 4.5.2 **Default** **:** false	This Boolean property determines whether to treat the locations of files recorded in the DAX as a supplemental replica catalog or not. By default, the LFN->PFN mappings contained in the DAX file overrides any specified in a replica catalog. This results in only the entry specified in the DAX file to be available for replica selection. Setting this property to true, results in the locations of files recorded in the DAX files to be treated as a supplemental replica catalog. This results in the mappings found in the replica catalog (as specified by pegasus.catalog.replica) to be merged with the ones found in the cache files. Thus, mappings for a particular LFN found in both the DAX and the replica catalog are available for replica selection.
Property Key: pegasus.catalog.replica.output**.*** **Profile Key :** N/A **Scope** **:** Properties **Since** **:** 4.5.3 **Default** **:** None	Normally, the registration jobs in the executable workflow register to the replica catalog specified by the user in the properties file . This property prefix allows the user to specify a separate output replica catalog that is different from the one used for discovery of input files. This is normally the case, when a Directory or MRC based replica catalog backend that don't support insertion of entries are used for discovery of input files. For example to specify a separate file based output replica catalog, specify <pre>pegasus.catalog.replica.output File
pegasus.catalog.replica.output.file /workflow/
output.rc</pre>

Table 13.17. Site Catalog Properties

Key Attributes	Description
Property Key: pegasus.catalog.site **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Default** : XML	Pegasus supports two different types of site catalogs in XML format conforming • sc-3.0.xsd http://pegasus.isi.edu/schema/sc-3.0.xsd • sc-4.0.xsd http://pegasus.isi.edu/schema/sc-4.0.xsd Pegasus is able to auto-detect what schema a user site catalog refers to. Hence, this property may no longer be set.
Property Key: pegasus.catalog.site.file **Profile Key :** N/A **Scope** : Properties **Since** : 2.0 **Default** : $PWD/sites.xml	The path to the site catalog file, that describes the various sites and their layouts to Pegasus.

Table 13.18. Transformation Catalog Properties

Key Attributes	Description

Property Key: pegasus.catalog.transformation
Profile Key: N/A
Scope : Properties
Since : 2.0
Default : Text
	The only recommended and supported version of Transformation Catalog for Pegasus is Text. For the old File based formats, users should use pegasus-tc-converter to convert File format to Text Format.
	
	Text In this mode, a multiline file based format is understood. The file is read and cached in memory. Any modifications, as adding or deleting, causes an update of the memory and hence to the file underneath. All queries are done against the memory representation.
	
	The file sample.tc.text in the etc directory contains an example
	
	Here is a sample textual format for transfomation catalog containing one transformation on two sites
	
	<pre>tr example::keg:1.0 {
	#specify profiles that apply for all the
	 sites for the transformation
	#in each site entry the profile can be
	 overriden
	profile env "APP_HOME" "/tmp/karan"
	profile env "JAVA_HOME" "/bin/app"
	site isi {
	profile env "me" "with"
	profile condor "more" "test"
	profile env "JAVA_HOME" "/bin/java.1.6"
	pfn "/path/to/keg"
	arch "x86"
	os "linux"
	osrelease "fc"
	osversion "4"
	type "INSTALLED"
	site wind {
	profile env "me" "with"
	profile condor "more" "test"
	pfn "/path/to/keg"
	arch "x86"
	os "linux"
	osrelease "fc"</pre>

```
                                            osversion "4"
                                            type "STAGEABLE"
```

Property Key: pegasus.catalog.transformation **Profile Key :** N/A **Scope** : Properties **Since** : 2.0 **Default** : $PWD/tc.txt	The path to the transformation catalog file, that describes the locations of the executables.

Replica Selection Properties

Table 13.19. Replica Selection Properties

Key Attributes	Description
Property Key: pegasus.selector.replica **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Type** : String **Default** : Default **See Also** : pegasus.selector.replica.*.ignore.stagein.sites **See Also** : pegasus.selector.replica.*.prefer.stagein.sites	Each job in the DAX maybe associated with input LFN's denoting the files that are required for the job to run. To determine the physical replica (PFN) for a LFN, Pegasus queries the replica catalog to get all the PFN's (replicas) associated with a LFN. Pegasus then calls out to a replica selector to select a replica amongst the various replicas returned. This property determines the replica selector to use for selecting the replicas.

<table>
<tr><td></td><td>Default</td><td>The selector orders the various candidate replica's according to the following rules

1. valid file URL's . That is URL's that have the site attribute matching the site where the executable *pegasus-transfer* is executed.

2. all URL's from preferred site (usually the compute site)

3. all other remotely accessible (non file) URL's</td></tr>
<tr><td></td><td>Regex</td><td>This replica selector allows the user allows the user to specific regular expressions that can be used to rank various PFN's returned from the Replica Catalog for a particular LFN. This replica selector orders the replicas based on the rank. Lower the rank higher the preference.

The regular expressions are assigned different rank, that determine the order in which the expressions are employed. The rank values for the regex can expressed in user properties using the property.

<code>pegasus.selector.replica.regex.rank.
[value]   regex-expression</code>

The value is an integer value that denotes the rank of an expression with a rank value of 1 being the highest rank.</td></tr>
</table>

	Please note that before applying any regular expressions on the PFN's, the file URL's that dont match the preferred site are explicitly filtered out.
Restricted	This replica selector, allows the user to specify good sites and bad sites for staging in data to a particular compute site. A good site for a compute site X, is a preferred site from which replicas should be staged to site X. If there are more than one good sites having a particular replica, then a random site is selected amongst these preferred sites.
	A bad site for a compute site X, is a site from which replica's should not be staged. The reason of not accessing replica from a bad site can vary from the link being down, to the user not having permissions on that site's data.
	The good | bad sites are specified by the properties
	`pegasus.replica.*.prefer.stagein.sites` `pegasus.replica.*.ignore.stagein.sites`
	where the * in the property name denotes the name of the compute site. A * in the property key is taken to mean all sites.
	The pegasus.replica.*.prefer.stagein.sites property takes precedence over pegasus.replica.*.ignore.stagein.sites property i.e. if for a site X, a site Y is specified both in the ignored and the preferred set, then site Y is taken to mean as only a preferred site for a site X.
Local	This replica selector prefers replicas from the local host and that start with a file: URL scheme. It is useful, when users want to stagin files to a remote site from your submit host using the Condor file transfer mechanism.

Property Key: pegasus.selector.replica.*.ignore.stagein.sites **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Default** : (no default) **See Also** : pegasus.selector.replica **See Also** : pegasus.selector.replica.*.prefer.stagein.sites	A comma separated list of storage sites from which to never stage in data to a compute site. The property can apply to all or a single compute site, depending on how the * in the property name is expanded. The * in the property name means all compute sites unless replaced by a site name. For e.g setting pegasus.selector.replica.*.ignore.stagein.sites to usc means that ignore all replicas from site usc for staging in to any compute site. Setting

	pegasus.replica.isi.ignore.stagein.sites to usc means that ignore all replicas from site usc for staging in data to site isi.
Property Key: pegasus.selector.replica.*.prefer.stagein.sites **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Default** : (no default) **See Also** : pegasus.selector.replica **See Also** : pegasus.selector.replica.*.ignore.stagein.sites	A comma separated list of preferred storage sites from which to stage in data to a compute site. The property can apply to all or a single compute site, depending on how the * in the property name is expanded. The * in the property name means all compute sites unless replaced by a site name. For e.g setting pegasus.selector.replica.*.prefer.stagein.sites to usc means that prefer all replicas from site usc for staging in to any compute site. Setting pegasus.replica.isi.prefer.stagein.sites to usc means that prefer all replicas from site usc for staging in data to site isi.

Property Key: pegasus.selector.replica.regex.rank.[value]
Profile Key: N/A
Scope   : Properties
Since   : 2.3.0
Default   : (no default)
See Also   : pegasus.selector.replica
	Specifies the regex expressions to be applied on the PFNs returned for a particular LFN. Refer to
	
	`http://java.sun.com/javase/6/docs/api/java/util/`
	`regex/Pattern.html`
	
	on information of how to construct a regex expression.
	
	The [value] in the property key is to be replaced by an int value that designates the rank value for the regex expression to be applied in the Regex replica selector.
	
	The example below indicates preference for file URL's over URL's referring to gridftp server at example.isi.edu
	
	`pegasus.selector.replica.regex.rank.1 file://.*`
	`pegasus.selector.replica.regex.rank.2 gsiftp://`
	`example\.isi\.edu.*`

Site Selection Properties

Table 13.20. Site Selection Properties

Key Attributes	Description
Property Key: pegasus.selector.site **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Type** : String **Default** : Random **See Also** : pegasus.selector.site.path **See Also** : pegasus.selector.site.timeout **See Also** : pegasus.selector.site.keep.tmp **See Also** : pegasus.selector.site.env.*	The site selection in Pegasus can be on basis of any of the following strategies. Random In this mode, the jobs will be randomly distributed among the sites that can execute them. RoundRobin In this mode. the jobs will be assigned in a round robin manner amongst the sites that can execute them. Since each site cannot execute everytype of job, the round robin scheduling is done per level on a sorted list. The sorting is on the basis of the number of jobs a particular site has been assigned in that level so far. If a job cannot be run on the first site in the queue (due to no matching entry

| | | in the transformation catalog for the transformation referred to by the job), it goes to the next one and so on. This implementation defaults to classic round robin in the case where all the jobs in the workflow can run on all the sites. |
| | NonJavaCallout | In this mode, Pegasus will call-out to an external site selector.In this mode a temporary file is prepared containing the job information that is passed to the site selector as an argument while invoking it. The path to the site selector is specified by setting the property pegasus.site.selector.path. The environment variables that need to be set to run the site selector can be specified using the properties with a pegasus.site.selector.env. prefix. The temporary file contains information about the job that needs to be scheduled. It contains key value pairs with each key value pair being on a new line and separated by a =.

The following pairs are currently generated for the site selector temporary file that is generated in the NonJavaCallout. |

version	is the version of the site selector api,currently 2.0.
transformation	is the fully-qualified definition identifier for the transformation (TR) namespace::name:version.
derivation	is teh fully qualified definition identifier for the derivation (DV), namespace::name:version.
job.level	is the job's depth in the tree of the workflow DAG.

	job.id	is the job's ID, as used in the DAX file.
	resource.id	is a site handle, followed by whitespace, followed by a gridftp server. Typically, each gridftp server is enumerated once, so you may have multiple occurances of the same site. There can be multiple occurances of this key.
	input.lfn	is an input LFN, optionally followed by a whitespace and file size. There can be multiple occurances of this key,one for each input LFN required by the job.
	wf.name	label of the dax, as found in the DAX's root element. wf.index is the DAX index, that is incremented for each partition in case of deferred planning.
	wf.time	is the mtime of the workflow.
	wf.manager	is the name of the workflow manager being used .e.g condor
	vo.name	is the name of the virtual organization that is running this workflow. It is

		currently set to NONE
	vo.group	unused at present and is set to NONE.
	Group	In this mode, a group of jobs will be assigned to the same site that can execute them. The use of the PEGASUS profile key group in the dax, associates a job with a particular group. The jobs that do not have the profile key associated with them, will be put in the default group. The jobs in the default group are handed over to the "Random" Site Selector for scheduling.
	Heft	In this mode, a version of the HEFT processor scheduling algorithm is used to schedule jobs in the workflow to multiple grid sites. The implementation assumes default data communication costs when jobs are not scheduled on to the same site. Later on this may be made more configurable.
		The runtime for the jobs is specified in the transformation catalog by associating the pegasus profile key runtime with the entries.
		The number of processors in a site is picked up from the attribute idle-nodes associated with the vanilla jobmanager of the site in the site catalog.

Property Key: pegasus.selector.site.path **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Default** : (no default)	If one calls out to an external site selector using the Non-JavaCallout mode, this refers to the path where the site selector is installed. In case other strategies are used it does not need to be set.
Property Key: pegasus.selector.site.env.* **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Default** : (no default)	The environment variables that need to be set while callout to the site selector. These are the variables that the user would set if running the site selector on the command line. The name of the environment variable is got by stripping the keys of the prefix "pegasus.site.selector.env." prefix from them. The value of the environment variable is the value of the property. e.g pegasus.site.selector.path.LD_LIBRARY_PATH / globus/lib would lead to the site selector being called with the LD_LIBRARY_PATH set to /globus/lib.

Key Attributes	Description
Property Key: pegasus.selector.site.timeout **Profile Key:** N/A **Scope** : Properties **Since** : 2.3.0 **Default** : 60 **See Also** : pegasus.selector.site	It sets the number of seconds Pegasus waits to hear back from an external site selector using the NonJavaCallout interface before timing out.
Property Key: pegasus.selector.site.keep.tmp **Profile Key:** N/A **Scope** : Properties **Since** : 2.3.0 **Values** : onerror|always|never **Default** : onerror **See Also** : pegasus.selector.site	It determines whether Pegasus deletes the temporary input files that are generated in the temp directory or not. These temporary input files are passed as input to the external site selectors. A temporary input file is created for each that needs to be scheduled.

Data Staging Configuration Properties

Table 13.21. Data Configuration Properties

Key Attributes	Description
Property Key: pegasus.data.configuration **Profile Key:** data.configuration **Scope** : Properties, Site Catalog **Since** : 4.0.0 **Values** : sharedfs|nonsharedfs|condorio **Default** : sharedfs **See Also** : pegasus.transfer.bypass.input.staging	This property sets up Pegasus to run in different environments. For Pegasus 4.5.0 and above, users can set the pegasus profile data.configuration with the sites in their site catalog, to run multisite workflows with each site having a different data configuration. sharedfs If this is set, Pegasus will be setup to execute jobs on the shared filesystem on the execution site. This assumes, that the head node of a cluster and the worker nodes share a filesystem. The staging site in this case is the same as the execution site. Pegasus adds a create dir job to the executable workflow that creates a workflow specific directory on the shared filesystem . The data transfer jobs in the executable workflow (stage_in_ , stage_inter_ , stage_out_) transfer the data to this directory.The compute jobs in the executable workflow are launched in the directory on the shared filesystem. condorio If this is set, Pegasus will be setup to run jobs in a pure condor pool, with the nodes not sharing a filesystem. Data is staged to the compute nodes from the submit host using Condor File IO. The planner is automatically setup to use the submit host (site local) as the staging site. All the auxillary jobs added by the planner to the executable workflow (create dir, data stagein and stage-out, cleanup) jobs refer to the workflow specific directory on the local site. The data transfer jobs in the executable workflow (stage_in_ , stage_inter_ , stage_out_) transfer the data to this directory. When the compute jobs start, the input data for each job is shipped

| | | from the workflow specific directory on the submit host to compute/worker node using Condor file IO. The output data for each job is similarly shipped back to the submit host from the compute/worker node. This setup is particularly helpful when running workflows in the cloud environment where setting up a shared filesystem across the VM's may be tricky.

`pegasus.gridstart`
`    PegasusLite`
`pegasus.transfer.worker.package`
`    true` |
| | nonsharedfs | If this is set, Pegasus will be setup to execute jobs on an execution site without relying on a shared filesystem between the head node and the worker nodes. You can specify staging site (using --staging-site option to pegasus-plan) to indicate the site to use as a central storage location for a workflow. The staging site is independant of the execution sites on which a workflow executes. All the auxillary jobs added by the planner to the executable workflow (create dir, data stagein and stage-out, cleanup) jobs refer to the workflow specific directory on the staging site. The data transfer jobs in the executable workflow (stage_in_ , stage_inter_ , stage_out_) transfer the data to this directory. When the compute jobs start, the input data for each job is shipped from the workflow specific directory on the submit host to compute/worker node using pegasus-transfer. The output data for each job is similarly shipped back to the submit host from the compute/worker node. The protocols supported are at this time SRM, GridFTP, iRods, S3. This setup is particularly helpful when running workflows on OSG where most of the execution sites don't have enough data storage. Only a few sites have large amounts of data storage exposed that can be used to place data during a workflow run. This setup is also helpful when running workflows in the cloud environment where setting up a shared filesystem across the VM's may be tricky. On loading this property, internally the following properies are set

`pegasus.gridstart`
`    PegasusLite`
`pegasus.transfer.worker.package`
`    true` |

| Property Key: pegasus.transfer.bypass.input.staging
Profile Key: N/A
Scope : Properties
Since : 4.3.0
Type : Boolean
Default : false
See Also : pegasus.data.configuration | When executiing in a non shared filesystem setup i.e data configuration set to nonsharedfs or condorio, Pegasus always stages the input files through the staging site i.e the stage-in job stages in data from the input site to the staging site. The PegasusLite jobs that start up on the worker nodes, then pull the input data from the staging site for each job.

This property can be used to setup the PegasusLite jobs to pull input data directly from the input site without going through the staging server. This is based on the assumption that the worker nodes can access the input site. If users set this to true, they should be aware that the access to the input site is no longer throttled (as in case of stage in jobs). If large number of compute jobs start at the same time in a workflow, the input server will see a connection from each job. |

Transfer Configuration Properties

Table 13.22. Transfer Configuration Properties

Key Attributes	Description

Property Key: pegasus.transfer.*.impl
Profile Key: N/A
Scope : Properties
Since : 2.0.0
Values : Transfer|GUC
Default : Transfer
See Also : pegasus.transfer.refiner
	Each compute job usually has data products that are required to be staged in to the execution site, materialized data products staged out to a final resting place, or staged to another job running at a different site. This property determines the underlying grid transfer tool that is used to manage the transfers.
	
	The * in the property name can be replaced to achieve finer grained control to dictate what type of transfer jobs need to be managed with which grid transfer tool.
	
	Usually,the arguments with which the client is invoked can be specified by
	
	`- the property pegasus.transfer.arguments`
	`- associating the PEGASUS profile key`
	`  transfer.arguments`
	
	The table below illustrates all the possible variations of the property.

Property Name	Applies to
pegasus.transfer.stagein.impl	the stage in transfer jobs
pegasus.transfer.stageout.impl	the stage out transfer jobs
pegasus.transfer.inter.impl	the inter site transfer jobs
pegasus.transfer.setup.impl	the setup transfer job
pegasus.transfer.*.impl	apply to types of transfer jobs

Note: Since version 2.2.0 the worker package is staged automatically during staging of executables to the remote site. This is achieved by adding a setup transfer job to the workflow. The setup transfer job by default uses GUC to stage the data. The implementation to use can be configured by setting the property

```
pegasus.transfer.setup.impl
```

property. However, if you have pegasus.transfer.*.impl set in your properties file, then you need to set pegasus.transfer.setup.impl to GUC

The various grid transfer tools that can be used to manage data transfers are explained below

Transfer	This results in pegasus-transfer to be used for transferring of files. It is a python based wrapper around various transfer clients like globus-url-copy, lcg-copy, wget, cp, ln . pegasus-transfer looks at source and destination url and figures out automatically which underlying client to use. pegasus-transfer is distributed with the PEGASUS and can be found at $PEGASUS_HOME/bin/pegasus-transfer.
	For remote sites, Pegasus constructs the default path to pegasus-transfer on the basis of PEGASUS_HOME env profile specified in the site catalog. To specify a different path to the pegasus-transfer client , users can add an entry into the transformation catalog with fully qualified logical name as pegasus::pegasus-transfer
GUC	This refers to the new guc client that does multiple file transfers per invocation. The globus-url-copy client distributed with Globus 4.x is compatible with this mode.

Property Key: pegasus.transfer.arguments **Profile Key:** transfer.arguments **Scope** : Properties **Since** : 2.0.0 **Type** : String **Default** : (no default) **See Also** : pegasus.transfer.lite.arguments	This determines the extra arguments with which the transfer implementation is invoked. The transfer executable that is invoked is dependant upon the transfer mode that has been selected. The property can be overloaded by associated the pegasus profile key transfer.arguments either with the site in the site catalog or the corresponding transfer executable in the transformation catalog.
Property Key: pegasus.transfer.threads **Profile Key:** transfer.threads **Scope** : Properties **Since** : 4.4.0 **Type** : Integer **Default** : 2	This property set the number of threads pegasus-transfer uses to transfer the files. This property to applies to the separate data transfer nodes that are added by Pegasus to the executable workflow. The property can be overloaded by associated the pegasus profile key transfer.threads either with the site in the site catalog or the corresponding transfer executable in the transformation catalog.
Property Key: pegasus.transfer.lite.arguments **Profile Key:** transfer.lite.arguments **Scope** : Properties **Since** : 4.4.0	This determines the extra arguments with which the PegasusLite transfer implementation is invoked. The transfer executable that is invoked is dependant upon the PegasusLite transfer implementation that has been selected.

Type : String **Default** : (no default) **See Also** : pegasus.transfer.arguments	
Property Key: pegasus.transfer.worker.package **Profile Key:** N/A **Scope** : Properties **Since** : 2.0.0 **Type** : Boolean **Default** : false **See Also** : pegasus.data.configuration	By default, Pegasus relies on the worker package to be installed in a directory accessible to the worker nodes on the remote sites . Pegasus uses the value of PEGASUS_HOME environment profile in the site catalog for the remote sites, to then construct paths to pegasus auxillary executables like kickstart, pegasus-transfer, seqexec etc. If the Pegasus worker package is not installed on the remote sites users can set this property to true to get Pegasus to deploy worker package on the nodes. In the case of sharedfs setup, the worker package is deployed on the shared scratch directory for the workflow , that is accessible to all the compute nodes of the remote sites. When running in nonsharefs environments, the worker package is first brought to the submit directory and then transferred to the worker node filesystem using Condor file IO.
Property Key: pegasus.transfer.worker.package.autodownload **Profile Key:** N/A **Scope** : Properties **Since** : 4.6.1 **Type** : Boolean **Default** : true **See Also** : pegasus.transfer.worker.package	If PegasusLite does not find a worker package install matching the pegasus lite job on the worker node, it automatically downloads the correct worker package from the Pegasus website. However, this can mask user errors in configuration. This property can be set to false to disable auto downloads.
Property Key: pegasus.transfer.worker.package.strict **Profile Key:** N/A **Scope** : Properties **Since** : 4.6.1 **Type** : Boolean **Default** : true **See Also** : pegasus.transfer.worker.package	In PegasusLite mode, the pegasus worker package for the jobs is shipped along with the jobs. This property controls whether PegasusLite will do a strict match against the architecture and os on the local worker node, along with pegasus version. If the strict match fails, then PegasusLite will revert to the pegasus website to download the correct worker package.
Property Key: pegasus.transfer.links **Profile Key:** N/A **Scope** : Properties **Since** : 2.0.0 **Type** : Boolean **Default** : false	If this is set, and the transfer implementation is set to Transfer i.e. using the transfer executable distributed with the PEGASUS. On setting this property, if Pegasus while fetching data from the Replica Catalog sees a "site" attribute associated with the PFN that matches the execution site on which the data has to be transferred to, Pegasus instead of the URL returned by the Replica Catalog replaces it with a file based URL. This is based on the assumption that the if the "site" attributes match, the filesystems are visible to the remote execution directory where input data resides. On seeing both the source and destination urls as file based URLs the transfer executable spawns a job that creates a symbolic link by calling ln -s on the remote site.
Property Key: pegasus.transfer.*.remote.sites **Profile Key:** N/A **Scope** : Properties **Since** : 2.0.0 **Type** : comma separated list of sites **Default** : (no default)	By default Pegasus looks at the source and destination URL's for to determine whether the associated transfer job runs on the submit host or the head node of a remote site, with preference set to run a transfer job to run on submit host.

Pegasus will run transfer jobs on the remote sites

```
-  if the file server for the compute site is a
 file server i.e url prefix file://
-  symlink jobs need to be added that require
 the symlink transfer jobs to
be run remotely.
```

This property can be used to change the default behaviour of Pegasus and force pegasus to run different types of transfer jobs for the sites specified on the remote site.

The table below illustrates all the possible variations of the property.

Property Name	Applies to
pegasus.transfer.stagein.remote.sites	the stage in transfer jobs
pegasus.transfer.stageout.remote.sites	the stage out transfer jobs
pegasus.transfer.inter.remote.sites	the inter site transfer jobs
pegasus.transfer.*.remote.sites	apply to types of transfer jobs

In addition * can be specified as a property value, to designate that it applies to all sites.

Property Key: pegasus.transfer.staging.delimiter
Profile Key: N/A
Scope   : Properties
Since   : 2.0.0
Type   : String
Default   : :

Pegasus supports executable staging as part of the workflow. Currently staging of statically linked executables is supported only. An executable is normally staged to the work directory for the workflow/partition on the remote site. The basename of the staged executable is derived from the namespace,name and version of the transformation in the transformation catalog. This property sets the delimiter that is used for the construction of the name of the staged executable.

Property Key: pegasus.transfer.disable.chmod.sites
Profile Key: N/A
Scope   : Properties
Since   : 2.0.0
Type   : comma separated list of sites
Default   : (no default)

During staging of executables to remote sites, chmod jobs are added to the workflow. These jobs run on the remote sites and do a chmod on the staged executable. For some sites, this maynot be required. The permissions might be preserved, or there maybe an automatic mechanism that does it.

This property allows you to specify the list of sites, where you do not want the chmod jobs to be executed. For those sites, the chmod jobs are replaced by NoOP jobs. The NoOP jobs are executed by Condor, and instead will immediately have a terminate event written to the job log file and removed from the queue.

Property Key: pegasus.transfer.setup.source.base.url
Profile Key: N/A
Scope   : Properties
Since   : 2.0.0
Type   : URL
Default   : (no default)

This property specifies the base URL to the directory containing the Pegasus worker package builds. During Staging of Executable, the Pegasus Worker Package is also staged to the remote site. The worker packages are by default pulled from the http server at pegasus.isi.edu. This property can be used to override the location from where the worker package are staged. This maybe required if the

	remote computes sites don't allows files transfers from a http server.

Monitoring Properties

Table 13.23. Monitoring Properties

Key Attributes	Description
Property Key: pegasus.monitord.events **Profile Key:** N/A **Scope** : Properties **Since** : 3.0.2 **Type** : String **Default** : true **See Also** : pegasus.catalog.workflow.url	This property determines whether pegasus-monitord generates log events. If log events are disabled using this property, no bp file, or database will be created, even if the pegasus.monitord.output property is specified.

Property Key: pegasus.catalog.workflow.url
Profile Key: N/A
Scope   : Properties
Since   : 4.5
Type   : String
Default   : SQlite database in submit
              directory.
See Also : pegasus.monitord.events
	This property specifies the destination for generated log events in pegasus-monitord. By default, events are stored in a sqlite database in the workflow directory, which will be created with the workflow's name, and a ".stampede.db" extension. Users can specify an alternative database by using a SQLAlchemy connection string. Details are available at:
	
	`http://www.sqlalchemy.org/docs/05/reference/`
	`dialects/index.html`
	
	It is important to note that users will need to have the appropriate db interface library installed. Which is to say, SQLAlchemy is a wrapper around the mysql interface library (for instance), it does not provide a MySQL driver itself. The Pegasus distribution includes both SQLAlchemy and the SQLite Python driver. As a final note, it is important to mention that unlike when using SQLite databases, using SQLAlchemy with other database servers, e.g. MySQL or Postgres , the target database needs to exist. Users can also specify a file name using this property in order to create a file with the log events.
	
	Example values for the SQLAlchemy connection string for various end points are listed below
	
	<table><tr><td>SQL Alchemy End Point</td><td>Example Value</td></tr><tr><td>Netlogger BP File</td><td>file:///submit/dir/myworkflow.bp</td></tr><tr><td>SQL Lite Database</td><td>sqlite:///submit/dir/myworkflow.db</td></tr><tr><td>MySQL Database</td><td>mysql://user:password@host:port/databasename</td></tr></table>

Property Key: pegasus.catalog.master.url **Profile Key:** N/A **Scope** : Properties **Since** : 4.2 **Type** : String **Default** : sqlite database in $HOME/.pegasus/workflow.db **See Also** : pegasus.catalog.workflow.url	This property specifies the destination for the workflow dashboard database. By default, the workflow dashboard datbase defaults to a sqlite database named workflow.db in the $HOME/.pegasus directory. This is database is shared for all workflows run as a particular user. Users can specify an alternative database by using a SQLAlchemy connection string. Details are available at:

```
http://www.sqlalchemy.org/docs/05/reference/
dialects/index.html
```

It is important to note that users will need to have the appropriate db interface library installed. Which is to say, SQLAlchemy is a wrapper around the mysql interface library (for instance), it does not provide a MySQL driver itself. The Pegasus distribution includes both SQLAlchemy and the SQLite Python driver. As a final note, it is important to mention that unlike when using SQLite databases, using SQLAlchemy with other database servers, e.g. MySQL or Postgres , the target database needs to exist. Users can also specify a file name using this property in order to create a file with the log events.

Example values for the SQLAlchemy connection string for various end points are listed below

SQL Alchemy End Point	Example Value
SQL Lite Database	sqlite:///shared/myworkflow.db
MySQL Database	mysql://user:password@host:port/database-name

Property Key: pegasus.monitord.output **Profile Key:** N/A **Scope** : Properties **Since** **:** 3.0.2 **Type** : String **Default** : SQlite database in submit directory. **See Also** : pegasus.monitord.events	This property has been deprecated in favor of pegasus.catalog.workflow.url that introduced in 4.5 release. Support for this property will be dropped in future releases.
Property Key: pegasus.dashboard.output **Profile Key:** N/A **Scope** : Properties **Since** **:** 4.2 **Type** : String **Default** : sqlite database in $HOME/.pegasus/workflow.db **See Also** : pegasus.monitord.output	This property has been deprecated in favor of pegasus.catalog.master.url that introduced in 4.5 release. Support for this property will be dropped in future releases.
Property Key: pegasus.monitord.notifications **Profile Key:** N/A **Scope** : Properties **Since** **:** 3.1.0 **Type** : Boolean **Default** : true **See Also** : pegasus.monitord.notifications.max **See Also** : pegasus.monitord.notifications.timeout	This property determines how many notification scripts pegasus-monitord will call concurrently. Upon reaching this limit, pegasus-monitord will wait for one notification script to finish before issuing another one. This is a way to keep the number of processes under control at the submit host. Setting this property to 0 will disable notifications completely.
Property Key: pegasus.monitord.notifications.max **Profile Key:** N/A **Scope** : Properties **Since** **:** 3.1.0 **Type** : Integer **Default** : 10 **See Also** : pegasus.monitord.notifications **See Also** : pegasus.monitord.notifications.timeout	This property determines whether pegasus-monitord processes notifications. When notifications are enabled, pegasus-monitord will parse the .notify file generated by pegasus-plan and will invoke notification scripts whenever conditions matches one of the notifications.

Property Key: pegasus.monitord.notifications.timeout **Profile Key:** N/A **Scope** : Properties **Since** : 3.1.0 **Type** : Integer **Default** : true **See Also** : pegasus.monitord.notifications. **See Also** : pegasus.monitord.notifications.max	This property determines how long will pegasus-monitord let notification scripts run before terminating them. When this property is set to 0 (default), pegasus-monitord will not terminate any notification scripts, letting them run indefinitely. If some notification scripts missbehave, this has the potential problem of starving pegasus-monitord's notification slots (see the pegasus.monitord.notifications.max property), and block further notifications. In addition, users should be aware that pegasus-monitord will not exit until all notification scripts are finished.
Property Key: pegasus.monitord.stdout.disable.parsing **Profile Key:** N/A **Scope** : Properties **Since** : 3.1.1 **Type** : Boolean **Default** : false	By default, pegasus-monitord parses the stdout/stderr section of the kickstart to populate the applications captured stdout and stderr in the job instance table for the stampede schema. For large workflows, this may slow down monitord especially if the application is generating a lot of output to it's stdout and stderr. This property, can be used to turn of the database population.
Property Key: pegasus.monitord.arguments **Profile Key:** N/A **Scope** : Properties **Since** : 4.6 **Type** : String **Default** : N/A	This property specifies additional command-line arguments that should be passed to pegasus-monitord at startup. These additional arguments are appended to the arguments given to pegasus-monitord.

Job Clustering Properties

Table 13.24. Job Clustering Properties

Key Attributes	Description
Property Key: pegasus.clusterer.job.aggregator **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Type** : String **Values** : seqexec|mpiexec **Default** : seqexec	A large number of workflows executed through the Virtual Data System, are composed of several jobs that run for only a few seconds or so. The overhead of running any job on the grid is usually 60 seconds or more. Hence, it makes sense to collapse small independent jobs into a larger job. This property determines, the executable that will be used for running the larger job on the remote site. seqexec In this mode, the executable used to run the merged job is "pegasus-cluster" that runs each of the smaller jobs sequentially on the same node. The executable "pegasus-cluster" is a PEGASUS tool distributed in the PEGASUS worker package, and can be usually found at {pegasus.home}/bin/seqexec. mpiexec In this mode, the executable used to run the merged job is "pegasus-mpi-cluster" (PMC) that runs the smaller jobs via mpi on n nodes where n is the nodecount associated with the merged job. The executable "pegasus-mpi-cluster" is a PEGASUS tool distributed in the PEGASUS distribution and is built only if mpi compiler is available.
Property Key: pegasus.clusterer.job.aggregator.seqexec.log **Profile Key:** N/A **Scope** : Properties **Since** : 2.3	The tool pegasus-cluster logs the progress of the jobs that are being run by it in a progress file on the remote cluster where it is executed.

Key Attributes	Description
Type : Boolean **Default** : false **See Also** : pegasus.clusterer.job.aggregator **See Also** : pegasus.clusterer.job.aggregator.seqexec.log.global	This property sets the Boolean flag, that indicates whether to turn on the logging or not.
Property Key: pegasus.clusterer.job.aggregator.seqexec.log **Profile Key:** N/A **Scope** : Properties **Since** : 2.3 **Type** : Boolean **Default** : false **See Also** : pegasus.clusterer.job.aggregator **See Also** : pegasus.clusterer.job.aggregator.seqexec.log.global	The tool pegasus-cluster logs the progress of the jobs that are being run by it in a progress file on the remote cluster where it is executed. The progress log is useful for you to track the progress of your computations and remote grid debugging. The progress log file can be shared by multiple pegasus-cluster jobs that are running on a particular cluster as part of the same workflow. Or it can be per job. This property sets the Boolean flag, that indicates whether to have a single global log for all the pegasus-cluster jobs on a particular cluster or progress log per job.
Property Key: pegasus.clusterer.job.aggregator.seqexec.firstjobfail **Profile Key:** N/A **Scope** : Properties **Since** : 2.2 **Type** : Boolean **Default** : true **See Also** : pegasus.clusterer.job.aggregator	By default "pegasus-cluster" does not stop execution even if one of the clustered jobs it is executing fails. This is because "pegasus-cluster" tries to get as much work done as possible. This property sets the Boolean flag, that indicates whether to make "pegasus-cluster" stop on the first job failure it detects.
Property Key: pegasus.clusterer.label.key **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Type** : String **Default** : label	While clustering jobs in the workflow into larger jobs, you can optionally label your graph to control which jobs are clustered and to which clustered job they belong. This done using a label based clustering scheme and is done by associating a profile/label key in the PEGASUS namespace with the jobs in the DAX. Each job that has the same value/label value for this profile key, is put in the same clustered job. This property allows you to specify the PEGASUS profile key that you want to use for label based clustering.

Logging Properties

Table 13.25. Logging Properties

Key Attributes	Description
Property Key: pegasus.log.manager **Profile Key:** N/A **Scope** : Properties **Since** : 2.2.0 **Type** : String **Values** : Default|Log4J **Default** : Default **See Also** :pegasus.log.manager.formatter	This property sets the logging implementation to use for logging. Default This implementation refers to the legacy Pegasus logger, that logs directly to stdout and stderr. It however, does have the concept of levels similar to log4j or syslog. Log4j This implementation, uses Log4j to log messages. The log4j properties can be specified in a properties file, the location of which is specified by the property `pegasus.log.manager.log4j.conf`
Property Key: pegasus.log.manager.formatter **Profile Key:** N/A **Scope** : Properties	This property sets the formatter to use for formatting the log messages while logging.

Since : 2.2.0 **Type** : String **Values** : Simple|Netlogger **Default** : Simple **See Also** :pegasus.log.manager	Simple	This formats the messages in a simple format. The messages are logged as is with minimal formatting. Below are sample log messages in this format while ranking a dax according to performance.	

```
event.pegasus.ranking dax.id se18-
gda.dax  - STARTED
event.pegasus.parsing.dax dax.id
 se18-gda-nested.dax  - STARTED
event.pegasus.parsing.dax dax.id
 se18-gda-nested.dax  - FINISHED
job.id jobGDA
job.id jobGDA query.name
 getpredicted performace time
 10.00
event.pegasus.ranking dax.id se18-
gda.dax  - FINISHED
```

	Netlogger	This formats the messages in the Netlogger format , that is based on key value pairs. The netlogger format is useful for loading the logs into a database to do some meaningful analysis. Below are sample log messages in this format while ranking a dax according to performance.	

```
ts=2008-09-06T12:26:20.100502Z
 event=event.pegasus.ranking.start
 \
msgid=6bc49c1f-112e-4cdb-
af54-3e0afb5d593c \
eventId=event.pegasus.ranking_8d7c0a3c-9271-4c9c-
a0f2-1fb57c6394d5 \
dax.id=se18-gda.dax prog=Pegasus
ts=2008-09-06T12:26:20.100750Z
 event=event.pegasus.parsing.dax.start
 \
msgid=fed3ebdf-68e6-4711-8224-
a16bb1ad2969 \
eventId=event.pegasus.parsing.dax_887134a8-39cb-40f1-
b11c-b49def0c5232\
dax.id=se18-gda-nested.dax
 prog=Pegasus
ts=2008-09-06T12:26:20.100894Z
 event=event.pegasus.parsing.dax.end
 \
msgid=a81e92ba-27df-451f-bb2b-
b60d232ed1ad \
eventId=event.pegasus.parsing.dax_887134a8-39cb-40f1-
b11c-b49def0c5232
ts=2008-09-06T12:26:20.100395Z
 event=event.pegasus.ranking \
msgid=4dcecb68-74fe-4fd5-aa9e-
ea1cee88727d \
eventId=event.pegasus.ranking_8d7c0a3c-9271-4c9c-
a0f2-1fb57c6394d5 \
job.id="jobGDA"
ts=2008-09-06T12:26:20.100395Z
 event=event.pegasus.ranking \
msgid=4dcecb68-74fe-4fd5-aa9e-
ea1cee88727d \
eventId=event.pegasus.ranking_8d7c0a3c-9271-4c9c-
a0f2-1fb57c6394d5 \
job.id="jobGDA"
 query.name="getpredicted
 performace" time="10.00"
ts=2008-09-06T12:26:20.102003Z
 event=event.pegasus.ranking.end \
```

msgid=31f50f39-
efe2-47fc-9f4c-07121280cd64 \
eventId=event.pegasus.ranking_8d7c0a3c-9271-4c9c-
a0f2-1fb57c6394d5

Key Attributes	Description
Property Key: pegasus.log.* **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Type** : file path **Default** : no default	This property sets the path to the file where all the logging for Pegasus can be redirected to. Both stdout and stderr are logged to the file specified.
Property Key: pegasus.log.memory.usage **Profile Key:** N/A **Scope** : Properties **Since** : 4.3.4 **Type** : Boolean **Default** : false	This property if set to true, will result in the planner writing out JVM heap memory statistics at the end of the planning process at the INFO level. This is useful, if users want to fine tune their java memory settings by setting JAVA_HEAPMAX and JAVA_HEAPMIN for large workflows.

Property Key: pegasus.metrics.app
Profile Key: N/A
Scope : Properties
Since : 4.3.0
Type : String
Default : (no default)
	This property namespace allows users to pass application level metrics to the metrics server. The value of this property is the name of the application.
	
	Additional application specific attributes can be passed by using the prefix pegasus.metrics.app
	
	`pegasus.metrics.app.[arribute-name]`
	` attribute-value`
	
	Note: the attribute cannot be named name. This attribute is automatically assigned the value from pegasus.metrics.app

Cleanup Properties

Table 13.26. Cleanup Properties

Key Attributes	Description
Property Key: pegasus.file.cleanup.strategy **Profile Key:** N/A **Scope** : Properties **Since** : 2.2- **Type** : String **Default** : InPlace	This property is used to select the strategy of how the cleanup nodes are added to the executable workflow. InPlace — The default cleanup strategy. Adds cleanup nodes per level of the workflow. Constraint — Adds cleanup nodes to constraint the amount of storage space used by a workflow. Note that this property is overridden by the --cleanup option used in pegasus-plan.
Property Key: pegasus.file.cleanup.impl **Profile Key:** N/A **Scope** : Properties **Since** : 2.2 **Type** : String **Default** : Cleanup	This property is used to select the executable that is used to create the working directory on the compute sites. Cleanup — The default executable that is used to delete files is the "pegasus-transfer" executable shipped with Pegasus. It is found at $PEGASUS_HOME/bin/pegasus-transfer in the Pegasus distribution. An entry for transformation pegasus::dirmanager needs to exist in the Transformation Catalog or the

	PEGASUS_HOME environment variable should be specified in the site catalog for the sites for this mode to work.
	RM This mode results in the rm executable to be used to delete files from remote directories. The rm executable is standard on *nix systems and is usually found at /bin/rm location.
Property Key: pegasus.file.cleanup.clusters.num **Profile Key:** N/A **Scope** : Properties **Since** : 4.2.0 **Type** : Integer	In case of the InPlace strategy for adding the cleanup nodes to the workflow, this property specifies the maximum number of cleanup jobs that are added to the executable workflow on each level.
Property Key: pegasus.file.cleanup.clusters.size **Profile Key:** N/A **Scope** : Properties **Since** : 4.2.0 **Type** : Integer **Default** : 2	In case of the InPlace strategy this property sets the number of cleanup jobs that get clustered into a bigger cleanup job. This parameter is only used if pegasus.file.cleanup.clusters.num is not set.
Property Key: pegasus.file.cleanup.scope **Profile Key:** N/A **Scope** : Properties **Since** : 2.3.0 **Type** : Enumeration **Value** : fullahead|deferred **Default** : fullahead	By default in case of deferred planning InPlace file cleanup is turned OFF. This is because the cleanup algorithm does not work across partitions. This property can be used to turn on the cleanup in case of deferred planning. **fullahead** This is the default scope. The pegasus cleanup algorithm does not work across partitions in deferred planning. Hence the cleanup is always turned OFF , when deferred planning occurs and cleanup scope is set to full ahead. **deferred** If the scope is set to deferred, then Pegasus will not disable file cleanup in case of deferred planning. This is useful for scenarios where the partitions themselves are independant (i.e. dont share files). Even if the scope is set to deferred, users can turn off cleanup by specifying --no-cleanup option to pegasus-plan.
Property Key: pegasus.file.cleanup.constraint.*.maxspace **Profile Key:** N/A **Scope** : Properties **Since** : 4.6.0 **Type** : String **Default** : 10737418240	This property is used to set the maximum avaialble space (i.e., constraint) per site in Bytes. The * in the property name denotes the name of the compute site. A * in the property key is taken to mean all sites.
Property Key: pegasus.file.cleanup.constraint.deferstagein **Profile Key:** N/A **Scope** : Properties **Since** : 4.6.0 **Type** : Boolean **Default** : False	This property is used to determine whether stage in jobs may be deferred. If this property is set to False (default), all stage in jobs will be marked as executing on the current compute site and will be executed before any task. This property has no effect when running in a multi site case.
Property Key: pegasus.file.cleanup.constraint.csv **Profile Key:** N/A **Scope** : Properties **Since** : 4.6.1 **Type** : String	This property is used to specify a CSV file with a list of LFNs and their respective sizes in Bytes. The CSV file must be composed of two columns: **filename** and **length**.

Default : (no default)	

Miscellaneous Properties

Table 13.27. Miscellaneous Properties

Key Attributes	Description
Property Key: pegasus.code.generator **Profile Key:** N/A **Scope** : Properties **Since** : 3.0 **Type** : String **Values** : Condor|Shell|PMC **Default** : Condor **See Also** : pegasus.log.manager.formatter	This property is used to load the appropriate Code Generator to use for writing out the executable workflow. Condor This is the default code generator for Pegasus . This generator generates the executable workflow as a Condor DAG file and associated job submit files. The Condor DAG file is passed as input to Condor DAGMan for job execution. Shell This Code Generator generates the executable workflow as a shell script that can be executed on the submit host. While using this code generator, all the jobs should be mapped to site local i.e specify --sites local to pegasus-plan. PMC This Code Generator generates the executable workflow as a PMC task workflow. This is useful to run on platforms where it not feasible to run Condor such as the new XSEDE machines such as Blue Waters. In this mode, Pegasus will generate the executable workflow as a PMC task workflow and a sample PBS submit script that submits this workflow.
Property Key: pegasus.condor.concurrency.limits **Profile Key:** N/A **Scope** : Properties **Since** : 4.5.3 **Type** : Boolean **Default** : False	This Boolean property is used to determine whether Pegasus associates default HTCondor concurrency limits with jobs or not. Setting this property to true, allows you to throttle jobs across workflows, if the workflow are set to run in pure condor environment.
Property Key: pegasus.register **Profile Key:** N/A **Scope** : Properties **Since** : 4.1.- **Type** : Boolean **Default** : true	Pegasus creates registration jobs to register the output files in the replica catalog. An output file is registered only if 1) a user has configured a replica catalog in the properties 2) the register flags for the output files in the DAX are set to true This property can be used to turn off the creation of the registration jobs even though the files maybe marked to be registered in the replica catalog.
Property Key: pegasus.register.deep **Profile Key:** N/A **Scope** : Properties **Since** : 4.5.3.- **Type** : Boolean **Default** : true	By default, Pegasus always registers the complete LFN that is associated with the output files in the DAX i.e if the LFN has / in it, then lfn registered in the replica catalog has the whole part. For example, if in your DAX you have rupture/0001.rx as the name attribute for the uses tag, then in the Replica Catalog the LFN is registered as rupture/0001.rx On setting this property to false, only the basename is considered while registering in the replica catalog. In the

	above case, 0001.rx will be registered instead of rupture/0001.rx
Property Key: pegasus.data.reuse.scope **Profile Key:** N/A **Scope** : Properties **Since** : 4.5.0 **Type** : Enumeration **Value** : none|partial|full **Default** : full	This property is used to control the behavior of the data reuse algorithm in Pegasus none — This is same as disabling data reuse. It is equivalent to passing the --force option to pegasus-plan on the command line. partial — In this case, only certain jobs (those that have pegasus profile key enable_for_data_reuse set to true) are checked for presence of output files in the replica catalog. This gives users control over what jobs are deleted as part of the data reuse algorithm. full — This is the default behavior, where all the jobs output files are looked up in the replica catalog.
Property Key: pegasus.catalog.transformation.mapper **Profile Key:** N/A **Scope** : Properties **Since** : 2.0 **Type** : Enumeration **Value** : All|Installed|Staged|Submit **Default** : All	Pegasus supports transfer of statically linked executables as part of the executable workflow. At present, there is only support for staging of executables referred to by the compute jobs specified in the DAX file. Pegasus determines the source locations of the binaries from the transformation catalog, where it searches for entries of type STATIC_BINARY for a particular architecture type. The PFN for these entries should refer to a globus-url-copy valid and accessible remote URL. For transfer of executables, Pegasus constructs a soft state map that resides on top of the transformation catalog, that helps in determining the locations from where an executable can be staged to the remote site. This property determines, how that map is created. All — In this mode, all sources with entries of type STATIC_BINARY for a particular transformation are considered valid sources for the transfer of executables. This the most general mode, and results in the constructing the map as a result of the cartesian product of the matches. Installed — In this mode, only entries that are of type INSTALLED are used while constructing the soft state map. This results in Pegasus never doing any transfer of executables as part of the workflow. It always prefers the installed executables at the remote sites. Staged — In this mode, only entries that are of type STATIC_BINARY are used while constructing the soft state map. This results in the concrete workflow referring only to the staged executables, irrespective of the fact that the executables are already installed at the remote end. Submit — In this mode, only entries that are of type STATIC_BINARY and reside at the sub-

<table>
<tr><td></td><td>mit host ("site" local), are used while constructing the soft state map. This is especially helpful, when the user wants to use the latest compute code for his computations on the grid and that relies on his submit host.</td></tr>
<tr><td>

Property Key: pegasus.selector.transformation
Profile Key: N/A
Scope   : Properties
Since   : 2.0
Type   : Enumeration
Value   : Random|Installed|Staged|Submit
Default   : Random

</td><td>

In case of transfer of executables, Pegasus could have various transformations to select from when it schedules to run a particular compute job at a remote site. For e.g it can have the choice of staging an executable from a particular remote site, from the local (submit host) only, use the one that is installed on the remote site only.

This property determines, how a transformation amongst the various candidate transformations is selected, and is applied after the property pegasus.tc has been applied. For e.g specifying pegasus.tc as Staged and then pegasus.transformation.selector as INSTALLED does not work, as by the time this property is applied, the soft state map only has entries of type STAGED.

Random	In this mode, a random matching candidate transformation is selected to be staged to the remote execution site.
Installed	In this mode, only entries that are of type INSTALLED are selected. This means that the concrete workflow only refers to the transformations already pre installed on the remote sites.
Staged	In this mode, only entries that are of type STATIC_BINARY are selected, ignoring the ones that are installed at the remote site.
Submit	In this mode, only entries that are of type STATIC_BINARY and reside at the submit host ("site" local), are selected as sources for staging the executables to the remote execution sites.

</td></tr>
<tr><td>

Property Key: pegasus.parser.dax.preserver.linebreaks
Profile Key: N/A
Scope   : Properties
Since   : 2.2.0
Type   : Boolean
Default   : false

</td><td>

The DAX Parser normally does not preserve line breaks while parsing the CDATA section that appears in the arguments section of the job element in the DAX. On setting this to true, the DAX Parser preserves any line line breaks that appear in the CDATA section.

</td></tr>
<tr><td>

Property Key: pegasus.parser.dax.data.dependencies
Profile Key: N/A
Scope   : Properties
Since   : 4.4.0
Type   : Boolean
Default   : true

</td><td>

If this property is set to true, then the planner will automatically add edges between jobs in the DAX on the basis of exisitng data dependencies between jobs. For example, if a JobA generates an output file that is listed as input for JobB, then the planner will automatically add an edge between JobA and JobB.

</td></tr>
</table>

Chapter 14. Submit Directory Details

This chapter describes the submit directory content after Pegasus has planned a workflow. Pegasus takes in an abstract workflow (DAX) and generates an executable workflow (DAG) in the submit directory.

This document also describes the various Replica Selection Strategies in Pegasus.

Layout

Each executable workflow is associated with a submit directory, and includes the following:

1. **<daxlabel-daxindex>.dag**

   This is the Condor DAGMman dag file corresponding to the executable workflow generated by Pegasus. The dag file describes the edges in the DAG and information about the jobs in the DAG. Pegasus generated .dag file usually contains the following information for each job

   a. The job submit file for each job in the DAG.

   b. The post script that is to be invoked when a job completes. This is usually located at **$PEGASUS_HOME/bin/ exitpost** and parses the kickstart record in the job's **.out file** and determines the exitcode.

   c. JOB RETRY - the number of times the job is to be retried in case of failure. In Pegasus, the job postscript exits with a non zero exitcode if it determines a failure occurred.

2. **<daxlabel-daxindex>.dag.dagman.out**

   When a DAG (.dag file) is executed by Condor DAGMan , the DAGMan writes out it's output to the **<daxla- bel-daxindex>.dag.dagman.out file** . This file tells us the progress of the workflow, and can be used to determine the status of the workflow. Most of pegasus tools mine the **dagman.out** or **jobstate.log** to determine the progress of the workflows.

3. **<daxlabel-daxindex>.static.bp**

   This file contains netlogger events that link jobs in the DAG with the jobs in the DAX. This file is parsed by pegasus-monitord when a workflow starts and populated to the stampede backend.

4. **<daxlabel-daxindex>.notify**

   This file contains all the notifications that need to be set for the workflow and the jobs in the executable workflow. The format of notify file is described here

5. **<daxlabel-daxindex>.replica.store**

   This is a file based replica catalog, that only lists file locations are mentioned in the DAX.

6. **<daxlabel-daxindex>.dot**

   Pegasus creates a dot file for the executable workflow in addition to the .dag file. This can be used to visualize the executable workflow using the dot program.

7. **<job>.sub**

   Each job in the executable workflow is associated with it's own submit file. The submit file tells Condor how to execute the job.

8. **<job>.out.00n**

   The stdout of the executable referred in the job submit file. In Pegasus, most jobs are launched via kickstart. Hence, this file contains the kickstart XML provenance record that captures runtime provenance on the remote node where the job was executed. n varies from 1-N where N is the JOB RETRY value in the .dag file. The exitpost executable

is invoked on the <job>.out file and it moves the <job>.out to <job>.out.00n so that the the job's .out files are preserved across retries.

9. **<job>.err.00n**

The stderr of the executable referred in the job submit file. In case of Pegasus, mostly the jobs are launched via kickstart. Hence, this file contains stderr of kickstart. This is usually empty unless there in an error in kickstart e.g. kickstart segfaults , or kickstart location specified in the submit file is incorrect. The exitpost executable is invoked on the **<job>.out** file and it moves the **<job>.err to <job>.err.00n** so that the the job's **.out** files are preserved across retries.

10. **jobstate.log**

The jobstate.log file is written out by the pegasus-monitord daemon that is launched when a workflow is submitted for execution by pegasus-run. The pegasus-monitord daemon parses the dagman.out file and writes out the jobstate.log that is easier to parse. The jobstate.log captures the various states through which a job goes during the workflow. There are other monitoring related files that are explained in the monitoring chapter.

11. **braindump.txt**

Contains information about pegasus version, dax file, dag file, dax label.

Condor DAGMan File

The Condor DAGMan file (.dag) is the input to Condor DAGMan (the workflow executor used by Pegasus) .

Pegasus generated .dag file usually contains the following information for each job:

1. The job submit file for each job in the DAG.

2. The post script that is to be invoked when a job completes. This is usually found in **$PEGASUS_HOME/bin/ exitpost** and parses the kickstart record in the job's .out file and determines the exitcode.

3. JOB RETRY - the number of times the job is to be retried in case of failure. In case of Pegasus, job postscript exits with a non zero exitcode if it determines a failure occurred.

4. The pre script to be invoked before running a job. This is usually for the dax jobs in the DAX. The pre script is pegasus-plan invocation for the subdax.

In the last section of the DAG file the relations between the jobs (that identify the underlying DAG structure) are highlighted.

Sample Condor DAG File

```
######################################################################
# PEGASUS WMS GENERATED DAG FILE
# DAG blackdiamond
# Index = 0, Count = 1
######################################################################

JOB create_dir_blackdiamond_0_isi_viz create_dir_blackdiamond_0_isi_viz.sub
SCRIPT POST create_dir_blackdiamond_0_isi_viz /pegasus/bin/pegasus-exitcode   \
                                    /submit-dir/create_dir_blackdiamond_0_isi_viz.out
RETRY create_dir_blackdiamond_0_isi_viz 3

JOB create_dir_blackdiamond_0_local create_dir_blackdiamond_0_local.sub
SCRIPT POST create_dir_blackdiamond_0_local /pegasus/bin/pegasus-exitcode
                                    /submit-dir/create_dir_blackdiamond_0_local.out

JOB pegasus_concat_blackdiamond_0 pegasus_concat_blackdiamond_0.sub

JOB stage_in_local_isi_viz_0 stage_in_local_isi_viz_0.sub
SCRIPT POST stage_in_local_isi_viz_0 /pegasus/bin/pegasus-exitcode   \
                                      /submit-dir/stage_in_local_isi_viz_0.out

JOB chmod_preprocess_ID000001_0 chmod_preprocess_ID000001_0.sub
SCRIPT POST chmod_preprocess_ID000001_0 /pegasus/bin/pegasus-exitcode \
                                        /submit-dir/chmod_preprocess_ID000001_0.out
```

```
JOB preprocess_ID000001 preprocess_ID000001.sub
SCRIPT POST preprocess_ID000001 /pegasus/bin/pegasus-exitcode    \
                                    /submit-dir/preprocess_ID000001.out

JOB subdax_black_ID000002 subdax_black_ID000002.sub
SCRIPT PRE subdax_black_ID000002 /pegasus/bin/pegasus-plan  \
     -Dpegasus.user.properties=/submit-dir/./dag_1/test_ID000002/
pegasus.3862379342822189446.properties\
     -Dpegasus.log.*=/submit-dir/subdax_black_ID000002.pre.log \
     -Dpegasus.dir.exec=app_domain/app -Dpegasus.dir.storage=duncan -Xmx1024 -Xms512\
     --dir /pegasus-features/dax-3.2/dags \
     --relative-dir user/pegasus/blackdiamond/run0005/user/pegasus/blackdiamond/run0005/./dag_1 \
     --relative-submit-dir user/pegasus/blackdiamond/run0005/./dag_1/test_ID000002\
     --basename black --sites dax_site \
     --output local --force  --nocleanup  \
     --verbose  --verbose  --verbose  --verbose  --verbose  --verbose  --verbose \
     --verbose  --monitor  --deferred  --group pegasus --rescue 0 \
     --dax /submit-dir/./dag_1/test_ID000002/dax/blackdiamond_dax.xml

JOB stage_out_local_isi_viz_0_0 stage_out_local_isi_viz_0_0.sub
SCRIPT POST stage_out_local_isi_viz_0_0 /pegasus/bin/pegasus-exitcode   /submit-dir/
stage_out_local_isi_viz_0_0.out

SUBDAG EXTERNAL subdag_black_ID000003 /Users/user/Pegasus/work/dax-3.2/black.dag DIR /duncan/test

JOB clean_up_stage_out_local_isi_viz_0_0 clean_up_stage_out_local_isi_viz_0_0.sub
SCRIPT POST clean_up_stage_out_local_isi_viz_0_0 /lfs1/devel/Pegasus/pegasus/bin/pegasus-exitcode  \
                                    /submit-dir/clean_up_stage_out_local_isi_viz_0_0.out

JOB clean_up_preprocess_ID000001 clean_up_preprocess_ID000001.sub
SCRIPT POST clean_up_preprocess_ID000001 /lfs1/devel/Pegasus/pegasus/bin/pegasus-exitcode  \
                                    /submit-dir/clean_up_preprocess_ID000001.out

PARENT create_dir_blackdiamond_0_isi_viz CHILD pegasus_concat_blackdiamond_0
PARENT create_dir_blackdiamond_0_local CHILD pegasus_concat_blackdiamond_0
PARENT stage_out_local_isi_viz_0_0 CHILD clean_up_stage_out_local_isi_viz_0_0
PARENT stage_out_local_isi_viz_0_0 CHILD clean_up_preprocess_ID000001
PARENT preprocess_ID000001 CHILD subdax_black_ID000002
PARENT preprocess_ID000001 CHILD stage_out_local_isi_viz_0_0
PARENT subdax_black_ID000002 CHILD subdag_black_ID000003
PARENT stage_in_local_isi_viz_0 CHILD chmod_preprocess_ID000001_0
PARENT stage_in_local_isi_viz_0 CHILD preprocess_ID000001
PARENT chmod_preprocess_ID000001_0 CHILD preprocess_ID000001
PARENT pegasus_concat_blackdiamond_0 CHILD stage_in_local_isi_viz_0
######################################################################
# End of DAG
######################################################################
```

Kickstart XML Record

Kickstart is a light weight C executable that is shipped with the pegasus worker package. All jobs are launced via Kickstart on the remote end, unless explicitly disabled at the time of running pegasus-plan.

Kickstart does not work with:

1. Condor Standard Universe Jobs

2. MPI Jobs

Pegasus automatically disables kickstart for the above jobs.

Kickstart captures useful runtime provenance information about the job launched by it on the remote note, and puts in an XML record that it writes to its own stdout. The stdout appears in the workflow submit directory as <job>.out.00n . The following information is captured by kickstart and logged:

1. The exitcode with which the job it launched exited.

2. The duration of the job

3. The start time for the job

4. The node on which the job ran

5. The stdout and stderr of the job

6. The arguments with which it launched the job

7. The environment that was set for the job before it was launched.

8. The machine information about the node that the job ran on

Amongst the above information, the dagman.out file gives a coarser grained estimate of the job duration and start time.

Reading a Kickstart Output File

The kickstart file below has the following fields highlighted:

1. The host on which the job executed and the ipaddress of that host

2. The duration and start time of the job. The time here is in reference to the clock on the remote node where the job is executed.

3. The exitcode with which the job executed

4. The arguments with which the job was launched.

5. The directory in which the job executed on the remote site

6. The stdout of the job

7. The stderr of the job

8. The environment of the job

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<invocation xmlns="http://pegasus.isi.edu/schema/invocation" \
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" \
       xsi:schemaLocation="http://pegasus.isi.edu/schema/invocation http://pegasus.isi.edu/schema/
iv-2.0.xsd" \
       version="2.0" start="2009-01-30T19:17:41.157-06:00" duration="0.321"
 transformation="pegasus::dirmanager"\
      derivation="pegasus::dirmanager:1.0" resource="cobalt" wf-label="scb" \
      wf-stamp="2009-01-30T17:12:55-08:00" hostaddr="141.142.30.219" hostname="co-
login.ncsa.uiuc.edu"\
      pid="27714" uid="29548" user="vahi" gid="13872" group="bvr" umask="0022">

<mainjob start="2009-01-30T19:17:41.426-06:00" duration="0.052" pid="27783">

<usage utime="0.036" stime="0.004" minflt="739" majflt="0" nswap="0" nsignals="0" nvcsw="36"
 nivcsw="3"/>

<status raw="0"><regular exitcode="0"/></status>

<statcall error="0">
<!-- deferred flag: 0 -->
<file name="/u/ac/vahi/SOFTWARE/pegasus/default/bin/dirmanager">23212F7573722F62696E2F656E762070</
file>
<statinfo mode="0100755" size="8202" inode="85904615883" nlink="1" blksize="16384" \
    blocks="24" mtime="2008-09-22T18:52:37-05:00" atime="2009-01-30T14:54:18-06:00" \
    ctime="2009-01-13T19:09:47-06:00" uid="29548" user="vahi" gid="13872" group="bvr"/>
</statcall>

<argument-vector>
<arg nr="1">--create</arg>
<arg nr="2">--dir</arg>
<arg nr="3">/u/ac/vahi/globus-test/EXEC/vahi/pegasus/scb/run0001</arg>
</argument-vector>

</mainjob>

<cwd>/u/ac/vahi/globus-test/EXEC</cwd>

<usage utime="0.012" stime="0.208" minflt="4232" majflt="0" nswap="0" nsignals="0" nvcsw="15"
 nivcsw="74"/>
<machine page-size="16384" provider="LINUX">
```

```
<stamp>2009-01-30T19:17:41.157-06:00</stamp>
<uname system="linux" nodename="co-login" release="2.6.16.54-0.2.5-default" machine="ia64">#1 SMP
 Mon Jan 21\
         13:29:51 UTC 2008</uname>
<ram total="148299268096" free="123371929600" shared="0" buffer="2801664"/>
<swap total="1179656486912" free="1179656486912"/>
<boot idle="1315786.920">2009-01-15T10:19:50.283-06:00</boot>
<cpu count="32" speed="1600" vendor=""></cpu>
<load min1="3.50" min5="3.50" min15="2.60"/>
<proc total="841" running="5" sleeping="828" stopped="5" vmsize="10025418752" rss="2524299264"/>
<task total="1125" running="6" sleeping="1114" stopped="5"/>
</machine>
<statcall error="0" id="stdin">
<!-- deferred flag: 0 -->
<file name="/dev/null"/>
<statinfo mode="020666" size="0" inode="68697" nlink="1" blksize="16384" blocks="0" \
    mtime="2007-05-04T05:54:02-05:00" atime="2007-05-04T05:54:02-05:00" \
   ctime="2009-01-15T10:21:54-06:00" uid="0" user="root" gid="0" group="root"/>
</statcall>

<statcall error="0" id="stdout">
<temporary name="/tmp/gs.out.s9rTJL" descriptor="3"/>
<statinfo mode="0100600" size="29" inode="203420686" nlink="1" blksize="16384" blocks="128" \
 mtime="2009-01-30T19:17:41-06:00" atime="2009-01-30T19:17:41-06:00"\
 ctime="2009-01-30T19:17:41-06:00" uid="29548" user="vahi" gid="13872" group="bvr"/>
<data>mkdir finished successfully.
</data>
</statcall>
<statcall error="0" id="stderr">
<temporary name="/tmp/gs.err.kobn3S" descriptor="5"/>
<statinfo mode="0100600" size="0" inode="203420689" nlink="1" blksize="16384" blocks="0" \
 mtime="2009-01-30T19:17:41-06:00" atime="2009-01-30T19:17:41-06:00" \
ctime="2009-01-30T19:17:41-06:00" uid="29548" user="vahi" gid="13872" group="bvr"/>
</statcall>

<statcall error="0" id="gridstart">
<!-- deferred flag: 0 -->
<file name="/u/ac/vahi/SOFTWARE/pegasus/default/bin/kickstart">7F454C4602010100000000000000000000</file>
<statinfo mode="0100755" size="255445" inode="85904615876" nlink="1" blksize="16384" blocks="504" \
  mtime="2009-01-30T18:06:28-06:00" atime="2009-01-30T19:17:41-06:00"\
 ctime="2009-01-30T18:06:28-06:00" uid="29548" user="vahi" gid="13872" group="bvr"/>
</statcall>
<statcall error="0" id="logfile">
<descriptor number="1"/>
<statinfo mode="0100600" size="0" inode="53040253" nlink="1" blksize="16384" blocks="0" \
 mtime="2009-01-30T19:17:39-06:00" atime="2009-01-30T19:17:39-06:00" \
ctime="2009-01-30T19:17:39-06:00" uid="29548" user="vahi" gid="13872" group="bvr"/>
</statcall>
<statcall error="0" id="channel">
<fifo name="/tmp/gs.app.Ien1m0" descriptor="7" count="0" rsize="0" wsize="0"/>
<statinfo mode="010640" size="0" inode="203420696" nlink="1" blksize="16384" blocks="0" \
  mtime="2009-01-30T19:17:41-06:00" atime="2009-01-30T19:17:41-06:00" \
ctime="2009-01-30T19:17:41-06:00" uid="29548" user="vahi" gid="13872" group="bvr"/>
</statcall>

<environment>
<env key="GLOBUS_GRAM_JOB_CONTACT">https://co-login.ncsa.uiuc.edu:50001/27456/1233364659/</env>
<env key="GLOBUS_GRAM_MYJOB_CONTACT">URLx-nexus://co-login.ncsa.uiuc.edu:50002/</env>
<env key="GLOBUS_LOCATION">/usr/local/prews-gram-4.0.7-r1/</env>
....
</environment>

<resource>
<soft id="RLIMIT_CPU">unlimited</soft>
<hard id="RLIMIT_CPU">unlimited</hard>
<soft id="RLIMIT_FSIZE">unlimited</soft>
....
</resource>
</invocation>
```

Jobstate.Log File

The jobstate.log file logs the various states that a job goes through during workflow execution. It is created by the **pegasus-monitord** daemon that is launched when a workflow is submitted to Condor DAGMan by pegasus-run.

pegasus-monitord parses the dagman.out file and writes out the jobstate.log file, the format of which is more amenable to parsing.

Note

The jobstate.log file is not created if a user uses condor_submit_dag to submit a workflow to Condor DAG-Man.

The jobstate.log file can be created after a workflow has finished executing by running **pegasus-monitord** on the .dagman.out file in the workflow submit directory.

Below is a snippet from the jobstate.log for a single job executed via condorg:

```
1239666049 create_dir_blackdiamond_0_isi_viz SUBMIT 3758.0 isi_viz - 1
1239666059 create_dir_blackdiamond_0_isi_viz EXECUTE 3758.0 isi_viz - 1
1239666059 create_dir_blackdiamond_0_isi_viz GLOBUS_SUBMIT 3758.0 isi_viz - 1
1239666059 create_dir_blackdiamond_0_isi_viz GRID_SUBMIT 3758.0 isi_viz - 1
1239666064 create_dir_blackdiamond_0_isi_viz JOB_TERMINATED 3758.0 isi_viz - 1
1239666064 create_dir_blackdiamond_0_isi_viz JOB_SUCCESS 0 isi_viz - 1
1239666064 create_dir_blackdiamond_0_isi_viz POST_SCRIPT_STARTED - isi_viz - 1
1239666069 create_dir_blackdiamond_0_isi_viz POST_SCRIPT_TERMINATED 3758.0 isi_viz - 1
1239666069 create_dir_blackdiamond_0_isi_viz POST_SCRIPT_SUCCESS - isi_viz - 1
```

Each entry in jobstate.log has the following:

1. The ISO timestamp for the time at which the particular event happened.

2. The name of the job.

3. The event recorded by DAGMan for the job.

4. The condor id of the job in the queue on the submit node.

5. The pegasus site to which the job is mapped.

6. The job time requirements from the submit file.

7. The job submit sequence for this workflow.

Table 14.1. The job lifecycle when executed as part of the workflow

STATE/EVENT	DESCRIPTION
SUBMIT	job is submitted by condor schedd for execution.
EXECUTE	condor schedd detects that a job has started execution.
GLOBUS_SUBMIT	the job has been submitted to the remote resource. It's only written for GRAM jobs (i.e. gt2 and gt4).
GRID_SUBMIT	same as GLOBUS_SUBMIT event. The ULOG_GRID_SUBMIT event is written for all grid universe jobs./
JOB_TERMINATED	job terminated on the remote node.
JOB_SUCCESS	job succeeded on the remote host, condor id will be zero (successful exit code).
JOB_FAILURE	job failed on the remote host, condor id will be the job's exit code.
POST_SCRIPT_STARTED	post script started by DAGMan on the submit host, usually to parse the kickstart output
POST_SCRIPT_TERMINATED	post script finished on the submit node.
POST_SCRIPT_SUCCESS	POST_SCRIPT_FAILURE	post script succeeded or failed.

There are other monitoring related files that are explained in the monitoring chapter.

Pegasus Workflow Job States and Delays

The various job states that a job goes through (as caputured in the dagman.out and jobstate.log file) during it's lifecycle are illustrated below. The figure below highlights the various local and remote delays during job lifecycle.

Braindump File

The braindump file is created per workflow in the submit file and contains metadata about the workflow.

Table 14.2. Information Captured in Braindump File

KEY	DESCRIPTION
user	the username of the user that ran pegasus-plan
grid_dn	the Distinguished Name in the proxy
submit_hostname	the hostname of the submit host
root_wf_uuid	the workflow uuid of the root workflow
wf_uuid	the workflow uuid of the current workflow i.e the one whose submit directory the braindump file is.
dax	the path to the dax file
dax_label	the label attribute in the adag element of the dax
dax_index	the index in the dax.
dax_version	the version of the DAX schema that DAX referred to.
pegasus_wf_name	the workflow name constructed by pegasus when planning
timestamp	the timestamp when planning occured
basedir	the base submit directory
submit_dir	the full path for the submit directory
properties	the full path to the properties file in the submit directory

planner	the planner used to construct the executable workflow. always pegasus
planner_version	the versions of the planner
pegasus_build	the build timestamp
planner_arguments	the arguments with which the planner is invoked.
jsd	the path to the jobstate file
rundir	the rundir in the numbering scheme for the submit directories
pegasushome	the root directory of the pegasus installation
vogroup	the vo group to which the user belongs to. Defaults to pegasus
condor_log	the full path to condor common log in the submit directory
notify	the notify file that contains any notifications that need to be sent for the workflow.
dag	the basename of the dag file created
type	the type of executable workflow. Can be dag	shell

A Sample Braindump File is displayed below:

```
user vahi
grid_dn null
submit_hostname obelix
root_wf_uuid a4045eb6-317a-4710-9a73-96a745cb1fe8
wf_uuid a4045eb6-317a-4710-9a73-96a745cb1fe8
dax /data/scratch/vahi/examples/synthetic-scec/Test.dax
dax_label Stampede-Test
dax_index 0
dax_version 3.3
pegasus_wf_name Stampede-Test-0
timestamp 20110726T153746-0700
basedir /data/scratch/vahi/examples/synthetic-scec/dags
submit_dir /data/scratch/vahi/examples/synthetic-scec/dags/vahi/pegasus/Stampede-Test/run0005
properties pegasus.6923599674234553065.properties
planner /data/scratch/vahi/software/install/pegasus/default/bin/pegasus-plan
planner_version 3.1.0cvs
pegasus_build 20110726221240Z
planner_arguments "--conf ./conf/properties --dax Test.dax --sites local --output local --dir dags
 --force --submit "
jsd jobstate.log
rundir run0005
pegasushome /data/scratch/vahi/software/install/pegasus/default
vogroup pegasus
condor_log Stampede-Test-0.log
notify Stampede-Test-0.notify
dag Stampede-Test-0.dag
type dag
```

Pegasus static.bp File

Pegasus creates a workflow.static.bp file that links jobs in the DAG with the jobs in the DAX. The contents of the file are in netlogger format. The purpose of this file is to be able to link an invocation record of a task to the corresponding job in the DAX

The workflow is replaced by the name of the workflow i.e. same prefix as the .dag file

In the file there are five types of events:

• task.info

  This event is used to capture information about all the tasks in the DAX(abstract workflow)

• task.edge

This event is used to capture information about the edges between the tasks in the DAX (abstract workflow)

- job.info

  This event is used to capture information about the jobs in the DAG (executable workflow generated by Pegasus)

- job.edge

  This event is used to capture information about edges between the jobs in the DAG (executable workflow).

- wf.map.task_job

  This event is used to associate the tasks in the DAX with the corresponding jobs in the DAG.

Chapter 15. Jupyter Notebooks

Introduction

The Jupyter Notebook [http://jupyter.org/] is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text. Its flexible and portable format resulted in a rapidly adoption by the research community to share and interact with experiments. Jupyter Notebooks has a strong potential to reduce the gap between researchers and the complex knowledge required to run large-scale scientific workflows via a programmatic high-level interface to access/manage workflow capabilities.

The Pegasus-Jupyter integration aims to facilitate the usage of Pegasus via Jupyter notebooks. In addition to easiness of usage, notebooks foster reproducibility (all the information to run an experiment is in a unique place) and reuse (notebooks are portable if running in equivalent environments). Since Pegasus 4.8.0 [/downloads], a Python API to declare and manage Pegasus workflows via Jupyter has been provided. The user can create a notebook and declare a workflow using the Pegasus DAX API, and then create an instance of the workflow for execution. This API encapsulates most of Pegasus commands (e.g., plan, run, statistics, among others), and also allows workflow creation, execution, and monitoring. The API also provides mechanisms to define Pegasus catalogs (sites, replica, and transformation), as well as to generate tutorial example workflows.

Requirements

In order to run Pegasus workflows via Jupyter the following software packages are required:

1. Python 2.7 or superior (Jupyter requires version 2.7+)

2. Java 1.6 or superior

3. Pegasus 4.8.0 or superior (submit node)

4. Jupyter (see installation notes [http://jupyter.org/install.html])

The Pegasus DAX and Jupyter Python APIs

The first step to enable Jupyter to use the Pegasus API is to import the Python Pegasus Jupyter API. The instance module will automatically load the Pegasus DAX3 API and the catalogs APIs.

```
from Pegasus.jupyter.instance import *
```

By default, the API automatically creates a folder in the user's $HOME directory based on the workflow name. However, a predefined path for the workflow files can also be defined as follows:

```
workflow_dir = '/home/pegasus/wf-split-tutorial'
```

Creating an Abstract Workflow

Workflow creation within Jupyter follows the same steps to generate a DAX with the DAX Generator API.

Creating the Catalogs

The *Replica Catalog* (RC) tells Pegasus where to find each of the input files for the workflow. We provide a Python API for creating the RC programmatically. For detailed information on how the RC works and its semantics can be found here, and the auto-generated python documentation for this API can be found here [python/replica_catalog.html].

```
rc = ReplicaCatalog(workflow_dir)
rc.add('pegasus.html', 'file:///home/pegasus/pegasus.html', site='local')
```

The *Transformation Catalog* (TC) describes all of the executables (called "transformations") used by the workflow. The Python Jupyter API also provides methods to manage this catalog. A detailed description of the TC properties

can be found here, and the auto-generated python documentation for this API can be found here [python/transformation_catalog.html].

```
e_split = Executable('split', arch=Arch.X86_64, os=OSType.LINUX, installed=True)
e_split.addPFN(PFN('file:///usr/bin/split', 'condorpool'))

e_wc = Executable('wc', arch=Arch.X86_64, os=OSType.LINUX, installed=True)
e_wc.addPFN(PFN('file:///usr/bin/wc', 'condorpool'))

tc = TransformationCatalog(workflow_dir)
tc.add(e_split)
tc.add(e_wc)
```

The *Site Catalog* (SC) describes the sites where the workflow jobs are to be executed. A detailed description of the SC properties and handlers can be found here, and the auto-generated python documentation for this API can be found here [python/sites_catalog.html].

```
sc = SitesCatalog(workflow_dir)
sc.add_site('condorpool', arch=Arch.X86_64, os=OSType.LINUX)
sc.add_site_profile('condorpool', namespace=Namespace.PEGASUS, key='style', value='condor')
sc.add_site_profile('condorpool', namespace=Namespace.CONDOR, key='universe', value='vanilla')
```

Workflow Execution

Workflow execution and management are performed using an *Instance* object. An instance receives a DAX object (created with the DAX Generator API), and the catalogs objects (replica, transformation, and site). A path to the workflow directory can also be provided:

```
instance = Instance(dax, replica_catalog=rc, transformation_catalog=tc, sites_catalog=sc,
 workflow_dir=workflow_dir)
```

An instance object represents a workflow run, from where the workflow execution can be launched, monitored, and managed. The *run* method starts the workflow execution.

```
instance.run(site='condorpool')
```

After the workflow has been submitted you can monitor it using the *status()* method. This method takes two arguments:

1. *loop*: whether the status command should be invoked once or continuously until the workflow is completed or a failure is detected.

2. *delay*: The delay (in seconds) the status will be refreshed. Default value is 10s.

```
instance.status(loop=True, delay=5)
```

JupyterHub

The Pegasus Jupyter API can also be used with JupyterHub [https://jupyterhub.readthedocs.io] portals. Due to the strict requirement of Python 3 for running the multi-user hub, our API requires the Python future [https://pypi.python.org/pypi/future] package in order to be compatible with Python 3.

API Reference

Refer to the auto-generated python documentation explaining the Jupyter API (instance) [python/instance.html], and for the catalogs (sites [python/sites_catalog.html], replica [python/replica_catalog.html], and transformation [python/transformation_catalog.html]).

Tutorial Example Notebook

The Pegasus Tutorial VM is configured with Jupyter and the example Pegasus Tutorial Jupyter Notebook. To start Jupyter, use the following command in the VM terminal:

```
$ jupyter-notebook --browser=firefox
```

This command will open the browser with a tab to the Jupyter dashboard, which will show your $HOME directory list of files. The Pegasus Tutorial Notebook can be found into the **'jupyter'** folder.

Chapter 16. API Reference

DAX XML Schema

The DAX format is described by the XML schema instance document dax-3.6.xsd [schemas/dax-3.6/dax-3.6.xsd]. A local copy of the schema definition is provided in the "etc" directory. The documentation of the XML schema and its elements can be found in dax-3.6.html [schemas/dax-3.6/dax-3.6.html] as well as locally in `doc/schemas/dax-3.6/dax-3.6.html` in your Pegasus distribution.

DAX XML Schema In Detail

The DAX file format has four major sections, with the second section divided into more sub-sections. The DAX format works on the abstract or logical level, letting you focus on the shape of the workflows, what to do and what to work upon.

1. Workflow level Metadata

   Metadata that is associated with the whole workflow. These are defined in the Metadata section.

2. Workflow-level Notifications

   Very simple workflow-level notifications. These are defined in the Notification section.

3. Catalogs

   The first section deals with included catalogs. While we do recommend to use external replica- and transformation catalogs, it is possible to include some replicas and transformations into the DAX file itself. Any DAX-included entry takes precedence over regular replica catalog (RC) and transformation catalog (TC) entries.

   The first section (and any of its sub-sections) is completely optional.

   a. The first sub-section deals with included replica descriptions.

   b. The second sub-section deals with included transformation descriptions.

   c. The third sub-section declares multi-item executables.

4. Job List

   The jobs section defines the job- or task descriptions. For each task to conduct, a three-part logical name declares the task and aides identifying it in the transformation catalog or one of the *executable* section above. During planning, the logical name is translated into the physical executable location on the chosen target site. By declaring jobs abstractly, physical layout consideration of the target sites do not matter. The job's *id* uniquley identifies the job within this workflow.

   The arguments declare what command-line arguments to pass to the job. If you are passing filenames, you should refer to the logical filename using the *file* element in the argument list.

   Important for properly planning the task is the list of files consumed by the task, its input files, and the files produced by the task, its output files. Each file is described with a *uses* element inside the task.

   Elements exist to link a logical file to any of the stdio file descriptors. The *profile* element is Pegasus's way to abstract site-specific data.

   Jobs are nodes in the workflow graph. Other nodes include unplanned workflows (DAX), which are planned and then run when the node runs, and planned workflows (DAG), which are simply executed.

5. Control-flow Dependencies

   The third section lists the dependencies between the tasks. The relationships are defined as child parent relationships, and thus impacts the order in which tasks are run. No cyclic dependencies are permitted.

Dependencies are directed edges in the workflow graph.

XML Intro

If you have seen the DAX schema before, not a lot of new items in the root element. *However*, we did retire the (old) attributes ending in *Count*.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated: 2011-07-28T18:29:57Z -->
<adag xmlns="http://pegasus.isi.edu/schema/DAX"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://pegasus.isi.edu/schema/DAX http://pegasus.isi.edu/schema/
dax-3.6.xsd"
      version="3.6"
      name="diamond"
      index="0"
      count="1">
```

The following attributes are supported for the root element *adag*.

Table 16.1. Root element attributes

attribute	optional?	type	meaning
version	required	*VersionPattern*	Version number of DAX instance document. Must be 3.6.
name	required	string	name of this DAX (or set of DAXes).
count	optional	positiveInteger	size of list of DAXes with this *name*. Defaults to 1.
index	optional	nonNegativeInteger	current index of DAX with same *name*. Defaults to 0.
fileCount	removed	nonNegativeInteger	Old 2.1 attribute, removed, do not use.
jobCount	removed	positiveInteger	Old 2.1 attribute, removed, do not use.
childCount	removed	nonNegativeInteger	Old 2.1 attribute, removed, do not use.

The *version* attribute is restricted to the regular expression $\d+(\.\d+(\.\d+)?)?$. This expression represents the *VersionPattern* type that is used in other places, too. It is a more restrictive expression than before, but allows us to compute comparable version number using the following formula:

version1: a.b.c	version2: d.e.f
n = a * 1,000,000 + b * 1,000 + c	m = d * 1,000,000 + e * 1,000 + f
version1 > version2 if n > m	

Workflow-level Metadata

Metadata associated with the whole workflow.

```
<metadata key="name">diamond</metadata>
<metadata key="createdBy">Karan Vahi</metadata>
```

The workflow level metadata maybe used to control the Pegasus Mapper behaviour at planning time or maybe propogated to external services while querying for job characteristics.

Workflow-level Notifications

Notifications that are generated when workflow level events happened.

```
<!-- part 1.1: invocations -->
<invoke when="at_end">/bin/date -Ins &gt;&gt; my.log</invoke>
```

The above snippet will append the current time to a log file in the current directory. This is with regards to the pegasus-monitord instance acting on the notification.

The Catalogs Section

The initial section features three sub-sections:

1. a catalog of files used,

2. a catalog of transformations used, and

3. compound transformation declarations.

The Replica Catalog Section

The file section acts as in in-file replica catalog (RC). Any files declared in this section take precedence over files in external replica catalogs during planning.

```
<!-- part 1.2: included replica catalog -->
<file name="example.a" >
  <!-- profiles are optional -->
  <!-- The "stat" namespace is ONLY AN EXAMPLE -->
  <profile namespace="stat" key="size">/* integer to be defined */</profile>
  <profile namespace="stat" key="md5sum">/* 32 char hex string */</profile>
  <profile namespace="stat" key="mtime">/* ISO-8601 timestamp */</profile>

  <!-- Metadata will be supported 4.6 onwards-->
  <metadata key="timestamp" >/* ISO-8601 *or* 20100417134523:int */</metadata>
  <metadata key="origin" >ocean</metadata>

  <!-- PFN to by-pass replica catalog -->
  <!-- The "site attribute is optional -->
  <pfn url="file:///tmp/example.a" site="local">
    <profile namespace="stat" key="owner">voeckler</profile>
  </pfn>
  <pfn url="file:///storage/funky.a" site="local"/>
</file>

<!-- a more typical example from the black diamond -->
<file name="f.a">
  <pfn url="file:///Users/voeckler/f.a" site="local"/>
</file>
```

The first *file* entry above is an example of a data file with two replicas. The *file* element requires a logical file *name*. Each logical filename may have additional information associated with it, enumerated by *profile* elements. Each file entry may have 0 or more *metadata* associated with it. Each piece of metadata has a *key* string and *type* attribute describing the element's value.

Warning

The *metadata* element is not support as of this writing! Details may change in the future.

The *file* element can provide 0 or more *pfn* locations, taking precedence over the replica catalog. A *file* element that does not name any *pfn* children-elements will still require look-ups in external replica catalogs. Each *pfn* element names a concrete location of a file. Multiple locations constitute replicas of the same file, and are assumed to be usable interchangably. The *url* attribute is mandatory, and typically would use a file schema URL. The *site* attribute is optional, and defaults to value *local* if missing. A *pfn* element may have *profile* children-elements, which refer to attributes of the physical file. The file-level profiles refer to attributes of the logical file.

Note

The stat profile namespace is ony an example, and details about stat are not yet implemented. The proper namespaces pegasus, condor, dagman, env, hints, globus and selector enjoy full support.

The second *file* entry above shows a usage example from the black-diamond example workflow that you are more likely to encouter or write.

The presence of an in-file replica catalog lets you declare a couple of interesting advanced features. The DAG and DAX file declarations are just files for all practical purposes. For deferred planning, the location of the site catalog (SC) can be captured in a file, too, that is passed to the job dealing with the deferred planning as logical filename.

```
<file name="black.dax" >
  <!-- specify the location of the DAX file -->
  <pfn url="file:///Users/vahi/Pegasus/work/dax-3.0/blackdiamond_dax.xml" site="local"/>
</file>

<file name="black.dag" >
  <!-- specify the location of the DAG file -->
  <pfn url="file:///Users/vahi/Pegasus/work/dax-3.0/blackdiamond.dag" site="local"/>
</file>

<file name="sites.xml" >
  <!-- specify the location of a site catalog to use for deferred planning -->
  <pfn url="file:///Users/vahi/Pegasus/work/dax-3.0/conf/sites.xml" site="local"/>
</file>
```

The Transformation Catalog Section

The executable section acts as an in-file transformation catalog (TC). Any transformations declared in this section take precedence over the external transformation catalog during planning.

```
<!-- part 1.3: included transformation catalog -->
<executable namespace="example" name="mDiffFit" version="1.0"
            arch="x86_64" os="linux" installed="true" >
  <!-- profiles are optional -->
  <!-- The "stat" namespace is ONLY AN EXAMPLE! -->
  <profile namespace="stat" key="size">5000</profile>
  <profile namespace="stat" key="md5sum">AB454DSSDA4646DS</profile>
  <profile namespace="stat" key="mtime">2010-11-22T10:05:55.470606000-0800</profile>

  <!-- metadata will be supported in 4.6 -->
  <metadata key="timestamp" >/* see above */</metadata>
  <metadata key="origin">ocean</metadata>

  <!-- PFN to by-pass transformation catalog -->
  <!-- The "site" attribute is optional -->
  <pfn url="file:///tmp/mDiffFit"          site="local"/>
  <pfn url="file:///tmp/storage/mDiffFit"  site="local"/>
</executable>

<!-- to be used in compound transformation later -->
<executable namespace="example" name="mDiff" version="1.0"
            arch="x86_64" os="linux" installed="true" >
  <pfn url="file:///tmp/mDiff" site="local"/>
</executable>

<!-- to be used in compound transformation later -->
<executable namespace="example" name="mFitplane" version="1.0"
            arch="x86_64" os="linux" installed="true" >
  <pfn url="file:///tmp/mDiffFitplane"  site="local">
    <profile namespace="stat" key="md5sum">0a9c38b919c7809cb645fc09011588a6</profile>
  </pfn>
  <invoke when="at_end">/path/to/my_send_email some args</invoke>
</executable>

<!-- a more likely example from the black diamond -->
<executable namespace="diamond" name="preprocess" version="2.0"
            arch="x86_64"
            os="linux"
            osversion="2.6.18">
  <pfn url="file:///opt/pegasus/default/bin/keg" site="local" />
</executable>
```

Logical filenames pertaining to a single executables in the transformation catalog use the *executable* element. Any *executable* element features the optional *namespace* attribute, a mandatory *name* attribute, and an optional *version* attribute. The *version* attribute defaults to "1.0" when absent. An executable typically needs additional attributes to describe it properly, like the architecture, OS release and other flags typically seen with transformations, or found in the transformation catalog.

Table 16.2. executable element attributes

attribute	optional?	type	meaning
name	required	string	logical transformation name
namespace	optional	string	namespace of logical transformation, default to *null* value.
version	optional	VersionPattern	version of logical transformation, defaults to "1.0".
installed	optional	boolean	whether to stage the file (false), or not (true, default).
arch	optional	Architecture	restricted set of tokens, see schema definition file.
os	optional	OSType	restricted set of tokens, see schema definition file.
osversion	optional	VersionPattern	kernel version as beginning of `uname -r`.
glibc	optional	VersionPattern	version of libc.

The rationale for giving these flags in the *executable* element header is that PFNs are just identical replicas or instances of a given LFN. If you need a different 32/64 bit-ed-ness or OS release, the underlying PFN would be different, and thus the LFN for it should be different, too.

Note

We are still discussing some details and implications of this decision.

The initial examples come with the same caveats as for the included replica catalog.

Warning

The *metadata* element is not support as of this writing! Details may change in the future.

Similar to the replica catalog, each *executable* element may have 0 or more *profile* elements abstracting away site-specific details, zero or more *metadata* elements, and zero or more *pfn* elements. If there are no *pfn* elements, the transformation must still be searched for in the external transformation catalog. As before, the *pfn* element may have *profile* children-elements, referring to attributes of the physical filename itself.

Each *executable* element may also feature *invoke* elements. These enable notifications at the appropriate point when every job that uses this executable reaches the point of notification. Please refer to the notification section for details and caveats.

The last example above comes from the black diamond example workflow, and presents the kind and extend of attributes you are most likely to see and use in your own workflows.

The Compound Transformation Section

The compound transformation section declares a transformation that comprises multiple plain transformation. You can think of a compound transformation like a script interpreter and the script itself. In order to properly run the application, you must start both, the script interpreter and the script passed to it. The compound transformation helps Pegasus to properly deal with this case, especially when it needs to stage executables.

```
<transformation namespace="example" version="1.0" name="mDiffFit" >
  <uses name="mDiffFit" />
  <uses name="mDiff" namespace="example" version="2.0" />
  <uses name="mFitPlane" />
  <uses name="mDiffFit.config" executable="false" />
</transformation>
```

A *transformation* element declares a set of purely logical entities, executables and config (data) files, that are all required together for the same job. Being purely logical entities, the lookup happens only when the transformation element is referenced (or instantiated) by a job element later on.

The *namespace* and *version* attributes of the transformation element are optional, and provide the defaults for the inner uses elements. They are also essential for matching the transformation with a job.

The *transformation* is made up of 1 or more *uses* element. Each *uses* has a boolean attribute *executable*, `true` by default, or `false` to indicate a data file. The *name* is a mandatory attribute, refering to an LFN declared previously in the File Catalog (*executable* is `false`), Executable Catalog (*executable* is `true`), or to be looked up as necessary at instantiation time. The lookup catalog is determined by the *executable* attribute.

After *uses* elements, any number of *invoke* elements may occur to add a notification each whenever this transformation is instantiated.

The *namespace* and *version* attributes' default values inside *uses* elements are inherited from the *transformation* attributes of the same name. There is no such inheritance for *uses* elements with *executable* attribute of `false`.

Graph Nodes

The nodes in the DAX comprise regular job nodes, already instantiated sub-workflows as dag nodes, and still to be instantiated dax nodes. Each of the graph nodes can has a mandatory *id* attribute. The *id* attribute is currently a restriction of type *NodeIdentifierPattern* type, which is a restriction of the `xs:NMTOKEN` type to letters, digits, hyphen and underscore.

The *level* attribute is deprecated, as the planner will trust its own re-computation more than user input. Please do not use nor produce any *level* attribute.

The *node-label* attribute is optional. It applies to the use-case when every transformation has the same name, but its arguments determine what it really does. In the presence of a *node-label* value, a workflow grapher could use the label value to show graph nodes to the user. It may also come in handy while debugging.

Any job-like graph node has the following set of children elements, as defined in the *AbstractJobType* declaration in the schema definition:

- 0 or 1 *argument* element to declare the command-line of the job's invocation.

- 0 or more *profile* elements to abstract away site-specific or job-specific details.

- 0 or 1 *stdin* element to link a logical file the the job's standard input.

- 0 or 1 *stdout* element to link a logical file to the job's standard output.

- 0 or 1 *stderr* element to link a logical file to the job's standard error.

- 0 or more *uses* elements to declare consumed data files and produced data files.

- 0 or more *invoke* elements to solicit notifications whence a job reaches a certain state in its life-cycle.

Job Nodes

A job element has a number of attributes. In addition to the *id* and *node-label* described in (Graph Nodes)above, the optional *namespace*, mandatory *name* and optional *version* identify the transformation, and provide the lookup handle: first in the DAX's *transformation* elements, then in the *executable* elements, and finally in an external transformation catalog.

```
<!-- part 2: definition of all jobs (at least one) -->
<job id="ID000001" namespace="example" name="mDiffFit" version="1.0"
    node-label="preprocess" >
 <argument>-a top -T 6  -i <file name="f.a"/>  -o <file name="f.b1"/></argument>

 <!-- profiles are optional -->
 <profile namespace="execution" key="site">isi_viz</profile>
 <profile namespace="condor" key="getenv">true</profile>

  <uses name="f.a" link="input" transfer="true" register="true">
      <metadata key="size">1024</metadata>
```

```
        </uses>
    <uses name="f.b" link="output" register="false" transfer="true" type="data" />

    <!-- 'WHEN' enumeration: never, start, on_error, on_success, at_end, all -->
    <!-- PEGASUS_* env-vars: event, status, submit dir, wf/job id, stdout, stderr -->
    <invoke when="start">/path/to arg arg</invoke>
    <invoke when="on_success"><![CDATA[/path/to arg arg]]></invoke>
    <invoke when="at_end"><![CDATA[/path/to arg arg]]></invoke>
  </job>
```

The *argument* element contains the complete command-line that is needed to invoke the executable. The only variable components are logical filenames, as included *file* elements.

The *profile* argument lets you encapsulate site-specific knowledge .

The *stdin*, *stdout* and *stderr* element permits you to connect a stdio file descriptor to a logical filename. Note that you will still have to declare these files in the *uses* section below.

The *uses* element enumerates all the files that the task consumes or produces. While it is not necessary nor required to have all files appear on the command-line, it is imperative that you declare even hidden files that your task requires in this section, so that the proper ancilliary staging- and clean-up tasks can be generated during planning.

The *invoke* element may be specified multiple times, as needed. It has a mandatory when attribute with the following value set:

Table 16.3. invoke element attributes

keyword	job life-cycle state	meaning
never	never	(default). Never notify of anything. This is useful to temporarily disable an existing notifications.
start	submit	create a notification when the job is submitted.
on_error	end	after a job finishes with failure (exit-code != 0).
on_success	end	after a job finishes with success (exit-code == 0).
at_end	end	after a job finishes, regardless of exit-code.
all	always	like start and at_end combined.

Warning

In clustered jobs, a notification can only be sent at the start or end of the clustered job, not for each member.

Each *invoke* is a simple local invocation of an executable or script with the specified arguments. The executable inside the invoke body will see the following environment variables:

Table 16.4. invoke/executable environment variables

variable	job life-cycle state	meaning
PEGASUS_EVENT	always	The value of the when attribute
PEGASUS_STATUS	end	The exit status of the graph node. Only available for end notifications.
PEGASUS_SUBMIT_DIR	always	In which directory to find the job (or workflow).
PEGASUS_JOBID	always	The job (or workflow) identifier. This is potentially more than merely the value of the *id* attribute.

variable	job life-cycle state	meaning
PEGASUS_STDOUT	always	The filename where *stdout* goes. Empty and possibly non-existent at submit time (though we still have the filename). The kickstart record for job nodes.
PEGASUS_STDERR	always	The filename where *stderr* goes. Empty and possibly non-existent at submit time (though we still have the filename).

Generators should use CDATA encapsulated values to the invoke element to minimize interference. Unfortunately, CDATA cannot be nested, so if the user invocation contains a CDATA section, we suggest that they use careful XML-entity escaped strings. The notifications section describes these in further detail.

DAG Nodes

A workflow that has already been concretized, either by an earlier run of Pegasus, or otherwise constructed for DAG-Man execution, can be included into the current workflow using the *dag* element.

```
<dag id="ID000003" name="black.dag" node-label="foo" >
  <profile namespace="dagman" key="DIR">/dag-dir/test</profile>
  <invoke> <!-- optional, should be possible --> </invoke>
  <uses file="sites.xml" link="input" register="false" transfer="true" type="data"/>
</dag>
```

The *id* and *node-label* attributes were described previously. The *name* attribute refers to a file from the File Catalog that provides the actual DAGMan DAG as data content. The *dag* element features optional *profile* elements. These would most likely pertain to the `dagman` and `env` profile namespaces. It should be possible to have the optional *notify* element in the same manner as for jobs.

A graph node that is a dag instead of a job would just use a different submit file generator to create a DAGMan invocation. There can be an *argument* element to modify the command-line passed to DAGMan.

DAX Nodes

A still to be planned workflow incurs an invocation of the Pegasus planner as part of the workflow. This still abstract sub-workflow uses the *dax* element.

```
<dax id="ID000002" name="black.dax" node-label="bar" >
  <profile namespace="env" key="foo">bar</profile>
  <argument>-Xmx1024 -Xms512 -Dpegasus.dir.storage=storagedir  -Dpegasus.dir.exec=execdir -o local
--dir ./datafind -vvvvv --force -s dax_site </argument>
  <invoke> <!-- optional, may not be possible here --> </invoke>
  <uses file="sites.xml" link="input" register="false" transfer="true" type="data" />
</dax>
```

In addition to the *id* and *node-label* attributes, See Graph Nodes. The *name* attribute refers to a file from the File Catalog that provides the to be planned DAX as external file data content. The *dax* element features optional *profile* elements. These would most likely pertain to the `pegasus`, `dagman` and `env` profile namespaces. It may be possible to have the optional *notify* element in the same manner as for jobs.

A graph node that is a *dax* instead of a job would just use yet another submit file and pre-script generator to create a DAGMan invocation. The *argument* string pertains to the command line of the to-be-generated DAGMan invocation.

Inner ADAG Nodes

While completeness would argue to have a recursive nesting of *adag* elements, such recursive nestings are currently not supported, not even in the schema. If you need to nest workflows, please use the *dax* or *dag* element to achieve the same goal.

The Dependency Section

This section describes the dependencies between the jobs.

```
<!-- part 3: list of control-flow dependencies -->
```

```
  <child ref="ID000002">
    <parent ref="ID000001" edge-label="edge1" />
  </child>
  <child ref="ID000003">
    <parent ref="ID000001" edge-label="edge2" />
  </child>
  <child ref="ID000004">
    <parent ref="ID000002" edge-label="edge3" />
    <parent ref="ID000003" edge-label="edge4" />
  </child>
```

Each *child* element contains one or more *parent* element. Either element refers to a *job*, *dag* or *dax* element id attribute using the *ref* attribute. In this version, we relaxed the xs:IDREF constraint in favor of a restriction on the xs:NM-TOKEN type to permit a larger set of identifiers.

The *parent* element has an optional *edge-label* attribute.

Warning

The *edge-label* attribute is currently unused.

Its goal is to annotate edges when drawing workflow graphs.

Closing

As any XML element, the root element needs to be closed.

```
</adag>
```

DAX XML Schema Example

The following code example shows the XML instance document representing the diamond workflow.

```
<?xml version="1.0" encoding="UTF-8"?>
<adag xmlns="http://pegasus.isi.edu/schema/DAX"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://pegasus.isi.edu/schema/DAX http://pegasus.isi.edu/schema/dax-3.6.xsd"
 version="3.6" name="diamond" index="0" count="1">
  <!-- part 1.1: invocations -->
  <invoke when="on_error">/bin/mailx -s &apos;diamond failed&apos; use@some.domain</invoke>

  <!-- part 1.2: included replica catalog -->
  <file name="f.a">
    <pfn url="file:///lfs/voeckler/src/svn/pegasus/trunk/examples/grid-blackdiamond-perl/f.a"
site="local" />
  </file>

  <!-- part 1.3: included transformation catalog -->
  <executable namespace="diamond" name="preprocess" version="2.0" arch="x86_64" os="linux"
installed="false">
    <profile namespace="globus" key="maxtime">2</profile>
    <profile namespace="dagman" key="RETRY">3</profile>
    <pfn url="file:///opt/pegasus/latest/bin/keg" site="local" />
  </executable>
  <executable namespace="diamond" name="analyze" version="2.0" arch="x86_64" os="linux"
installed="false">
    <profile namespace="globus" key="maxtime">2</profile>
    <profile namespace="dagman" key="RETRY">3</profile>
    <pfn url="file:///opt/pegasus/latest/bin/keg" site="local" />
  </executable>
  <executable namespace="diamond" name="findrange" version="2.0" arch="x86_64" os="linux"
installed="false">
    <profile namespace="globus" key="maxtime">2</profile>
    <profile namespace="dagman" key="RETRY">3</profile>
    <pfn url="file:///opt/pegasus/latest/bin/keg" site="local" />
  </executable>

  <!-- part 2: definition of all jobs (at least one) -->
  <job namespace="diamond" name="preprocess" version="2.0" id="ID000001">
    <argument>-a preprocess -T60 -i <file name="f.a" /> -o <file name="f.b1" /> <file name="f.b2" /
></argument>
    <uses name="f.b2" link="output" register="false" transfer="true" />
    <uses name="f.b1" link="output" register="false" transfer="true" />
```

```
    <uses name="f.a" link="input" />
  </job>
  <job namespace="diamond" name="findrange" version="2.0" id="ID000002">
    <argument>-a findrange -T60 -i <file name="f.b1" /> -o <file name="f.c1" /></argument>
    <uses name="f.b1" link="input" register="false" transfer="true" />
    <uses name="f.c1" link="output" register="false" transfer="true" />
  </job>
  <job namespace="diamond" name="findrange" version="2.0" id="ID000003">
    <argument>-a findrange -T60 -i <file name="f.b2" /> -o <file name="f.c2" /></argument>
    <uses name="f.b2" link="input" register="false" transfer="true" />
    <uses name="f.c2" link="output" register="false" transfer="true" />
  </job>
  <job namespace="diamond" name="analyze" version="2.0" id="ID000004">
    <argument>-a analyze -T60 -i <file name="f.c1" /> <file name="f.c2" /> -o <file name="f.d" /></
argument>
    <uses name="f.c2" link="input" register="false" transfer="true" />
    <uses name="f.d" link="output" register="false" transfer="true" />
    <uses name="f.c1" link="input" register="false" transfer="true" />
  </job>

  <!-- part 3: list of control-flow dependencies -->
  <child ref="ID000002">
    <parent ref="ID000001" />
  </child>
  <child ref="ID000003">
    <parent ref="ID000001" />
  </child>
  <child ref="ID000004">
    <parent ref="ID000002" />
    <parent ref="ID000003" />
  </child>
</adag>
```

The above workflow defines the black diamond from the abstract workflow section of the Introduction chapter. It will require minimal configuration, because the catalog sections include all necessary declarations.

The file element defines the location of the required input file in terms of the local machine. Please note that

- The **file** element declares the required input file "f.a" in terms of the local machine. Please note that if you plan the workflow for a remote site, the has to be some way for the file to be staged from the local site to the remote site. While Pegasus will augment the workflow with such ancillary jobs, the site catalog as well as local and remote site have to be set up properlyl. For a locally run workflow you don't need to do anything.

- The **executable** elements declare the same executable keg that is to be run for each the logical transformation in terms of the remote site *futuregrid*. To declare it for a local site, you would have to adjust the *site* attribute's value to `local`. This section also shows that the same executable may come in different guises as transformation.

- The **job** elements define the workflow's logical constituents, the way to invoke the `keg` command, where to put filenames on the commandline, and what files are consumed or produced. In addition to the direction of files, further attributes determine whether to register the file with a replica catalog and whether to transfer it to the output site in case of a product. We are only interested in the final data product "f.d" in this workflow, and not any intermediary files. Typically, you would also want to register the data products in the replica catalog, especially in larger scenarios.

- The **child** elements define the control flow between the jobs.

DAX Generator API

The DAX generating APIs support Java, Perl, Python, and R. This section will show in each language the necessary code, using Pegasus-provided libraries, to generate the diamond DAX example above. There may be minor differences in details, e.g. to show-case certain features, but effectively all generate the same basic diamond.

The Java DAX Generator API

The Java DAX API provided with the Pegasus distribution allows easy creation of complex and huge workflows. This API is used by several applications to generate their abstract DAX. SCEC, which is Southern California Earthquake Center, uses this API in their CyberShake workflow generator to generate huge DAX containing 10’s of thousands of tasks with 100’s of thousands of input and output files. The Java API [javadoc/index.html] is well documented using Javadoc for ADAGs [javadoc/edu/isi/pegasus/planner/dax/ADAG.html] .

The steps involved in creating a DAX using the API are

1. Create a new *ADAG* object

2. Add any metadata attributes associated with the whole workflow.

3. Add any Workflow notification elements

4. Create *File* objects as necessary. You can augment the files with physical information, if you want to include them into your DAX. Otherwise, the physical information is determined from the replica catalog.

5. (Optional) Create *Executable* objects, if you want to include your transformation catalog into your DAX. Otherwise, the translation of a job/task into executable location happens with the transformation catalog.

6. Create a new *Job* object.

7. Add arguments, files, profiles, notifications and other information to the *Job* object

8. Add the job object to the *ADAG* object

9. Repeat step 4-6 as necessary.

10. Add all dependencies to the *ADAG* object.

11. Call the *writeToFile()* method on the *ADAG* object to render the XML DAX file.

An example Java code that generates the diamond dax show above is listed below. This same code can be found in the Pegasus distribution in the `examples/grid-blackdiamond-java` directory as `BlackDiamonDAX.java`:

```
/**
 *  Copyright 2007-2008 University Of Southern California
 *
 *  Licensed under the Apache License, Version 2.0 (the "License");
 *  you may not use this file except in compliance with the License.
 *  You may obtain a copy of the License at
 *
 *  http://www.apache.org/licenses/LICENSE-2.0
 *
 *  Unless required by applicable law or agreed to in writing,
 *  software distributed under the License is distributed on an "AS IS" BASIS,
 *  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *  See the License for the specific language governing permissions and
 *  limitations under the License.
 */

import edu.isi.pegasus.planner.dax.*;


/**
 * An example class to highlight how to use the JAVA DAX API to generate a diamond
 * DAX.
 *
 */
public class Diamond {


    public class Diamond {


    public ADAG generate(String site_handle, String pegasus_location) throws Exception {

        java.io.File cwdFile = new java.io.File (".");
        String cwd = cwdFile.getCanonicalPath();

        ADAG dax = new ADAG("diamond");
        dax.addNotification(Invoke.WHEN.start,"/pegasus/libexec/notification/email -t
 notify@example.com");
        dax.addNotification(Invoke.WHEN.at_end,"/pegasus/libexec/notification/email -t
 notify@example.com");
        dax.addMetadata( "name", "diamond");
        dax.addMetadata( "createdBy", "Karan Vahi");
```

```
        File fa = new File("f.a");
        fa.addPhysicalFile("file://" + cwd + "/f.a", "local");
        fa.addMetaData( "size", "1024" );
        dax.addFile(fa);

        File fb1 = new File("f.b1");
        File fb2 = new File("f.b2");
        File fc1 = new File("f.c1");
        File fc2 = new File("f.c2");
        File fd = new File("f.d");
        fd.setRegister(true);

        Executable preprocess = new Executable("pegasus", "preprocess", "4.0");
        preprocess.setArchitecture(Executable.ARCH.X86).setOS(Executable.OS.LINUX);
        preprocess.setInstalled(true);
        preprocess.addPhysicalFile("file://" + pegasus_location + "/bin/keg", site_handle);
        preprocess.addMetaData( "size", "2048" );

        Executable findrange = new Executable("pegasus", "findrange", "4.0");
        findrange.setArchitecture(Executable.ARCH.X86).setOS(Executable.OS.LINUX);
        findrange.setInstalled(true);
        findrange.addPhysicalFile("file://" + pegasus_location + "/bin/keg", site_handle);

        Executable analyze = new Executable("pegasus", "analyze", "4.0");
        analyze.setArchitecture(Executable.ARCH.X86).setOS(Executable.OS.LINUX);
        analyze.setInstalled(true);
        analyze.addPhysicalFile("file://" + pegasus_location + "/bin/keg", site_handle);

        dax.addExecutable(preprocess).addExecutable(findrange).addExecutable(analyze);

        // Add a preprocess job
        Job j1 = new Job("j1", "pegasus", "preprocess", "4.0");
        j1.addArgument("-a preprocess -T 60 -i ").addArgument(fa);
        j1.addArgument("-o ").addArgument(fb1);
        j1.addArgument(" ").addArgument(fb2);
        j1.addMetadata( "time", "60" );
        j1.uses(fa, File.LINK.INPUT);
        j1.uses(fb1, File.LINK.OUTPUT);
        j1.uses(fb2, File.LINK.OUTPUT);
        j1.addNotification(Invoke.WHEN.start,"/pegasus/libexec/notification/email -t
notify@example.com");
        j1.addNotification(Invoke.WHEN.at_end,"/pegasus/libexec/notification/email -t
notify@example.com");
        dax.addJob(j1);

        // Add left Findrange job
        Job j2 = new Job("j2", "pegasus", "findrange", "4.0");
        j2.addArgument("-a findrange -T 60 -i ").addArgument(fb1);
        j2.addArgument("-o ").addArgument(fc1);
        j2.addMetadata( "time", "60" );
        j2.uses(fb1, File.LINK.INPUT);
        j2.uses(fc1, File.LINK.OUTPUT);
        j2.addNotification(Invoke.WHEN.start,"/pegasus/libexec/notification/email -t
notify@example.com");
        j2.addNotification(Invoke.WHEN.at_end,"/pegasus/libexec/notification/email -t
notify@example.com");
        dax.addJob(j2);

        // Add right Findrange job
        Job j3 = new Job("j3", "pegasus", "findrange", "4.0");
        j3.addArgument("-a findrange -T 60 -i ").addArgument(fb2);
        j3.addArgument("-o ").addArgument(fc2);
        j3.addMetadata( "time", "60" );
        j3.uses(fb2, File.LINK.INPUT);
        j3.uses(fc2, File.LINK.OUTPUT);
        j3.addNotification(Invoke.WHEN.start,"/pegasus/libexec/notification/email -t
notify@example.com");
        j3.addNotification(Invoke.WHEN.at_end,"/pegasus/libexec/notification/email -t
notify@example.com");
        dax.addJob(j3);

        // Add analyze job
        Job j4 = new Job("j4", "pegasus", "analyze", "4.0");
        j4.addArgument("-a analyze -T 60 -i ").addArgument(fc1);
        j4.addArgument(" ").addArgument(fc2);
        j4.addArgument("-o ").addArgument(fd);
        j4.addMetadata( "time", "60" );
```

```
        j4.uses(fc1, File.LINK.INPUT);
        j4.uses(fc2, File.LINK.INPUT);
        j4.uses(fd, File.LINK.OUTPUT);
        j4.addNotification(Invoke.WHEN.start,"/pegasus/libexec/notification/email -t
 notify@example.com");
        j4.addNotification(Invoke.WHEN.at_end,"/pegasus/libexec/notification/email -t
 notify@example.com");
        dax.addJob(j4);

        dax.addDependency("j1", "j2");
        dax.addDependency("j1", "j3");
        dax.addDependency("j2", "j4");
        dax.addDependency("j3", "j4");
        return dax;
    }

    /**
     * Create an example DIAMOND DAX
     * @param args
     */
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java GenerateDiamondDAX  <pegasus_location> ");
            System.exit(1);
        }

        try {
            Diamond diamond = new Diamond();
            String pegasusHome = args[0];
            String site = "TestCluster";
            ADAG dag = diamond.generate( site, pegasusHome );
            dag.writeToSTDOUT();
            //generate(args[0], args[1]).writeToFile(args[2]);
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

Of course, you will have to set up some catalogs and properties to run this example. The details are catpured in the examples directory `examples/grid-blackdiamond-java`.

The Python DAX Generator API

Refer to the auto-generated python documentation [python/] explaining this API.

```python
#!/usr/bin/env python

from Pegasus.DAX3 import *

# Create a DAX
diamond = ADAG("diamond")

# Add some metadata
diamond.metadata("name", "diamond")
diamond.metadata("createdby", "Gideon Juve")

# Add input file to the DAX-level replica catalog
a = File("f.a")
a.addPFN(PFN("gsiftp://site.com/inputs/f.a","site"))
a.metadata("size", "1024")
diamond.addFile(a)

# Add executables to the DAX-level replica catalog
e_preprocess = Executable(namespace="diamond", name="preprocess", version="4.0", os="linux",
 arch="x86_64")
e_preprocess.metadata("size", "2048")
e_preprocess.addPFN(PFN("gsiftp://site.com/bin/preprocess","site"))
diamond.addExecutable(e_preprocess)

e_findrange = Executable(namespace="diamond", name="findrange", version="4.0", os="linux",
 arch="x86_64")
```

```
e_findrange.addPFN(PFN("gsiftp://site.com/bin/findrange","site"))
diamond.addExecutable(e_findrange)

e_analyze = Executable(namespace="diamond", name="analyze", version="4.0", os="linux",
 arch="x86_64")
e_analyze.addPFN(PFN("gsiftp://site.com/bin/analyze","site"))
diamond.addExecutable(e_analyze)

# Add a preprocess job
preprocess = Job(e_preprocess)
preprocess.metadata("time", "60")
b1 = File("f.b1")
b2 = File("f.b2")
preprocess.addArguments("-a preprocess","-T60","-i",a,"-o",b1,b2)
preprocess.uses(a, link=Link.INPUT)
preprocess.uses(b1, link=Link.OUTPUT, transfer=True)
preprocess.uses(b2, link=Link.OUTPUT, transfer=True)
diamond.addJob(preprocess)

# Add left Findrange job
frl = Job(e_findrange)
frl.metadata("time", "60")
c1 = File("f.c1")
frl.addArguments("-a findrange","-T60","-i",b1,"-o",c1)
frl.uses(b1, link=Link.INPUT)
frl.uses(c1, link=Link.OUTPUT, transfer=True)
diamond.addJob(frl)

# Add right Findrange job
frr = Job(e_findrange)
frr.metadata("time", "60")
c2 = File("f.c2")
frr.addArguments("-a findrange","-T60","-i",b2,"-o",c2)
frr.uses(b2, link=Link.INPUT)
frr.uses(c2, link=Link.OUTPUT, transfer=True)
diamond.addJob(frr)

# Add Analyze job
analyze = Job(e_analyze)
analyze.metadata("time", "60")
d = File("f.d")
analyze.addArguments("-a analyze","-T60","-i",c1,c2,"-o",d)
analyze.uses(c1, link=Link.INPUT)
analyze.uses(c2, link=Link.INPUT)
analyze.uses(d, link=Link.OUTPUT, transfer=True, register=True)
diamond.addJob(analyze)

# Add dependencies
diamond.depends(parent=preprocess, child=frl)
diamond.depends(parent=preprocess, child=frr)
diamond.depends(parent=frl, child=analyze)
diamond.depends(parent=frr, child=analyze)

# Write the DAX to stdout
import sys
diamond.writeXML(sys.stdout)

# Write the DAX to a file
f = open("diamond.dax","w")
diamond.writeXML(f)
f.close()
```

The Perl DAX Generator

The Perl API example below can be found in file blackdiamond.pl in directory examples/grid-black-diamond-perl. It requires that you set the environment variable PEGASUS_HOME to the installation directory of Pegasus, and include into PERL5LIB the path to the directory lib/perl of the Pegasus installation. The actual code is longer, and will not require these settings, only the example below does. The Perl API is documented using perldoc [perl/]. For each of the modules you can invoke perldoc, if your PERL5LIB variable is set.

The steps to generate a DAX from Perl are similar to the Java steps. However, since most methods to the classes are deeply within the Perl class modules, the convenience module Perl::DAX::Factory makes most constructors accessible without you needing to type your fingers raw:

1. Create a new *ADAG* object.

2. Create *Job* objects as necessary.

3. As example, the required input file "f.a" is declared as *File* object and linked to the *ADAG* object.

4. The first job arguments and files are filled into the job, and the job is added to the *ADAG* object.

5. Repeat step 4 for the remaining jobs.

6. Add dependencies for all jobs. You have the option of assigning label text to edges, though these are not used (yet).

7. To generate the DAX file, invoke the *toXML()* method on the *ADAG* object. The first argument is an opened file handle or IO::Handle descriptor scalar to write to, the second the default indentation for the root element, and the third the XML namespace to use for elements and attributes. The latter is typically unused unless you want to include your output into another XML document.

```perl
#!/usr/bin/env perl
#
use 5.006;
use strict;
use IO::Handle;
use Cwd;
use File::Spec;
use File::Basename;
use Sys::Hostname;
use POSIX ();

BEGIN { $ENV{'PEGASUS_HOME'} ||= `pegasus-config --nocrlf --home` }
use lib File::Spec->catdir( $ENV{'PEGASUS_HOME'}, 'lib', 'perl' );

use Pegasus::DAX::Factory qw(:all);
use constant NS => 'diamond';

my $adag = newADAG( name => NS );

# Workflow MetaData
my $meta = newMetaData('name', 'diamond');
$adag->addMetaData( $meta );
$adag->metaData( 'createdBy', 'Rajiv Mayani' );

my $job1 = newJob( namespace => NS, name => 'preprocess', version => '2.0' );
my $job2 = newJob( namespace => NS, name => 'findrange', version => '2.0' );
my $job3 = newJob( namespace => NS, name => 'findrange', version => '2.0' );
my $job4 = newJob( namespace => NS, name => 'analyze', version => '2.0' );

# create "f.a" locally
my $fn = "f.a";
open( F, ">$fn" ) || die "FATAL: Unable to open $fn: $!\n";
my @now = gmtime();
printf F "%04u-%02u-%02u %02u:%02u:%02uZ\n",
        $now[5]+1900, $now[4]+1, @now[3,2,1,0];
close F;

my $file = newFile( name => 'f.a' );
$file->addPFN( newPFN( url => 'file://' . Cwd::abs_path($fn),
                       site => 'local' ) );
$file->metaData( 'size', '1024' );
$adag->addFile($file);

# follow this path, if the PEGASUS_HOME was determined
if ( exists $ENV{'PEGASUS_HOME'} ) {
    my $keg = File::Spec->catfile( $ENV{'PEGASUS_HOME'}, 'bin', 'keg' );
    my @os = POSIX::uname();
    # $os[2] =~ s/^(\d+(\.\d+(\.\d+)?)?).*/$1/;  ## create a proper osversion
    $os[4] =~ s/i.86/x86/;

    # add Executable instances to DAX-included TC. This will only work,
    # if we know how to access the keg executable. HOWEVER, for a grid
    # workflow, these entries are not used, and you need to
    # [1] install the work tools remotely
    # [2] create a TC with the proper entries
    if ( -x $keg ) {
        for my $j ( $job1, $job2, $job4 ) {
            my $app = newExecutable( namespace => $j->namespace,
```

```
                                   name => $j->name,
                                   version => $j->version,
                                   installed => 'false',
                                   arch => $os[4],
                                   os => lc($^O) );
                $app->addProfile( 'globus', 'maxtime', '2' );
                $app->addProfile( 'dagman', 'RETRY', '3' );
                $app->addPFN( newPFN( url => "file://$keg", site => 'local' ) );
                $app1->metaData( 'size', '2048' );
                $adag->addExecutable($app);
            }
        }
}

my %hash = ( link => LINK_OUT, register => 'false', transfer => 'true' );
my $fna = newFilename( name => $file->name, link => LINK_IN );
my $fnb1 = newFilename( name => 'f.b1', %hash );
my $fnb2 = newFilename( name => 'f.b2', %hash );
$job1->addArgument( '-a', $job1->name, '-T60', '-i', $fna,
                    '-o', $fnb1, $fnb2 );
$job1->metaData( 'time', '60' );
$adag->addJob($job1);

my $fnc1 = newFilename( name => 'f.c1', %hash );
$fnb1->link( LINK_IN );
$job2->addArgument( '-a', $job2->name, '-T60', '-i', $fnb1,
                    '-o', $fnc1 );
$job2->metaData( 'time', '60' );
$adag->addJob($job2);

my $fnc2 = newFilename( name => 'f.c2', %hash );
$fnb2->link( LINK_IN );
$job3->addArgument( '-a', $job3->name, '-T60', '-i', $fnb2,
                    '-o', $fnc2 );
$job3->metaData( 'time', '60' );
$adag->addJob($job3);
# a convenience function -- you can specify multiple dependents
$adag->addDependency( $job1, $job2, $job3 );

my $fnd = newFilename( name => 'f.d', %hash );
$fnc1->link( LINK_IN );
$fnc2->link( LINK_IN );
$job4->separator('');                   # just to show the difference wrt default
$job4->addArgument( '-a ', $job4->name, ' -T60 -i ', $fnc1, ' ', $fnc2,
                    ' -o ', $fnd );
$job4->metaData( 'time', '60' );
$adag->addJob($job4);
# this is a convenience function adding parents to a child.
# it is clearer than overloading addDependency
$adag->addInverse( $job4, $job2, $job3 );

# workflow level notification in case of failure
# refer to Pegasus::DAX::Invoke for details
my $user = $ENV{USER} || $ENV{LOGNAME} || scalar getpwuid($>);
$adag->invoke( INVOKE_ON_ERROR,
               "/bin/mailx -s 'blackdiamond failed' $user" );

my $xmlns = shift;
$adag->toXML( \*STDOUT, '', $xmlns );
```

The R DAX Generator API

The R DAX API provided with the Pegasus distribution allows easy creation of complex and large workflows in R environments. The API follows the Google' R style guide [http://google.github.io/styleguide/Rguide.xml], and all objects and methods are defined using the *S3* OOP system.

The API can be installed as follows:

1. Installing from source package (.tar.gz) in an R environment:

   ```
   install.packages("/path/to/source/package.tar.gz", repo=NULL)
   ```

   The source package can be obtained using `pegasus-config --r` or from the Pegasus' downloads [http://pegasus.isi.edu/downloads] page.

The R API is well documented using Roxygen [http://http://roxygen.org]. In an R environment, it can be accessed using `help(package=dax3)`. A PDF manual [r/dax3-manual.pdf] is also available.

The steps involved in creating a DAX using the API are

1. Create a new *ADAG* object

2. Add any metadata attributes associated with the whole workflow.

3. Add any Workflow notification elements.

4. Create *File* objects as necessary. You can augment the files with physical information, if you want to include them into your DAX. Otherwise, the physical information is determined from the replica catalog.

5. (Optional) Create *Executable* objects, if you want to include your transformation catalog into your DAX. Otherwise, the translation of a job/task into executable location happens with the transformation catalog.

6. Create a new *Job* object.

7. Add arguments, files, profiles, notifications and other information to the *Job* object

8. Add the job object to the *ADAG* object

9. Repeat step 4-6 as necessary.

10. Add all dependencies to the *ADAG* object.

11. Call the `WriteXML()` method on the *ADAG* object to render the XML DAX file.

An example R code that generates the diamond dax show previously is listed below. A workflow example code can be found in the Pegasus distribution in the `examples/grid-blackdiamond-r` directory as `blackdiamond.R`:

```
#!/usr/bin/Rscript
#
library(dax3)

# Create a DAX
diamond <- ADAG("diamond")

# Add some metadata
diamond <- Metadata(diamond, "name", "diamond")
diamond <- Metadata(diamond, "createdby", "Rafael Ferreira da Silva")

# Add input file to the DAX-level replica catalog
a <- File("f.a")
a <- AddPFN(a, PFN("gsiftp://site.com/inputs/f.a","site"))
a <- Metadata(a, "size", "1024")
diamond <- AddFile(diamond, a)

# Add executables to the DAX-level replica catalog
e_preprocess <- Executable(namespace="diamond", name="preprocess", version="4.0", os="linux",
 arch="x86_64")
e_preprocess <- Metadata(e_preprocess, "size", "2048")
e_preprocess <- AddPFN(e_preprocess, PFN("gsiftp://site.com/bin/preprocess","site"))
diamond <- AddExecutable(diamond, e_preprocess)

e_findrange <- Executable(namespace="diamond", name="findrange", version="4.0", os="linux",
 arch="x86_64")
e_findrange <- AddPFN(e_findrange, PFN("gsiftp://site.com/bin/findrange","site"))
diamond <- AddExecutable(diamond, e_findrange)

e_analyze <- Executable(namespace="diamond", name="analyze", version="4.0", os="linux",
 arch="x86_64")
e_analyze <- AddPFN(e_analyze, PFN("gsiftp://site.com/bin/analyze","site"))
diamond <- AddExecutable(diamond, e_analyze)

# Add a preprocess job
preprocess <- Job(e_preprocess)
preprocess <- Metadata(preprocess, "time", "60")
b1 <- File("f.b1")
b2 <- File("f.b2")
preprocess <- AddArguments(preprocess, list("-a preprocess","-T60","-i",a,"-o",b1,b2))
```

```
preprocess <- Uses(preprocess, a, link=DAX3.Link$INPUT)
preprocess <- Uses(preprocess, b1, link=DAX3.Link$OUTPUT, transfer=TRUE)
preprocess <- Uses(preprocess, b2, link=DAX3.Link$OUTPUT, transfer=TRUE)
diamond <- AddJob(diamond, preprocess)

# Add left Findrange job
frl <- Job(e_findrange)
frl <- Metadata(frl, "time", "60")
c1 <- File("f.c1")
frl <- AddArguments(frl, list("-a findrange","-T60","-i",b1,"-o",c1))
frl <- Uses(frl, b1, link=DAX3.Link$INPUT)
frl <- Uses(frl, c1, link=DAX3.Link$OUTPUT, transfer=TRUE)
diamond <- AddJob(diamond, frl)

# Add right Findrange job
frr <- Job(e_findrange)
frr <- Metadata(frr, "time", "60")
c2 <- File("f.c2")
frr <- AddArguments(frr, list("-a findrange","-T60","-i",b2,"-o",c2))
frr <- Uses(frr, b2, link=DAX3.Link$INPUT)
frr <- Uses(frr, c2, link=DAX3.Link$OUTPUT, transfer=TRUE)
diamond <- AddJob(diamond, frr)

# Add Analyze job
analyze <- Job(e_analyze)
analyze <- Metadata(analyze, "time", "60")
d <- File("f.d")
analyze <- AddArguments(analyze, list("-a analyze","-T60","-i",c1,c2,"-o",d))
analyze <- Uses(analyze, c1, link=DAX3.Link$INPUT)
analyze <- Uses(analyze, c2, link=DAX3.Link$INPUT)
analyze <- Uses(analyze, d, link=DAX3.Link$OUTPUT, transfer=TRUE)
diamond <- AddJob(diamond, analyze)

# Add dependencies
diamond <- Depends(diamond, parent=preprocess, child=frl)
diamond <- Depends(diamond, parent=preprocess, child=frr)
diamond <- Depends(diamond, parent=frl, child=analyze)
diamond <- Depends(diamond, parent=frr, child=analyze)

# Get generated diamond dax
WriteXML(diamond, stdout())
```

DAX Generator without a Pegasus DAX API

If you are using some other scripting or programming environment, you can directly write out the DAX format using the provided schema using any language. For instance, LIGO, the Laser Interferometer Gravitational Wave Observatory, generate their DAX files as XML using their own Python code, not using our provided API.

If you write your own XML, you *must* ensure that the generated XML is well formed and valid with respect to the DAX schema. You can use the **pegasus-dax-validator** to verify the validity of your generated file. Typically, you generate a smallish test file to, validate that your generator creates valid XML using the validator, and then ramp it up to produce the full workflow(s) you want to run. At this point the **pegasus-dax-validator** is a very simple program that will only take exactly one argument, the name of the file to check.The following snippet checks a black-diamond file that uses an improper *osversion* attribute in its *executable* element:

```
$ pegasus-dax-validator blackdiamond.dax
ERROR: cvc-pattern-valid: Value '2.6.18-194.26.1.el5' is not facet-valid
 with respect to pattern '[0-9]+(\.[0-9]+(\.[0-9]+)?)?' for type 'VersionPattern'.
ERROR: cvc-attribute.3: The value '2.6.18-194.26.1.el5' of attribute 'osversion'
 on element 'executable' is not valid with respect to its type, 'VersionPattern'.

0 warnings, 2 errors, and 0 fatal errors detected.
```

We are working on improving this program, e.g. provide output with regards to the line number where the issue occurred. However, it will return with a non-zero exit code whenever errors were detected.

Monitoring

Monitoring REST API allows developers to query a Pegasus workflow's STAMPEDE database.

Resource Definition

Root Workflow

```
{
    "wf_id"             : <int:wf_id>,
    "wf_uuid"           : <string:wf_uuid>,
    "submit_hostname"   : <string:submit_hostname>,
    "submit_dir"        : <string:submit_dir>,
    "planner_arguments" : <string:planner_arguments>,
    "planner_version"   : <string:planner_version>,
    "user"              : <string:user>,
    "grid_dn"           : <string:grid_dn>,
    "dax_label"         : <string:dax_label>,
    "dax_version"       : <string:dax_version>,
    "dax_file"          : <string:dax_file>,
    "dag_file_name"     : <string:dag_file_name>,
    "timestamp"         : <int:timestamp>,
    "workflow_state"    : <object:workflow_state>,
    "_links"            : {
        "workflow" : <href:workflow>
    }
}
```

Workflow

```
{
    "wf_id"             : <int:wf_id>,
    "root_wf_id"        : <int:root_wf_id>,
    "parent_wf_id"      : <int:parent_wf_id>,
    "wf_uuid"           : <string:wf_uuid>,
    "submit_hostname"   : <string:submit_hostname>,
    "submit_dir"        : <string:submit_dir>,
    "planner_arguments" : <string:planner_arguments>,
    "planner_version"   : <string:planner_version>,
    "user"              : <string:user>,
    "grid_dn"           : <string:grid_dn>,
    "dax_label"         : <string:dax_label>,
    "dax_version"       : <string:dax_version>,
    "dax_file"          : <string:dax_file>,
    "dag_file_name"     : <string:dag_file_name>,
    "timestamp"         : <int:timestamp>,
    "_links"            : {
        "workflow_meta"  : <href:workflow_meta>,
        "workflow_state" : <href:workflow_state>,
        "job"            : <href:job>,
        "task"           : <href:task>,
        "host"           : <href:host>,
        "invocation"     : <href:invocation>
    }
}
```

Workflow Metadata

```
{
    "key"    : <string:key>,
    "value"  : <string:value>,
    "_links" : {
        "workflow" : <href:workflow>
    }
}
```

Workflow Files

```
{
    "wf_id"  : <int:wf_id>,
    "lfn_id" : <string:lfn_id>,
    "lfn"    : <string:lfn>,
    "pfns"   : [
        {
            "pfn_id" : <int:pfn_id>,
            "pfn"    : <string:pfn>,
```

```
                "site"    : <string:site>
            }
        ],
        "meta" : [
            {
                "meta_id" : <int:meta_id>,
                "key"     : <string:key>,
                "value"   : <string:value>
            }
        ],
        "_links"      : {
            "workflow" : <href:workflow>
        }
    }
```

Workflow State

```
    {
        "wf_id"         : int:wf_id,
        "state"         : <string:state>,
        "status"        : <int:status>,
        "restart_count" : <int:restart_count>,
        "timestamp"     : <datetime:timestamp>,
        "_links"        : {
            "workflow" : "<href:workflow>"
        }
    }
```

Job

```
    {
        "job_id"      : <int: job_id>,
        "exec_job_id" : <string: exec_job_id>,
        "submit_file" : <string: submit_file>,
        "type_desc"   : <string: type_desc>,
        "max_retries" : <int: max_retries>,
        "clustered"   : <bool: clustered>,
        "task_count"  : <int: task_count>,
        "executable"  : <string: executable>,
        "argv"        : <string: argv>,
        "task_count"  : <int:task_count>,
        "_links"      : {
            "workflow"     : <href:workflow>,
            "task"         : <href:task>,
            "job_instance" : <href:job_instance>
        }
    }
```

Host

```
    {
        "host_id"      : <int:host_id>,
        "site_name"    : <string:site_name>,
        "hostname"     : <string:hostname>,
        "ip"           : <string:ip>,
        "uname"        : <string:uname>,
        "total_memory" : <string:total_memory>,
        "_links"       : {
            "workflow" : <href:workflow>
        }
    }
```

Job State

```
    {
        "job_instance_id"     : <int:job_instance_id>,
        "state"               : <string:state>,
        "jobstate_submit_seq" : <int:jobstate_submit_seq>,
        "timestamp"           : <int:timestamp>,
        "_links"              : {
            "job_instance" : "href:job_instance"
        }
    }
```

Task

```
{
    "task_id"        : <int:task_id>,
    "abs_task_id"    : <string:abs_task_id>,
    "type_desc"      : <string: type_desc>,
    "transformation" : <string:transformation>,
    "argv"           : <string:argv>,
    "_links"         : {
        "workflow"  : <href:workflow>,
        "job"       : <href:job>,
        "task_meta" : <href:task_meta>
    }
}
```

Task Metadata

```
{
    "key"    : <string:key>,
    "value"  : <string:value>,
    "_links" : {
        "task" : <href:task>
    }
}
```

Job Instance

```
{
    "job_instance_id"   : <int:job_instance_id>,
    "host_id"           : <int:host_id>,
    "job_submit_seq"    : <int:job_submit_seq>,
    "sched_id"          : <string:sched_id>,
    "site_name"         : <string:site_name>,
    "user"              : <string:user>,
    "work_dir"          : <string:work_dir>,
    "cluster_start"     : <int:cluster_start>,
    "cluster_duration"  : <int:cluster_duration>,
    "local_duration"    : <int:local_duration>,
    "subwf_id"          : <int:subwf_id>,
    "stdout_text"       : <string:stdout_text>,
    "stderr_text"       : <string:stderr_text>,
    "stdin_file"        : <string:stdin_file>,
    "stdout_file"       : <string:stdout_file>,
    "stderr_file"       : <string:stderr_file>,
    "multiplier_factor" : <int:multiplier_factor>,
    "exitcode"          : <int:exitcode>,
    "_links"            : {
        "job_state"  : <href:job_state>,
        "host"       : <href:host>,
        "invocation" : <href:invocation>,
        "job"        : <href:job>
    }
}
```

Invocation

```
{
    "invocation_id"   : <int:invocation_id>,
    "job_instance_id" : <int:job_instance_id>,
    "abs_task_id"     : <string:abs_task_id>,
    "task_submit_seq" : <int:task_submit_seq>,
    "start_time"      : <int:start_time>,
    "remote_duration" : <int:remote_duration>,
    "remote_cpu_time" : <int:remote_cpu_time>,
    "exitcode"        : <int:exitcode>,
    "transformation"  : <string:transformation>,
    "executable"      : <string:executable>,
    "argv"            : <string:argv>,
    "_links"          : {
        "workflow"     : <href:workflow>,
        "job_instance" : <href:job_instance>
    }
}
```

RC LFN

```
{
    "lfn_id" : <int:pfn_id>,
    "lfn"    : <string:pfn>
}
```

RC PFN

```
{
    "pfn_id" : <int:pfn_id>,
    "pfn"    : <string:pfn>,
    "site"   : <string:site>
}
```

RC Metadata

```
{
    "meta_id" : <int:meta_id>,
    "key"     : <string:key>,
    "value"   : <string:value>
}
```

Endpoints

All URIs are prefixed by **/api/v1/user/<string:user>**.

All endpoints return response with content-type as application/json.

All endpoints support `**pretty-print**` query argument, to return a formatted JSON response.

All endpoints return status code **401** for **Authentication failure**.

All endpoints return status code **403** for **Authorization failure**.

GET /root OR POST /root/query

Returns a collection of the Root Workflow resource.

Table 16.5. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 indexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.

Table 16.6. Returns

Status Code	Description
200	OK
204	No content; when no workflows found.
400	Bad request

GET /root/<m_wf_id>

Returns a Root Workflow resource identified by m_wf_id.

Table 16.7. Returns

Status Code	Description
200	OK
404	Not found

GET /root/<m_wf_id>/workflow OR POST /root/<m_wf_id>/workflow/query

Returns a collection of the Workflow resource.

Table 16.8. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 in-dexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.

Table 16.9. Returns

Status Code	Description
200	OK
204	No content; when no workflows found.
400	Bad request

GET /root/<m_wf_id>/workflow/<wf_id>

Returns a Workflow resource identified by m_wf_id, wf_id.

Table 16.10. Returns

Status Code	Description
200	OK
404	Not found

GET /root/<m_wf_id>/workflow/<wf_id>/meta OR POST /root/<m_wf_id>/workflow/<wf_id>/meta/query

Returns a collection of the WorkflowMetadata resource.

Table 16.11. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 in-dexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.

Table 16.12. Returns

Status Code	Description
200	OK

Status Code	Description
204	No content; when no workflows found.
400	Bad request

GET /root/<m_wf_id>/workflow/<wf_id>/files OR POST /root/<m_wf_id>/ workflow/<wf_id>/files/query

Returns a collection of the WorkflowFiles resource.

Table 16.13. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 in-dexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.

Table 16.14. Returns

Status Code	Description
200	OK
204	No content; when no workflows found.
400	Bad request

GET /root/<m_wf_id>/workflow/<wf_id>/state[;recent=true] OR POST /root/ <m_wf_id>/workflow/<wf_id>/state[;recent=true]/query

Returns a collection of the Workflow State resource.

Table 16.15. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 in-dexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.
recent	Get most recent results. See Recent.

Table 16.16. Returns

Status Code	Description
200	OK
204	No content; when no workflow-state found.
400	Bad request

GET /root/<m_wf_id>/workflow/<wf_id>/host OR POST /root/<m_wf_id>/ workflow/<wf_id>/host/query

Returns a collection of the Host resource.

Table 16.17. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 in-dexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.

Table 16.18. Returns

Status Code	Description
200	OK
204	No content; when no hosts found.
400	Bad request

GET /root/<m_wf_id>/workflow/<wf_id>/host/<host_id>

Returns a Host resource identified by m_wf_id, wf_id, host_id.

Table 16.19. Returns

Status Code	Description
200	OK
404	Not found

GET /root/<m_wf_id>/workflow/<wf_id>/task OR POST /root/<m_wf_id>/ workflow/<wf_id>/task/query

Returns a collection of the Task resource.

Table 16.20. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 in-dexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.

Table 16.21. Returns

Status Code	Description
200	OK
204	No content; when no tasks found.
400	Bad request

GET /root/<m_wf_id>/workflow/<wf_id>/task/<task_id>

Returns a Task resource identified by m_wf_id, wf_id, task_id.

Table 16.22. Returns

Status Code	Description
200	OK
404	Not found

GET /root/<m_wf_id>/workflow/<wf_id>/task/<task_id>/meta OR POST /root/<m_wf_id>/workflow/<wf_id>/task/<task_id>/meta/query

Returns a collection of the TaskMetadata resource.

Table 16.23. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 indexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.

Table 16.24. Returns

Status Code	Description
200	OK
204	No content; when no workflows found.
400	Bad request

GET /root/<m_wf_id>/workflow/<wf_id>/invocation OR POST /root/<m_wf_id>/workflow/<wf_id>/invocation/query

Returns a collection of the Invocation resource.

Table 16.25. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 indexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.

Table 16.26. Returns

Status Code	Description
200	OK
204	No content; when no invocations found.
400	Bad request

GET /root/<m_wf_id>/workflow/<wf_id>/invocation/<invocation_id>

Returns a Invocation resource identified by m_wf_id, wf_id, invocation_id.

Table 16.27. Returns

Status Code	Description
200	OK
404	Not found

GET /root/<m_wf_id>/workflow/<wf_id>/job OR POST /root/<m_wf_id>/workflow/<wf_id>/job/query

Returns a collection of the Job resource.

Table 16.28. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 indexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.

Table 16.29. Returns

Status Code	Description
200	OK
204	No content; when no jobs found.
400	Bad request

GET /root/<m_wf_id>/workflow/<wf_id>/job/<job_id>

Returns a Job resource identified by m_wf_id, wf_id, job_id.

Table 16.30. Returns

Status Code	Description
200	OK
404	Not foun

GET /root/<m_wf_id>/workflow/<wf_id>/job/<job_id>/task OR POST /root/<m_wf_id>/workflow/<wf_id>/job/<job_id>/task/query

Returns a collection of the Task resource.

Table 16.31. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 indexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.

Table 16.32. Returns

Status Code	Description
200	OK
204	No content; when no tasks found.
400	Bad request

GET /root/<m_wf_id>/workflow/<wf_id>/job/<job_id>/job-instance[;recent=true] OR POST /root/<m_wf_id>/workflow/<wf_id>/job/<job_id>/job-instance[;recent=true]/query

Returns a collection of the Job Instance resource.

Table 16.33. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 indexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.
recent	Get most recent results. See Recent.

Table 16.34. Returns

Status Code	Description
200	OK
204	No content; when no job-instances found.
400	Bad request

GET /root/<m_wf_id>/workflow/<wf_id>/job/<job_id>/job-instance/<job_instance_id>

Returns a Job Instance resource identified by m_wf_id, wf_id, job_id, job_instance_id.

Table 16.35. Returns

Status Code	Description
200	OK
404	Not found

GET /root/<m_wf_id>/workflow/<wf_id>/job/<job_id>/job-instance/<job_instance_id>/state[;recent=true] OR POST /root/<m_wf_id>/workflow/<wf_id>/job/<job_id>/job-instance/<job_instance_id>/state[;recent=true]/query

Returns a collection of the Job State resource.

Table 16.36. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 indexed)

Argument	Description
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.
recent	Get most recent results. See Recent.

Table 16.37. Returns

Status Code	Description
200	OK
204	No content; when no job-state found.
400	Bad request

GET /root/<m_wf_id>/workflow/<wf_id>/job/<job_id>/job-instance/<job_instance_id>/invocation OR POST /root/<m_wf_id>/workflow/<wf_id>/job/<job_id>/job-instance/<job_instance_id>/invocation/query

Returns a collection of the Invocation resource.

Table 16.38. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 indexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.

Table 16.39. Returns

Status Code	Description
200	OK
204	No content; when no invocations found.
400	Bad request

POST /batch

Returns an array of responses; one entry for each request.

Batch Request

```
[
    {
        "method" : <string:method>,
        "path"   : <string:path>,
        "body"   : <dict:body>
    },
    {
        "method" : <string:method>,
        "path"   : <string:path>,
        "body"   : <dict:body>
    }
]
```

Batch Response

```
[
    {
        "status"   : <int:status_code>,
        "response" : <string:response>
    },
    {
        "status"   : <int:status_code>,
        "response" : <string:response>
    }
]
```

Table 16.40. Returns

Status Code	Description
207	Multi status

GET /root/<m_wf_id>/workflow/<wf_id>/job/<[running|successful|failed|failing]> OR POST /root/<m_wf_id>/workflow/<wf_id>/job/<[running|successful|failed|failing]>/query

Returns a collection of running, successful, failed, or failing Job resource.

Note: Queries, Sorting can include fields from Job and JobInstance resource.

Table 16.41. Options

Argument	Description
start-index	Return results starting from record <start-index> (0 indexed)
max-results	Return a maximum of <max-results> records
query	Search criteria. See Querying.
order	Order criteria. See Ordering.

Table 16.42. Returns

Status Code	Description
200	OK
204	No content; when no jobs found.
400	Bad request

Querying

Querying is supported through query string argument `**query**`.

Querying is supported only on endpoints returning collections.

Syntax

Query clauses are rudimentary and must follow some rules.

- Supported comparators are =, !=, <, <=, >, >=, LIKE, IN.

- Supported operators are AND, OR.

- Comparision clauses must have the form <FIELDNAME> SPACE <COMPARATOR> SPACE <STRING LITERAL OR INTEGER LITERAL OR FLOAT LITERAL>

- <FIELDNAME> must be prefixed with resource query prefix Example: **r.wf_id** is valid, but **wf_id** is not.

- <FIELDNAMES> which can be used in a query caluse depends on the resource being queries. Example: For end-point /api/v1/user/user-a/root/1/**workflow**/1/**job**/2/**state** query clause can only contain fields from the Job State resource.

- Only exceptions for the previous rules are

  Querying Root Workflow where fields from both Root Workflow and Workflow State can be included.

  Querying the /api/v1/user/user-a/root/1/workflow/1/files where fields from RC LFN, RC PFN, an RC Metadata can be included.

  Views endpoint /api/v1/user/user-a/root/1/workflow/1/job/<[running|successful|failed|failing]> where fields from Job and JobInstance resource can be included.

Example

```
For Root Workflow https://www.domain.com/api/v1/user/user-a/root?query<QUERY>
```

```
Where QUERY can be( r.wf_id >= 5 AND r.planner_version LIKE '4.5%' ) OR ( r.wf_id IN ( 1, 2 ) )
```

Resource - Query Prefix

Table 16.43. Query Prefix

Resource	Query Prefix	Example
Root Workflow	r	r.wf_id
Workflow	w	w.wf_uuid
Workflow Metadata	wm	wm.key
Workflow Files	wf	wf.lfn
Workflow State	ws	ws.state
Job	j	j.type_desc
Host	h	h.site
Job State	js	js.state
Task	t	t.abs_task_id
Task Metadata	tm	tm.value
Job Instance	ji	ji.exitcode
Job	i	i.argv
RC LFN	l	l.lfn
RC PFN	p	p.pfn
RC Metadata	rm	rm.key

Recent

Workflow State, Job State, and Job Instance resources have historical records.

For use cases where developers need to get the most recent record, we set **path** argument `**recent**` to true. Recent argument is always true when requesting for root-workflow's resource.

Ordering

Ordering is supported through query string argument `**order**`.

Ordering is supported only on endpoints returning collections.

Order clause can only contain fields which are part of the resource being returned. Fields may or may not be prefixed by the Resource Query Prefix

Example: Order clause for an endpoint returning a Workflow resource can only contain fields that are part of the Workflow resource.

Syntax

Order clause consists of one or more field names optionally followed by order direction (ASC or DESC), separated by commas.

```
https://www.domain.com/api/v1/user/user-a/root?order=r.submit_hostname ASC, wf_id DESC
```

Examples

Resource - Single

```
$ curl --request GET \
       --user user-a:user-a-password \
       https://www.domain.com/api/v1/user/user-a/root/1/workflow/1?pretty-print=true


HTTP/1.1 200 OK

{
    "wf_id"             : 1,
    "root_wf_id"        : 1,
    "parent_wf_id"      : null,
    "wf_uuid"           : "7193de8c-a28d-4eca-b576-1b1c3c4f668b",
    "submit_hostname"   : "isis.isi.edu",
    "submit_dir"        : "/home/tutorial/submit/",
    "planner_arguments" : "--conf pegasusrc --sites condorpool --output-site local --dir dags --dax
 dax.xml --submit",
    "planner_version"   : "4.5.0dev",
    "user"              : "user-a",
    "grid_dn"           : null,
    "dax_label"         : "hello_world",
    "dax_version"       : "3.5",
    "dax_file"          : "/home/tutorial/hello-world.xml",
    "dag_file_name"     : "hello_world-0.dag",
    "timestamp"         : 1421432530.0,
    "_links"            : {
        "workflow_state" : "/user/user-a/root/1/workflow/1/state",
        "job"            : "/user/user-a/root/1/workflow/1/job",
        "task"           : "/user/user-a/root/1/workflow/1/task",
        "host"           : "/user/user-a/root/1/workflow/1/host",
        "invocation"     : "/user/user-a/root/1/workflow/1/job"
    }
}
```

Resource - Collection

```
$ curl --request POST \
       --user user-a:user-a-password \
       --data 'pretty-print=true' \
       https://www.domain.com/api/v1/user/user-a/root/1/workflow/query

HTTP/1.1 200 OK

{
    "records" : [
        {
            "wf_id"             : 1,
            "root_wf_id"        : 1,
            "parent_wf_id"      : null,
            "wf_uuid"           : "7193de8c-a28d-4eca-b576-1b1c3c4f668b",
            "submit_hostname"   : "isis.isi.edu",
            "submit_dir"        : "/home/tutorial/dags/20150116T102210-0800",
            "planner_arguments" : "--conf pegasusrc --sites condorpool --output-site local --dir
 dags  --dax dax.xml --submit",
            "planner_version"   : "4.5.0dev",
            "user"              : "user-a",
            "grid_dn"           : null,
            "dax_label"         : "hello_world",
            "dax_version"       : "3.5",
```

```
                     "dax_file"           : "/home/tutorial/hello-world.xml",
                     "dag_file_name"      : "hello_world-0.dag",
                     "timestamp"          : 1421432530.0,
                     "_links"             : {
                         "workflow_state" : "/user/user-a/root/1/workflow/1/state",
                         "job"            : "/user/user-a/root/1/workflow/1/job",
                         "task"           : "/user/user-a/root/1/workflow/1/task",
                         "host"           : "/user/user-a/root/1/workflow/1/host",
                         "invocation"     : "/user/user-a/root/1/workflow/1/job"
                     }
                 },
                 {
                     "wf_id"              : 2,
                     "root_wf_id"         : 2,
                     "parent_wf_id"       : null,
                     "wf_uuid"            : "41920a57-7882-4990-854e-658b7a797745",
                     "submit_hostname"    : "isis.isi.edu",
                     "submit_dir"         : "/home/tutorial/dags/20150330T165231-0700",
                     "planner_arguments"  : "--conf pegasusrc --sites condorpool --output-site local --dir
     dags --dax dax.xml --submit",
                     "planner_version"    : "4.5.0dev",
                     "user"               : "user-a",
                     "grid_dn"            : null,
                     "dax_label"          : "hello_world",
                     "dax_version"        : "3.5",
                     "dax_file"           : "/home/tutorial/hello-world.xml",
                     "dag_file_name"      : "hello_world-0.dag",
                     "timestamp"          : 1427759551.0,
                     "_links"             : {
                         "workflow_state" : "/user/user-a/root/2/workflow/1/state",
                         "job"            : "/user/user-a/root/2/workflow/1/job",
                         "task"           : "/user/user-a/root/2/workflow/1/task",
                         "host"           : "/user/user-a/root/2/workflow/1/host",
                         "invocation"     : "/user/user-a/root/2/workflow/1/job"
                     }
                 },
                 {
                     "wf_id"              : 3,
                     "root_wf_id"         : 3,
                     "parent_wf_id"       : null,
                     "wf_uuid"            : "fce67b41-df67-4b3c-8fa4-d77e6e2b9769",
                     "submit_hostname"    : "isis.isi.edu",
                     "submit_dir"         : "/home/tutorial/dags/20150330T170228-0700",
                     "planner_arguments"  : "--conf pegasusrc --sites condorpool --output-site local --dir
     dags --dax dax.xml --submit",
                     "planner_version"    : "4.5.0dev",
                     "user"               : "user-a",
                     "grid_dn"            : null,
                     "dax_label"          : "hello_world",
                     "dax_version"        : "3.5",
                     "dax_file"           : "/home/tutorial/hello-world.xml",
                     "dag_file_name"      : "hello_world-0.dag",
                     "timestamp"          : 1427760148.0,
                     "_links"             : {
                         "workflow_state" : "/user/user-a/root/3/workflow/1/state",
                         "job"            : "/user/user-a/root/3/workflow/1/job",
                         "task"           : "/user/user-a/root/3/workflow/1/task",
                         "host"           : "/user/user-a/root/3/workflow/1/host",
                         "invocation"     : "/user/user-a/root/3/workflow/1/job"
                     }
                 }
             ],
             "_meta"    : {
                 "records_total"     : 3,
                 "records_filtered" : 3
             }
         }
     }
```

Querying

```
$ curl --request GET \
       --get \
       --data-urlencode "pretty-print=true" \
       --data-urlencode "query=w.wf_uuid = '41920a57-7882-4990-854e-658b7a797745'" \
       --user user-a:user-a-password \
```

```
https://www.domain.com/api/v1/user/user-a/root/1/workflow

HTTP/1.1 200 OK

{
    "records" : [
        {
            "wf_id"             : 2,
            "root_wf_id"        : 2,
            "parent_wf_id"      : null,
            "wf_uuid"           : "41920a57-7882-4990-854e-658b7a797745",
            "submit_hostname"   : "isis.isi.edu",
            "submit_dir"        : "/home/tutorial/dags/20150330T165231-0700",
            "planner_arguments" : "--conf pegasusrc --sites condorpool --output-site local --dir
 dags --dax dax.xml --submit",
            "planner_version"   : "4.5.0dev",
            "user"              : "user-a",
            "grid_dn"           : null,
            "dax_label"         : "hello_world",
            "dax_version"       : "3.5",
            "dax_file"          : "/home/tutorial/hello-world.xml",
            "dag_file_name"     : "hello_world-0.dag",
            "timestamp"         : 1427759551.0,
            "_links"            : {
                "workflow_state" : "/user/user-a/root/2/workflow/1/state",
                "job"            : "/user/user-a/root/2/workflow/1/job",
                "task"           : "/user/user-a/root/2/workflow/1/task",
                "host"           : "/user/user-a/root/2/workflow/1/host",
                "invocation"     : "/user/user-a/root/2/workflow/1/job"
            }
        }
    ],
    "_meta"    : {
        "records_total"    : 3,
        "records_filtered" : 1
    }
}
```

Ordering

```
$ curl --request GET \
       --user user-a:user-a-password \
       https://www.domain.com/api/v1/user/user-a/root/1/workflow?pretty-print=true&order=wf_id desc

HTTP/1.1 200 OK

{
    "records" : [
        {
            "wf_id"             : 3,
            "root_wf_id"        : 3,
            "parent_wf_id"      : null,
            "wf_uuid"           : "fce67b41-df67-4b3c-8fa4-d77e6e2b9769",
            "submit_hostname"   : "isis.isi.edu",
            "submit_dir"        : "/home/tutorial/dags/20150330T170228-0700",
            "planner_arguments" : "--conf pegasusrc --sites condorpool --output-site local --dir
 dags --dax dax.xml --submit",
            "planner_version"   : "4.5.0dev",
            "user"              : "user-a",
            "grid_dn"           : null,
            "dax_label"         : "hello_world",
            "dax_version"       : "3.5",
            "dax_file"          : "/home/tutorial/hello-world.xml",
            "dag_file_name"     : "hello_world-0.dag",
            "timestamp"         : 1427760148.0,
            "_links"            : {
                "workflow_state" : "/user/user-a/root/3/workflow/1/state",
                "job"            : "/user/user-a/root/3/workflow/1/job",
                "task"           : "/user/user-a/root/3/workflow/1/task",
                "host"           : "/user/user-a/root/3/workflow/1/host",
                "invocation"     : "/user/user-a/root/3/workflow/1/job"
            }
        },
        {
            "wf_id"             : 2,
            "root_wf_id"        : 2,
```

```
                    "parent_wf_id"       : null,
                    "wf_uuid"            : "41920a57-7882-4990-854e-658b7a797745",
                    "submit_hostname"    : "isis.isi.edu",
                    "submit_dir"         : "/home/tutorial/dags/20150330T165231-0700",
                    "planner_arguments"  : "--conf pegasusrc --sites condorpool --output-site local --dir
  dags --dax dax.xml --submit",
                    "planner_version"    : "4.5.0dev",
                    "user"               : "user-a",
                    "grid_dn"            : null,
                    "dax_label"          : "hello_world",
                    "dax_version"        : "3.5",
                    "dax_file"           : "/home/tutorial/hello-world.xml",
                    "dag_file_name"      : "hello_world-0.dag",
                    "timestamp"          : 1427759551.0,
                    "_links"             : {
                        "workflow_state" : "/user/user-a/root/2/workflow/1/state",
                        "job"            : "/user/user-a/root/2/workflow/1/job",
                        "task"           : "/user/user-a/root/2/workflow/1/task",
                        "host"           : "/user/user-a/root/2/workflow/1/host",
                        "invocation"     : "/user/user-a/root/2/workflow/1/job"
                    }
                },
                {
                    "wf_id"              : 1,
                    "root_wf_id"         : 1,
                    "parent_wf_id"       : null,
                    "wf_uuid"            : "7193de8c-a28d-4eca-b576-1b1c3c4f668b",
                    "submit_hostname"    : "isis.isi.edu",
                    "submit_dir"         : "/home/tutorial/dags/20150116T102210-0800",
                    "planner_arguments"  : "--conf pegasusrc --sites condorpool --output-site local --dir
  dags  --dax dax.xml --submit",
                    "planner_version"    : "4.5.0dev",
                    "user"               : "user-a",
                    "grid_dn"            : null,
                    "dax_label"          : "hello_world",
                    "dax_version"        : "3.5",
                    "dax_file"           : "/home/tutorial/hello-world.xml",
                    "dag_file_name"      : "hello_world-0.dag",
                    "timestamp"          : 1421432530.0,
                    "_links"             : {
                        "workflow_state" : "/user/user-a/root/1/workflow/1/state",
                        "job"            : "/user/user-a/root/1/workflow/1/job",
                        "task"           : "/user/user-a/root/1/workflow/1/task",
                        "host"           : "/user/user-a/root/1/workflow/1/host",
                        "invocation"     : "/user/user-a/root/1/workflow/1/job"
                    }
                }
            ],
            "_meta"    : {
                "records_total"    : 3,
                "records_filtered" : 3
            }
        }
    }
```

Recent

```
$ curl --request GET \
      --user user-a:user-a-password \
      https://www.domain.com/api/v1/user/user-a/root/1/workflow/1/state;recent=true?pretty-
print=true

HTTP/1.1 200 OK

{
    "records": [
        {
            "wf_id": 1,
            "state": "WORKFLOW_TERMINATED",
            "status": 1,
            "restart_count": 3,
            "timestamp": 1421885063.0,
            "_links": {
                "workflow": "/api/v1/user/user-a/root/1/workflow/1"
            }
        }
    ],
```

```
            "_meta": {
                "records_total": 8,
                "records_filtered": 1
            }
        }
    }
```

Batch Request

Example

```
$ curl --request POST \
       --user user-a:user-a-password \
       --header "Content-Type: application/json" \
       --data '[
            {
                "path"   : "/api/v1/user/user-a/root?query=r.wf_id = 1&pretty-print=True",
                "method" : "GET"
            },
            {
                "path"   : "/api/v1/user/user-a/root",
                "method" : "POST",
                "body"   : {
                    "query"        : "r.wf_id = 2",
                    "pretty-print" : "True"
                }
            }
       ]' \
       https://www.domain.com/api/v1/user/user-a/batch

[
    {
        "status"   : 200,
        "response" : {
            "records" : [
                {
                    "wf_id"              : 1,
                    "wf_uuid"            : "7193de8c-a28d-4eca-b576-1b1c3c4f668b",
                    ..
                    "_links"            : {
                        "workflow" : "/api/v1/user/user-a/root/1/workflow"
                    }
                }
            ],
            "_meta"    : {
                "records_total"    : 5,
                "records_filtered" : 1
            }
        }
    },
    {
        "status"   : 200,
        "response" : {
            "records" : [
                {
                    "wf_id"              : 2,
                    "wf_uuid"            : "41920a57-7882-4990-854e-658b7a797745",
                    ..
                    "_links"            : {
                        "workflow" : "/api/v1/user/user-a/root/2/workflow"
                    }
                }
            ],
            "_meta"    : {
                "records_total"    : 5,
                "records_filtered" : 1
            }
        }
    }
]
```

Chapter 17. Command Line Tools

Name

pegasus-analyzer — debugs a workflow.

Synopsis

pegasus-analyzer [**--help**|**-h**] [**--quiet**|**-q**] [**--strict**|**-s**]
        [**--monitord**|**-m**|**-t**] [**--verbose**|**-v**]
        [**--output-dir**|**-o** *output_dir*]
        [**--dag** *dag_filename*] [**--dir**|**-d**|**-i** *input_dir*]
        [**--print**|**-p** *print_options*] [**--type** *workflow_type*]
        [**--debug-job** *job*][**--debug-dir** *debug_dir*]
        [**--local-executable** *local user executable*]
        [**--conf**|**-c** *property_file*] [**--files**]
        [**--top-dir** *dir_name*] [**--recurse**|**-r**]
        [*workflow_directory*]

Description

pegasus-analyzer is a command-line utility for parsing the *jobstate.log* file and reporting successful and failed jobs. When executed without any options, it will query the **SQLite** or **MySQL** database and retrieve failed job information for the particular workflow. When invoked with the **--files** option, it will retrieve information from several log files, isolating jobs that did not complete successfully, and printing their *stdout* and *stderr* so that users can get detailed information about their workflow runs.

Options

-h , **--help**	Prints a usage summary with all the available command-line options.
-q , **--quiet**	Only print the the output and error filenames instead of their contents.
-s , **--strict**	Get jobs' output and error filenames from the job's submit file.
-m , **-t** , **--monitord**	Invoke **pegasus-monitord** before analyzing the *jobstate.log* file. Although **pegasus-analyzer** can be executed during the workflow execution as well as after the workflow has already completed execution, **pegasus-monitord"** is always invoked with the **--replay** option. Since multiple instances of **pegasus-monitord"** should not be executed simultaneously in the same workflow directory, the user should ensure that no other instances of **pegasus-monitord** are running. If the *run_directory* is writable, **pegasus-analyzer** will create a *jobstate.log* file there, rotating an older log, if it is found. If the *run_directory* is not writable (e.g. when the user debugging the workflow is not the same user that ran the workflow), **pegasus-analyzer** will exit and ask the user to provide the **--output-dir** option, in order to provide an alternative location for **pegasus-monitord** log files.
-v , **--verbose**	Sets the log level for **pegasus-analyzer**. If omitted, the default *level* will be set to *WARNING*. When this option is given, the log level is changed to *INFO*. If this option is repeated, the log level will be changed to *DEBUG*.
-o *output_dir* , **--output-dir** *output_dir*	This option provides an alternative location for all monitoring log files for a particular workflow. It is mainly used when an user does not have write privileges to a workflow directory and needs to generate the log files needed by **pegasus-analyzer**. If this option is used in conjunction with the **--monitord** option, it will invoke **pegasus-monitord** using *output_dir* to store all output files. Because workflows can have sub-workflows, **pegasus-monitord** will create its files prepending the workflow *wf_uuid* to each filename. This way, multiple workflow files can be stored in the same directory. **pegasus-analyzer** has built-in logic to find the specific *jobstate.log* file by looking at the workflow *braindump.txt* file first and figuring out the corresponding *wf_uuid*. If *output_dir* does not exist, it will be created.

--dag 'dag_filename	In this option, *dag_filename* specifies the path to the *DAG* file to use. **pegasus-analyzer** will get the directory information from the *dag_filename*. This option overrides the **--dir** option below.
-d *input_dir* , **-i** *input_dir* , **--dir** *input_dir*	Makes **pegasus-analyzer** look for the *jobstate.log* file in the *input_dir* directory. If this option is omitted, **pegasus-analyzer** will look in the current directory.
-p *print_options* , **--print** *print_options*	Tells **pegasus-analyzer** what extra information it should print for failed jobs. *print_options* is a comma-delimited list of options, that include *pre*, *invocation*, and/or *all*, which activates all printing options. With the *pre* option, **pegasus-analyzer** will print the *pre-script* information for failed jobs. For the *invocation* option, **pegasus-analyzer** will print the *invocation* command, so users can manually run the failed job.
--debug-job *job*	When given this option, **pegasus-analyzer** turns on its *debug_mode*, when it can be used to debug a particular Pegasus Lite job. In this mode, **pegasus-analyzer** will create a shell script in the *debug_dir* (see below, for specifying it) and copy all necessary files to this local directory and then execute the job locally.
--debug-dir *debug_dir*	When in *debug_mode*, **pegasus-analyzer** will create a temporary debug directory. Users can give this option in order to specify a particular *debug_dir* directory to be used instead.
--local-executable *local user executable*	When in debug job mode for Pegasus Lite jobs, pegasus-analyzer creates a shell script to execute the Pegasus Lite job locally in a debug directory. The Pegasus Lite script refers to remote user executable path. This option can be used to pass the local path to the user executable on the submit host. If the path to the user executable in the Pegasus Lite job is same as the local installation.
--type *workflow_type*	In this options, users specify what *workflow_type* they want to debug. At this moment, the only *workflow_type* available is **condor** and it is the default value if this option is not specified.
-c *property_file* , **--conf** *property_file*	This option is used to specify an alternative property file, which may contain the path to the database to be used by **pegasus-analyzer**. If this option is not specified, the config file specified in the **braindump.txt** file will take precedence.
--files	This option allows users to run **pegasus-analyzer** using the files in the workflow directory instead of the database as the source of information. **pegasus-analyzer** will output the same information, this option only changes where the data comes from.
--top-dir *dir_name*	This option enables **pegasus-analyzer** to show information about sub-workflows when using the database mode. When debugging a top-level workflow with failures in sub-workflows, the analyzer will automatically print the command users should use to debug a failed sub-workflow. This allows the analyzer to find the database it needs to access.
-r , **--recurse**	This option sets **pegasus-analyzer** to automatically recurse into sub workflows in case of failure. By default, if a workflow has a sub workflow in it, and that sub workflow fails , **pegasus-analyzer** reports that the sub workflow node failed, and lists a command invocation that the user must execute to determine what jobs in the sub workflow failed. If this option is set, then the analyzer automatically issues the command invocation and in addition displays the failed jobs in the sub workflow.

Environment Variables

pegasus-analyzer does not require that any environmental variables be set. It locates its required Python modules based on its own location, and therefore should not be moved outside of Pegasus' bin directory.

Example

The simplest way to use **pegasus-analyzer** is to go to the *run_directory* and invoke the analyzer:

```
$ pegasus-analyzer .
```

which will cause **pegasus-analyzer** to print information about the workflow in the current directory.

pegasus-analyzer output contains a summary, followed by detailed information about each job that either failed, or is in an unknown state. Here is the summary section of the output:

```
**************************Summary***************************

 Total jobs         :      75 (100.00%)
 # jobs succeeded   :      41 (54.67%)
 # jobs failed      :       0 (0.00%)
 # jobs held        :       1 (1.33%)
 # jobs unsubmitted :      33 (44.00%)
 # jobs unknown     :       1 (1.33%)
```

jobs_succeeded are jobs that have completed successfully. *jobs_failed* are jobs that have finished, but that did not complete successfully. *jobs_unsubmitted* are jobs that are listed in the *dag_file*, but no information about them was found in the *jobstate.log* file. *jobs_held* are jobs that were in HTCondor HELD state on the last retry of the job. With default, pegasus added periodic_remove expression with the jobs, a held job can eventually fail. In that case, held job appears as a failed job also. Finally, *jobs_unknown* are jobs that have started, but have not reached completion.

After the summary section, **pegasus-analyzer** will display information about each job in the *job_failed* and *job_unknown* categories.

```
*******************************Held jobs' details*******************************

==================================sleep_j2==================================

        submit file         : sleep_j2.sub
        last_job_instance_id : 7
        reason               :  Error from slot1@corbusier.isi.edu:
                                 STARTER at 128.9.64.188 failed to
                                 send file(s) to
                                 <128.9.64.188:62639>: error reading from
                                 /opt/condor/8.4.8/local.corbusier/execute/dir_76205/f.out:
                                 (errno 2) No such file or directory;
                                 SHADOW failed to receive file(s) from <128.9.64.188:62653>
```

In the above example, the *sleep_j2* job was held, and the analyzer displays the reason why it was held, as determined from the dagman.out file for the workflow. The last_job_instance_id is the database id for the job in the job instance table of the monitoring database.

```
*******************Failed jobs' details*******************

======================findrange_j3==========================

  last state: POST_SCRIPT_FAILURE
        site: local
 submit file: /home/user/diamond-submit/findrange_j3.sub
 output file: /home/user/diamond-submit/findrange_j3.out.000
  error file: /home/user/diamond-submit/findrange_j3.err.000

-------------------Task #1 - Summary----------------------

 site        : local
 hostname    : server-machine.domain.com
 executable  : (null)
 arguments   : -a findrange -T 60 -i f.b2 -o f.c2
 error       : 2
 working dir :
```

In the example above, the *findrange_j3* job has failed, and the analyzer displays information about the job, showing that the job finished with a *POST_SCRIPT_FAILURE*, and lists the *submit*, *output* and *error* files for this job. Whenever **pegasus-analyzer** detects that the output file contains a kickstart record, it will display the breakdown containing each task in the job (in this case we only have one task). Because **pegasus-analyzer** was not invoked with the **--quiet** flag,

it will also display the contents of the *output* and *error* files (or the stdout and stderr sections of the kickstart record), which in this case are both empty.

In the case of *SUBDAG* and *subdax* jobs, **pegasus-analyzer** will indicate it, and show the command needed for the user to debug that sub-workflow. For example:

```
==================subdax_black_ID000009====================

  last state: JOB_FAILURE
        site: local
 submit file: /home/user/run1/subdax_black_ID000009.sub
 output file: /home/user/run1/subdax_black_ID000009.out
  error file: /home/user/run1/subdax_black_ID000009.err
  This job contains sub workflows!
  Please run the command below for more information:
  pegasus-analyzer -d /home/user/run1/blackdiamond_ID000009.000

-----------------subdax_black_ID000009.out-----------------

Executing condor dagman ...

-----------------subdax_black_ID000009.err-----------------
```

tells the user the *subdax_black_ID000009* sub-workflow failed, and that it can be debugged by using the indicated **pegasus-analyzer** command.

See Also

pegasus-status(1), pegasus-monitord(1), pegasus-statistics(1).

Authors

Fabio Silva `<fabio at isi dot edu>`

Karan Vahi `<vahi at isi dot edu>`

Pegasus Team http://pegasus.isi.edu

Name

pegasus-cluster — run a list of applications

Synopsis

pegasus-cluster [-d] [-e | -f] [-S ec] [-s fn] [-R fn] [-n nr] [**inputfile**]

Description

The **pegasus-cluster** tool executes a list of application in the order specified (assuming sequential mode.) It is generally used to do horizontal clustering of independent application, and does not care about any application failures. Such failures should be caught by using **pegasus-kickstart** to start application.

In vertical clustering mode, the *hard failure* mode is encouraged, ending execution as soon as one application fails. When running a complex workflow through **pegasus-cluster** , the order of applications in the input file must be topologically sorted.

Applications are usually using **pegasus-kickstart** to execute. In the **pegasus-kickstart** case, all invocations of **pegasus-kickstart** except the first should add the **pegasus-kickstart** option *-H* to supress repeating the XML preamble and certain other headers of no interest when repeated.

pegasus-cluster permits shell-style quoting. One level of quoting is removed from the arguments. Please note that **pegasus-kickstart** will also remove one level of quoting.

Arguments

-d	This option increases the debug level. Debug message are generated on *stdout* . By default, debugging is minimal.
-e	This flag turns on the old behavior of **pegasus-cluster** to always run everything *and* return success no matter what. The **-e** flag is mutually exclusive with the **-f** flag. By default, all applications are executed regardles of failures. Any detected application failure results in a non-zero exit status from **pegasus-cluster**.
-f	In hard failure mode, as soon as one application fails, either through a non-zero exit code, or by dying on a signal, further execution is stopped. In parallel execution mode, one or more other applications later in the sequence file may have been started already by the time failure is detected. **Pegasus-cluster** will wait for the completion of these applications, but not start new ones. The **-f** flag is mutually exclusive with the **-e** flag. By default, all applications are executed regardless of failures. Any detected application failure results in a non-zero exit status from **pegasus-cluster**.
-h	This option prints the help message and exits the program.
-s fn	This option will send protocol message (for Mei) to the specified file. By default, all message are written to *stdout* .
-R fn	The progress reporting feature, if turned on, will write one event record whenever an application is started, and one event record whenever an application finished. This is to enable tracking of jobs in progress. By default, track logs are not written, unless the environment variable *SEQEXEC_PROGRESS_REPORT* is set. If set, progress reports are appended to the file pointed to by the environment variable.
-S ec	This option is a multi-option, which may be used multiple times. For each given non-zero exit-code of an application, mark it as a form of success. In **-f** mode, this means that **pegasus-cluster** will not fail when seeing this exit code from any application it runs. By default, all non-zero exit code constitute failure.
-n nr	This option determines the amount of parallel execution. Typically, parallel execution is only recommended on multi-core systems, and must be deployed rather carefully, i.e. only completely independent jobs across of whole *inputfile* should ever be attempted to be run in parallel. The argu-

ment **nr** is the number of parallel jobs that should be used. In addition to a non-negative integer, the word *auto* is also understood. When *auto* is specified, **pegasus-cluster** will attempt to automatically determine the number of cores available in the system. Strictly sequential execution, as if *nr* was 1, is the default. If the environment variable *SEQEXEC_CPUS* is set, it will determine the default number of CPUs.

inputfile   The input file specifies a list of application to run, one per line. Comments and empty lines are permitted. The comment character is the octothorpe (#), and extends to the end of line. By default, **pegasus-cluster** uses *stdin* to read the list of applications to execute.

Return Value

The **pegasus-cluster** tool returns 1, if an illegal option was used. It returns 2, if the status file from option **-s** cannot be opened. It returns 3, if the input file cannot be opened. It does *not* return any failure for failed applications in old-exit **-e** mode. In *default* and hard failure **-f** mode, it will return 5 for true failure. The determination of failure is modified by the **-S** option.

All other internal errors being absent, **pegasus-cluster** will always return 0 when run without **-f** . Unlike shell, it will *not* return the last application's exit code. In *default* mode, it will return 5, if any application failed. Unlike shell, it will *not* return the last application's exit code. However, it will execute all applications. The determination of failure is modified by the **-S** flag. In **-f** mode, *pegasus-cluster returns either 0 if all main sequence applications succeeded, or 5 if one failed; or more than one in parallel execution mode. It will run only as long as applications were successful. As before, the *-S flag determines what constitutes a failure.

The **pegasus-cluster** application will also create a small summary on *stdout* for each job, and one for itself, about the success and failure. The field **failed** reports any exit code that was not zero or a signal of death termination. It does *not* include non-zero exit codes that were marked as success using the **-S** option.

Task Summary

Each task executed by **pegasus-cluster** generates a record bracketed by square brackets like this (each entry is broken over two lines for readability):

```
[cluster-task id=1, start="2011-04-27T14:31:25.340-07:00", duration=0.521,
 status=0, line=1, pid=18543, app="/bin/usleep"]
[cluster-task id=2, start="2011-04-27T14:31:25.342-07:00", duration=0.619,
 status=0, line=2, pid=18544, app="/bin/usleep"]
[cluster-task id=3, start="2011-04-27T14:31:25.862-07:00", duration=0.619,
 status=0, line=3, pid=18549, app="/bin/usleep"]
```

Each record is introduced by the string *cluster-task* with the following constituents, where strings are quoted:

id   This is a numerical value for main sequence application, indicating the application's place in the sequence file. The setup task uses the string *setup* , and the cleanup task uses the string *cleanup* .

start   is the ISO 8601 time stamp, with millisecond resolution, when the application was started. This string is quoted.

duration   is the application wall-time duration in seconds, with millisecond resolution.

status   is the *raw* exit status as returned by the *wait* family of system calls. Typically, the exit code is found in the high byte, and the signal of death in the low byte. Typically, 0 indicates a successful execution, and any other value a problem. However, details could differ between systems, and exit codes are only meaningful on the same os and architecture.

line   is the line number where the task was found in the main sequence file. Setup- and cleanup tasks don't have this attribute.

pid   is the process id under which the application had run.

app   is the path to the application that was started. As with the progress record, any **pegasus-kickstart** will be parsed out so that you see the true application.

pegasus-cluster Summary

The final summary of counts is a record bracketed by square brackets like this (broken over two lines for readability):

```
[cluster-summary stat="ok", lines=3, tasks=3, succeeded=3, failed=0, extra=0,
 duration=1.143, start="2011-04-27T14:31:25.338-07:00", pid=18542, app="./seqexec"]
```

The record is introduced by the string *cluster-summary* with the following constituents:

stat	The string *fail* when **pegasus-cluster** would return with an exit status of 5. Concretely, this is any failure in *default* mode, and first failure in **-f** mode. Otherwise, it will always be the string *ok* , if the record is produced.
lines	is the stopping line number of the input sequence file, indicating how far processing got. Up to the number of cores additional lines may have been parsed in case of **-f** mode.
tasks	is the number of tasks processed.
succeeded	is the number of main sequence jobs that succeeded.
failed	is the number of main sequence jobs that failed. The failure condition depends on the **-S** settings, too.
extra	is 0, 1 or 2, depending on the existence of setup- and cleanup jobs.
duration	is the duration in seconds, with millisecond resolution, how long *pegasus-cluster ran.
start	is the start time of **pegasus-cluster** as ISO 8601 time stamp.

See Also

pegasus-kickstart(1)

Caveats

The **-S** option sets success codes globally. It is not possible to activate success codes only for one specific application, and doing so would break the shell compatibility. Due to the global nature, use success codes sparingly as last resort emergency handler. In better plannable environments, you should use an application wrapper instead.

Example

The following shows an example input file to **pegasus-cluster** making use of **pegasus-kickstart** to track applications.

```
#
# mkdir
/path/to/pegasus-kickstart -R HPC -n mkdir /bin/mkdir -m 2755 -p split-corpus split-ne-corpus
#
# drop-dian
/path/to/pegasus-kickstart -H -R HPC -n drop-dian -o '^f-new.plain' /path/to/drop-dian /path/to/f-
tok.plain /path/to/f-tok.NE
#
# split-corpus
/path/to/pegasus-kickstart -H -R HPC -n split-corpus /path/to/split-seq-new.pl 23 f-new.plain split-
corpus/corpus.
#
# split-corpus
/path/to/pegasus-kickstart -H -R HPC -n split-corpus /path/to/split-seq-new.pl 23 /path/to/f-tok.NE
 split-ne-corpus/corpus.
```

Environment Variables

A number of environment variables permits to influence the behavior of **pegasus-cluster** during run-time.

SEQEXEC_PROGRESS_RE-PORT	If this variable is set, and points to a writable file location, progress report records are appended to the file. While care is taken to atomically append

records to the log file, in case concurrent instances of **pegasus-cluster** are running, broken Linux NFS may still garble some content.

SEQEXEC_CPUS	If this variable is set to a non-negative integer, that many CPUs are attempted to be used. The special value *auto* permits to auto-detect the number of CPUs available to **pegasus-cluster** on the system.
SEQEXEC_SETUP	If this variable is set, and contains a single fully-qualified path to an executable and arguments, this executable will be run before any jobs are started. The exit code of this setup job will have no effect upon the main job sequence. Success or failure will not be counted towards the summary.
SEQEXEC_CLEANUP	If this variable is set, and contains a single fully-qualified path to an executable and arguments, this executable will be before **pegasus-cluster** quits. Failure of any previous job will have no effect on the ability to run this job. The exit code of the cleanup job will have no effect on the overall success or failure state. Success or failure will not be counted towards the summary.

History

As you may have noticed, **pegasus-cluster** had the name **seqexec** in previous incantations. We are slowly moving to the new name to avoid clashes in a larger OS installation setting. However, there is no pertinent need to change the internal name, too, as no name clashes are expected.

Authors

Jens-S. Vöckler <voeckler at isi dot edu>

Pegasus **http://pegasus.isi.edu/**

Name

pegasus-configure-glite — install Pegasus-specific glite configuration

Synopsis

pegasus-configure-glite [**GLITE_LOCATION**]

Description

pegasus-configure-glite installs the Pegasus-specific scripts and configuration used by Pegasus to submit jobs via Glite. It installs:

1. *_local_submit_attributes.sh scripts that map Pegasus profiles to batch system-specifc job requirements.

2. Scripts for Moab and modifications to batch_gahp.config to enable Moab job submission.

Options

GLITE_LOCATION   The directory where glite is installed. If this is not provided, then **condor_config_val** will be called to get the value of **GLITE_LOCATION** from the Condor configuration files.

Authors

Pegasus Team http://pegasus.isi.edu

Name

pegasus-config — Can be used to find installed Pegasus tools and libraries.

Synopsis

pegasus-config [**-h**] [**--help**] [**-V**] [**--version**] [**--noeoln**]
[**--perl-dump**] [**--perl-hash**] [**--python-dump**] [**--sh-dump**]
[**--bin**] [**--conf**] [**--java**] [**--perl**] [**--python**]
[**--python-externals**] [**--r**] [**--schema**] [**--classpath**]
[**--local-site**] [**--full-local**]

Description

pegasus-config is used to find locations of Pegasus system components. The tool is used internally in Pegasus and by users who need to find paths for DAX generator libraries and schemas.

Options

-h , **--help**	Prints help and exits.
-V , **--version**	Prints Pegasus version information
--perl-dump	Dumps all settings in perl format as separate variables.
--perl-hash	Dumps all settings in perl format as single perl hash.
--python-dump	Dumps all settings in python format.
--sh-dump	Dumps all settings in shell format.
--bin	Print the directory containing Pegasus binaries.
--conf	Print the directory containing configuration files.
--java	Print the directory containing the jars.
--perl	Print the directory to include into your PERL5LIB.
--python	Print the directory to include into your PYTHONLIB.
--python-externals	Print the directory to the external Python libraries.
--r	Print the path to the R DAX API source package.
--schema	Print the directory containing schemas.
--classpath	Builds a classpath containing the Pegasus jars.
--noeoln	Do not produce a end-of-line after output. This is useful when being called from non-shell backticks in scripts. However, order is important for this option: If you intend to use it, specify it first.
--local-site [d]	Create a site catalog entry for site "local". This is only an XML snippet without root element nor XML headers. The optional argument "d" points to the mount point to use. If not specified, defaults to the user's $HOME directory.
--full-local [d]	Create a complete site catalog with only site "local". The an XML snippet without root element nor XML headers. The optional argument "d" points to the mount point to use. If not specified, defaults to the user's $HOME directory.

Example

To set the PYTHONPATH variable in your shell for using the Python DAX API:

```
export PYTHONPATH=`pegasus-config --python`
```

To set the same path inside Python:

```
config = subprocess.Popen("pegasus-config --python-dump", stdout=subprocess.PIPE,
 shell=True).communicate()[0]
exec config
```

To set the PERL5LIB variable in your shell for using the Perl DAX API:

```
export PERL5LIB=`pegasus-config --perl`
```

To set the same path inside Perl:

```
eval `pegasus-config --perl-dump`;
die("Unable to eval pegasus-config output: $@") if $@;
```

will set variables a number of lexically local-scoped **my** variables with prefix "pegasus_" and expand Perl's search path for this script.

Alternatively, you can fail early and collect all Pegasus-related variables into a single global `%pegasus` variable for convenience:

```
BEGIN {
    eval `pegasus-config --perl-hash`;
    die("Unable to eval pegasus-config output: $@") if $@;
}
```

Author

Pegasus Team http://pegasus.isi.edu

Name

pegasus-dagman — Wrapper around *condor_dagman*. Not to be run by user.

Description

The **pegasus-dagman** is a python wrapper that invokes **pegasus-monitord** and **condor_dagman** both. This is started automatically by **pegasus-submit-dag** and ultimately **condor_submit_dag**. **DO NOT USE DIRECTLY**

Return Value

If the **condor_dagman** and **pegasus-monitord** exit successfully, **pegasus-dagman** exits with 0, else exits with non-zero.

Environment Variables

PATH   The path variable is used to locate binary for **condor_dagman** and **pegasus-monitord**

See Also

pegasus-run(1) pegasus-monitord(1) pegasus-submit-dag(1)

Authors

Gaurang Mehta `<gmehta at isi dot edu>`

Pegasus Team http://pegasus.isi.edu

Name

pegasus-dax-validator — determines if a given DAX file is valid.

Synopsis

pegasus-dax-validator *daxfile* [*verbose*]

Description

The **pegasus-dax-validator** is a simple application that determines, if a given DAX file is valid XML. For this, it parses the file with as many XML validity checks that the Apache Xerces XML parser framework supports.

Options

daxfile   The location of the file containing the DAX.

verbose   If any kind of second argument was specified, not limited to the string *verbose*, the verbose output mode is switched on.

Return Value

If the DAX was parsed successfully, or only *warning's were issued, the exit code is 0. Any 'error* or *fatal error* will result in an exit code of 1.

Additionally, a summary statistics with counts of warnings, errors, and fatal errors will be displayed.

Example

The following shows the parsing of a DAX file that uses the wrong kind of value for certain enumerations. The output shows the errors with the respective line number and column number of the input DAX file, so that one can find and fix them more easily. (The lines in the example were broken to fit the manpage format.)

```
$ pegasus-dax-validator bd.dax
ERROR in line 14, col 110: cvc-enumeration-valid: Value 'i386' is not
 facet-valid with respect to enumeration '[x86, x86_64, ppc, ppc_64,
 ia64, sparcv7, sparcv9, amd64]'. It must be a value from the
 enumeration.
ERROR in line 14, col 110: cvc-attribute.3: The value 'i386' of
 attribute 'arch' on element 'executable' is not valid with respect to
 its type, 'ArchitectureType'.
ERROR in line 14, col 110: cvc-enumeration-valid: Value 'darwin' is
 not facet-valid with respect to enumeration '[aix, sunos, linux, macosx,
 windows]'. It must be a value from the enumeration.
ERROR in line 14, col 110: cvc-attribute.3: The value 'darwin' of
 attribute 'os' on element 'executable' is not valid with respect to
 its type, 'OSType'.

0 warnings, 4 errors, and 0 fatal errors detected.
```

See Also

Apache Xerces-J http://xerces.apache.org/xerces2-j/

Authors

Jens-S. Vöckler `<voeckler at isi dot edu>`

Pegasus Team http://pegasus.isi.edu/

Name

pegasus-db-admin — Manage Pegasus databases.

Synopsis

pegasus-db-admin *COMMAND* [options] [DATABASE_URL]

Description

pegasus-db-admin is used to manage Pegasus databases. The tool can operate directly over a database URL, or can read configuration parameters from the properties file or a submit directory. In the later case, a database type should be provided to indicate which properties should be used to connect to the database. For example, the tool will seek for pegasus.catalog.replica.db.* properties to connect to the JDBCRC database; or seek for pegasus.catalog.master.url (or pegasus.dashboard.output, which is deprecated) property to connect to the MASTER database; or seek for the pegasus.catalog.workflow.url (or pegasus.monitord.output, which is deprecated) property to connect to the WORKFLOW database. If none of these properties are found, the tool will connect to the default database

The **pegasus-db-admin** tool should always be followed by a **COMMAND** listed below. To see the available options for each command, please use the **-h** option after the command. For example: **pegasus-db-admin update -h**

Commands

create DATABASE_URL	Creates Pegasus databases from new or empty databases, or updates current database to the latest version. If a database already exists, it will create a backup (SQLite only) of the current database in the database folder as a 3-digit integer (e.g., workflow.db.000). Pegasus databases can be created by 1) passing a database URL, 2) from the properties file, and 3) from the submit directory. Note that if the properties file or the submit directory is used, a database type (JDBCRC, MASTER, or WORKFLOW) should be provided.
update *[-a] [-V]* **DATA-BASE_URL**	Updates the database to the latest or a given Pegasus version provided with the **-V** or **--version** option. If a database already exists, it will create a backup (SQLite only) of the current database in the database folder as a 3-digit integer (e.g., workflow.db.000). The **-a** or **--all** option will also update databases from completed workflows in the MASTER database.
downgrade *[-a] [-V]* **DATA-BASE_URL**	Downgrades the database to the previous or a given Pegasus version provided with the **-V** or **--version** option. If a database already exists, it will create a backup (SQLite only) of the current database in the database folder as a 3-digit integer (e.g., workflow.db.000). The **-a** or **--all** option will also downgrade databases from completed workflows in the MASTER database.
check *[-V] [-e]* **DATABASE_URL**	Verifies if the database is updated to the latest or a given Pegasus version provided with the **-V** or **--version** option.
version *[-V] [-e]* **DATA-BASE_URL**	Prints the current version of the database.

Global Options

-h , **--help**	Prints a usage summary with all the available command-line options.
-c CONFIG_PROPERTIES , **--conf=CONFIG_PROPERTIES**	Specifies the properties file. This overrides all other property files. Should be used with *-t*.
-s SUBMIT_DIR , **--submit-dir=SUBMIT_DIR**	Specifies the submit directory. Should be used with *-t*.
-t DB_TYPE , **--type=DB_TYPE**	Type of the database (JDBCRC, MASTER, or WORKFLOW). Should be used with *-c* or *-s*.

| **-D PROPERTIES** | Commandline overwrite for properties. Must be in the *prop=val* format. |
| **-d** , **--debug** | Enables debugging. |

Update and Downgrade Options

| **-a** , **--all** | Update/Downgrade all databases of completed workflows in MASTER. |
| **-V PEGASUS_VERSION** , **--ver-sion=PEGASUS_VERSION** | Pegasus version that the database will be updated/downgraded to. |

Check and Version Options

| **-V PEGASUS_VERSION** , **--ver-sion=PEGASUS_VERSION** | Pegasus version that the database will be updated/downgraded to. |
| **-e** , **--version-value** | Show actual version values (an integer number). |

Database Upgrades From Pegasus 4.5.X to Pegasus current version

Databases will be automatically updated when **pegasus-plan** is invoked, but WORKFLOW databases from past runs may not be updated accordingly. Since Pegasus 4.6.0, the pegasus-db-admin tool provides an option to automatically update all databases from completed workflows in the MASTER database. To enable this option, run the following command:

```
$ pegasus-db-admin update -a
Your database has been updated.
Your database is compatible with Pegasus version: 4.7.0

Verifying and updating workflow databases:
21/21

Summary:
Verified/Updated: 21/21
Failed: 0/21
Unable to connect: 0/21
Unable to update (active workflows): 0/21

Log files:
20161006T134415-dbadmin.out (Succeeded operations)
20161006T134415-dbadmin.err (Failed operations)
```

This option generates a log file for succeeded operations, and a log file for failed operations. Each file contains the list of URLs of the succeeded/failed databases.

Note that, if no URL is provided, the tool will create/use a SQLite

Examples

```
# Create a database by passing a database URL.
$ pegasus-db-admin create sqlite:///${HOME}/.pegasus/workflow.db
$ pegasus-db-admin create mysql://localhost:3306/pegasus

# Create a database from the properties file. Note that a database
# type should be provided.
$ pegasus-db-admin create -c pegasus.properties -t MASTER
$ pegasus-db-admin create -c pegasus.properties -t JDBCRC
$ pegasus-db-admin create -c pegasus.properties -t WORKFLOW

# Create a database from the submit directory. Note that a database
# type should be provided.
$ pegasus-db-admin update -s /path/to/submitdir -t WORKFLOW
$ pegasus-db-admin update -s /path/to/submitdir -t MASTER
$ pegasus-db-admin update -s /path/to/submitdir -t JDBCRC
```

```
# Update the database schema by passing a database URL.
$ pegasus-db-admin update sqlite:///${HOME}/.pegasus/workflow.db
$ pegasus-db-admin update mysql://localhost:3306/pegasus

# Update the database schema from the properties file. Note that a
# database type should be provided.
$ pegasus-db-admin update -c pegasus.properties -t MASTER
$ pegasus-db-admin update -c pegasus.properties -t JDBCRC
$ pegasus-db-admin update -c pegasus.properties -t WORKFLOW

# Update the database schema from the submit directory. Note that a
# database type should be provided.
$ pegasus-db-admin update -s /path/to/submitdir -t WORKFLOW
$ pegasus-db-admin update -s /path/to/submitdir -t MASTER
$ pegasus-db-admin update -s /path/to/submitdir -t JDBCRC
```

Troubleshooting

Error 2013: Lost connection to MySQL server during query when dumping table

When updating MySQL databases, pegasus-db-admin uses *mysqldump* to create a backup .sql file for the current database. For very large databases, the dump may fail due to timeout limits of the MySQL database (which are set to 30 seconds for read, and 60 seconds for write). You can change these limits in the *my.cnf* config file by setting the following configuration parameters (the values below are only an example, you should adjust them as you may like):

```
net_read_timeout = 120
net_write_timeout = 900
```

After making these changes to my.cnf you must restart MySQL.

Authors

Rafael Ferreira da Silva <rafsilva@isi.edu>

Pegasus Team http://pegasus.isi.edu

Name

pegasus-em — Submit and monitor ensembles of workflows

Synopsis

pegasus-em *COMMAND* [options] [ARGUMENT…]

Commands

server [-d]	Start the ensemble manager server.
ensembles	List ensembles.
create *ENSEMBLE* [-R *MAX_RUNNING*] [-P *MAX_PLAN-NING*]	Create an ensemble.
pause *ENSEMBLE*	Pause ensemble.
activate *ENSEMBLE*	Activate a paused ensemble.
config *ENSEMBLE* [-R *MAX_RUNNING*] | [-P *MAX_PLANNING*]	Configure an ensemble.
submit *ENSEMBLE.WORKFLOW plan_command* [ARGUMENT…]	Submit a workflow. The command is either **pegasus-plan**, or a shell script that calls **pegasus-plan**. The output of *plan_command* must contain the output of **pegasus-plan**.
workflows *ENSEMBLE* [-l]	List the workflows in an ensemble.
replan *ENSEMBLE.WORKFLOW*	Replan a failed workflow.
rerun *ENSEMBLE.WORKFLOW*	Rerun a failed workflow.
status *ENSEMBLE.WORKFLOW*	Display the status of a workflow.
analyze *ENSEMBLE.WORKFLOW*	Analyze the current state of a workflow.
priority *ENSEMBLE.WORKFLOW* -p *PRIORITY*	Alter the priority of a workflow.

Common Options

-h , **--help**   Print help message

-d , **--debug**   Enable debugging

Create and Config Options

-R *N* , **--max-running** *N*   Maximum number of concurrently running workflows.

-P *N* , **--max-planning** *N*   Maximum number of workflows being planned simultaneously.

Workflows Options

-l , **--long**   Use long listing format.

Authors

Pegasus Team <pegasus@isi.edu>

Name

pegasus-exitcode — Used post-job to check the stdout/stderr for errors

Synopsis

pegasus-exitcode [**-h**][**-r** *rv*][**-n**][**-s** *msg*][**-f** *msg*] *job.out*

Description

pegasus-exitcode is a utility that examines the STDOUT of a job to determine if the job failed, and renames the STDOUT and STDERR files of a job to preserve them in case the job is retried.

Pegasus uses **pegasus-exitcode** as the DAGMan postscript for jobs submitted via Globus GRAM. This tool exists as a workaround to a known problem with Globus and Condor-G where the exitcodes of GRAM jobs are not returned. This is a problem because Pegasus uses the exitcode of a job to determine if the job failed or not.

In order to get around the exitcode problem, Pegasus can wrap GRAM jobs with Kickstart, which records the exitcode of the job in an XML invocation record, which it writes to the job's STDOUT. The STDOUT is transferred from the execution host back to the submit host when the job terminates. After the job terminates, DAGMan runs the job's postscript, which Pegasus sets to be **pegasus-exitcode**. **pegasus-exitcode** looks at the invocation record generated by kickstart to see if the job succeeded or failed. If the invocation record indicates a failure, then **pegasus-exitcode** returns a non-zero result, which indicates to DAGMan that the job has failed. If the invocation record indicates that the job succeeded, then **pegasus-exitcode** returns 0, which tells DAGMan that the job succeeded.

In addition, clustered jobs executed with **pegasus-cluster** or **pegasus-mpi-cluster** will have a [cluster-summary] record in their STDOUT. **pegasus-exitcode** can examine these records to determine if any of the tasks in the clustered job failed.

pegasus-exitcode performs several checks (some optional) to determine whether a job failed or not. These checks include:

1. Is the Condor exitcode non-zero? If so, then the job failed.

2. Is STDOUT empty? If it is empty, then the job failed.

3. Are there any failure messages in the STDOUT or STDERR? If so, the job failed.

4. Are all of the success messages in the STDOUT or STDERR? If not, then the job failed.

5. Does the [cluster-summary] record indicate that the job was successful. If not, then the job failed.

6. Are there any <status> tags with a non-zero value? If there are, then the job failed. Note that, if this is a clustered job, there could be multiple <status> tags, one for each task. If any of them are non-zero, then the job failed.

7. Is there at least one <status> tag with a zero value? There must be at least one successful invocation or the job has failed.

In addition, **pegasus-exitcode** allows the caller to specify the exitcode returned by Condor using the **--return** argument. This can be passed to **pegasus-exitcode** in a DAGMan post script by using the $RETURN variable. If this value is non-zero, then **pegasus-exitcode** returns a non-zero result before performing any other checks. For GRAM jobs, the value of $RETURN will always be 0 regardless of whether the job failed or not.

In addition to checking the success/failure of a job, **pegasus-exitcode** also renames the STDOUT and STDERR files of the job so that if the job is retried, the STDOUT and STDERR of the previous run are not lost. It does this by appending a sequence number to the end of the files. For example, if the STDOUT file is called "job.out", then the first time the job is run **pegasus-exitcode** will rename the file "job.out.000". If the job is run again, then **pegasus-exitcode** sees that "job.out.000" already exists and renames the file "job.out.001". It will continue to rename the file by incrementing the sequence number every time the job is executed.

Options

-h , **--help**   Prints a usage summary with all the available command-line options.

-r *rv* , **--return** *rv*	Return value reported by DAGMan. This can be specified in the DAG using the $RETURN variable. If this is non-zero, then **pegasus-exitcode** immediately fails with a non-zero return value itself. If it is zero, then just rotate the file and don't check for proper kickstart output. This option can be used in cases where kickstart cannot be used (such as pegasus-create-dir) to enable file rotation.
-n , **--no-rename**	Don't rename *job.out* and *job.err* to *.out.XXX* and *.err.XXX*. This option is used primarily for testing.
-f *msg* , **--failure-message** *msg*	Failure message to find in job stdout/stderr. If this message exists in the stdout/stderr of the job, then the job will be considered a failure no matter what other output exists. If multiple failure messages are provided, then none of them can exist in the output or the job is considered a failure.
-s *msg* , **--success-message** *msg*	Success message to find in job stdout/stderr. If this message does not exist in the stdout/stderr of the job, then the job will be considered a failure no matter what other output exists. If multiple success messages are provided, then they must all exist in the output or the job is considered a failure.
-l *filename* , **--log** *filename*	Name of the common log file in which stdout/stderr will be redirected.

Authors

Gideon Juve <`juve@usc.edu`> Rafael Ferreira da Silva <`rafsilva@isi.edu`>

Pegasus Team http://pegasus.isi.edu

Name

pegasus-globus-online-init — Initializes OAuth tokens for Globus Online authentication.

Synopsis

pegasus-globus-online-init [**-h**]
[**--permanent**]

Description

pegasus-globus-online-init initializes OAuth tokens, to be used with Globus Online transfers. It redirects the user to globus website, in order to authorize Pegasus wms to perform transfers with the user's Globus account. By default this tool requests tokens that cannot be refreshed and could potentially expire within a couple of days. In order to provide pegasus with refreshable tokens please use --permanent option. The acquired tokens are placed in globus.conf inside .pegasus folder of the user's home directory.

Note this tool should be used before starting a workflow that relies on Globus Online transfers, unless the user has initialized the tokens with another way or has acquired refreshable tokens previously.

Options

-h , **--help**   Prints a usage summary with all the available command-line options.

--permanent   Requests a refresh token that can be used indefinetely. Access can be revoked from globus web interface (manage consents)

Author

Pegasus Team http://pegasus.isi.edu

Name

pegasus-globus-online — Interfaces with Globus Online for managed transfers.

Synopsis

pegasus-globus-online [**--mkdir**]
[**--transfer**]
[**--remove**]
[**--file** *inputfile*]
[**--debug**]

Description

pegasus-globus-online takes a JSON input from the pegasus-transfer tool and executes the list by interacting with the Globus Online service.

It assumes that the endpoints already have been activated using the web interface. To authenticate with Globus Online, OAuth tokens must be provided inside the JSON that defines the operation. Tokens can be initialized with **pegasus-globus-online-init** tool.

Note that pegasus-globus-online is a tool mostly used internally in Pegasus workflows, in particular by pegasus-transfer.

Options

--mkdir	The input JSON is for a mkdir request
--transfer	The input JSON is for a transfer request
--remove	The input JSON is for a remove request
--file *inputfile*	JSON transfer specification. If not given, stdin will be used.
--debug	Enables debugging output.

Author

Pegasus Team http://pegasus.isi.edu

Name

pegasus-graphviz — Convert a DAX or DAG into a graphviz dot file

Synopsis

pegasus-graphviz [options] FILE

Description

pegasus-graphviz is a tool that generates a graphviz DOT file based on a Pegasus DAX file or DAGMan DAG file.

Options

-h , **--help**	Show the help message
-s , **--nosimplify**	Do not simplify the graph by removing redundant edges. [default: False]
-l *LABEL* , **--label** *LABEL*	What attribute to use for labels. One of *label*,*xform*, or *id*. For *label*, the transformation is used for jobs that have no node-label. [default: label]
-o *FILE* , **--output** *FILE*	Write output to FILE [default: stdout]
-r *XFORM* , **--remove** *XFORM*	Remove jobs from the workflow by transformation name
-W *WIDTH* , **--width** *WIDTH*	Width of the digraph.
-H *HEIGHT* , **--height** *HEIGHT*	Height of the digraph.
-f , **--files**	Include files. This option is only valid for DAX files. [default: false]

Author

Gideon Juve <gideon@isi.edu>

Pegasus Team http://pegasus.isi.edu

Name

pegasus-gridftp — Perform file and directory operations on remote GridFTP servers

Synopsis

pegasus-gridftp ls [options] [*URL...*]
pegasus-gridftp mkdir [options] [*URL...*]
pegasus-gridftp rm [options] [*URL...*]

Description

pegasus-gridftp is a client for Globus GridFTP servers. It enables remote operations on files and directories via the GridFTP protocol. This tool was created to enable more efficient remote directory creation and file cleanup tasks in Pegasus.

Options

Global Options

-v   Turn on verbose output. Verbosity can be increased by specifying multiple -v arguments.

-i *FILE*   Read a list of URLs to operate on from FILE.

rm Options

-f   If the URL does not exist, then ignore the error.

-r   Recursively delete files and directories.

ls Options

-a   List files beginning with a ".".

-l   Create a long-format listing with file size, creation date, type, permissions, etc.

mkdir Options

-p   Create intermediate directories as necessary.

-f   Ignore error if directory already exists

Subcommands

pegasus-gridftp has several subcommands to implement different operations.

ls   The **ls** subcommand lists the details of a file, or the contents of a directory on the remote server.

mkdir   The **mkdir** subcommand creates one or more directories on the remote server.

rm   The **rm** subcommand deletes one or more files and directories from the remote server.

URL Format

All URLs supplied to **pegasus-gridftp** should begin with "gsiftp://".

Configuration

pegasus-gridftp uses the CoG JGlobus API to communicate with remote GridFTP servers. Refer to the CoG JGlobus documentation for information about configuring the API, such as how to specify the user's proxy, etc.

Return Value

pegasus-gridftp returns a zero exist status if the operation is successful. A non-zero exit status is returned in case of failure.

Author

Gideon Juve <`gideon@isi.edu`>

Pegasus Team http://pegasus.isi.edu

Name

pegasus-halt — stops a workflow gracefully, current jobs will finish

Synopsis

pegasus-halt [**rundir**]

Description

pegasus-halt stops a workflow gracefully by allowing the jobs already running to finish on their own. No new jobs will be submitted. Once all jobs have finished, the workflow will stop. A stopped workflow can be restarted with the pegasus-run command.

Another way to remove a workflow is with the pegasus-remove command. The difference is that pegasus-remove will stop running jobs.

Options

rundir   The run directory of the workflow you want to stop

Authors

Pegasus Team http://pegasus.isi.edu

Name

pegasus-init — create a new workflow configuration

Synopsis

pegasus-init WORKFLOW_DIR

Description

pegasus-init creates a new workflow configuration based by asking the user a series of questions. Based on the responses to these questions, **pegasus-init** generates a workflow configuration including a DAX generator, site catalog, properties file, and other artifacts that can be edited to meet the user's needs.

Options

WORK-FLOW_DIR	The directory where you want to create the new workflow configuration.

Authors

Pegasus Team http://pegasus.isi.edu

Name

pegasus-integrity — Generates and verifies data integrity with checksums

Synopsis

pegasus-integrity [**-h**]
[**--generate-sha256** *file*]
[**--verify** *file*]
[**--debug**]

Description

pegasus-integrity either generates a file checksum (usually called from **pegasus-kickstart**) or verifies a checksum for a file using metadata in the current working directory.

Note that pegasus-integrity is a tool mostly used internally in Pegasus workflows, but the tool can be used stand alone as well.

Options

-h , **--help**	Prints a usage summary with all the available command-line options.
--generate-sha256 *file*	Generates a sha256 checksum for a file.
--verify *file*	Verifies a file checksum as compared to what is provided in metadata.
-d , **--debug**	Enables debugging output.

Author

Pegasus Team http://pegasus.isi.edu

Name

pegasus-invoke — invokes a command from a file

Synopsis

pegasus-invoke (**app** | **@fn**) [**arg** | ***@fn** [..]]

Description

The **pegasus-invoke** tool invokes a single application with as many arguments as your Unix permits (128k characters for Linux). Arguments are come from two places, either the command-line as regular arguments, or from a special file, which contains one argument per line.

The **pegasus-invoke** tool became necessary to work around the 4k argument length limit in Condor. It also permits to use arguments inside argument files without worry about shell, Condor or Globus escape necessities. All argument file contents are passed as is, one line per argument entry.

Arguments

-d   This option increases the debug level. Currently, only debugging or no debugging is distinguished. Debug message are generated on *stdout* . By default, debugging is disabled.

-h   This option prints the help message and exits the program.

--   This option stops any option processing. It may only be necessary, if the application is stated on the command-line, and starts with a hyphen itself.The first argument must either be the application to run as fully-specified location (either absolute, or relative to current wd), or a file containing one argument per line. The *PATH* environment variables is **not** used to locate an application. Subsequent arguments may either be specified explicitly on the commandline. Any argument that starts with an at (@) sign is taken to introduce a filename, which contains one argument per line. The textual file may contain long arguments and filenames. However, Unices still impose limits on the maximum length of a directory name, and the maximum length of a file name. These lengths are not checked, because **pegasus-invoke** is oblivious of the application (e.g. what argument is a filename, and what argument is a mere string resembling a filename).

Return Value

The **pegasus-invoke** tool returns 127, if it was unable to find the application. It returns 126, if there was a problem parsing the file. All other exit status, including 126 and 127, come from the application.

See Also

pegasus-kickstart(1)

Example

```
$ echo "/bin/date" > X
$ echo "-Isec" >> X
$ pegasus-invoke @X
2005-11-03T15:07:01-0600
```

Recursion is also possible. Please mind not to use circular inclusions. Also note how duplicating the initial at (@) sign will escape its meaning as inclusion symbol.

```
$ cat test.3
This is test 3

$ cat test.2
/bin/echo
@test.3
@@test.3
```

```
$ pegasus-invoke @test.2
This is test 3 @test.3
```

Restrictions

While the arguments themselves may contain files with arguments to parse, starting with an at (@) sign as before, the maximum recursion limit is 32 levels of inclusions. It is not possible (yet) to use *stdin* as source of inclusion.

History

As you may have noticed, **pegasus-invoke** had the name **invoke** in previous incantations. We are slowly moving to the new name to avoid clashes in a larger OS installation setting. However, there is no pertinent need to change the internal name, too, as no name clashes are expected.

Authors

Mike Wilde <wilde at mcs dot anl dot gov>

Jens-S. Vöckler <voeckler at isi dot edu>

Pegasus **http://pegasus.isi.edu/**

Name

pegasus-keg — kanonical executable for grids

Synopsis

pegasus-keg [-a appname] [-t interval |-T interval] [-l logname]
[-P prefix] [-o fn [..]] [-i fn [..]] [-G sz [..]] [-m memory]
[-C] [-e env [..]] [-p parm [..]] [-u data_unit]

Description

The kanonical executable is a stand-in for regular binaries in a DAG - but not for their arguments. It allows to trace the shape of the execution of a DAG, and thus is an aid to debugging DAG related issues.

Key feature of **pegasus-keg** is that it can copy any number of input files, including the *generator* case, to any number of output files, including the *datasink* case. In addition, it protocols the IPv4 and hostname of the host it ran upon, the current timestamp, and the run time from start til the point of logging the information, the current working directory and some information on the system environment. **pegasus-keg** will also report all input files, the current output files and any requested string and environment value.

The workflow of the Keg tool is as follows: - if **-m** - allocate a memory buffer of the specified amount - if **-i** - read all input files into the memory buffer - if **-o** - write either the input files content (or a generated content if **-G**) to output files - if **-T** - generate CPU load for the specified time period decreased by the time period spent on IO stuff; if the IO stuff time period exceeds the time period specified here the program exits with code status 3 - if **-t** - wait/sleep for the specified time period decreased by time periods spent on IO stuff (and CPU load generating if any); if the time period spent on previous activities exceeds the amount specified here the program exits with code status 3 - if **-l** - write info to the specified log file.

Arguments

The **-e**, **-i**, **-o**, **-p** and **-G** arguments allow lists with arbitrary number of arguments. These options may also occur repeatedly on the command line. The file options may be provided with the special filename - to indicate *stdout* in append mode for writing, or *stdin* for reading. The **-a**, **-l** , **-P** , **-T** and **-t** arguments should only occur a single time with a single argument.

If **pegasus-keg** is called without any arguments, it will display its usage and exit with success.

-a appname
This option allows **pegasus-keg** to display a different name as its applications. This mode of operation is useful in make-believe mode. The default is the basename of *argv[0].*

-e env [..]
This option names any number of environment variables, whose value should be reported as part of the data dump. By default, no environment variables are reported.

-i infile [..]
The **pegasus-keg** binary can work on any number of input files. For each output file, every input file will be opened, and its content copied to the output file. Textual input files are assumed. Each input line is indented by two spaces. The input file content is bracketed between an start and end section, see below. By default, **pegasus-keg** operates in *generator* mode.

-l logfile
The *logfile* is the name of a file to append atomically the self-info, see below. The atomic write guarantees that the multi-line information will not interleave with other processes that simultaneously write to the same file. The default is not to use any log file.

-o outfile [..]
The **pegasus-keg** can work on any number of output files. For each output file, every input file will be opened, and its content copied to the output file. Textual input files are assumed. Each input line is indented by two spaces. The input file content is bracketed between an start and end section, see 2nd example. After all input files are copied, the data dump from this instance of **pegasus-keg** is appended to the output file. Without output files, **pegasus-keg** operates in *data sink* mode. Accept also *<filename>=<filesize><data_unit>* form, where <data_unit> is a character supported by the **-u** switch.

-G size [..]	If you want **pegasus-keg** to generate a lot of output, the generator option will do that for you. Just specify how much, in bytes (but you can change it with **-u** switch), you want. You can specify more than 1 value here if you specify more than 1 output file. Subsequent values specified here will correspond to sizes of subsequent output files. This option is off by default.
-u data_unit	By default, the output data generator (the **-G** switch) generates the specified amount of data in Bytes. You can alter this behavior with this switch. It accepts one of the following characters as *data_unit* value: B for Bytes, K for KiloBytes, M for MegaBytes, and G for GigaBytes.
-C	This option causes **pegasus-keg** to list all environment variables that start with the prefix _*CONDOR* The option is useful, if .B pegasus-keg is run as (part of) a Condor job. This option is off by default.
-p string [..]	Any number of parameters can be reported, without being specific on their content. Effectively, these strings are copied straight from the command line. By default, no extra arguments are shown.
-P prefix	Each line from every input file is indented with a prefix string to visually emphasize the provenance of an input files through multiple instances of **pegasus-keg**. By default, two spaces are used as prefix string.
-t interval	The interval is an amount of sleep time that the **pegasus-keg** executable is to sleep in seconds. This can be used to emulate light work without straining the pool resources. If used together with the **-T** spin option, the sleep interval comes before the spin interval. The default is no sleep time.
-T interval	The interval is an amount of busy spin time that the **pegasus-keg** executable is to simulate intense computation in seconds. The simulation is done by random julia set calculations. This option can be used to emulate an intense work to strain pool resources. If used together with the **-t** sleep option, the sleep interval comes before the spin interval. The default is no spin time.
-m memory	The amount of memory ([MB]) the Keg process should use. This option can be used to emulated application's memory requirements. The default is not to allocate anything.

Return Value

Execution as planned will return 0. The failure to open an input file will return 1, the failure to open an output file, including the log file, will return with exit code 2. If the time spent on IO exceeds the specified time CPU load period with **-T** or the time spent on IO and CPU load exceeds the specified wall time with **-T** the return code will be 3.

Example

The example shows the bracketing of an input file, and the copy produced on the output file. For illustration purposes, the output file is connected to *stdout* :

```
$ date > xx
$ pegasus-keg -i xx -p a b c -o -
--- start xx ----
  Thu May  5 10:55:45 PDT 2011
--- final xx ----
Timestamp Today: 20110505T105552.910-07:00 (1304618152.910;0.000)
Applicationname: pegasus-keg [3661M] @ 128.9.xxx.xxx (xxx.isi.edu)
Current Workdir: /opt/pegasus/default/bin/pegasus-keg
Systemenvironm.: x86_64-Linux 2.6.18-238.9.1.el5
Processor Info.: 4 x Intel(R) Core(TM) i5 CPU         750  @ 2.67GHz @ 2660.068
Load Averages  : 0.298 0.135 0.104
Memory Usage MB: 11970 total, 8089 free, 0 shared, 695 buffered
Swap Usage   MB: 12299 total, 12299 free
Filesystem Info: /                       ext3    62GB total,    20GB avail
Filesystem Info: /lfs/balefire           ext4  1694GB total,  1485GB avail
Filesystem Info: /boot                   ext2   493MB total,   447MB avail
```

```
Output Filename: -
Input Filenames: xx
Other Arguments: a b c
```

Restrictions

The input file must be textual files. The behaviour with binary files is unspecified.

The host address is determined from the primary interface. If there is no active interface besides loopback, the host address will default to 0.0.0.0. If the host address is within a *virtual private network* address range, only *(VPN)* will be displayed as hostname, and no reverse address lookup will be attempted.

The *processor info* line is only available on Linux systems. The line will be missing on other operating systems. Its information is assuming symmetrical multi processing, reflecting the CPU name and speed of the last CPU available in */dev/cpuinfo* .

There is a limit of *4 * page size* to the output buffer of things that .B pegasus-keg can report in its self-info dump. There is no such restriction on the input to output file copy.

Authors

Jens-S. Vöckler <voeckler at isi dot edu>

Mike Wilde

Yong Zhao

Pegasus - http://pegasus.isi.edu/

Name

pegasus-kickstart — remote job wrapper

Synopsis

pegasus-kickstart [**-n** *tr*] [**-N** *dv*] [**-H**] [**-R** *site*] [**-W** | **-w** *dir*]
[**-L** *lbl* **-T** *iso*] [**-s** *p* | @*fn*] [**-S** *p* | @*fn*] [**-i** *fn*]
[**-o** *fn*] [**-e** *fn*] [**-X**] [**-l** *fn sz*] [**-F**] (**-I** *fn* | *app* [*appflags*])
pegasus-kickstart -V

Description

pegasus-kickstart is a wrapper program which manages and monitors the execution of jobs on remote resources.

Sitting in between the remote scheduler and the application process, it is possible for **pegasus-kickstart** to gather additional information about the process' run-time behavior and resource usage, including the exit status of jobs. This information is important for Pegasus invocation tracking as well as detecting Globus GRAM job failures.

pegasus-kickstart allows the optional execution of jobs before and after the main application job that run in chained execution with the main application job. See section **SUBJOBS** for details about this feature.

It also allows stdin, stdout, and stderr to be redirected from/to specific files.

All jobs with relative path specifications to the application are part of search relative to the current working directory (yes, this is unsafe), and by prepending each component from the *PATH* environment variable. The first match is used. Jobs that use absolute pathnames, starting in a slash, are exempt. Using an absolute path to your executable is the safe and recommended option.

pegasus-kickstart rewrites the command line of any job (pre, post and main) with variable substitutions from Unix environment variables. See section **VARIABLE REWRITING** below for details on this feature.

Options

-n *tr*
In order to associate the minimal performance information of the job with the invocation records, the jobs needs to carry which *transformation* was responsible for producing it. The format is the textual notation for fully-qualified definition names, like namespace::name:version, with only the name portion being mandatory.

There is no default. If no value is given, "null" will be reported.

-N *dv*
The jobs may carry which instantiation of a transformation was responsible for producing it. The format is the textual notation for fully-qualified definition names, like namespace::name:version, with only the name portion being mandatory.

There is no default. If no value is given, "null" will be reported.

-H
This option avoids pegasus-kickstart writing the XML preamble (entity), if you need to combine multiple pegasus-kickstart records into one document.

Additionally, if specified, the environment and the resource usage segments will not be written, assuming that a in a concatenated record version, the initial run will have captured those settings.

-R *site*
In order to provide the greater picture, pegasus-kickstart can reflect the site handle (resource identifier) into its output.

There is no default. If no value is given, the attribute will not be generated.

-L *lbl* , **-T** *iso*
These optional arguments denote the workflow label (from DAX) and the workflow's last modification time (from DAX). The label *lbl* can be any sensible string of up to 32 characters, but should use C identifier characters. The timestamp *iso* must be an ISO 8601 compliant time-stamp.

-S *l=p*   If stat information on any file is required *before* any jobs were started, logical to physical file mappings to stat can be passed using the **-S** option. The LFN and PFN are concatenated by an equals (=) sign. The LFN is optional: If no equals sign is found, the argument is taken as sole PFN specification without LFN.

This option may be specified multiple times. To reduce and overcome command line length limits, if the argument is prefixed with an at (@) sign, the argument is taken to be a textual file of LFN to PFN mappings. The optionality mentioned above applies. Each line inside the file argument is the name of a file to stat. Comments (#) and empty lines are permitted.

Each PFN will incur a *statcall* record (element) with attribute *id* set to value *initial*. The optional *lfn* attribute is set to the LFN stat'ed. The filename is part of the *statinfo* record inside.

There is no default.

-s *fn*   If stat information on any file is required *after* all jobs have finished, logical to physical file mappings to stat can be passed using the **-s** option. The LFN and PFN are concatenated by an equals (=) sign. The LFN is optional: If no equals sign is found, the argument is taken as sole PFN specification without LFN.

This option may be specified multiple times. To reduce and overcome commandline length limits, if the argument is prefixed with an at (@) sign, the argument is taken to be a textual file of LFN to PFN mappings. The optionality mentioned above applies. Each line inside the file argument is the name of a file to stat. Comments (#) and empty lines are permitted.

Each PFN will incur a *statcall* record (element) with attribute *id* set to value *final*. The optional *lfn* attribute is set to the LFN stat'ed. The filename is part of the *statinfo* record inside.

There is no default.

-i *fn*   This option allows **pegasus-kickstart** to re-connect the stdin of the application that it starts. Use a single hyphen to share *stdin* with the one provided to **pegasus-kickstart**.

The default is to connect *stdin* to */dev/null*.

-o *fn*   This option allows **pegasus-kickstart** to re-connect the *stdout* of the application that it starts. The mode is used whenever an application produces meaningful results on its *stdout* that need to be tracked by Pegasus. The real *stdout* of Globus jobs is staged via GASS (GT2) or RFT (GT4). The real *stdout* is used to propagate the invocation record back to the submit site. Use the single hyphen to share the application's *stdout* with the one that is provided to **pegasus-kickstart**. In that case, the output from **pegasus-kickstart** will interleave with application output. For this reason, such a mode is not recommended.

In order to provide an un-captured *stdout* as part of the results, it is the default to connect the *stdout* of the application to a temporary file. The content of this temporary file will be transferred as payload data in the **pegasus-kickstart** results. The content size is subject to payload limits, see the **-B** option. If the content grows large, only the last portion will become part of the payload. If the temporary file grows too large, it may flood the worker node's temporary space. The temporary file will be deleted after **pegasus-kickstart** finishes.

If the filename is prefixed with an exclamation point, the file will be opened in append mode instead of overwrite mode. Note that you may need to escape the exclamation point from the shell.

The default is to connect *stdout* to a temporary file.

-e *fn*   This option allows **pegasus-kickstart** to re-connect the *stderr* of the application that it starts. This option is used whenever an application produces meaningful results on *stderr* that needs tracking by Pegasus. The real *stderr* of Globus jobs is staged via GASS (GT2) or RFT (GT4). It is used to propagate abnormal behavior from both, **pegasus-kickstart** and the application that it starts, though its main use is to propagate application dependent data and heartbeats. Use a single hyphen to share *stderr* with the *stderr* that is provided to **pegasus-kickstart**. This is the backward compatible behavior.

In order to provide an un-captured *stderr* as part of the results, by default the *stderr* of the application will be connected to a temporary file. Its content is transferred as payload data in the **pegasus-kickstart** results. If too large, only the last portion will become part of the payload. If the temporary file grows too large, it may flood the worker node's temporary space. The temporary file will be deleted after **pegasus-kickstart** finishes.

If the filename is prefixed with an exclamation point, the file will be opened in append mode instead of overwrite mode. Note that you may need to escape the exclamation point from the shell.

The default is to connect *stderr* to a temporary file.

-l *logfn*   allows to append the performance data to the specified file. Thus, multiple XML documents may end up in the same file, including their XML preamble. *stdout* is normally used to stream back the results. Usually, this is a GASS-staged stream. Use a single hyphen to generate the output on the *stdout* that was provided to **pegasus-kickstart**, the default behavior.

Default is to append the invocation record onto the provided *stdout*.

-w *dir*   permits the explicit setting of a new working directory once pegasus-kickstart is started. This is useful in a remote scheduling environment, when the chosen working directory is not visible on the job submitting host. If the directory does not exist, **pegasus-kickstart** will fail. This option is mutually exclusive with the **-W** *dir* option.

Default is to use the working directory that the application was started in. This is usually set up by a remote scheduling environment.

-W *dir*   permits the explicit creation and setting of a new working directory once pegasus-kickstart is started. This is useful in a remote scheduling environment, when the chosen working directory is not visible on the job submitting host. If the directory does not exist, **pegasus-kickstart** will attempt to create it, and then change into it. Both, creation and directory change may still fail. This option is mutually exclusive with the **-w** *dir* option.

Default is to use the working directory that the application was started in. This is usually set up by a remote scheduling environment.

-X   make an application executable, no matter what. It is a work-around code for a weakness of **globus-url-copy** which does not copy the permissions of the source to the destination. Thus, if an executable is staged-in using GridFTP, it will have the wrong permissions. Specifying the **-X** flag will attempt to change the mode to include the necessary x (and r) bits to make the application executable.

Default is not to change the mode of the application. Note that this feature can be misused by hackers, as it is attempted to call chmod on whatever path is specified.

-B *sz*   Changes the amount of stdout and stderr data to include in the output. The last *sz* bytes of the stdout and stderr of the process will be copied into kickstart's output. All other data will be discarded. The special value *all* can be used to capture all the stdout/stderr of the process. The default is 256KB.

-F   This flag will issue an explicit **fsync()** call on kickstart's own *stdout* file. Typically you won't need this flag. Albeit, certain shared file system situations may improve when adding a flush after the written invocation record.

The default is to just use kickstart's NFS alleviation strategy by locking and unlocking *stdout*.

-I *fn*   In this mode, the application name and any arguments to the application are specified inside of file *fn*. The file contains one argument per line. Escaping from Globus, Condor and shell meta characters is not required. This mode permits to use the maximum possible command line length of the underlying operating system, e.g. 128k for Linux. Using the **-I** mode stops any further command line processing of **pegasus-kickstart** command lines.

Default is to use the *app flags* mode, where the application is specified explicitly on the command-line.

-f	This flag causes kickstart to output full information, including the environment and resource limits under which the job ran, and any useful auxilliary statcalls. If the job fails, then **-f** is implied.
-k *S*	This flag causes kickstart to send the job a SIGTERM if it is still running after S seconds. The default value is 0, which disables the timeout.
-K *S*	This flag causes kickstart to send the job a SIGKILL if it is still running S seconds after recieving a SIGTERM sent as a result of the **-k** flag. The default value is 5. If **-k** is not set, or is set to 0, then this flag is ignored.
-t	This flag causes kickstart to use ptrace() to collect resource usage info for the process by intercepting the process start and stop events. This flag only exists when kickstart is compiled for Linux.
-z	This flag causes kickstart to use ptrace() to intercept system calls and report a list of files accessed and I/O performed. This flag only exists when kickstart is compiled for Linux.
-Z	This flag causes kickstart to use LD_PRELOAD to intercept library calls and report a list of files accessed and I/O performed. This flag only exists when kickstart is compiled for Linux. There are several environment variables documented below that control what file accesses are traced.
-q	This flag causes kickstart to omit the <data> part of the <statcall> records when the job exits successfully. This is designed to reduce the size of the output logs for large workflows.
-c	This flag causes kickstart to output <data> from stdout and stderr as a CDATA section instead of quoting it.
app	The path to the application has to be completely specified. The application is a mandatory option.
appflags	Application may or may not have additional flags.

Return Value

pegasus-kickstart will return the return value of the main job. In addition, the error code 127 signals that the call to exec failed, and 126 that reconnecting the stdio failed. A job failing with the same exit codes is indistinguishable from **pegasus-kickstart** failures.

See Also

pegasus-plan(1), condor_submit_dag(1), condor_submit(1), getrusage(3c).

Subjobs

Subjobs are a new feature and may have a few wrinkles left.

In order to allow specific setups and assertion checks for compute nodes, **pegasus-kickstart** allows the optional execution of a *prejob*. This *prejob* is anything that the remote compute node is capable of executing. For modern Unix systems, this includes #! scripts interpreter invocations, as long as the x bits on the executed file are set. The main job is run if and only if the prejob returned regularly with an exit code of zero.

With similar restrictions, the optional execution of a *postjob* is chained to the success of the main job. The postjob will be run, if the main job terminated normally with an exit code of zero.

In addition, a user may specify a *setup* and a *cleanup* job. The *setup* job sets up the remote execution environment. The *cleanup* job may tear down and clean-up after any job ran. Failure to run the setup job has no impact on subsequent jobs. The cleanup is a job that will even be attempted to run for all failed jobs. No job information is passed. If you need to invoke multiple setup or clean-up jobs, bundle them into a script, and invoke the clean-up script. Failure of the clean-up job is not meant to affect the progress of the remote workflow (DAGMan). This may change in the future.

The setup-, pre-, and post- and cleanup-job run on the same compute node as the main job to execute. However, since they run in separate processes as children of **pegasus-kickstart**, they are unable to influence each others nor the main jobs environment settings.

All jobs and their arguments are subject to variable substitutions as explained in the next section.

To specify the prejob, insert the the application invocation and any optional commandline argument into the environment variable *KICKSTART_PREJOB*. If you are invoking from a shell, you might want to use single quotes to protect against the shell. If you are invoking from Globus, you can append the RSL string feature. From Condor, you can use Condor's notion of environment settings. In Pegasus use the *profile* command to set generic scripts that will work on multiple sites, or the transformation catalog to set environment variables in a pool-specific fashion. Please remember that the execution of the main job is chained to the success of the prejob.

To set up the postjob, use the environment variable *KICKSTART_POSTJOB* to point to an application with potential arguments to execute. The same restrictions as for the prejob apply. Please note that the execution of the post job is chained to the main job.

To provide the independent setup job, use the environment variable *KICKSTART_SETUP*. The exit code of the setup job has no influence on the remaining chain of jobs. To provide an independent cleanup job, use the environment variable *KICKSTART_CLEANUP* to point to an application with possible arguments to execute. The same restrictions as for prejob and postjob apply. The cleanup is run regardless of the exit status of any other jobs.

Variable Rewriting

Variable substitution is a new feature and may have a few wrinkles left.

The variable substitution employs simple rules from the Bourne shell syntax. Simple quoting rules for backslashed characters, double quotes and single quotes are obeyed. Thus, in order to pass a dollar sign to as argument to your job, it must be escaped with a backslash from the variable rewriting.

For pre- and postjobs, double quotes allow the preservation of whitespace and the insertion of special characters like \a (alarm), \b (backspace), \n (newline), \r (carriage return), \t (horizontal tab), and \v (vertical tab). Octal modes are *not* allowed. Variables are still substituted in double quotes. Single quotes inside double quotes have no special meaning.

Inside single quotes, no variables are expanded. The backslash only escapes a single quote or backslash.

Backticks are not supported.

Variables are only substituted once. You cannot have variables in variables. If you need this feature, please request it.

Outside quotes, arguments from the pre- and postjob are split on linear whitespace. The backslash makes the next character verbatim.

Variables that are rewritten must start with a dollar sign either outside quotes or inside double quotes. The dollar may be followed by a valid identifier. A valid identifier starts with a letter or the underscore. A valid identifier may contain further letters, digits or underscores. The identifier is case sensitive.

The alternative use is to enclose the identifier inside curly braces. In this case, almost any character is allowed for the identifier, including whitespace. This is the *only* curly brace expansion. No other Bourne magic involving curly braces is supported.

One of the advantages of variable substitution is, for example, the ability to specify the application as *$HOME/bin/ app1* in the transformation catalog, and thus to gridstart. As long as your home directory on any compute node has a *bin* directory that contains the application, the transformation catalog does not need to care about the true location of the application path on each pool. Even better, an administrator may decide to move your home directory to a different place. As long as the compute node is set up correctly, you don't have to adjust any Pegasus data.

Mind that variable substitution is an expert feature, as some degree of tricky quoting is required to protect substitutable variables and quotes from Globus, Condor and Pegasus in that order. Note that Condor uses the dollar sign for its own variables.

The variable substitution assumptions for the main job differ slightly from the prejob and postjob for technical reasons. The pre- and postjob command lines are passed as one string. However, the main jobs command line is already split into pieces by the time it reaches **pegasus-kickstart**. Thus, any whitespace on the main job's command line must be preserved, and further argument splitting avoided.

It is highly recommended to experiment on the Unix command line with the *echo* and *env* applications to obtain a feeling for the different quoting mechanisms needed to achieve variable substitution.

Example

You can run the **pegasus-kickstart** executable locally to verify that it is functioning well. In the initial phase, the format of the performance data may be slightly adjusted.

```
$ env KICKSTART_PREJOB='/bin/usleep 250000' \\
  KICKSTART_POSTJOB='/bin/date -u' \\
  pegasus-kickstart -l xx \\$PEGASUS_HOME/bin/keg -T1 -o-
$ cat xx
<?xml version="1.0" encoding="ISO-8859-1"?>
  ...
  </statcall>
</invocation>
```

Please take note a few things in the above example:

The output from the postjob is appended to the output of the main job on *stdout*. The output could potentially be separated into different data sections through different temporary files. If you truly need the separation, request that feature.

The log file is reported with a size of zero, because the log file did indeed barely exist at the time the data structure was (re-) initialized. With regular GASS output, it will report the status of the socket file descriptor, though.

The file descriptors reported for the temporary files are from the perspective of **pegasus-kickstart**. Since the temporary files have the close-on-exec flag set, **pegasus-kickstart**s *file descriptors are invisible to the job processes. Still, the 'stdio* of the job processes are connected to the temporary files.

Even this output already appears large. The output may already be too large to guarantee that the append operation on networked pipes (GASS, NFS) are atomically written.

The current format of the performance data is as follows:

Timeouts

Kickstart sets timeouts for the job based on the **-k** and **-K** flags. The **-k** flag sets the time kickstart will wait before it sends the job a SIGTERM, and the **-K** flag sets the time kickstart will wait after delivering a SIGTERM until it delivers a SIGKILL. The **-K** timeout is designed to give the job some time to write a checkpoint, which it can trigger by handling the SIGTERM. If the job runs for longer than the timeout specified using **-k**, then then the job exits with a non-zero exit status.

If the job has KICKSTART_SETUP, KICKSTART_PREJOB, or KICKSTART_POSTJOB, then their runtimes are included in the timeout and they will be sent SIGTERM/SIGKILL in the same manner as the main job. If KICKS-TART_CLEANUP is set, then it will run regardless of whether processes from the other stages were signalled. If KICKSTART_SETUP is specified, and it runs longer than the timeout, then it will be signalled, and the other stages will be skipped.

Output Format

Refer to https://pegasus.isi.edu/documentation/schemas/iv-2.2/iv-2.2.html for an up-to-date description of elements and their attributes. Check with https://pegasus.isi.edu/documentation for invocation schemas with a higher version number.

Restrictions

There is no version for the Condor *standard* universe. It is simply not possible within the constraints of Condor.

Due to its very nature, **pegasus-kickstart** will also prove difficult to port outside the Unix environment.

Any of the pre-, main-, cleanup and postjob are unable to influence one another's visible environment.

Do not use a Pegasus transformation with just the name *null* and no namespace nor version.

First Condor, and then Unix, place a limit on the length of the command line. The additional space required for the gridstart invocation may silently overflow the maximum space, and cause applications to fail. If you suspect to work with many argument, try an argument-file based approach.

A job failing with exit code 126 or 127 is indistinguishable from **pegasus-kickstart** failing with the same exit codes. Sometimes, careful examination of the returned data can help.

If the logfile is collected into a shared file, due to the size of the data, simultaneous appends on a shared filesystem from different machines may still mangle data. Currently, file locking is not even attempted, although all data is written atomically from the perspective of **pegasus-kickstart**.

The upper limit of characters of command line characters is currently not checked by **pegasus-kickstart**. Thus, some variable substitutions could potentially result in a command line that is larger than permissible.

If the output or error file is opened in append mode, but the application decides to truncate its output file, as in the above example by opening */dev/fd/1* inside *keg*, the resulting file will still be truncated. This is correct behavior, but sometimes not obvious.

Files

/usr/share/pegasus/schema/iv-2.2.xsd	is the suggested location of the latest XML schema describing the data on the submit host.

Metadata

Kickstart creates a file to which the job should write metadata "key=value" pairs. The contents of the file are inserted into the invocation record by Kickstart, and transferred with the job's stdio. If the job is run under Pegasus, then pegasus-monitord will parse this metadata and merge it with the metadata for the job in the Pegasus workflow database. Kickstart uses the environment variable **KICKSTART_METADATA** to tell the job to which file it should write its metadata.

Environment Variables

Note: Pegasus 4.6 deprecated the "GRIDSTART_" prefix for environment variables and replaced it with "KICKSTART_". The "GRIDSTART_" versions of the old variables should still work.

KICKSTART_TMP	is the hightest priority to look for a temporary directory, if specified. This rather special variable was introduced to overcome some peculiarities with the FNAL cluster.
TMP	is the next hightest priority to look for a temporary directory, if specified.
TEMP	is the next priority for an environment variable denoting a temporary files directory.
TMPDIR	is next in the checklist. If none of these are found, either the *stdio* definition *P_tmpdir* is taken, or the fixed string */tmp*.
KICKSTART_SETUP	contains a string that starts a job to be executed unconditionally before any other jobs, see above for a detailed description.
KICKSTART_PREJOB	contains a string that starts a job to be executed before the main job, see above for a detailed description.
KICKSTART_POSTJOB	contains a string that starts a job to be executed conditionally after the main job, see above for a detailed description.
KICKSTART_CLEANUP	contains a string that starts a job to be executed unconditionally after any of the previous jobs, see above for a detailed description.

KICKS-TART_PREPEND_PATH	the value of this variable is prepended to the PATH variable seen by Kickstart and passed to the job. The modified PATH is also used to look up executables for the main job and any pre/post/setup/cleanup jobs.
KICKSTART_WRAPPER	the value of this variable is prepended to the job arguments. It can be used to wrap the task with a wrapper or launcher. For example, you can set it to "mpiexec -n 128" to run an MPI job, or you can set it to "tau_exec" to profile the job with TAU.

KICKSTART_TRACE_ALL If this variable is set, then the **-Z** option will trace everything, including stdio and directories. By default, stdio and directories are ignored.

KICKSTART_TRACE_CWD If this variable is set, then the **-Z** option will only trace files in the current working directory of the process.

KICKS-TART_TRACE_MATCH	If this variable is set, then the **-Z** option will only trace files that match one of the patterns specified. The value of this variable should be a list of **fnmatch()** patterns separated by $:$.

KICKSTART_TRACE_IGNORE This is the inverse of **KICKSTART_TRACE_MATCH**. Any files matching one of the patterns will be ignored, and all other files will be traced.

KICKSTART_METADATA Kickstart passes this environment variable to the job. The value of the variable is the path to the metadata file to which the job should write its metadata. See the **METADATA** section for more information.

History

As you may have noticed, **pegasus-kickstart** had the name **kickstart** in previous incantations. We are slowly moving to the new name to avoid clashes in a larger OS installation setting. However, there is no pertinent need to change the internal name, too, as no name clashes are expected.

Authors

Michael Milligan <`mbmillig@uchicago.edu`>

Mike Wilde <`wilde@mcs.anl.gov`>

Yong Zhao <`yongzh@cs.uchicago.edu`>

Jens-S. Vöckler <`voeckler@isi.edu`>

Gideon Juve <`gideon@isi.edu`>

Pegasus Team http://pegasus.isi.edu/

Name

pegasus-metadata — Query metadata collected for Pegasus workflows

Synopsis

pegasus-metadata *COMMAND* [options] <SUBMIT_DIR>

Description

pegasus-metadata is a tool to query metadata collected for a workflow. The tools can query workflow, task, and file metadata.

Commands

workflow   Query metadata for a workflow

task   Query metadata for a workflow task

file   Query metadata for files

Global Options

-v , **--verbose**   Increase logging verbosity

-h , **--help**   Prints a usage summary with all the available command-line options.

Workflow Options

-r , **--recursive**   Query workflow metadata for the entire workflow; including sub-workflows

Task Options

-i *ABS_TASK_ID* , **--task-id** *ABS_TASK_ID*   Specifies the absolute task id whose metadata should be queried.

File Options

-l , **--list**   Queries metadata for all files

-n *FILE_NAME* , **--file-name** *FILE_NAME*   Specifies name of the file whose metadata should be queried.

-t , **--trace**   Queries metadata for the file, the task that generated the file, the workflow which contains the task, and the root workflow which contains the task.

Examples

```
# Query metadata for a workflow
$ pegasus-metadata workflow /path/to/submit-dir

# Query metadata for all workflows i.e. including sub-workflows
$ pegasus-metadata workflow --recursive /path/to/submit-dir

# Query task metadata for a given task
$ pegasus-metadata task --task-id ID0000001 /path/to/submit-dir

# Query metadata for all files
$ pegasus-metadata file --list /path/to/submit-dir

# Query metadata for a given file
```

```
$ pegasus-metadata file --file-name f.a /path/to/submit-dir

# Trace entire metadata for a given file
$ pegasus-metadata file --file-name f.a --trace /path/to/submit-dir
```

Authors

Pegasus Team http://pegasus.isi.edu

Name

pegasus-monitord — tracks a workflow progress, mining information

Synopsis

pegasus-monitord [**--help**|**-help**] [**--verbose**|**-v**]
      [**--adjust**|**-a** *i*] [**--condor-daemon**|**-N**]
      [**--job**|**-j** *jobstate.log file*]
      [**--conf** *properties file*]
      [**--no-recursive**] [**--no-database** | **--no-events**]
      [**--replay**|**-r**] [**--no-notifications**]
      [**--notifications-max** *max_notifications*]
      [**--notifications-timeout** *timeout*]
      [**--sim**|**-s** *millisleep*] [**--db-stats**]
      [**--skip-stdout**] [**--force**|**-f**]
      [**--output-dir** | **-o** *dir*]
      [**--dest**|**-d** *PATH* or *URL*] [**--encoding**|**-e** *bp* | *bson*]
      DAGMan output file

Description

This program follows a workflow, parsing the output of DAGMAN's dagman.out file. In addition to generating the jobstate.log file, **pegasus-monitord** can also be used mine information from the workflow dag file and jobs' submit and output files, and either populate a database or write a NetLogger events file with that information. **pegasus-monitord** can also perform notifications when tracking a workflow's progress in real-time.

Options

-h , **--help**	Prints a usage summary with all the available command-line options.
-v , **--verbose**	Sets the log level for **pegasus-monitord**. If omitted, the default *level* will be set to *WARNING*. When this option is given, the log level is changed to *INFO*. If this option is repeated, the log level will be changed to *DEBUG*.
	The log level in **pegasus-monitord** can also be adjusted interactively, by sending the *USR1* and *USR2* signals to the process, respectively for incrementing and decrementing the log level.
-a *i* , **--adjust** *i*	For adjusting time zone differences by *i* seconds, default is 0.
-N , **--condor-daemon**	Condor daemon mode. This is used when monitord is invoked by pegasus-dagman. It just causes monitord to create a new process group.
-j *jobstate.log file* , **--job** *job-state.log file*	Alternative location for the *jobstate.log* file. The default is to write a *jobstate.log* in the workflow directory. An absolute file name should only be used if the workflow does not have any sub-workflows, as each sub-workflow will generate its own *jobstate.log* file. If an alternative, non-absolute, filename is given with this option, **pegasus-monitord** will create one file in each workflow (and sub-workflow) directory with the filename provided by the user with this option. If an absolute filename is provided and sub-workflows are found, a warning message will be printed and **pegasus-monitord** will not track any sub-workflows.
--conf *properties_file*	is an alternative file containing properties in the *key=value* format, and allows users to override values read from the *braindump.txt* file. This option has precedence over the properties file specified in the *braindump.txt* file. Please note that these properties will apply not only to the main workflow, but also to all sub-workflows found.

--no-recursive	This options disables **pegasus-monitord** to automatically follow any sub-workflows that are found.
--nodatabase , **--no-database** , **--no-events**	Turns off generating events (when this option is given, **pegasus-monitord** will only generate the jobstate.log file). The default is to automatically log information to a SQLite database (see the **--dest** option below for more details). This option overrides any parameter given by the **--dest** option.
-r , **--replay**	This option is used to replay the output of an already finished workflow. It should only be used after the workflow is finished (not necessarily successfully). If a *jobstate.log* file is found, it will be rotated. However, when using a database, all previous references to that workflow (and all its sub-workflows) will be erased from it. When outputing to a bp file, the file will be deleted. When running in replay mode, **pegasus-monitord** will always run with the **--no-daemon** option, and any errors will be output directly to the terminal. Also, **pegasus-monitord** will not process any notifications while in replay mode.
--no-notifications	This options disables notifications completely, making **pegasus-monitord** ignore all the .notify files for all workflows it tracks.
--notifications-max *max_notifications*	This option sets the maximum number of concurrent notifications that **pegasus-monitord** will start. When the *max_notifications* limit is reached, **pegasus-monitord** will queue notifications and wait for a pending notification script to finish before starting a new one. If *max_notifications* is set to 0, notifications will be disabled.
--notifications-timeout *timeout*	Normally, **pegasus-monitord** will start a notification script and wait indefinitely for it to finish. This option allows users to set up a maximum *timeout* that **pegasus-monitord** will wait for a notification script to finish before terminating it. If notification scripts do not finish in a reasonable amount of time, it can cause other notification scripts to be queued due to the maximum number of concurrent scripts allowed by **pegasus-monitord**. Additionally, until all notification scripts finish, **pegasus-monitord** will not terminate.
-s *millisleep* , **--sim** *millisleep*	This option simulates delays between reads, by sleeping *millisleep* milliseconds. This option is mainly used by developers.
--db-stats	This option causes the database module to collect and print database statistics at the end of the execution. It has no effect if the **--no-database** option is given.
--skip-stdout	This option causes **pegasus-monitord** not to populate jobs' stdout and stderr into the BP file or the Stampede database. It should be used to avoid increasing the database size substantially in cases where jobs are very verbose in their output.
-f , **--force**	This option causes **pegasus-monitord** to skip checking for another instance of itself already running on the same workflow directory. The default behavior prevents two or more **pegasus-monitord** instances from starting and running simultaneously (which would cause the bp file and database to be left in an unstable state). This option should noly be used when the user knows the previous instance of **pegasus-monitord** is **NOT** running anymore.
-o *dir* , **--ouput-dir** *dir*	When this option is given, **pegasus-monitord** will create all its output files in the directory specified by *dir*. This option is useful for allowing a user to debug a workflow in a directory the user does not have write permissions. In this case, all files generated by **pegasus-monitord** will have the workflow *wf_uuid* as a prefix so that files from multiple sub-workflows can be placed in the same directory. This option is mainly used by **pegasus-analyzer**. It is important to note that the location for the output BP file or database is not changed by this option and should be set via the **--dest** option.
-d *URL params* , **--dest** *URL params*	This option allows users to specify the destination for the log events generated by **pegasus-monitord**. If this option is omitted, **pegasus-monitord** will cre-

ate a SQLite database in the workflow's run directory with the same name as the workflow, but with a *.stampede.db* prefix. For an *empty* scheme, *params* are a file path with **-** meaning standard output. For a *x-tcp* scheme, *params* are *TCP_host[:port=14380]*. For a database scheme, *params* are a *SQLAlchemy engine URL* with a database connection string that can be used to specify different database engines. Please see the examples section below for more information on how to use this option. Note that when using a database engine other than **sqlite**, the necessary Python database drivers will need to be installed.

-e *encoding* , **--encoding** *encoding* — This option specifies how to encode log events. The two available possibilities are *bp* and *bson*. If this option is not specified, events will be generated in the *bp* format.

DAGMan_output_file — The *DAGMan_output_file* is the only requires command-line argument in **pegasus-monitord** and must have the *.dag.dagman.out* extension.

Return Value

If the plan could be constructed, **pegasus-monitord** returns with an exit code of 0. However, in case of error, a non-zero exit code indicates problems. In that case, the *logfile* should contain additional information about the error condition.

Environment Variables

pegasus-monitord does not require that any environmental variables be set. It locates its required Python modules based on its own location, and therefore should not be moved outside of Pegasus' bin directory.

Examples

Usually, **pegasus-monitord** is invoked automatically by **pegasus-run** and tracks the workflow progress in real-time, producing the *jobstate.log* file and a corresponding SQLite database. When a workflow fails, and is re-submitted with a rescue DAG, **pegasus-monitord** will automatically pick up from where it left previously and continue the *jobstate.log* file and the database.

If users need to create the *jobstate.log* file after a workflow is already finished, the **--replay | -r** option should be used when running **pegasus-monitord** manually. For example:

```
$ pegasus_monitord -r diamond-0.dag.dagman.out
```

will launch **pegasus-monitord** in replay mode. In this case, if a *jobstate.log* file already exists, it will be rotated and a new file will be created. If a *diamond-0.stampede.db* SQLite database already exists, **pegasus-monitord** will purge all references to the workflow id specified in the *braindump.txt* file, including all sub-workflows associated with that workflow id.

```
$ pegasus_monitord -r --no-database diamond-0.dag.dagman.out
```

will do the same thing, but without generating any log events.

```
$ pegasus_monitord -r --dest `pwd`/diamond-0.bp diamond-0.dag.dagman.out
```

will create the file *diamond-0.bp* in the current directory, containing NetLogger events with all the workflow data. This is in addition to the *jobstate.log* file.

For using a database, users should provide a database connection string in the format of:

```
dialect://username:password@host:port/database
```

Where *dialect* is the name of the underlying driver (*mysql*, *sqlite*, *oracle*, *postgres*) and *database* is the name of the database running on the server at the *host* computer.

If users want to use a different *SQLite* database, **pegasus-monitord** requires them to specify the absolute path of the alternate file. For example:

```
$ pegasus_monitord -r --dest sqlite:////home/user/diamond_database.db diamond-0.dag.dagman.out
```

Here are docs with details for all of the supported drivers: http://www.sqlalchemy.org/docs/05/reference/dialects/index.html

Additional per-database options that work into the connection strings are outlined there.

It is important to note that one will need to have the appropriate db interface library installed. Which is to say, *SQLAlchemy* is a wrapper around the mysql interface library (for instance), it does not provide a *MySQL* driver itself. The **Pegasus** distribution includes both **SQLAlchemy** and the **SQLite** Python driver.

As a final note, it is important to mention that unlike when using *SQLite* databases, using **SQLAlchemy** with other database servers, e.g. *MySQL* or *Postgres*, the target database needs to exist. So, if a user wanted to connect to:

```
mysql://pegasus-user:supersecret@localhost:localport/diamond
```

it would need to first connect to the server at *localhost* and issue the appropriate create database command before running **pegasus-monitord** as **SQLAlchemy** will take care of creating the tables and indexes if they do not already exist.

See Also

pegasus-run(1)

Authors

Gaurang Mehta `<gmehta at isi dot edu>`

Fabio Silva `<fabio at isi dot edu>`

Karan Vahi `<vahi at isi dot edu>`

Jens-S. Vöckler `<voeckler at isi dot edu>`

Pegasus Team http://pegasus.isi.edu

Name

pegasus-mpi-cluster — Enables running DAGs (Directed Acyclic Graphs) on clusters using MPI.

Synopsis

pegasus-mpi-cluster [options] *workflow.dag*

Description

pegasus-mpi-cluster is a tool used to run HTC (High Throughput Computing) scientific workflows on systems designed for HPC (High Performance Computing). Many HPC systems have custom architectures that are optimized for tightly-coupled, parallel applications. These systems commonly have exotic, low-latency networks that are designed for passing short messages very quickly between compute nodes. Many of these networks are so highly optimized that the compute nodes do not even support a TCP/IP stack. This makes it impossible to run HTC applications using software that was designed for commodity clusters, such as Condor.

pegasus-mpi-cluster was developed to enable loosely-coupled HTC applications such as scientific workflows to take advantage of HPC systems. In order to get around the network issues outlined above, **pegasus-mpi-cluster** uses MPI (Message Passing Interface), a commonly used API for writing SPMD (Single Process, Multiple Data) parallel applications. Most HPC systems have an MPI implementation that works on whatever exotic network architecture the system uses.

An **pegasus-mpi-cluster** job consists of a single master process (this process is rank 0 in MPI parlance) and several worker processes. The master process manages the workflow and assigns workflow tasks to workers for execution. The workers execute the tasks and return the results to the master. Any output written to stdout or stderr by the tasks is captured (see **TASK STDIO**).

pegasus-mpi-cluster applications are expressed as DAGs (Directed Acyclic Graphs) (see **DAG FILES**). Each node in the graph represents a task, and the edges represent dependencies between the tasks that constrain the order in which the tasks are executed. Each task is a program and a set of parameters that need to be run (i.e. a command and some optional arguments). The dependencies typically represent data flow dependencies in the application, where the output files produced by one task are needed as inputs for another.

If an error occurs while executing a DAG that causes the workflow to stop, it can be restarted using a rescue file, which records the progress of the workflow (see **RESCUE FILES**). This enables **pegasus-mpi-cluster** to pick up running the workflow where it stopped.

pegasus-mpi-cluster was designed to work either as a standalone tool or as a complement to the Pegasus Workflow Managment System (WMS). For more information about using PMC with Pegasus see the section on **PMC AND PEGASUS**.

pegasus-mpi-cluster allows applications expressed as a DAG to be executed in parallel on a large number of compute nodes. It is designed to be simple, lightweight and robust.

Options

-h , **--help**	Print help message
-V , **--version**	Print version information
-v , **--verbose**	Increase logging verbosity. Adding multiple **-v** increases the level more. The default log level is *INFO*. (see **LOGGING**)
-q , **--quiet**	Decrease logging verbosity. Adding multiple **-q** decreases the level more. The default log level is *INFO*. (see **LOGGING**)
-s , **--skip-rescue**	Ignore the rescue file for *workflow.dag* if it exists. Note that **pegasus-mpi-cluster** will still create a new rescue file for the current run. The default behavior is to use the rescue file if one is found. (see **RESCUE FILES**)
-o *path* , **--stdout** *path*	Path to file for task stdout. (see **TASK STDIO** and **--per-task-stdio**)

-e *path* , **--stderr** *path*	Path to file for task stderr. (see **TASK STDIO** and **--per-task-stdio**)
-m *M* , **--max-failures** *M*	Stop submitting new tasks after *M* tasks have failed. Once *M* has been reached, **pegasus-mpi-cluster** will finish running any tasks that have been started, but will not start any more tasks. This option is used to prevent **pegasus-mpi-cluster** from continuing to run a workflow that is suffering from a systematic error, such as a missing binary or an invalid path. The default for *M* is 0, which means unlimited failures are allowed.
-t *T* , **--tries** *T*	Attempt to run each task *T* times before marking the task as failed. Note that the *T* tries do not count as failures for the purposes of the **-m** option. A task is only considered failed if it is tried *T* times and all *T* attempts result in a non-zero exitcode. The value of *T* must be at least 1. The default is 1.
-n , **--nolock**	Do not lock DAGFILE. By default, **pegasus-mpi-cluster** will attempt to acquire an exclusive lock on DAGFILE to prevent multiple MPI jobs from running the same DAG at the same time. If this option is specified, then the lock will not be acquired.
-r , **--rescue** *path*	Path to rescue log. If the file exists, and **-s** is not specified, then the log will be used to recover the state of the workflow. The file is truncated after it is read and a new rescue log is created in its place. The default is to append *.rescue* to the DAG file name. (see **RESCUE FILES**)
--host-script *path*	Path to a script or executable to launch on each unique host that **pegasus-mpi-cluster** is running on. This path can also be set using the PMC_HOST_SCRIPT environment variable. (see **HOST SCRIPTS**)
--host-memory *size*	Amount of memory available on each host in MB. The default is to determine the amount of physical RAM automatically. This value can also be set using the PMC_HOST_MEMORY environment variable. (see **RESOURCE-BASED SCHEDULING**)
--host-cpus *cpus*	Number of CPUs available on each host. The default is to determine the number of CPU cores automatically. This value can also be set using the PMC_HOST_CPUS environment variable. (see **RESOURCE-BASED SCHEDULING**)
--strict-limits	This enables strict memory usage limits for tasks. When this option is specified, and a task tries to allocate more memory than was requested in the DAG, the memory allocation operation will fail.
--max-wall-time *minutes*	This is the maximum number of minutes that **pegasus-mpi-cluster** will allow the workflow to run. When this time expires **pegasus-mpi-cluster** will abort the workflow and merge all of the stdout/stderr files of the workers. The value is in minutes, and the default is unlimited wall time. This option was added so that the output of a workflow will be recorded even if the workflow exceeds the max wall time of its batch job. This value can also be set using the PMC_MAX_WALL_TIME environment variable.
--per-task-stdio	This causes PMC to generate a .out.XXX and a .err.XXX file for each task instead of writing task stdout/stderr to **--stdout** and **--stderr**. The name of the files are "TASKNAME.out.XXX" and "TASKNAME.err.XXX", where "TASKNAME" is the name of the task from the DAG and "XXX" is a sequence number that is incremented each time the task is tried. This option overrides the values for **--stdout** and **--stderr**. This argument is used by Pegasus when workflows are planned in PMC-only mode to facilitate debugging and monitoring.
--jobstate-log	This option causes PMC to generate a jobstate.log file for the workflow. The file is named "jobstate.log" and is placed in the same directory where the DAG file is located. If the file already exists, then PMC appends new lines to the

existing file. This option is used by Pegasus when workflows are planned in PMC-only mode to facilitate monitoring.

--monitord-hack	This option causes PMC to generate a .dagman.out file for the workflow. This file mimics the contents of the .dagman.out file generated by Condor DAGMan. The point of this option is to trick monitord into thinking that it is dealing with DAGMan so that it will generate the appropriate events to populate the STAMPEDE database for monitoring purposes. The file is named "DAG.dagman.out" where "DAG" is the path to the PMC DAG file.
--no-resource-log	Do not generate a *workflow.dag.resource* file for the workflow.
--no-sleep-on-recv	Do not use polling with sleep() to implement message receive. (see **Known Issues: CPU Usage**)
--maxfds	Set the maximum number of file descriptors that can be left open by the master for I/O forwarding. By default this value is set automatically based on the value of getrlimit(RLIMIT_NOFILE). The value must be at least 1, and cannot be more than RLIMIT_NOFILE.
--keep-affinity	By default PMC attempts to clear the CPU and memory affinity. This is to ensure that all available CPUs and memory can be used by PMC tasks on systems that are not configured properly. This flag tells PMC to keep the affinity settings inherited from its parent. Note that the memory policy can only be cleared if PMC was compiled with libnuma. CPU affinity is cleared using **sched_setaffinity()**, and memory policy is cleared with **set_mempolicy()**.
--set-affinity	If this flag is set, then PMC will allocate CPUs to tasks and call **sched_setaffinity()** to bind the task to those CPUs. This only applies to multicore tasks (i.e. those tasks that specify -c N where N > 1). Single core tasks are not bound to a CPU to reduce the possibility of fragmentation. PMC does not currently have any mechanism to handle resource fragmentation that may occur if a workflow contains several tasks with different core counts. In the case that fragmentation would result in a task not being bound to a minimal number of sockets and cores, PMC will not bind the task to any CPUs. For example, if a 2 socket, 8 core machine without hyperthreading is being used to run 2, 4-core tasks, each task will be bound to a full socket. If the same machine is running 4, 2-core tasks, each task will get 2-cores on one socket. If 2 of the 2-core tasks finish, but they free up cores on two different sockets, and PMC wants to run a 4-core task, it will not bind the 4-core task to any CPUs, because that would result in the 4-core task being bound to two different sockets. Instead, PMC lets the 4-core task float, so that the scheduler can find a better placement when another one of the 2-core tasks finishes. In order to fix this issue we need to rearchitect PMC, which is on the roadmap.

DAG Files

pegasus-mpi-cluster workflows are expressed using a simple text-based format similar to that used by Condor DAGMan. There are only two record types allowed in a DAG file: **TASK** and **EDGE**. Any blank lines in the DAG (lines with all whitespace characters) are ignored, as are any lines beginning with # (note that # can only appear at the beginning of a line, not in the middle).

The format of a **TASK** record is:

```
"TASK" id [options...] executable [arguments...]
```

Where *id* is the ID of the task, *options* is a list of task options, *executable* is the path to the executable or script to run, and *arguments...* is a space-separated list of arguments to pass to the task. An example is:

```
TASK t01 -m 10 -c 2 /bin/program -a -b
```

This example specifies a task *t01* that requires 10 MB memory and 2 CPUs to run */bin/program* with the arguments *-a* and *-b*. The available task options are:

-m *M* , **--request-memory** *M*	The amount of memory required by the task in MB. The default is 0, which means memory is not considered for this task. This option can be set for a job in the DAX by specifying the pegasus::pmc_request_memory profile. (see **RESOURCE-BASED SCHEDULING**)
-c *N* , **--request-cpus** *N*	The number of CPUs required by the task. The default is 1, which implies that the number of slots on a host should be less than or equal to the number of physical CPUs in order for all the slots to be used. This option can be set for a job in the DAX by specifying the pegasus::pmc_request_cpus profile. (see **RESOURCE-BASED SCHEDULING**)
-t *T* , **--tries** *T*	The number of times to try to execute the task before failing permanently. This is the task-level equivalent of the **--tries** command-line option.
-p *P* , **--priority** *P*	The priority of the task. P should be an integer. Larger values have higher priority. The default is 0. Priorities are simply hints and are not strict—if a task cannot be matched to an available slot (e.g. due to resource availability), but a lower-priority task can, then the task will be deferred and the lower priority task will be executed. This option can be set for a job in the DAX by specifying the pegasus::pmc_priority profile.
-f *VAR=FILE* , **--pipe-forward** *VAR=FILE*	Forward I/O to file *FILE* using pipes to communicate with the task. The environment variable *VAR* will be set to the value of a file descriptor for a pipe to which the task can write to get data into *FILE*. For example, if a task specifies: -f FOO=/tmp/foo then the environment variable FOO for the task will be set to a number (e.g. 3) that represents the file /tmp/foo. In order to specify this argument in a Pegasus DAX you need to set the pegasus::pmc_arguments profile (note that the value of pmc_arguments must contain the "-f" part of the argument, so a valid value would be: <profile namespace="pegasus" key="pmc_arguments">-f A=/tmp/a </profile>). (see **I/O FORWARDING**)
-F *SRC=DEST* , **--file-forward** *SRC=DEST*	Forward I/O to the file *DEST* from the file *SRC*. When the task finishes, the worker will read the data from *SRC* and send it to the master where it will be written to the file *DEST*. After *SRC* is read it is deleted. In order to specify this argument in a Pegasus DAX you need to set the pegasus::pmc_arguments profile. (see **I/O FORWARDING**)

The format of an **EDGE** record is:

```
"EDGE" parent child
```

Where *parent* is the ID of the parent task, and *child* is the ID of the child task. An example **EDGE** record is:

```
EDGE t01 t02
```

A simple diamond-shaped workflow would look like this:

```
# diamond.dag
TASK A /bin/echo "I am A"
TASK B /bin/echo "I am B"
TASK C /bin/echo "I am C"
TASK D /bin/echo "I am D"

EDGE A B
EDGE A C
EDGE B D
EDGE C D
```

Rescue Files

Many different types of errors can occur when running a DAG. One or more of the tasks may fail, the MPI job may run out of wall time, **pegasus-mpi-cluster** may segfault (we hope not), the system may crash, etc. In order to ensure that the DAG does not need to be restarted from the beginning after an error, **pegasus-mpi-cluster** generates a rescue file for each workflow.

The rescue file is a simple text file that lists all of the tasks in the workflow that have finished successfully. This file is updated each time a task finishes, and is flushed periodically so that if the work- flow fails and the user restarts it, **pegasus-mpi-cluster** can determine which tasks still need to be executed. As such, the rescue file is a sort-of transaction log for the workflow.

The rescue file contains zero or more DONE records. The format of these records is:

```
"DONE" *taskid*
```

Where *taskid* is the ID of the task that finished successfully.

By default, rescue files are named *DAGNAME.rescue* where *DAGNAME* is the path to the input DAG file. The file name can be changed by specifying the **-r** argument.

PMC and Pegasus

Using PMC for Pegasus Task Clustering

PMC can be used as the wrapper for executing clustered jobs in Pegasus. In this mode Pegasus groups several tasks together and submits them as a single clustered job to a remote system. PMC then executes the individual tasks in the cluster and returns the results.

PMC can be specified as the task manager for clustered jobs in Pegasus in three ways:

1. Globally in the properties file

   The user can set a property in the properties file that results in all the clustered jobs of the workflow being executed by PMC. In the Pegasus properties file specify:

   ```
   #PEGASUS PROPERTIES FILE
   pegasus.clusterer.job.aggregator=mpiexec
   ```

   In the above example, all the clustered jobs on all remote sites will be launched via PMC as long as the property value is not overridden in the site catalog.

2. By setting the profile key "job.aggregator" in the site catalog:

   ```
   <site handle="siteX" arch="x86" os="LINUX">
       ...
       <profile namespace="pegasus" key="job.aggregator">mpiexec</profile>
   </site>
   ```

   In the above example, all the clustered jobs on a siteX are going to be executed via PMC as long as the value is not overridden in the transformation catalog.

3. By setting the profile key "job.aggregator" in the transformation catalog:

   ```
   tr B {
       site siteX {
           pfn "/path/to/mytask"
           arch "x86"
           os "linux"
           type "INSTALLED"
           profile pegasus "clusters.size" "3"
           profile pegasus "job.aggregator" "mpiexec"
       }
   }
   ```

   In the above example, all the clustered jobs for transformation B on siteX will be executed via PMC.

It is usually necessary to have a pegasus::mpiexec entry in your transformation catalog that specifies a) the path to PMC on the remote site and b) the relevant globus profiles such as xcount, host_xcount and maxwalltime to control size of the MPI job. That entry would look like this:

```
tr pegasus::mpiexec {
    site siteX {
        pfn "/path/to/pegasus-mpi-cluster"
        arch "x86"
        os "linux"
```

```
        type "INSTALLED"
        profile globus "maxwalltime" "240"
        profile globus "host_xcount" "1"
        profile globus "xcount" "32"
    }
}
```

If this transformation catalog entry is not specified, Pegasus will attempt create a default path on the basis of the environment profile PEGASUS_HOME specified in the site catalog for the remote site.

PMC can be used with both horizontal and label-based clustering in Pegasus, but we recommend using label-based clustering so that entire sub-graphs of a Pegasus DAX can be clustered into a single PMC job, instead of only a single level of the workflow.

Pegasus Profiles for PMC

There are several Pegasus profiles that map to PMC task options:

pmc_request_memory This profile is used to set the --request-memory task option and is usually specified in the DAX or transformation catalog.

pmc_request_cpus This key is used to set the --request-cpus task option and is usually specified in the DAX or transformation catalog.

pmc_priority This key is used to set the --priority task option and is usually specified in the DAX.

These profiles are used by Pegasus when generating PMC's input DAG when PMC is used as the task manager for clustered jobs in Pegasus.

The profiles can be specified in the DAX like this:

```
<job id="ID0000001" name="mytask">
    <arguments>-a 1 -b 2 -c 3</arguments>
    ...
    <profile namespace="pegasus" key="pmc_request_memory">1024</profile>
    <profile namespace="pegasus" key="pmc_request_cpus">4</profile>
    <profile namespace="pegasus" key="pmc_priority">10</profile>
</job>
```

This example specifies a PMC task that requires 1GB of memory and 4 cores, and has a priority of 10. It produces a task in the PMC DAG that looks like this:

```
TASK mytask_ID00000001 -m 1024 -c 4 -p 10 /path/to/mytask -a 1 -b 2 -c 3
```

Using PMC for the Entire Pegasus DAX

Pegasus can also be configured to run the entire workflow as a single PMC job. In this mode Pegasus will generate a single PMC DAG for the entire workflow as well as a PBS script that can be used to submit the workflow.

In contrast to using PMC as a task clustering tool, in this mode there are no jobs in the workflow executed without PMC. The entire workflow, including auxilliary jobs such as directory creation and file transfers, is managed by PMC. If Pegasus is configured in this mode, then DAGMan and Condor are not required.

To run in PMC-only mode, set the property "pegasus.code.generator" to "PMC" in the Pegasus properties file:

```
pegasus.code.generator=PMC
```

In order to submit the resulting PBS job you may need to make changes to the .pbs file generated by Pegasus to get it to work with your cluster. This mode is experimental and has not been used extensively.

Logging

By default, all logging messages are printed to stderr. If you turn up the logging using **-v** then you may end up with a lot of stderr being forwarded from the workers to the master.

The log levels in order of severity are: FATAL, ERROR, WARN, INFO, DEBUG, and TRACE.

The default logging level is INFO. The logging levels can be increased with **-v** and decreased with **-q**.

Task STDIO

By default the stdout and stderr of tasks will be redirected to the master's stdout and stderr. You can change the path of these files with the **-o** and **-e** arguments. You can also enable per-task stdio files using the **--per-task-stdio** argument. Note that if per-task stdio files are not used then the stdio of all workers will be merged into one out and one err file by the master at the end, so I/O from different workers will not be interleaved, but I/O from each worker will appear in the order that it was generated. Also note that, if the job fails for any reason, the outputs will not be merged, but instead there will be one file for each worker named DAGFILE.out.X and DAGFILE.err.X, where DAGFILE is the path to the input DAG, and *X* is the worker's rank.

Host Scripts

A host script is a shell script or executable that **pegasus-mpi-cluster** launches on each unique host on which it is running. They can be used to start auxilliary services, such as memcached, that the tasks in a workflow require.

Host scripts are specified using either the **--host-script** argument or the **PMC_HOST_SCRIPT** environment variable.

The host script is started when **pegasus-mpi-cluster** starts and must exit with an exitcode of 0 before any tasks can be executed. If it the host script returns a non-zero exitcode, then the workflow is aborted. The host script is given 60 seconds to do any setup that is required. If it doesn't exit in 60 seconds then a SIGALRM signal is delivered to the process, which, if not handled, will cause the process to terminate.

When the workflow finishes, **pegasus-mpi-cluster** will deliver a SIGTERM signal to the host script's process group. Any child processes left running by the host script will receive this signal unless they created their own process group. If there were any processes left to receive this signal, then they will be given a few seconds to exit, then they will be sent SIGKILL. This is the mechanism by which processes started by the host script can be informed of the termination of the workflow.

Resource-Based Scheduling

High-performance computing resources often have a low ratio of memory to CPUs. At the same time, workflow tasks often have high memory requirements. Often, the memory requirements of a workflow task exceed the amount of memory available to each CPU on a given host. As a result, it may be necessary to disable some CPUs in order to free up enough memory to run the tasks. Similarly, many codes have support for multicore hosts. In that case it is necessary for efficiency to ensure that the number of cores required by the tasks running on a host do not exceed the number of cores available on that host.

In order to make this process more efficient, **pegasus-mpi-cluster** supports resource-based scheduling. In resource-based scheduling the tasks in the workflow can specify how much memory and how many CPUs they require, and **pegasus-mpi-cluster** will schedule them so that the tasks running on a given host do not exceed the amount of physical memory and CPUs available. This enables **pegasus-mpi-cluster** to take advantage of all the CPUs available when the tasks' memory requirement is low, but also disable some CPUs when the tasks' memory requirement is higher. It also enables workflows with a mixture of single core and multi-core tasks to be executed on a heterogenous pool.

If there are no hosts available that have enough memory and CPUs to execute one of the tasks in a workflow, then the workflow is aborted.

Memory

Users can specify both the amount of memory required per task, and the amount of memory available per host. If the amount of memory required by any task exceeds the available memory of all the hosts, then the workflow will be aborted. By default, the host memory is determined automatically, however the user can specify **--host-memory** to "lie" to **pegasus-mpi-cluster**. The amount of memory required for each task is specified in the DAG using the **-m**/**--request-memory** argument (see **DAG Files**).

CPUs

Users can specify the number of CPUs required per task, and the total number of CPUs available on each host. If the number of CPUs required by a task exceeds the available CPUs on all hosts, then the workflow will be aborted.

By default, the number of CPUs on a host is determined automatically, but the user can specify **--host-cpus** to over- or under-subscribe the host. The number of CPUs required for each task is specified in the DAG using the **-c/--request-cpus** argument (see **DAG Files**).

I/O Forwarding

In workflows that have lots of small tasks it is common for the I/O written by those tasks to be very small. For example, a workflow may have 10,000 tasks that each write a few KB of data. Typically each task writes to its own file, resulting in 10,000 files. This I/O pattern is very inefficient on many parallel file systems because it requires the file system to handle a large number of metadata operations, which are a bottleneck in many parallel file systems.

One way to handle this problem is to have all 10,000 tasks write to a single file. The problem with this approach is that it requires those tasks to synchronize their access to the file using POSIX locks or some other mutual exclusion mechanism. Otherwise, the writes from different tasks may be interleaved in arbitrary order, resulting in unusable data.

In order to address this use case PMC implements a feature that we call "I/O Forwarding". I/O forwarding enables each task in a PMC job to write data to an arbitrary number of shared files in a safe way. It does this by having PMC worker processes collect data written by the task and send it over over the high-speed network using MPI messaging to the PMC master process, where it is written to the output file. By having one process (the PMC master process) write to the file all of the I/O from many parallel tasks can be synchronized and written out to the files safely.

There are two different ways to use I/O forwarding in PMC: pipes and files. Pipes are more efficient, but files are easier to use.

I/O forwarding using pipes

I/O forwarding with pipes works by having PMC worker processes collect data from each task using UNIX pipes. This approach is more efficient than the file-based approach, but it requires the code of the task to be changed so that the task writes to the pipe instead of a regular file.

In order to use I/O forwarding a PMC task just needs to specify the **-f/--pipe-forward** argument to specify the name of the file to forward data to, and the name of an environment variable through which the PMC worker process can inform it of the file descriptor for the pipe.

For example, if there is a task "mytask" that needs to forward data to two files: "myfile.a" and "myfile.b", it would look like this:

```
TASK mytask -f A=/tmp/myfile.a -f B=/tmp/myfile.b /bin/mytask
```

When the /bin/mytask process starts it will have two variables in its environment: "A=3" and "B=4", for example. The value of these variables is the file descriptor number of the corresponding files. In this case, if the task wants to write to "/tmp/myfile.a", it gets the value of environment variable "A", and calls write() on that descriptor number. In C the code for that looks like this:

```
char *A = getenv("A");
int fd = atoi(A);
char *message = "Hello, World\n";
write(fd, message, strlen(message));
```

In some programming languages it is not possible to write to a file descriptor directly. Fortran, for example, refers to files by unit number instead of using file descriptors. In these languages you can either link C I/O functions into your binary and call them from routines written in the other language, or you can open a special file in the Linux /proc file system to get another handle to the pipe you want to access. For the latter, the file you should open is "/proc/self/fd/NUMBER" where NUMBER is the file descriptor number you got from the environment variable. For the example above, the pipe for myfile.a (environment variable A) is "/proc/self/fd/3".

If you are using **pegasus-kickstart**, which is probably the case if you are using PMC for a Pegasus workflow, then there's a trick you can do to avoid modifying your code. You use the /proc file system, as described above, but you let pegasus-kickstart handle the path construction. For example, if your application has an argument, -o, that allows you to specify the output file then you can write your task like this:

```
TASK mytask -f A=/tmp/myfile.a /bin/pegasus-kickstart /bin/mytask -o /proc/self/fd/$A
```

In this case, pegasus-kickstart will replace the $A in your application arguments with the file descriptor number you want. Your code can open that path normally, write to it, and then close it as if it were a regular file.

I/O forwarding using files

I/O forwarding with files works by having tasks write out data in files on the local disk. The PMC worker process reads these files and forwards the data to the master where it can be written to the desired output file. This approach may be much less efficient than using pipes because it involves the file system, which has more overhead than a pipe.

File forwarding can be enabled by giving the **-F/--file-forward** argument to a task.

Here's an example:

```
TASK mytask -F /tmp/foo.0=/scratch/foo /bin/mytask -o /tmp/foo.0
```

In this case, the worker process will expect to find the file /tmp/foo.0 when mytask exits successfully. It reads the data from that file and sends it to the master to be written to the end of /scratch/foo. After /tmp/foo.0 is read it will be deleted by the worker process.

This approach works best on systems where the local disk is a RAM file system such as Cray XT machines. Alternatively, the task can use /dev/shm on a regular Linux cluster. It might also work relatively efficiently on a local disk if the file system cache is able to absorb all of the reads and writes.

I/O forwarding caveats

When using I/O forwarding it is important to consider a few caveats.

First, if the PMC job fails for any reason (including when the workflow is aborted for violating **--max-wall-time**), then the files containing forwarded I/O may be corrupted. They can include **partial records**, meaning that only part of the I/O from one or more tasks was written, and they can include **duplicate records**, meaning that the I/O was written, but the PMC job failed before the task could be marked as successful, and the workflow was restarted later. We make no guarantees about the contents of the data files in this case. It is up to the code that reads the files to a) detect and b) recover from such problems. To eliminate duplicates the records should include a unique identifier, and to eliminate partials the records should include a checksum.

Second, you should not use I/O forwarding if your task is going to write a lot of data to the file. Because the PMC worker is reading data off the pipe/file into memory and sending it in an MPI message, if you write too much, then the worker process will run the system out of memory. Also, all the data needs to fit in a single MPI message. In pipe forwarding there is no hard limit on the size, but in file forwarding the limit is 1MB. We haven't benchmarked the performance on large I/O, but anything larger than about 1 MB is probably too much. At any rate, if your data is larger than 1MB, then I/O forwarding probably won't have much of a performance benefit anyway.

Third, the I/O is not written to the file if the task returns a non-zero exitcode. We assume that if the task failed that you don't want the data it produced.

Fourth, the data from different tasks is not interleaved. All of the data written by a given task will appear sequentially in the output file. Note that you can still get partial records, however, if any data from a task appears it will never be split among non-adjacent ranges in the output file. If you have 3 tasks that write: "I am a task" you can get:

```
I am a taskI am a taskI am a task
```

and:

```
I am a taskI amI am a task
```

but not:

```
I am a taskI amI am a task a task
```

Fifth, data from different tasks appears in arbitrary order in the output file. It depends on what order the tasks were executed by PMC, which may be arbitrary if there are no dependencies between the tasks. The data that is written should contain enough information that you are able to determine which task produced it if you require that. PMC does not add any headers or trailers to the data.

Sixth, a task will only be marked as successful if all of its I/O was successfully written. If the workflow completed successfully, then the I/O is guaranteed to have been written.

Seventh, if the master is not able to write to the output file for any reason (e.g. the master tries to write the I/O to the destination file, but the write() call returns an error) then the task is marked as failed even if the task produced a non-zero exitcode. In other words, you may get a non-zero kickstart record even when PMC marks the task failed.

Eighth, the pipes are write-only. If you need to read and write data from the file you should use file forwarding and not pipe forwarding.

Ninth, all files are opened by the master in append mode. This is so that, if the workflow fails and has to be restarted, or if a task fails and is retried, the data that was written previously is not lost. PMC never truncates the files. This is one of the reasons why you can have partial records and duplicate records in the output file.

Finally, in file forwarding the output file is removed when the task exits. You cannot rely on the file to be there when the next task runs even if you write it to a shared file system.

Misc

Resource Utilization

At the end of the workflow run, the master will report the resource utilization of the job. This is done by adding up the total runtimes of all the tasks executed (including failed tasks) and dividing by the total wall time of the job times N, where N is both the total number of processes including the master, and the total number of workers. These two resource utilization values are provided so that users can get an idea about how efficiently they are making use of the resources they allocated. Low resource utilization values suggest that the user should use fewer cores, and longer wall time, on future runs, while high resource utilization values suggest that the user could use more cores for future runs and get a shorter wall time.

Known Issues

Cray Compiler Wrappers

On Cray machines, the CC compiler wrapper for C++ code should be used to compile PMC. That wrapper links in all the required MPI libraries. **Cray compiler wrappers should not be used to compile tasks that run under PMC.** If you use a Cray wrapper to compile a task that runs under PMC, then the task will hang, or exit immediately with a 0 exit code without doing anything. This appears to happen only when the application binary is dynamically linked. It seems to be a problem with the libraries that are linked into the code when it is compiled with a Cray wrapper. To summarize: on Cray machines, compile PMC with the CC wrapper, but compile code that runs under PMC without any wrappers.

fork() and exec()

In order for the worker processes to start tasks on the compute node the compute nodes must support the **fork()** and **exec()** system calls. If your target machine runs a stripped-down OS on the compute nodes that does not support these system calls, then **pegasus-mpi-cluster** will not work.

CPU Usage

Many MPI implementations are optimized so that message sends and receives do busy waiting (i.e. they spin/poll on a message send or receive instead of sleeping). The reasoning is that sleeping adds overhead and, since many HPC systems use space sharing on dedicated hardware, there are no other processes competing, so spinning instead of sleeping can produce better performance. On those implementations MPI processes will run at 100% CPU usage even when they are just waiting for a message. This is a big problem for multicore tasks in **pegasus-mpi-cluster** because idle slots consume CPU resources. In order to solve this problem **pegasus-mpi-cluster** processes sleep for a short period between checks for waiting messages. This reduces the load significantly, but causes a short delay in receiving messages. If you are using an MPI implementation that sleeps on message send and receive instead of doing busy waiting, then you can disable the sleep by specifying the **--no-sleep-on-recv** option. Note that the master will always sleep if **--max-wall-time** is specified because there is no way to interrupt or otherwise timeout a blocking call in MPI (e.g. SIGALRM does not cause MPI_Recv to return EINTR).

Task Environment

PMC sets a few environment variables when it launches a task. In addition to the environment variables for pipe forwarding, it sets:

PMC_TASK   The name of the task from the DAG file.

PMC_MEMORY   The amount of memory requested by the task.

PMC_CPUS   The number of CPUs requested by the task.

PMC_RANK   The rank of the MPI worker that launched the task.

PMC_HOST_RANK   The host rank of the MPI worker that launched the task.

In addition, if **--set-affinity** is specified, and PMC has allocated some CPUs to the task, then it will export:

PMC_AFFINITY   A comma-separated list of CPUs to which the task is/should be bound.

Environment Variables

The environment variables below are aliases for command-line options. If the environment variable is present, then it is used as the default for the associated option. If both are present, then the command-line option is used.

PMC_HOST_SCRIPT   Alias for the **--host-script** option.

PMC_HOST_MEMORY   Alias for the **--host-memory** option.

PMC_HOST_CPUS   Alias for the **--host-cpus** option.

PMC_MAX_WAL-   Alias for the **--max-wall-time** option.
L_TIME

Author

Gideon Juve `<gideon@isi.edu>`

Mats Rynge `<rynge@isi.edu>`

Name

pegasus-mpi-keg — MPI version of KEG

Synopsis

pegasus-mpi-keg [-a appname] [-t interval |-T interval] [-l logname]
[-P prefix] [-o fn [..]] [-i fn [..]] [-G sz] [-m memory]
[-r root_memory_allocation] [-C] [-e env [..]] [-p parm [..]]

Description

The parallel version of kanonical executable is a stand-in for parallel binaries in a DAG - but not for their arguments. It allows to trace the shape of the execution of a DAG, and thus is an aid to debugging DAG related issues.

It works in the same way as the sequential version of **pegasus-keg** but it is intended to be executed as an MPI task. **pegasus-mpi-keg** accepts the same parameters as **pegasus-keg**, so please refer to the **pegasus-keg** manual page for more details.

Arguments

The same as **pegasus-keg**. But there are some MPI-specific arguments.

-r root_memory_allocation_only   Works use only with the **-m** option. When set, the memory allocation will take place in the root MPI process only. By default, each MPI processe allocates the amount of memory set by the **-m** option.

Return Value

The same as **pegasus-keg**.

Example

The example shows the bracketing of an input file, and the copy produced on the output file. For illustration purposes, the output file is connected to *stdout* :

```
$ date > xx
$ mpiexec -n 2 ./pegasus-mpi-keg -i xx -p a b c -o -
--- start xx ----
  Tue Dec  2 17:35:39 PST 2014
--- final xx ----
Timestamp Today: 20141202T173553.184-08:00 (1417570553.184;0.001)
Applicationname: pegasus-mpi-keg [36116e11c0735993bf54264953194e626fe4ab7e 2014-11-25] @
 138.25.147.42 (myc-2.local)
Current Workdir: /opt/pegasus/default/bin/pegasus-mpi-keg
Systemenvironm.: x86_64-Darwin 14.0.0
Processor Info.: 4 x Intel(R) Core(TM) i5-4278U CPU @ 2.60GHz
Load Averages  : 1.240 1.354 1.434
Memory Usage MB: 8192 total, 161 avail, 3599 active, 2496 inactive, 1077 wired
Swap Usage   MB: 2048 total, 1256 free
Filesystem Info: /                      hfs   232GB total,   66GB avail
Output Filename: -
Input Filenames: xx
Other Arguments: a b c
```

Restrictions

The same as **pegasus-keg**.

Authors

Pegasus - http://pegasus.isi.edu/

Name

pegasus-plan — runs Pegasus to generate the executable workflow

Synopsis

pegasus-plan [**-v**] [**-q**] [**-V**] [**-h**]
        [**-D***prop=value*…]] [**-b** prefix]
        [**--conf** *propsfile*]
        [**-c** *cachefile*[*,cachefile*…]] [**--cleanup** *cleanup strategy*]
        [**-C** *style*[*,style*…]]
        [**--dir** *dir*]
        [**--force**] [**--force-replan**]
        [**--inherited-rc-files** *file1*[*,file2*…]] [**-j** *prefix*]
        [**-n**][**-I** *input-dir1*[*,input-dir2*…]][**-O** *output-dir*] [**-o** *site*]
        [**-s** *site1*[*,site2*…]]
        [**--staging-site** s1=ss1[,s2=ss2[..]]]
        [**--randomdir**[*=dirname*]]
        [**--relative-dir** *dir*]
        [**--relative-submit-dir** *dir*]
        -d *daxfile*

Description

The **pegasus-plan** command takes in as input the DAX and generates an executable workflow usually in form of **condor** submit files, which can be submitted to an *execution* site for execution.

As part of generating an executable workflow, the planner needs to discover:

data
    The Pegasus Workflow Planner ensures that all the data required for the execution of the executable workflow is transferred to the execution site by adding transfer nodes at appropriate points in the DAG. This is done by looking up an appropriate **Replica Catalog** to determine the locations of the input files for the various jobs. By default, a file based replica catalog is used.

    The Pegasus Workflow Planner also tries to reduce the workflow, unless specified otherwise. This is done by deleting the jobs whose output files have been found in some location in the Replica Catalog. At present no cost metrics are used. However preference is given to a location corresponding to the execution site

    The planner can also add nodes to transfer all the materialized files to an output site. The location on the output site is determined by looking up the site catalog file, the path to which is picked up from the **pegasus.catalog.site.file** property value.

executables
    The planner looks up a Transformation Catalog to discover locations of the executables referred to in the executable workflow. Users can specify INSTALLED or STAGEABLE executables in the catalog. Stageable executables can be used by Pegasus to stage executables to resources where they are not pre-installed.

resources
    The layout of the sites, where Pegasus can schedule jobs of a workflow are described in the Site Catalog. The planner looks up the site catalog to determine for a site what directories a job can be executed in, what servers to use for staging in and out data and what jobmanagers (if applicable) can be used for submitting jobs.

The data and executable locations can now be specified in DAX'es conforming to DAX schema version 3.2 or higher.

Options

Any option will be displayed with its long options synonym(s).

-D*property=value*
    The **-D** option allows an experienced user to override certain properties which influence the program execution, among them the default location of the user's properties file and the PEGASUS home location. One may set several CLI prop-

erties by giving this option multiple times. The **-D** option(s) must be the first option on the command line. A CLI property take precedence over the properties file property of the same key.

-d *file* , **--dax** *file*	The DAX is the XML input file that describes an abstract workflow. This is a mandatory option, which has to be used.
-b *prefix* , **--basename** *prefix*	The basename prefix to be used while constructing per workflow files like the dagman file (.dag file) and other workflow specific files that are created by Condor. Usually this prefix, is taken from the name attribute specified in the root element of the dax files.

-c *file*[*,file,…*] , **--cache** *file*[*,file, …*]

A comma separated list of paths to replica cache files that override the results from the replica catalog for a particular LFN.

Each entry in the cache file describes a LFN , the corresponding PFN and the associated attributes. The site attribute should be specified for each entry.

```
LFN_1 PFN_1 site=[site handle 1]
LFN_2 PFN_2 site=[site handle 2]
 ...
LFN_N PFN_N [site handle N]
```

To treat the cache files as supplemental replica catalogs set the property **pegasus.catalog.replica.cache.asrc** to true. This results in the mapping in the cache files to be merged with the mappings in the replica catalog. Thus, for a particular LFN both the entries in the cache file and replica catalog are available for replica selection.

-C *style*[*,style,…*] , **--cluster** *style*[*,style,…*]

Comma-separated list of clustering styles to apply to the workflow. This mode of operation results in clustering of n compute jobs into a larger jobs to reduce remote scheduling overhead. You can specify a list of clustering techniques to recursively apply them to the workflow. For example, this allows you to cluster some jobs in the workflow using horizontal clustering and then use label based clustering on the intermediate workflow to do vertical clustering.

The clustered jobs can be run at the remote site, either sequentially or by using MPI. This can be specified by setting the property **pegasus.job.aggregator**. The property can be overridden by associating the PEGASUS profile key *collapser* either with the transformation in the transformation catalog or the execution site in the site catalog. The value specified (to the property or the profile), is the logical name of the transformation that is to be used for clustering jobs. Note that clustering will only happen if the corresponding transformations are catalogued in the transformation catalog.

PEGASUS ships with a clustering executable *pegasus-cluster* that can be found in the *$PEGASUS_HOME/bin* directory. It runs the jobs in the clustered job sequentially on the same node at the remote site.

In addition, an MPI based clustering tool called pegasus-mpi-cluster', is also distributed and can be found in the bin directory. pegasus-mpi-cluster can also be used in the sharedfs setup and needs to be compiled against the remote site MPI install. directory. The wrapper is run on every MPI node, with the first one being the master and the rest of the ones as workers.

By default, *pegasus-cluster* is used for clustering jobs unless overridden in the properties or by the pegasus profile key *collapser*.

The following type of clustering styles are currently supported:

- **horizontal** is the style of clustering in which jobs on the same level are aggregated into larger jobs. A level of the workflow is defined as the greatest distance of a node, from the root of the workflow. Clustering occurs only on

jobs of the same type i.e they refer to the same logical transformation in the transformation catalog.

Horizontal Clustering can operate in one of two modes. a. Job count based.

The granularity of clustering can be specified by associating either the PEGASUS profile key *clusters.size* or the PEGASUS profile key *clusters.num* with the transformation.

The *clusters.size* key indicates how many jobs need to be clustered into the larger clustered job. The clusters.num key indicates how many clustered jobs are to be created for a particular level at a particular execution site. If both keys are specified for a particular transformation, then the clusters.num key value is used to determine the clustering granularity.

a. Runtime based.

To cluster jobs according to runtimes user needs to set one property and two profile keys. The property pegasus.clusterer.preference must be set to the value *runtime*. In addition user needs to specify two Pegasus profiles. a. clusters.maxruntime which specifies the maximum duration for which the clustered job should run for. b. job.runtime which specifies the duration for which the job with which the profile key is associated, runs for. Ideally, clusters.maxruntime should be set in transformation catalog and job.runtime should be set for each job individually.

- **label** is the style of clustering in which you can label the jobs in your workflow. The jobs with the same level are put in the same clustered job. This allows you to aggregate jobs across levels, or in a manner that is best suited to your application.

  To label the workflow, you need to associate PEGASUS profiles with the jobs in the DAX. The profile key to use for labeling the workflow can be set by the property *pegasus.clusterer.label.key*. It defaults to label, meaning if you have a PEGASUS profile key label with jobs, the jobs with the same value for the pegasus profile key label will go into the same clustered job.

--cleanup *cleanup strategy* The cleanup strategy to be used for workflows. Pegasus can add cleanup jobs to the executable workflow that can remove files and directories during the workflow execution. The default strategy is inplace .

The following type of cleanup strategies are currently supported:

- **none** disables cleanup altogether. The planner does not add any cleanup jobs in the executable workflow whatsoever.

- **leaf** the planner adds a leaf cleanup node per staging site that removes the directory created by the create dir job in the workflow.

- **inplace** the planner adds in addition to leaf cleanup nodes, cleanup nodes per level of the workflow that remove files no longer required during execution. For example, an added cleanup node will remove input files for a particular compute job after the job has finished successfully.

- **constraint** the planner adds in addition to leaf cleanup nodes, cleanup nodes to constraint the amount of storage space used by a workflow. The added cleanup node guarantees limits on disk usage.

By default, for hierarchal workflows the inplace cleanup is always turned off. This is because the cleanup algorithm (InPlace) does not work across the sub workflows. For example, if you have two DAX jobs in your top level workflow and the child DAX job refers to a file generated during the execu-

tion of the parent DAX job, the InPlace cleanup algorithm when applied to the parent dax job will result in the file being deleted, when the sub workflow corresponding to parent DAX job is executed. This would result in failure of sub workflow corresponding to the child DAX job, as the file deleted is required to present during it's execution.

In case there are no data dependencies across the dax jobs, then yes you can enable the InPlace algorithm for the sub dax'es . To do this you can set the property

pegasus.file.cleanup.scope deferred

This will result in cleanup option to be picked up from the arguments for the DAX job in the top level DAX.

--conf *propfile*	The path to properties file that contains the properties planner needs to use while planning the workflow. Defaults to pegasus.properties file in the current working directory, if no conf option is specified.
--dir *dir*	The base directory where you want the output of the Pegasus Workflow Planner usually condor submit files, to be generated. Pegasus creates a directory structure in this base directory on the basis of username, VO Group and the label of the workflow in the DAX.
	By default the base directory is the directory from which one runs the **pegasus-plan** command.
-f , **--force**	This bypasses the reduction phase in which the abstract DAG is reduced, on the basis of the locations of the output files returned by the replica catalog. This is analogous to a **make** style generation of the executable workflow.
--force-replan	By default, for hierarichal workflows if a DAX job fails, then on job retry the rescue DAG of the associated workflow is submitted. This option causes Pegasus to replan the DAX job in case of failure instead.
-g , **--group**	The VO Group to which the user belongs to.
-h , **--help**	Displays all the options to the **pegasus-plan** command.
--inherited-rc-files *file*[,*file*,…]	A comma separated list of paths to replica files. Locations mentioned in these have a lower priority than the locations in the DAX file. This option is usually used internally for hierarchical workflows, where the file locations mentioned in the parent (encompassing) workflow DAX, passed to the sub workflows (corresponding) to the DAX jobs.
-I , **--input-dir**	A comma separated list of input directories on the submit host where the input files reside. This internally loads a Directory based Replica Catalog backend, that constructs does a directory listing to create the LFN#PFN mappings for the files in the input directory. You can specify additional properties either on the command line or the properties file to control the site attribute and url prefix associated with the mappings.
	pegasus.catalog.replica.directory.site specifies the site attribute to associate with the mappings. Defaults to local
	pegasus.catalog.replica.directory.url.prefix specifies the URL prefix to use while constructing the PFN. Defaults to file://
-j *prefix* , **--job-prefix** *prefix*	The job prefix to be applied for constructing the filenames for the job submit files.
-n , **--nocleanup**	This option is deprecated. Use --cleanup none instead.

-o *site* , **--output-site** *site*	The output site to which the output files of the DAX are transferred to.
	By default the **materialized data** remains in the working directory on the **execution** site where it was created. Only those output files are transferred to an output site for which transfer attribute is set to true in the DAX.
-O *output directory* , **--output-dir** *output directory*	The output directory to which the output files of the DAX are transferred to.
	If -o is specified the storage directory of the site specified as the output site is updated to be the directory passed. If no output site is specified, then this option internally sets the output site to local with the storage directory updated to the directory passed.
-q , **--quiet**	Decreases the logging level.
-r[*dirname*] , **--randomdir**[=*dirname*]	Pegasus Worfklow Planner adds create directory jobs to the executable workflow that create a directory in which all jobs for that workflow execute on a particular site. The directory created is in the working directory (specified in the site catalog with each site).
	By default, Pegasus duplicates the relative directory structure on the submit host on the remote site. The user can specify this option without arguments to create a random timestamp based name for the execution directory that are created by the create dir jobs. The user can can specify the optional argument to this option to specify the basename of the directory that is to be created.
	The create dir jobs refer to the **dirmanager** executable that is shipped as part of the PEGASUS worker package. The transformation catalog is searched for the transformation named **pegasus::dirmanager** for all the remote sites where the workflow has been scheduled. Pegasus can create a default path for the dirmanager executable, if **PEGASUS_HOME** environment variable is associated with the sites in the site catalog as an environment profile.
--relative-dir *dir*	The directory relative to the base directory where the executable workflow it to be generated and executed. This overrides the default directory structure that Pegasus creates based on username, VO Group and the DAX label.
--relative-submit-dir *dir*	The directory relative to the base directory where the executable workflow it to be generated. This overrides the default directory structure that Pegasus creates based on username, VO Group and the DAX label. By specifying **--relative-dir** and **--relative-submit-dir** you can have different relative execution directory on the remote site and different relative submit directory on the submit host.
-s *site*[,*site*,…] , **--sites** *site*[,*site*, …]	A comma separated list of execution sites on which the workflow is to be executed. Each of the sites should have an entry in the site catalog, that is being used.
	In case this option is not specified, all the sites in the site catalog other than site **local** are picked up as candidates for running the workflow.
--staging-site *s1=ss1*[,s2=ss2[..]]	A comma separated list of key=value pairs , where the key is the execution site and value is the staging site for that execution site.
	In case of running on a shared filesystem, the staging site is automatically associated by the planner to be the execution site. If only a value is specified, then that is taken to be the staging site for all the execution sites. e.g **--staging-site** local means that the planner will use the local site as the staging site for all jobs in the workflow.
-s , **--submit**	Submits the generated **executable workflow** using **pegasus-run** script in $PEGASUS_HOME/bin directory. By default, the Pegasus Workflow Planner only generates the Condor submit files and does not submit them.

-v , **--verbose**	Increases the verbosity of messages about what is going on. By default, all FATAL, ERROR, CONSOLE and WARN messages are logged. The logging hierarchy is as follows:

1. FATAL

2. ERROR

3. CONSOLE

4. WARN

5. INFO

6. CONFIG

7. DEBUG

8. TRACE

For example, to see the INFO, CONFIG and DEBUG messages additionally, set **-vvv**.

-V , **--version**	Displays the current version number of the Pegasus Workflow Management System.

Return Value

If the Pegasus Workflow Planner is able to generate an executable workflow successfully, the exitcode will be 0. All runtime errors result in an exitcode of 1. This is usually in the case when you have misconfigured your catalogs etc. In the case of an error occurring while loading a specific module implementation at run time, the exitcode will be 2. This is usually due to factory methods failing while loading a module. In case of any other error occurring during the running of the command, the exitcode will be 1. In most cases, the error message logged should give a clear indication as to where things went wrong.

Controlling pegasus-plan Memory Consumption

pegasus-plan will try to determine memory limits automatically using factors such as total system memory and potential memory limits (ulimits). The automatic limits can be overridden by setting the JAVA_HEAPMIN and JA-VA_HEAPMAX environment variables before invoking pegasus-plan. The values are in megabytes. As a rule of thumb, JAVA_HEAPMIN can be set to half of the value of JAVA_HEAPMAX.

Pegasus Properties

This is not an exhaustive list of properties used. For the complete description and list of properties refer to **$PE-GASUS_HOME/doc/advanced-properties.pdf**

pegasus.selector.site	Identifies what type of site selector you want to use. If not specified the default value of **Random** is used. Other supported modes are **RoundRobin** and **Non-JavaCallout** that calls out to a external site selector.
pegasus.catalog.replica	Specifies the type of replica catalog to be used.
	If not specified, then the value defaults to **File**.
pegasus.catalog.replica.url	Contact string to access the replica catalog. In case of File it is path to the file based replica catalog. If not specified, then default value of $PWD/rc.txt is used for the default File based Replica Catalog.
pegasus.dir.exec	A suffix to the workdir in the site catalog to determine the current working directory. If relative, the value will be appended to the working directory from the site.config file. If absolute it constitutes the working directory.

pegasus.catalog.transformation	Specifies the type of transformation catalog to be used. One can use only a file based transformation catalog, with the value as **Text**.
pegasus.catalog.transformation.file	The location of file to use as transformation catalog. If not specified, then the default location of $PWD/tc.txt
pegasus.catalog.site	Specifies the type of site catalog to be used. One can use either a text based or an xml based site catalog. At present the default is **XML**.
pegasus.catalog.site.file	The location of file to use as a site catalog. If not specified, then default value of $PWD/sites.xml is used in case of the xml based site catalog.
pegasus.data.configuration	This property sets up Pegasus to run in different environments. This can be set to **sharedfs** If this is set, Pegasus will be setup to execute jobs on the shared filesystem on the execution site. This assumes, that the head node of a cluster and the worker nodes share a filesystem. The staging site in this case is the same as the execution site. **nonsharedfs** If this is set, Pegasus will be setup to execute jobs on an execution site without relying on a shared filesystem between the head node and the worker nodes. **condorio** If this is set, Pegasus will be setup to run jobs in a pure condor pool, with the nodes not sharing a filesystem. Data is staged to the compute nodes from the submit host using Condor File IO.
pegasus.code.generator	The code generator to use. By default, Condor submit files are generated for the executable workflow. Setting to **Shell** results in Pegasus generating a shell script that can be executed on the submit host.

Files

$PEGASUS_HOME/etc/dax-3.3.xsd	is the suggested location of the latest DAX schema to produce DAX output.
$PEGASUS_HOME/etc/sc-4.0.xsd	is the suggested location of the latest Site Catalog schema that is used to create the XML version of the site catalog
$PEGASUS_HOME/etc/tc.data.text	is the suggested location for the file corresponding to the Transformation Catalog.
$PEGASUS_HOME/etc/sites.xml4** \| **$PEGASUS_HOME/etc/sites.xml3	is the suggested location for the file containing the site information.
$PEGASUS_HOME/lib/pegasus.jar	contains all compiled Java bytecode to run the Pegasus Workflow Planner.

See Also

pegasus-run(1), pegasus-status(1), pegasus-remove(1), pegasus-rc-client(1), pegasus-analyzer(1)

Authors

Karan Vahi <vahi at isi dot edu>

Pegasus Team http://pegasus.isi.edu

Name

pegasus-plots — A tool to generate graphs and charts to visualize workflow run.

Synopsis

pegasus-plots [**-h**|**--help**]
        [**-o**|**--output** *outdir*]
        [**-c**|**--conf** *propfile*]
        [**-m**|**--max-graph-nodes** *max*]
        [**-p**|**--plotting-level** *level*]
        [**-i**|**--ignore-db-inconsistency**]
        [**-v**|**--verbose**]
        [**-q**|**--quiet**]
        [*submitdir*]

Description

pegasus-plots generates graphs and charts to visualize workflow run. It generates workflow execution Gantt chart, job over time chart, time chart, dax and dag graph. It uses executable 'dot\' to generate graphs. pegasus-plots looks for the executable in your path and generates graphs based on it's availability .

Options

-h , **--help**	Prints a usage summary with all the available command-line options.
-o *outdir* , **--output** *outdir*	Writes the output to the given directory
-c *propfile* , **--conf** *propfile*	The properties file to use. This option overrides all other property files.
-m *max* , **--max-graph-nodes** *max*	Maximum limit on the number of tasks/jobs in the dax/dag up to which the graph should be generated. The default value is 100.
-p *level* , **--plotting-level** *level*	Specifies the charts and graphs to generate. Valid levels are: **all**, **all_charts**, **all_graphs**, **dax_graph**, **dag_graph**, **gantt_chart**, **host_chart**, **time_chart**, **breakdown_chart**. Default is **all_charts**. The output generated by **pegasus-plots** is based on the *level* set:

- **all**: generates all charts and graphs.

- **all_charts**: generates all charts.

- **all_graphs**: generates all graphs.

- **dax_graph**: generates dax graph.

- **dag_graph**: generates dag graph.

- **gantt_chart**: generates the workflow execution Gantt chart.

- **host_chart**: generates the host over time chart.

- **time_chart**: generates the time chart which shows the job instance/invocation count and runtime over time.

- **breakdown_chart**: generates the breakdown chart which shows the invocation count and runtime grouped by transformation name.

-i , **--ignore-db-inconsistency**	Turn off the the check for database consistency.
-v , **--verbose**	Increases the log level. If omitted, the default level will be set to WARNING. When this option is given, the log level is changed to INFO. If this option is repeated, the log level will be changed to DEBUG.

-q , **--quiet**	Decreases the log level. If omitted, the default level will be set to WARNING. When this option is given, the log level is changed to ERROR.

Example

Runs pegasus-plots and writes the output to the given directory:

```
pegasus-plots  -o /scratch/plot /scratch/grid-setup/run0001
```

Authors

Prasanth Thomas

Pegasus Team http://pegasus.isi.edu

Name

pegasus-rc-client — shell client for replica implementations

Synopsis

pegasus-rc-client [**-D**property=value[…]] [**-V**]
[**-c** fn] [**-p** k=v]
[[**-f** fn]|[**-i**|**-d** fn]|[cmd [args]]

Description

The shell interface to replica catalog implementations is a prototype. It determines from various property setting which class implements the replica manager interface, and loads that driver at run-time. Some commands depend on the implementation.

Options

Any option will be displayed with its long options synonym(s).

-Dproperty=value	The **-D** option allows an experienced user to override certain properties which influence the program execution, among them the default location of the user's properties file and the PEGASUS home location. One may set several CLI properties by giving this option multiple times. The **-D** option(s) must be the first option on the command line. A CLI property take precedence over the properties file property of the same key.
-c fn , **--conf** fn	Path to the property file
-f fn , **--file** fn	The optional input file argument permits to enter non-interactive bulk mode. If this option is not present, replica manager specific commands should be issued on the command-line. The special filename hyphen (-) can be used to read from pipes.
	Default is to use an interactive interface reading from *stdin*.
-i fn , **--insert** fn	The optional input file argument permits insertion of entries from the Replica Catalog in a bulk mode, wherever supported by the underlying implementation.
	Each line in the file denotes one mapping of the format **<lfn> <pfn> [k=v [..]]**
-d fn , **--delete** fn	The optional input file argument permits deletion of entries from the Replica Catalog in a bulk mode, wherever supported by the underlying implementation.
	Each line in the file denotes one mapping of the format: **<lfn> <pfn> [k=v [..]]**
-p k=v , **--pref** k=v	This option may be specified multiple times. Each specification populates instance preferences. Preferences control the extend of log information, or the output format string to use in listings.
	The keys **format** and **level** are recognized as of this writing.
	There are no defaults.
cmd [args]	If not in file-driven mode, a single command can be specified with its arguments.
	Default is to use interactive mode.
-V , **--version**	displays the version of Pegasus you are using.

Return Value

Regular and planned program terminations will result in an exit code of 0. Abnormal termination will result in a non-zero exit code.

Files

$PEGASUS_HOME/etc/proper-ties	contains the basic properties with all configurable options.
$HOME/.pegasusrc	contains the basic properties with all configurable options.
pegasus.jar	contains all compiled Java bytecode to run the replica manager.

Environment Variables

PEGASUS_HOME	is the suggested base directory of your the execution environment.
JAVA_HOME	should be set and point to a valid location to start the intended Java virtual machine as *$JA-VA_HOME/bin/java*.
CLASSPATH	should be set to contain all necessary files for the execution environment. Please make sure that your *CLASSPATH* includes pointer to the replica implementation required jar files.

Properties

The complete branch of properties *pegasus.catalog.replica* including itself are interpreted by the prototype. While the *pegasus.catalog.replica* property itself steers the backend to connect to, any meaning of branched keys is dependent on the backend. The same key may have different meanings for different backends.

pegasus.catalog.replica	determines the name of the implementing class to load at run-time. If the class resides in *org.griphyn.common.catalog.replica* no prefix is required. Otherwise, the fully qualified class name must be specified.
pegasus.catalog.replica.file	is used by the SimpleFile implementation. It specifies the path to the file to use as the backend for the catalog.
pegasus.catalog.replica.db.driver	is used by a simple rDBMs implementation. The string is the fully-qualified class name of the JDBC driver used by the RDBMS implementer.
pegasus.catalog.replica.db.url	is the JDBC URL to use to connect to the database.
pegasus.catalog.replica.db.user	is used by a simple rDBMS implementation. It constitutes the database user account that contains the *RC_LFN* and *RC_ATTR* tables.
pegasus.catalog.replica.db.pass-word	is used by a simple RDBMS implementation. It constitutes the database user account that contains the *RC_LFN* and *RC_ATTR* tables.
pegasus.catalog.repli-ca.chunk.size	is used by **the pegasus-rc-client** for the bulk insert and delete operations. The value determines the number of lines that are read in at a time, and worked upon at together.

Commands

The command line tool provides a simplified shell-wrappable interface to manage a replica catalog backend. The commands can either be specified in a file in bulk mode, in a pipe, or as additional arguments to the invocation.

Note that you must escape special characters from the shell.

help	displays a small resume of the commands.
exit , **quit**	should only be used in interactive mode to exit the interactive mode.
clear	drops all contents from the backend. Use with special care!
insert <lfn> <pfn> [k=v […]]	inserts a given **lfn** and **pfn**, and an optional **site** string into the backend. If the site is not specified, a *null* value is inserted for the **site**.

delete <lfn> <pfn> [k=v […]]	removes a triple of **lfn**, **pfn** and, optionally, **site** from the replica backend. If the site was not specified, all matches of the **lfn pfn** pairs will be removed, regardless of the **site**.
lookup <lfn> [<lfn> […]]	retrieves one or more mappings for a given **lfn** from the replica backend.
remove <lfn> [<lfn> […]]	removes all mappings for each **lfn** from the replica backend.
list [lfn <pat>] [pfn <pat>] [<name> <pat>]	obtains all matches from the replica backend. If no arguments were specified, all contents of the replica backend are matched. You must use the word **lfn**, **pfn** or **<name>** before specifying a pattern. The pattern is meaningful only to the implementation. Thus, a SQL implementation may chose to permit SQL wild-card characters. A memory-resident service may chose to interpret the pattern as regular expression.
set [var [value]]	sets an internal variable that controls the behavior of the front-end. With no arguments, all possible behaviors are displayed. With one argument, just the matching behavior is listed. With two arguments, the matching behavior is set to the value.

Database Schema

The tables are set up as part of the PEGASUS database setup. The files concerned with the database have a suffix *-rc.sql*.

Authors

Karan Vahi `<vahi at isi dot edu>`

Gaurang Mehta `<gmetha at isi dot edu>`

Jens-S. Vöckler `<voeckler at isi dot dot edu>`

Pegasus Team http://pegasus.isi.edu/

Name

pegasus-remove — removes a workflow that has been planned and submitted using pegasus-plan and pegasus-run

Synopsis

pegasus-remove [**-d** *dagid*] [**-v**] [*rundir*]

Description

The pegasus-remove command remove a submitted/running workflow that has been planned and submitted using **pegasus-plan** and **pegasus-run**. The command can be invoked either in the planned directory with no options and arguments or just the full path to the run directory.

Another way to remove a workflow is with the pegasus-halt command. The difference is that pegasus-halt will allow current jobs to finish gracefully before stopping the workflow.

Options

By default pegasus-remove does not require any options or arguments if invoked from within the planned workflow directory. If running the command outside the workflow directory then a full path to the workflow directory needs to be specified or the *dagid* of the workflow to be removed.

pegasus-remove takes the following options:

-d *dagid* , **--dagid** *dagid*
The workflow dagid to remove

-v , **--verbose**
Raises debug level. Each invocation increase the level by 1.

rundir
Is the full qualified path to the base directory containing the planned workflow DAG and submit files. This is optional if pegasus-remove command is invoked from within the run directory.

Return Value

If the workflow is removed successfully pegasus-remove returns with an exit code of 0. However, in case of error, a non-zero exit code indicates problems. An error message clearly marks the cause.

Files

The following files are opened:

braindump
This file is located in the rundir. pegasus-remove uses this file to find out paths to several other files.

Environment Variables

PATH
The path variable is used to locate binary for **condor_rm**.

See Also

pegasus-plan(1), pegasus-run(1)

Authors

Gaurang Mehta <gmehta at isi dot edu>

Jens-S. Vöckler <voeckler at isi dot edu>

Pegasus Team http://pegasus.isi.edu

Name

pegasus-run — executes a workflow that has been planned using *pegasus-plan*.

Synopsis

pegasus-run [-**D***property=value*…][**-c** *propsfile*][**-d** *level*]
[**-v**][--grid*][rundir]

Description

The **pegasus-run** command executes a workflow that has been planned using **pegasus-plan**. By default pegasus-run can be invoked either in the planned directory with no options and arguments or just the full path to the run directory. **pegasus-run** also can be used to resubmit a failed workflow by running the same command again.

Options

By default **pegasus-run** does not require any options or arguments if invoked from within the planned workflow directory. If running the command outside the workflow directory then a full path to the workflow directory needs to be specified.

pegasus-run takes the following options

-**D***property=value*	The **-D** option allows an advanced user to override certain properties which influence **pegasus-run**. One may set several CLI properties by giving this option multiple times.
	The **-D** option(s) must be the first option on the command line. CLI properties take precedence over the file-based properties of the same key.
	See the **PROPERTIES** section below.
-**c** *propsfile* , **--conf** *propsfile*	Provide a property file to override the default Pegasus properties file from the planning directory. Ordinary users do not need to use this option unless the specifically want to override several properties
-**d** *level* , **--debug** *level*	Set the debug level for the client. Default is 0.
-**v** , **--verbose**	Raises debug level. Each invocation increase the level by 1.
--grid	Enable grid checks to see if your submit machine is GRID enabled.
rundir	Is the full qualified path to the base directory containing the planned workflow DAG and submit files. This is optional if the **pegasus-run** command is invoked from within the run directory.

Return Value

If the workflow is submitted for execution **pegasus-run** returns with an exit code of 0. However, in case of error, a non-zero return value indicates problems. An error message clearly marks the cause.

Files

The following files are created, opened or written to:

braindump	This file is located in the rundir. pegasus-run uses this file to find out paths to several other files, properties configurations etc.
pegasus.?????????.properties	This file is located in the rundir. pegasus-run uses this properties file by default to configure its internal settings.

workflowname.dag	pegasus-run uses the workflowname.dag or workflowname.sh file and submits it either to condor for execution or runs it locally in a shell environment

Properties

pegasus-run reads its properties from several locations.

RUNDIR/pega-sus.??????????.properties	The default location for pegasus-run to read the properties from
--conf propfile	properties file provided in the conf option replaces the default properties file used.
$HOME/.pegasusrc	will be used if neither default rundir properties or --conf propertiesfile are found.
	Additionally properties can be provided individually using the **-Dprop-key**=*propvalue* option on the command line before all other options. These properties will override properties provided using either **--conf** or *RUNDIR/pegasus.???????.properties* or the *$HOME/.pegasusrc*
	The merge logic is CONF PROPERTIES || DEFAULT RUNDIR PROPERTIES || PEGASUSRC overriden by Command line properties

Environment Variables

PATH	The path variable is used to locate binaries for condor-submit-dag, condor-dagman, condor-submit,pegasus-submit-dag, pegasus-dagman and pegasus-monitord

See Also

pegasus-plan(1)

Authors

Gaurang Mehta `<gmehta at isi dot edu>`

Jens-S. Vöckler `<voeckler at isi dot edu>`

Pegasus Team http://pegasus.isi.edu

Name

pegasus-s3 — Upload, download, delete objects in Amazon S3

Synopsis

pegasus-s3 help
pegasus-s3 ls [options] *URL*
pegasus-s3 mkdir [options] *URL*...
pegasus-s3 rmdir [options] URL…
pegasus-s3 rm [options] [*URL*...]
pegasus-s3 put [options] *FILE URL*
pegasus-s3 get [options] *URL* [*FILE*]
pegasus-s3 lsup [options] *URL*
pegasus-s3 rmup [options] *URL* [*UPLOAD*]
pegasus-s3 cp [options] *SRC... DEST*

Description

pegasus-s3 is a client for the Amazon S3 object storage service and any other storage services that conform to the Amazon S3 API, such as Eucalyptus Walrus.

Options

Global Options

-h , **--help**	Show help message for subcommand and exit
-d , **--debug**	Turn on debugging
-v , **--verbose**	Show progress messages
-C *FILE* , **--con-f**=*FILE*	Path to configuration file

ls Options

-l , **--long**	Use long listing format that includes size, etc.

rm Options

-f , **--force**	If the URL does not exist, then ignore the error.
-F *FILE* , **--file**=*FILE*	File containing a list of URLs to delete

put Options

-r , **--recursive**	Upload all files in the directory named FILE to keys with prefix URL.
-c *X* , **--chunksize**=*X*	Set the chunk size for multipart uploads to X MB. A value of 0 disables multipart uploads. The default is 10MB, the min is 5MB and the max is 1024MB. This parameter only applies for sites that support multipart uploads (see multipart_uploads configuration parameter in the **CONFIGURATION** section). The maximum number of chunks is 10,000, so if you are uploading a large file, then the chunk size is automatically increased to enable the upload. Choose smaller values to reduce the impact of transient failures.
-p *N* , **--parallel**=*N*	Use N threads to upload *FILE* in parallel. The default value is 4, which enables parallel uploads with 4 threads. This parameter is only valid if the site supports mulipart uploads and the **--chunksize** parameter is not 0. Otherwise parallel uploads are disabled.
-b , **--create-bucket**	Create the destination bucket if it does not already exist

get Options

-r , **--recursive**	Download all keys that match URL exactly or begin with URL+"/". For example, *pegasus-s3 get -r s3://u@h/bucket/key* will match both *key* and *key/foo* but not *keyfoo*. Since S3 allows names to exist as both keys (the bare *key*) and folders (the *key* in *key/foo*), but file systems do not, you will get an error when using **-r**/**--recursive** on a bucket that contains such duplicate names. An entire bucket can be downloaded at once by specifying only the bucket name in URL.
-c *X* , **--chunksize**=*X*	Set the chunk size for parallel downloads to X megabytes. A value of 0 will avoid chunked reads. This option only applies for sites that support ranged downloads (see ranged_downloads configuration parameter). The default chunk size is 10MB, the min is 1MB and the max is 1024MB. Choose smaller values to reduce the impact of transient failures.
-p *N* , **--parallel**=*N*	Use N threads to upload FILE in parallel. The default value is 4, which enables parallel downloads with 4 threads. This parameter is only valid if the site supports ranged downloads and the **--chunksize** parameter is not 0. Otherwise parallel downloads are disabled.

rmup Options

-a , **--all**	Cancel all uploads for the specified bucket

cp Options

-c , **--create-dest**	Create the destination bucket if it does not exist.
-r , **--recursive**	If SRC is a bucket, copy all of the keys in that bucket to DEST. In that case DEST must be a bucket.
-f , **--force**	If DEST exists, then overwrite it.

Subcommands

pegasus-s3 has several subcommands for different storage service operations.

help	The help subcommand lists all available subcommands.
ls	The **ls** subcommand lists the contents of a URL. If the URL does not contain a bucket, then all the buckets owned by the user are listed. If the URL contains a bucket, but no key, then all the keys in the bucket are listed. If the URL contains a bucket and a key, then all keys in the bucket that begin with the specified key are listed.
mkdir	The **mkdir** subcommand creates one or more buckets.
rmdir	The **rmdir** subcommand deletes one or more buckets from the storage service. In order to delete a bucket, the bucket must be empty.
rm	The **rm** subcommand deletes one or more keys from the storage service.
put	The **put** subcommand stores the file specified by FILE in the storage service under the bucket and key specified by URL. If the URL contains a bucket, but not a key, then the file name is used as the key. If URL ends with a "/", then the file name is appended to the URL to create the key name (e.g. *pegasus-s3 put foo s3://u@h/bucket/key* will create a key called "key", while *pegasus-s3 put foo s3://u@h/bucket/key/* will create a key called "key/foo". The same is true of directories when used with the **-r**/**--recursive** option.

The **put** subcommand can do both chunked and parallel uploads if the service supports multipart uploads (see **multipart_uploads** in the **CONFIGURATION** section). Currently only Amazon S3 supports multipart uploads.

This subcommand will check the size of the file to make sure it can be stored before attempting to store it.

Chunked uploads are useful to reduce the probability of an upload failing. If an upload is chunked, then **pegasus-s3** issues separate PUT requests for each chunk of the file. Specifying smaller chunks (using **--**

chunksize) will reduce the chances of an upload failing due to a transient error. Chunksizes can range from 5 MB to 1GB (chunk sizes smaller than 5 MB produced incomplete uploads on Amazon S3). The maximum number of chunks for any single file is 10,000, so if a large file is being uploaded with a small chunksize, then the chunksize will be increased to fit within the 10,000 chunk limit. By default, the file will be split into 10 MB chunks if the storage service supports multipart uploads. Chunked uploads can be disabled by specifying a chunksize of 0. If the upload is chunked, then each chunk is retried independently under transient failures. If any chunk fails permanently, then the upload is aborted.

Parallel uploads can increase performance for services that support multipart uploads. In a parallel upload the file is split into N chunks and each chunk is uploaded concurrently by one of M threads in first-come, first-served fashion. If the chunksize is set to 0, then parallel uploads are disabled. If M > N, then the actual number of threads used will be reduced to N. The number of threads can be specified using the -- parallel argument. If --parallel is 1, then only a single thread is used. The default value is 4. There is no maximum number of threads, but it is likely that the link will be saturated by 4 to 8 threads.

Under certain circumstances, when a multipart upload fails it could leave behind data on the server. When a failure occurs the **put** subcommand will attempt to abort the upload. If the upload cannot be aborted, then a partial upload may remain on the server. To check for partial uploads run the **lsup** subcommand. If you see an upload that failed in the output of **lsup**, then run the **rmup** subcommand to remove it.

get
The **get** subcommand retrieves an object from the storage service identified by URL and stores it in the file specified by FILE. If FILE is not specified, then the part of the key after the last "/" is used as the file/ directory name, and the results are placed in the current working directory. If FILE ends with a "/", then the last component of the key name is appended to FILE to create the output path (e.g. *pegasus-s3 get s3://u@h/bucket/key /tmp/* will create a file called */tmp/key* while *pegasus-s3 get s3://u@h/bucket/key / tmp/foo* will put the contents of *key* in a file called */tmp/foo*). The same is true of folders/directories with the **-r**/**--recursive** option.

The **get** subcommand can do both chunked and parallel downloads if the service supports ranged downloads (see **ranged_downloads** in the **CONFIGURATION** section). Currently only Amazon S3 has good support for ranged downloads. Eucalyptus Walrus supports ranged downloads, but version 1.6 is inconsistent with the Amazon interface and has a bug that causes ranged downloads to hang in some cases. It is recommended that ranged downloads not be used with Walrus 1.6.

Chunked downloads can be used to reduce the probability of a download failing. When a download is chunked, **pegasus-s3** issues separate GET requests for each chunk of the file. Specifying smaller chunks (using **--chunksize**) will reduce the chances that a download will fail to do a transient error. Chunk sizes can range from 1 MB to 1 GB. By default, a download will be split into 10 MB chunks if the site supports ranged downloads. Chunked downloads can be disabled by specifying a **--chunksize** of 0. If a download is chunked, then each chunk is retried independently under transient failures. If any chunk fails permanently, then the download is aborted.

Parallel downloads can increase performance for services that support ranged downloads. In a parallel download, the file to be retrieved is split into N chunks and each chunk is downloaded concurrently by one of M threads in a first-come, first-served fashion. If the chunksize is 0, then parallel downloads are disabled. If M > N, then the actual number of threads used will be reduced to N. The number of threads can be specified using the --parallel argument. If --parallel is 1, then only a single thread is used. The default value is 4. There is no maximum number of threads, but it is likely that the link will be saturated by 4 to 8 threads.

lsup
The **lsup** subcommand lists active multipart uploads. The URL specified should point to a bucket. This command is only valid if the site supports multipart uploads. The output of this command is a list of keys and upload IDs.

This subcommand is used with **rmup** to help recover from failures of multipart uploads.

rmup
The **rmup** subcommand cancels and active upload. The URL specified should point to a bucket, and UPLOAD is the long, complicated upload ID shown by the **lsup** subcommand.

This subcommand is used with **lsup** to recover from failures of multipart uploads.

cp
The **cp** subcommand copies keys on the server. Keys cannot be copied between accounts.

URL Format

All URLs for objects stored in S3 should be specified in the following format:

```
s3[s]://USER@SITE[/BUCKET[/KEY]]
```

The protocol part can be *s3://* or *s3s://*. If *s3s://* is used, then **pegasus-s3** will force the connection to use SSL and override the setting in the configuration file. If s3:// is used, then whether the connection uses SSL or not is determined by the value of the *endpoint* variable in the configuration for the site.

The *USER@SITE* part is required, but the *BUCKET* and *KEY* parts may be optional depending on the context.

The *USER@SITE* portion is referred to as the "identity", and the *SITE* portion is referred to as the "site". Both the identity and the site are looked up in the configuration file (see **CONFIGURATION**) to determine the parameters to use when establishing a connection to the service. The site portion is used to find the host and port, whether to use SSL, and other things. The identity portion is used to determine which authentication tokens to use. This format is designed to enable users to easily use multiple services with multiple authentication tokens. Note that neither the *USER* nor the *SITE* portion of the URL have any meaning outside of **pegasus-s3**. They do not refer to real usernames or hostnames, but are rather handles used to look up configuration values in the configuration file.

The BUCKET portion of the URL is the part between the 3rd and 4th slashes. Buckets are part of a global namespace that is shared with other users of the storage service. As such, they should be unique.

The KEY portion of the URL is anything after the 4th slash. Keys can include slashes, but S3-like storage services do not have the concept of a directory like regular file systems. Instead, keys are treated like opaque identifiers for individual objects. So, for example, the keys *a/b* and *a/c* have a common prefix, but cannot be said to be in the same *directory*.

Some example URLs are:

```
s3://ewa@amazon
s3://juve@skynet/gideon.isi.edu
s3://juve@magellan/pegasus-images/centos-5.5-x86_64-20101101.part.1
s3s://ewa@amazon/pegasus-images/data.tar.gz
```

Configuration

Each user should specify a configuration file that **pegasus-s3** will use to look up connection parameters and authentication tokens.

Search Path

This client will look in the following locations, in order, to locate the user's configuration file:

1. The -C/--conf argument

2. The S3CFG environment variable

3. $HOME/.pegasus/s3cfg

4. $HOME/.s3cfg

If it does not find the configuration file in one of these locations it will fail with an error. The $HOME/.s3cfg location is only supported for backward-compatibility. $HOME/.pegasus/s3cfg should be used instead.

Configuration File Format

The configuration file is in INI format and contains two types of entries.

The first type of entry is a site entry, which specifies the configuration for a storage service. This entry specifies the service endpoint that **pegasus-s3** should connect to for the site, and some optional features that the site may support. Here is an example of a site entry for Amazon S3:

```
[amazon]
```

```
endpoint = http://s3.amazonaws.com/
```

The other type of entry is an identity entry, which specifies the authentication information for a user at a particular site. Here is an example of an identity entry:

```
[pegasus@amazon]
access_key = 90c4143642cb097c88fe2ec66ce4ad4e
secret_key = a0e3840e5baee6abb08be68e81674dca
```

It is important to note that user names and site names used are only logical—they do not correspond to actual hostnames or usernames, but are simply used as a convenient way to refer to the services and identities used by the client.

The configuration file should be saved with limited permissions. Only the owner of the file should be able to read from it and write to it (i.e. it should have permissions of 0600 or 0400). If the file has more liberal permissions, then **pegasus-s3** will fail with an error message. The purpose of this is to prevent the authentication tokens stored in the configuration file from being accessed by other users.

Configuration Variables

endpoint (site)	The URL of the web service endpoint. If the URL begins with *https*, then SSL will be used.
max_object_size (site)	The maximum size of an object in GB (default: 5GB)
multipart_uploads (site)	Does the service support multipart uploads (True/False, default: False)
ranged_downloads (site)	Does the service support ranged downloads? (True/False, default: False)
access_key (identity)	The access key for the identity
secret_key (identity)	The secret key for the identity

Example Configuration

This is an example configuration that specifies a two sites (amazon and magellan) and three identities (`pegasus@amazon`, `juve@magellan`, and `voeckler@magellan`). For the amazon site the maximum object size is 5TB, and the site supports both multipart uploads and ranged downloads, so both uploads and downloads can be done in parallel.

```
[amazon]
endpoint = https://s3.amazonaws.com/
max_object_size = 5120
multipart_uploads = True
ranged_downloads = True

[pegasus@amazon]
access_key = 90c4143642cb097c88fe2ec66ce4ad4e
secret_key = a0e3840e5baee6abb08be68e81674dca

[magellan]
# NERSC Magellan is a Eucalyptus site. It doesn't support multipart uploads,
# or ranged downloads (the defaults), and the maximum object size is 5GB
# (also the default)
endpoint = https://128.55.69.235:8773/services/Walrus

[juve@magellan]
access_key = quwefahsdpfwlkewqjsdoijldsdf
secret_key = asdfa9wejalsdjfljasldjfasdfa

[voeckler@magellan]
# Each site can have multiple associated identities
access_key = asdkfaweasdfbaeiwhkjfbaqwhei
secret_key = asdhfuinakwjelfuhalsdflahsdl
```

Example

List all buckets owned by identity *user@amazon*:

```
$ pegasus-s3 ls s3://user@amazon
```

List the contents of bucket *bar* for identity *user@amazon*:

```
$ pegasus-s3 ls s3://user@amazon/bar
```

List all objects in bucket *bar* that start with *hello*:

```
$ pegasus-s3 ls s3://user@amazon/bar/hello
```

Create a bucket called *mybucket* for identity *user@amazon*:

```
$ pegasus-s3 mkdir s3://user@amazon/mybucket
```

Delete a bucket called *mybucket*:

```
$ pegasus-s3 rmdir s3://user@amazon/mybucket
```

Upload a file *foo* to bucket *bar*:

```
$ pegasus-s3 put foo s3://user@amazon/bar/foo
```

Download an object *foo* in bucket *bar*:

```
$ pegasus-s3 get s3://user@amazon/bar/foo foo
```

Upload a file in parallel with 4 threads and 100MB chunks:

```
$ pegasus-s3 put --parallel 4 --chunksize 100 foo s3://user@amazon/bar/foo
```

Download an object in parallel with 4 threads and 100MB chunks:

```
$ pegasus-s3 get --parallel 4 --chunksize 100 s3://user@amazon/bar/foo foo
```

List all partial uploads for bucket *bar*:

```
$ pegasus-s3 lsup s3://user@amazon/bar
```

Remove all partial uploads for bucket *bar*:

```
$ pegasus-s3 rmup --all s3://user@amazon/bar
```

Return Value

pegasus-s3 returns a zero exist status if the operation is successful. A non-zero exit status is returned in case of failure.

Author

Gideon Juve <gideon@isi.edu>

Pegasus Team http://pegasus.isi.edu

Name

pegasus-sc-converter — A client to convert site catalog from one format to another format.

Synopsis

pegasus-sc-converter [**-v**] [**-V**] [**-h**] [**-D***property*=*value*…]
[**-I** *fmt*] [**-O** *fmt*]
-i *infile*[,*infile*,…] **-o** *outfile*

Description

The **pegasus-sc-converter** program is used to convert the site catalog from one format to another.

Currently, the following formats of site catalog exist.

XML4   This format is a superset of previous formats. All information about a site that can be described about a site can be described in this format. In addition, the user has finer grained control over the specification of directories and FTP servers that are accessible at the **head node** and the **worker node**. The user can also specify which different file-servers for read/write operations

A sample entry in this format looks as follows

```
<site handle="osg" arch="x86" os="LINUX" osrelease="" osversion="" glibc="">
       <grid  type="gt2" contact="viz-login.isi.edu/jobmanager-pbs" scheduler="PBS"
 jobtype="compute"/>
       <grid  type="gt2" contact="viz-login.isi.edu/jobmanager-fork" scheduler="Fork"
 jobtype="auxillary"/>

       <directory path="/tmp" type="local-scratch">
               <file-server operation="put" url="file:///tmp"/>
       </directory>

       <profile namespace="pegasus" key="style">condor</profile>
       <profile namespace="condor" key="universe">vanilla</profile>
</site>
```

This format conforms to the XML schema found at http://pegasus.isi.edu/schema/sc-4.0.xsd.

XML3   This format is a superset of previous formats. All information about a site that can be described about a site can be described in this format. In addition, the user has finer grained control over the specification of directories and FTP servers that are accessible at the **head node** and the **worker node**.

A sample entry in this format looks as follows

```
<site  handle="local" arch="x86" os="LINUX">
  <grid  type="gt2" contact="viz-login.isi.edu/jobmanager-pbs" scheduler="PBS"
 jobtype="compute"/>
  <grid  type="gt2" contact="viz-login.isi.edu/jobmanager-fork" scheduler="Fork"
 jobtype="auxillary"/>
  <head-fs>
    <scratch>
      <shared>
        <file-server protocol="gsiftp" url="gsiftp://viz-login.isi.edu" mount-point="/
scratch">
        </file-server>
        <internal-mount-point mount-point="/scratch" free-size="null" total-size="null"/>
      </shared>
    </scratch>
    <storage>
      <shared>
        <file-server protocol="gsiftp" url="gsiftp://viz-login.isi.edu" mount-point="/
scratch">
        </file-server>
        <internal-mount-point mount-point="/scratch" free-size="null" total-size="null"/>
      </shared>
    </storage>
  </head-fs>
  <replica-catalog  type="LRC" url="rlsn://smarty.isi.edu">
  </replica-catalog>
```

```
  <profile namespace="env" key="GLOBUS_LOCATION" >/nfs/software/globus/default</profile>
  <profile namespace="env" key="LD_LIBRARY_PATH" >/nfs/software/globus/default/lib</
profile>
  <profile namespace="env" key="PEGASUS_HOME" >/nfs/software/pegasus/default</profile>
</site>
```

This format conforms to the XML schema found at http://pegasus.isi.edu/schema/sc-3.0.xsd.

Options

-i *infile*[,*infile*,…] , **--input** *infile*[,*infile*,…]	The comma separated list of input files that need to be converted to a file in the format specified by **--oformat** option.
-o *outfile* , **--output** *outfile*	The output file to which the output needs to be written out to.

Other Options

-O *fmt* , **--oformat** *fmt*	The output format of the output file.
	Valid values for the output format is **XML3**, **XML4**.
-v , **--verbose**	Increases the verbosity of messages about what is going on.
	By default, all FATAL ERROR, ERROR , WARNINGS and INFO messages are logged.
-V , **--version**	Displays the current version number of the Pegasus Workflow Planner Software.
-h , **--help**	Displays all the options to the **pegasus-plan** command.

Example

```
pegasus-sc-converter -i sites.xml -o sites.xml.new -O XML3 -vvvvv
```

Authors

Karan Vahi `<vahi at isi dot edu>`

Gaurang Mehta `<gmehta at isi dot edu>`

Pegasus Team http://pegasus.isi.edu

Name

pegasus-service — Runs the Pegasus Service server

Synopsis

pegasus-service [options]

Options

-H , **--host**   Hostname on which the service listens for request. Default: 127.0.0.1s

-p , **--port**   Port on which the service listens for requests. Default: 5000

-d , **--debug**   Enable debugging

-h , **--help**   Print help message

Configuration

The authentication/authorization settings can be specified in the configuration file.

Authors

Pegasus Team <pegasus@isi.edu>

Name

pegasus-statistics — A tool to generate statistics about the workflow run.

Synopsis

pegasus-statistics [**-h**|**--help**]
            [**-o**|**--output** *dir*]
            [**-c**|**--conf** *propfile*]
            [**-p**|**--statistics-level** *level*]
            [**-t**|**--time-filter** *filter*]
            [**-i**|**--ignore-db-inconsistency**]
            [**-v**|**--verbose**]
            [**-q**|**--quiet**]
            [**-m**|**--multiple-wf**]
            [**-p**|**--ispmc**]
            [**-u**|**--isuuid**]
            [[*submitdir* ..] | [*workflow_uuid* ..]]

Description

pegasus-statistics generates statistics about the workflow run like total jobs/tasks/sub workflows ran, how many succeeded/failed etc. It generates job instance statistics like run time, condor queue delay etc. It generates invocation statistics information grouped by transformation name. It also generates job instance and invocation statistics information grouped by time and host.

Options

-h , **--help**	Prints a usage summary with all the available command-line options.
-o *dir* , **--output** *dir*	Writes the output to the given directory.
-c *propfile* , **--conf** *propfile*	The properties file to use. This option overrides all other property files.
-s *level* , **--statistics-level** *level*	Specifies the statistics information to generate. Valid levels are: **all**, **summary**, **wf_stats**, **jb_stats**, **tf_stats**, and **ti_stats**. Default is **summary**. The output generated by pegasus-statistics is based on the the *level* set:

- **all**: generates all the statistics information.

- **summary**: generates the workflow statistics summary. In the case of a hierarchical workflow the summary is across all sub workflows.

- **wf_stats**: generates the workflow statistics information of each individual workflow. In case of a hierarchical workflow the workflow statistics are created for each sub workflow.

- **jb_stats**: generates the job statistics information of each individual workflow. In case of hierarchical workflow the job statistics is created for each sub workflows. Note: Not supported when generating statistics over multiple workflows.

- **tf_stats**: generates the invocation statistics information of each individual workflow grouped by transformation name .In case of hierarchical workflow the transformation statistics is created for each sub workflows.

- **ti_stats**: generates the job instance and invocation statistics like total count and runtime grouped by time and host.

-t *filter* , **--time-filter** *filter*	Specifies the time filter to group the time statistics. Valid *filter* values are: **month**, **week**, **day**, **hour**. Default is **day**.

-i , **--ignore-db-inconsistency**	Turn off the the check for database consistency.
-v , **--verbose**	Increases the log level. If omitted, the default level will be set to WARNING. When this option is given, the log level is changed to INFO. If this option is repeated, the log level will be changed to DEBUG.
-q , **--quiet**	Decreases the log level. If omitted, the default level will be set to WARNING. When this option is given, the log level is changed to ERROR.
-m , **--multiple-wf**	Set this option when generating statistics over more than one workflow. The tool automatically sets this flag if multiple submit directories or multiple workflow UUIDs are provided. This option would need to be set explicitly only to generate statistics over all workflows in a single STAMPEDE database. NOTE: When workflows are specified as UUIDs the --conf options needs to be set for the tool to determine the STAMPEDE database URL.
-p , **--ispmc**	Set this flag to generate statistics for workflows which are run with PMC clustering enabled. It is recommended that this option be used when calculating statistics over multiple workflow runs.
-u , **--isuuid**	Set this option if the positional argument are workflow UUIDs. NOTE: When workflows are specified as UUIDs the --conf options needs to be set for the tool to determine the STAMPEDE database URL.

Example

Runs pegasus-statistics and writes the output to the given directory:

```
$ pegasus-statistics  -o /scratch/statistics /scratch/grid-setup/run0001
```

Runs pegasus-statistics over a workflow run identified by a single workflow UUID:

```
$ pegasus-statistics  --conf pegasusrc --isuuid 316f2986-7754-44ec-8b38-fcd0cb602ce0
```

Runs pegasus-statistics over a workflow run identified by a multiple workflow UUID:

```
$ pegasus-statistics  --conf pegasusrc --isuuid 316f2986-7754-44ec-8b38-fcd0cb602ce0 \
7ef77af8-4eb2-45ca-b37d-c5a02186133a
```

Runs pegasus-statistics over all workflows in the STAMPEDE database:

```
$ pegasus-statistics  --conf pegasusrc --multiple-wf
```

Authors

Prasanth Thomas Rajiv Mayani

Pegasus Team http://pegasus.isi.edu

Name

pegasus-status — Pegasus workflow- and run-time status

Synopsis

pegasus-status [**-h**|**--help**]
           [**-V**|**--version**] [**-v**|**--verbose**] [**-d**|**--debug**]
           [**-w**|**--watch** [*s*]]
           [**-L**|**--[no]legend**] [**-c**|**--[no]color**] [**-U**|**--[no]utf8**]
           [**-Q**|**--[no]queue**] [**-i**|**--[no]idle**] [**--[no]held**]
           [**--[no]heavy**] [**-S**|**--[no]success**]
           [**-j**|**--jobtype** *jt*] [**-s**|**--site** *sid*]
           [**-u**|**--user** *name*]
           { [**-l**|**--long**] | [**-r**|**--rows**] }
           [*rundir*]

Description

pegasus-status shows the current state of the Condor Q and a workflow, depending on settings. If no valid run directory could be determined, including the current directory, **pegasus-status** will show all jobs of the current user and no workflows. If a run directory was specified, or the current directory is a valid run directory, status about the workflow will also be shown.

Many options will modify the behavior of this program, not withstanding a proper UTF-8 capable terminal, watch mode, the presence of jobs in the queue, progress in the workflow directory, etc.

Options

-h , **--help**	Prints a concise help and exits.
-V , **--version**	Prints the version information and exits.
-w [*sec*] , **--watch** [*sec*]	This option enables the *watch mode*. In watch mode, the program repeatedly polls the status sources and shows them in an updating window. The optional argument *sec* to this option determines how often these sources are polled.
	We *strongly* recommend to set this interval not too low, as frequent polling will degrade the scheduler performance and increase the host load. In watch mode, the terminal size is the limiting factor, and parts of the output may be truncated to fit it onto the given terminal.
	Watch mode is disabled by default. The *sec* argument defaults to 60 seconds.
-L , **--legend** , **--nolegend**	This option shows a legend explaining the columns in the output, or turns off legends.
	By default, legends are turned off to save terminal real estate.
-c , **--color** , **--nocolor**	This option turns on (or off) ANSI color escape sequences in the output. The single letter option can only switch on colors.
	By default, colors are turned off, as they will not display well on a terminal with black background.
-U , **--utf8** , **--noutf8**	This option turns on (or off) the output of Unicode box drawing characters as UTF-8 encoded sequences. The single option can only turn on box drawing characters.
	The defaults for this setting depend on the *LANG* environment variable. If the variable contains a value ending in something indicating UTF-8 capabilities, the option is turned on by default. It is off otherwise.
-Q , **--queue** , **--noqueue**	This option turns on (or off) the output from parsing Condor Q.

	By default, Condor Q will be parsed for jobs of the current user. If a workflow run directory is specified, it will furthermore be limited to jobs only belonging to the workflow.
-v , **--verbose**	This option increases the expert level, showing more information about the condor_q state. Being an incremental option, two increases are supported.
	Additionally, the signals *SIGUSR1* and *SIGUSR2* will increase and decrease the expert level respectively during run-time.
	By default, the simplest queue view is enabled.
-d , **--debug**	This is an internal debugging tool and should not be used outside the development team. As incremental option, it will show Pegasus-specific ClassAd tuples for each job, more in the second level.
	By default, debug mode is off.
-u *name* , **--user** *name*	This option permits to query the queue for a different user than the current one. This may be of interest, if you are debugging the workflow of another user.
	By default, the current user is assumed.
-i , **--idle** , **--noidle**	With this option, jobs in Condor state *idle* are omitted from the queue output.
	By default, *idle* jobs are shown.
--held , **--noheld**	This option enables or disabled showing of the reason a job entered Condor's *held* state. The reason will somewhat destroy the screen layout.
	By default, the reason is shown.
--heavy , **--noheavy**	If the terminal is UTF-8 capable, and output is to a terminal, this option decides whether to use heavyweight or lightweight line drawing characters.
	By default, heavy lines connect the jobs to workflows.
-j *jt* , **--jobtype** *jt*	This option filters the Condor jobs shown only to the Pegasus jobtypes given as argument or arguments to this option. It is a multi-option, and may be specified multiple times, and may use comma-separated lists. Use this option with an argument *help* to see all valid and recognized jobtypes.
	By default, all Pegasus jobtypes are shown.
-s *site* , **--site** *site*	This option limits the Condor jobs shown to only those pertaining to the (remote) site *site*. This is an multi-option, and may be specified multiple times, and may use comma-separated lists.
	By default, all sites are shown.
-l , **--long**	This option will show one line per sub-DAG, including one line for the workflow. If there is only a single DAG pertaining to the *rundir*, only total will be shown.
	This option is mutually exclusive with the **--rows** option. If both are specified, the **--long** option takes precedence.
	By default, only DAG totals (sums) are shown.
-r , **--rows** , **--norows**	This option is shows the workflow summary statistics in rows instead of columns. This option is useful for sending the statistics in email and later viewing them in a proportional font.
	This option is mutually exclusive with the **--long** option. If both are specified, the **--long** option takes precedence.

By default, the summary is shown in columns.

-S , **--success** , **--no-success**	This option modifies the previous **--long** option. It will omit (or show) fully successful sub-DAGs from the output.
	By default, all DAGs are shown.
rundir	This option show statistics about the given DAG that runs in *rundir*. To gather proper statistics, **pegasus-status** needs to traverse the directory and all sub-directories. This can become an expensive operation on shared filesystems.
	By default, the *rundir* is assumed to be the current directory. If the current directory is not a valid *rundir*, no DAG statistics will be shown.

Return Value

pegasus-status will typically return success in regular mode, and the termination signal in watch mode. Abnormal behavior will result in a non-zero exit code.

Example

pegasus-status	This invocation will parse the Condor Q for the current user and show all her jobs. Additionally, if the current directory is a valid Pegasus workflow directory, totals about the DAG in that directory are displayed.
pegasus-status -l rundir	As above, but providing a specific Pegasus workflow directory in argument *rundir* and requesting to itemize sub-DAGs.
pegasus-status -j help	This option will show all permissible job types and exit.
pegasus-status -vvw 300 -Ll	This invocation will parse the queue, print it in high-expert mode, show legends, itemize DAG statistics of the current working directory, and redraw the terminal every five minutes with updated statistics.

Restrictions

Currently only supports a single (optional) run directory. If you want to watch multiple run directories, I suggest to open multiple terminals and watch them separately. If that is not an option, or deemed too expensive, you can ask *pegasus-support at isi dot edu* to extend the program.

See Also

condor_q(1), pegasus-statistics(1)

Authors

Jens-S. Vöckler <voeckler at isi dot edu>

Gaurang Mehta <gmehta at isi dot edu>

Pegasus Team http://pegasus.isi.edu/

Name

pegasus-submit-dag — Wrapper around *condor_submit_dag*. Not to be run by user.

Description

The **pegasus-submit-dag** is a wrapper that invokes **condor_submit_dag**. This is started automatically by **pegasus-run**. **DO NOT USE DIRECTLY**

Return Value

If the workflow is submitted succesfully **pegasus-submit-dag** exits with 0, else exits with non-zero.

Environment Variables

PATH   The path variable is used to locate binary for **condor_submit_dag** and **pegasus-dagman**

See Also

pegasus-run(1) pegasus-dagman(1)

Authors

Gaurang Mehta <gmehta at isi dot edu>

Pegasus Team http://pegasus.isi.edu

Name

pegasus-submitdir — Manage a workflow submit directory.

Synopsis

pegasus-submitdir *COMMAND* [options] *SUBMITDIR*

Description

pegasus-submitdir is used to manage submit directories generated by the Pegasus planner.

The **archive** command significantly reduces the size of workflow submit directories by compressing the data in a way such that it remains accessible to tools such as pegasus-statistics, pegasus-plots, and pegasus-analyzer.

The **extract** command reverses the effect of the **archive** command.

The **move** command relocates a submit directory and updates relevant pointers in the database so that it can still be accessed through the dashboard.

The **delete** command removes the submit directory and cleans up any associated records in the user's master database.

The **attach** command adds a submit dir to the master database that drives the dashboard.

The **detach** command removes a submit dir from the master database that drives the dashboard.

Commands

archive SUBMITDIR	Compresses a workflow submit directory in a way that allows pegasus-dashboard, pegasus-statistics, pegasus-plots, and pegasus-analyzer to keep working. It creates a gzipped tar archive of the submit files and logs that excludes files such as the workflow database, braindump file, and monitord logs, which are used by pegasus reporting tools.
extract SUBMITDIR	Uncompresses a previously archived submit directory. This option returns the submit directory to the state it was before **pegasus-submitdir archive** was applied to it.
move SUBMITDIR DEST	Move a workflow submit dir from SUBMITDIR to DEST. This operation updates the relevant database records so that the dashboard continues to function. DEST can be either an existing directory, in which case the submit dir becomes a subdirectory, or a new path, in which case the submit dir is renamed. **IMPORTANT** This operation should only be performed on workflows that will not be resubmitted in the future. Moving a workflow does not update absolute paths in any of the submit files, so after a workflow has been moved it is not possible to rerun it.
delete SUBMITDIR	Delete a workflow submit dir. This operation removes all related records from the user's master database, including ensemble manager records. Deleted workflows do not appear in the dashboard.
attach SUBMITDIR	Add entries for the workflow in SUBMITDIR to the user's master db. If the workflow is already in the master db, then update the db_url and submit_dir fields to match the actual path of the submit dir. This command will create master_workflow and master_workflowstate entries in the master db for the root workflow in SUBMITDIR.
detach [--wf-uuid <WF_UUID>] SUBMITDIR	Remove entries for the workflow in SUBMITDIR from the user's master db. This command will delete any entries in the master_workflow and master_workflowstate tables.

Global Options

-h , **--help**   Prints a usage summary with all the available command-line options.

Authors

Gideon Juve <gideon@isi.edu [mailto:gideon@isi.edu]>

Pegasus Team http://pegasus.isi.edu

Name

pegasus-tc-client — A full featured generic client to handle adds, deletes and queries to the Transformation Catalog (TC).

Synopsis

pegasus-tc-client [**-D***property=value*…] [*-h*] [*-v*] [*-V*]
OPERATION TRIGGERS [*OPTIONS*]

Description

The **pegasus-tc-client** command is a generic client that performs the three basic operation of adding, deleting and querying of any Transformation Catalog implemented to the TC API. The client implements all the operations supported by the TC API. It is up to the TC implementation whether they support all operations or modes.

The following 3 operations are supported by the **pegasus-tc-client**. One of these operations have to be specified to run the client.

ADD   This operation allows the client to add or update entries in the Transformation Catalog. Entries can be added one by one on the command line or in bulk by using the *BULK* Trigger and providing a file with the necessary entries. Also Profiles can be added to either the logical transformation or the physical transformation.

DELETE   This operation allows the client to delete entries from the Transformation Catalog. Entries can be deleted based on logical transformation, by resource, by transformation type as well as the transformation system information. Also Profiles associated with the logical or physical transformation can be deleted.

QUERY   This operation allows the client to query for entries from the Transformation Catalog. Queries can be made for printing all the contents of the Catalog or for specific entries, for all the logical transformations or resources etc.

See the **TRIGGERS** and **VALID COMBINATIONS** section for more details.

Operations

To select one of the 3 operations.

-a, --add   Perform addition operations on the TC.

-d, --delete   Perform delete operations on the TC.

-q, --query   Perform query operations on the TC.

Triggers

Triggers modify the behavior of an **OPERATION**. For example, if you want to perform a bulk operation you would use a *BULK* Trigger or if you want to perform an operation on a Logical Transformation then you would use the *LFN* Trigger.

The following 7 Triggers are available. See the **VALID COMBINATIONS** section for the correct grouping and usage.

-B   Triggers a bulk operation.

-L   Triggers an operation on a logical transformation.

-P   Triggers an operation on a physical transformation

-R   Triggers an operation on a resource.

-E   Triggers an operation on a Profile.

-T	Triggers an operation on a Type.
-S	Triggers an operation on a System information.

Options

The following options are applicable for all the operations.

-D*property=value*	The -D options allows an experienced user to override certain properties which influence the program execution, among them the default location of the user's properties file and the PEGASUS home location. One may set several CLI properties by giving this option multiple times. The **-D** option(s) must be the first option on the command line. A CLI property take precedence over the properties file property of the same key.
-l, --lfn *logical*	The logical transformation to be added. The format is: **NAMESPACE::NAME:VERSION**. The name is always required, namespace and version are optional.
-p, --pfn *physical*	The physical transformation to be added. For INSTALLED executables its a local file path, for all others its a url.
-t, --type *type*	The type of physical transformation. Valid values are: INSTALLED, STATIC_BINARY, DYNAMIC_BINARY, SCRIPT, SOURCE, PACMAN_PACKAGE.
-r, --resource *resource*	The resourceID where the transformation is located.
-e, --profile *profiles*	The profiles for the transformation. Multiple profiles of same namespace can be added simultaneously by separating them with a comma **","**. Each profile section is written as **NAMESPACE::KEY=VALUE,KEY2=VALUE2** e.g. ENV::JAVA_HOME=/usr/bin/java2,PEGASUS_HOME=/usr/local/pegasus. To add multiple namespaces you need to repeat the -e option for each namespace. e.g. -e ENV::JAVA_HOME=/usr/bin/java -e GLOBUS::JobType=MPI,COUNT=10
-s, --system *systeminfo*	The architecture, os, osversion and glibc if any for the executable. Each system info is written in the form **ARCH::OS:OSVER:GLIBC**
-v, --verbose	Displays the output in verbose mode (Lots of Debugging info).
-V, --version	Displays the Pegasus version.
-h, --help	Generates help

Other Options

-o, --oldformat	Generates the output in the old single line format
-c, --conf	path to property file

Valid Combinations

The following are valid combinations of **OPERATIONS, TRIGGERS, OPTIONS** for the **pegasus-tc-client**.

ADD

Add TC Entry	-a -l *lfn* -p *pfn* -t *type* -r *resource* -s *system* [-e *profiles*…]
	Adds a single entry into the transformation catalog.

Add PFN Profile	-a -P -E -p *pfn* -t *type* -r *resource* -e *profiles* …	

Adds profiles to a specified physical transformation on a given resource and of a given type.

Add LFN Profile	-a -L -E -l *lfn* -e *profiles* …

Adds profiles to a specified logical transformation.

Add Bulk Entries	-a -B -f *file*

Adds entries in bulk mode by supplying a file containing the entries. The format of the file contains 6 columns. E.g.

```
#RESOURCE    LFN          PFN       TYPE       SYSINFO       PROFILES
#
isi NS::NAME:VER  /bin/date  INSTALLED  ARCH::OS:OSVERS:GLIBC
 NS::KEY=VALUE,KEY=VALUE;NS2::KEY=VALUE,KEY=VALUE
```

DELETE

Delete all TC	-d -BPRELST

Deletes the entire contents of the TC.

WARNING : USE WITH CAUTION.

Delete by LFN	-d -L -l *lfn* [-r *resource*] [-t *type*]

Deletes entries from the TC for a particular logical transformation and additionally a resource and or type.

Delete by PFN	-d -P -l *lfn* -p *pfn* [-r *resource*] [-t *type*]

Deletes entries from the TC for a given logical and physical transformation and additionally on a particular resource and or of a particular type.

Delete by Type	-d -T -t *type* [-r *resource*]

Deletes entries from TC of a specific type and/or on a specific resource.

Delete by Resource	-d -R -r *resource*

Deletes the entries from the TC on a particular resource.

Delete by SysInfo	-d -S -s *sysinfo*

Deletes the entries from the TC for a particular system information type.

Delete Pfn Profile	-d -P -E -p *pfn* -r *resource* -t *type* [-e *profiles* ..]

Deletes all or specific profiles associated with a physical transformation.

Delete Lfn Profile	-d -L -E -l *lfn* -e *profiles* ….

Deletes all or specific profiles associated with a logical transformation.

QUERY

Query Bulk	-q -B

Queries for all the contents of the TC. It produces a file format TC which can be added to another TC using the bulk option.

Query LFN	-q -L [-r *resource*] [-t *type*]

Queries the TC for logical transformation and/or on a particular resource and/or of a particular type.

Query PFN	-q -P -l *lfn* [-r *resource*] [-t *type*]

Queries the TC for physical transformations for a give logical transformation and/or on a particular resource and/or of a particular type.

Query Resource	-q -R -l *lfn* [-t *type*]

Queries the TC for resources that are registered and/or resources registered for a specific type of transformation.

Query LFN Profile	-q -L -E -l *lfn*

Queries for profiles associated with a particular logical transformation

Query Pfn Profile	-q -P -E -p *pfn* -r *resource* -t *type*

Queries for profiles associated with a particular physical transformation

Properties

These are the properties you will need to set to use either the **File** or **Database** TC.

For more details please check the **$PEGASUS_HOME/etc/sample.properties** file.

pegasus.catalog.transformation	Identifies what impelemntation of TC will be used. If relative name is used then the path org.griphyn.cPlanner.tc is prefixed to the name and used as the class name to load. The default value if **Text**. Other supported mode is **File**
pegasus.catalog.transformation.file	The file path where the text based TC is located. By default the path **$PEGASUS_HOME/var/tc.data** is used.

Files

$PEGASUS_HOME/var/tc.data	is the suggested location for the file corresponding to the Transformation Catalog
$PEGASUS_HOME/etc/properties	is the location to specify properties to change what Transformation Catalog Implementation to use and the implementation related **PROPERTIES**.
pegasus.jar	contains all compiled Java bytecode to run the Pegasus planner.

Environment Variables

PEGASUS_HOME	Path to the PEGASUS installation directory.
JAVA_HOME	Path to the JAVA 1.4.x installation directory.
CLASSPATH	The classpath should be set to contain all necessary PEGASUS files for the execution environment. To automatically add the *CLASSPATH* to you environment, in the *$PEGASUS_HOME* directory run the script *source setup-user-env.csh* or *source setup-user-env.sh*.

Authors

Gaurang Mehta <gmehta at isi dot edu>

Karan Vahi <vahi at isi dot edu>

Pegasus Team http://pegasus.isi.edu

Name

pegasus-tc-converter — A client to convert transformation catalog from one format to another format.

Synopsis

pegasus-tc-converter [-**D**property=value…] [-**v**] [-**q**] [-**V**] [-**h**]
        [-**I** fmt] [-**O** fmt]
        [-**N** dbusername] [-**P** dbpassword] [-**U** dburl] [-**H** dbhost]
        -**i** infile[,infile,…] -**o** outfile

Description

The tc-convert program is used to convert the transformation catalog from one format to another.

Currently, the following formats of transformation catalog exist:

Text   This is a easy to read multi line textual format.

A sample entry in this format looks as follows:

```
tr example::keg:1.0 {
        site isi {
        profile env "JAVA_HOME" "/bin/java.1.6"
        pfn "/path/to/keg"
        arch  "x86"
        os    "linux"
        osrelease "fc"
        osversion "4"
        type "installed"
    }
}
```

File   This is a tuple based format which contains 6 columns.

```
RESOURCE  LFN  PFN  TYPE  SYSINFO  PROFILES
```

A sample entry in this format looks as follows

```
isi  example::keg:1.0  /path/to/keg  INSTALLED  INTEL32::LINUX:fc_4:
 env::JAVA_HOME="/bin/java.1.6"
```

Database   Only MySQL is supported for the time being.

Options

-Dproperty=value   The **-D** option allows an experienced user to override certain properties which influence the program execution, among them the default location of the user's properties file and the **PEGASUS_HOME** location. One may set several CLI properties by giving this option multiple times.

The **-D** option(s) must be the first option on the command line. CLI properties take precedence over the file-based properties of the same key.

-I fmt , **--iformat** fmt   The input format of the input files. Valid values for the input format are: **File**, **Text**, and **Database**.

-O fmt **--oformat** fmt The output format of the output file. Valid values for the output format are: **File**, **Text**, and **Database**.

-i infile[,infile,…] **--input** infile[,infile,…] The comma separated list of input files that need to be converted to a file in the format specified by the **--oformat** option.

-o outfile , **--output** out-file   The output file to which the output needs to be written out to.

Other Options

-N *dbusername* , **--db-user-name** *dbusername*	The database user name.
-P *dbpassword* , **--db-user-pwd** *dbpassword*	The database user password.
-U *dburl* , **--db-url** *dburl*	The database url.
-H *dbhost* , **--db-host** *dbhost*	The database host.
-v , **--verbose**	Increases the verbosity of messages about what is going on. By default, all FATAL ERROR, ERROR , CONSOLE and WARNINGS messages are logged.
-q , **--quiet**	Decreases the verbosity of messages about what is going on. By default, all FATAL ERROR, ERROR , CONSOLE and WARNINGS messages are logged.
-V , **--version**	Displays the current version number of the Pegasus Workflow Planner Software.
-h , **--help**	Displays all the options to the **pegasus-tc-converter** command.

Example

Text to file format conversion

```
pegasus-tc-converter -i tc.data -I File -o tc.txt  -O Text -v
```

File to Database(new) format conversion

```
pegasus-tc-converter -i tc.data -I File -N mysql_user -P mysql_pwd -U jdbc:mysql://localhost:3306/tc
 -H localhost -O Database -v
```

Database (username, password, host, url specified in properties file) to text format conversion

```
pegasus-tc-converter -I Database -o tc.txt -O Text -vvvvv
```

Authors

Prasanth Thomas

Pegasus Team http://pegasus.isi.edu

Name

pegasus-transfer — Handles data transfers for Pegasus workflows.

Synopsis

pegasus-transfer [**-h**]
         [**--file** *inputfile*]
         [**--threads** *number_threads*]
         [**--max-attempts** *attempts*]
         [**--threads** *threads*]
         [**--debug**]

Description

pegasus-transfer takes a JSON defined list of urls, either on stdin or with an input file, determines the correct tool to use for the transfer and executes the transfer. Some of the protocols pegasus-transfer can handle are GridFTP, SCP, SRM, Amazon S3, Google Storage, XRootD, HTTP, and local cp/symlinking. Failed transfers are retried.

Note that pegasus-transfer is a tool mostly used internally in Pegasus workflows, but the tool can be used stand alone as well.

Options

-h , **--help**	Prints a usage summary with all the available command-line options.
-f *inputfile* , **--file** *inputfile*	JSON transfer specification. If not given, stdin will be used.
-m , **--max-attempts** *attempts*	Maximum number of attempts for retrying failed transfers.
-t , **--threads** *number_threads*	The number of threads to use. This controls the parallelism of transfers.
-d , **--debug**	Enables debugging output.

Example

```
$ pegasus-transfer
[
 { "type": "transfer",
   "id": 1,
   "src_urls": [ { "site_label": "web", "url": "http://pegasus.isi.edu" } ],
   "dest_urls": [ { "site_label": "local", "url": "file:///tmp/index.html" } ]
 }
]
CTRL+D
```

Credential Handling

Credentials used for transfers can be specified with a combination of site labels in the input JSON format and environment variables. For example, give the following input file:

```
[
 { "type": "transfer",
   "id": 1,
   "src_urls": [ { "site_label": "isi", "url": "gsiftp://workflow.isi.edu/data/file.dat" } ],
   "dest_urls": [ { "site_label": "tacc_stampede", "url": "gsiftp://
gridftp.stampede.tacc.utexas.edu/scratch/file.dat" } ]
 }
]
```

pegasus-transfer will expect either one environment variable specifying one credential to be used on both end of the connection (X509_USER_PROXY), or two separate environment variables specifying two different credentials to be used on the two ends of the connection. The the latter case, the environment variables are derived from the site labels. In

the example above, the environment variables would be named X509_USER_PROXY_isi and X509_USER_PROX-Y_tacc_stampede

Threading

In order to speed up data transfers, pegasus-transfer will start a set of transfers in parallel using threads.

Preference of GFAL over GUC

JGlobus is no longer actively supported and is not in compliance RFC 2818. As a result cleanup jobs using pegasus-gridftp client would fail against the servers supporting the strict mode. We have removed the pegasus-gridftp client and now use gfal clients as globus-url-copy does not support removes. If gfal is not available, globus-url-copy is used for cleanup by writing out zero bytes files instead of removing them.

If you want to force globus-url-copy to be preferred over GFAL, set the PEGASUS_FORCE_GUC=1 environment variable in the site catalog for the sites you want the preference to be enforced. Please note that we expect globus-url-copy support to be completely removed in future releases of Pegasus due to the end of life of Globus Toolkit in 2018.

Author

Pegasus Team http://pegasus.isi.edu

Name

pegasus-version — print or match the version of the toolkit.

Synopsis

pegasus-version [-**D***property=value*] [**-m** [**-q**]] [**-V**] [**-f**] [**-l**]

Description

This program prints the version string of the currently active Pegasus toolkit on *stdout*.

pegasus-version is a simple command-line tool that reports the version number of the Pegasus distribution being used. In its most basic invocation, it will show the current version of the Pegasus software you have installed:

```
$ pegasus-version
3.1.0cvs
```

If you want to know more details about the installed version, i.e. which system it was compiled for and when, use the long or full mode:

```
$ pegasus-version -f
3.1.0cvs-x86_64_cent_5.6-20110706191019Z
```

Options

-**D***property=value*	The **-D** option allows an experienced user to override certain properties which influence the program execution, among them the default location of the user's properties file and the **PEGASUS_HOME** location. One may set several CLI properties by giving this option multiple times.
	The **-D** option(s) must be the first option on the command line. CLI properties take precedence over the file-based properties of the same key.
-**f** , **--full**	The **--full** mode displays internal build metrics, like OS type and libc version, addition to the version number. It appends the build time as time stamp to the version. The time stamp uses ISO 8601 format, and is a UTC stamp.
-**l** , **--long**	This option is an alias for **--full**.
-**V** , **--version**	Displays the version of the Pegasus planner you are using.
--verbose	is ignored in this tool. However, to provide a uniform interface for all tools, the option is recognized and will not trigger an error.

Return Value

The program will usually return with success (0). In match mode, if the internal version does not match the external installation, an exit code of 1 is returned. If run-time errors are detected, an exit code of 2 is returned, 3 for fatal errors.

Environment Variables

JAVA_HOME	should be set and point to a valid location to start the intended Java virtual machine as *$JAVA_HOME/bin/java*.

Example

```
$ pegasus-version
3.1.0cvs

$ pegasus-version -f
3.1.0cvs-x86_64_cent_5.6-20110706191019Z
```

Authors

Jens-S. Vöckler `<voeckler at isi dot edu>`

Pegasus Team http://pegasus.isi.edu

Chapter 18. Useful Tips

Migrating From Pegasus 4.5.X to Pegasus current version

Most of the migrations from one version to another are related to database upgrades, that is addressed by running the tool **pegasus-db-admin**.

Database Upgrades From Pegasus 4.5.X to Pegasus current version

Since Pegasus 4.5 all databases are managed by a single tool: **pegasus-db-admin**. Databases will be automatically updated when **pegasus-plan** is invoked, but WORKFLOW databases from past runs may not be updated accordingly. Since Pegasus 4.6.0, the **pegasus-db-admin** tool provides an option to automatically update all databases from completed workflows in the MASTER database. To enable this option, run the following command:

```
$ pegasus-db-admin update -a
Your database has been updated.
Your database is compatible with Pegasus version: 4.7.0

Verifying and updating workflow databases:
21/21

Summary:
Verified/Updated: 21/21
Failed: 0/21
Unable to connect: 0/21
Unable to update (active workflows): 0/21

Log files:
20161006T134415-dbadmin.out (Succeeded operations)
20161006T134415-dbadmin.err (Failed operations)
```

This option generates a log file for succeeded operations, and a log file for failed operations. Each file contains the list of URLs of the succeeded/failed databases.

Note that, if no URL is provided, the tool will create/use a SQLite database in the user's home directory: *${HOME}/.pegasus/workflow.db*.

For complete description of pegasus-db-admin, see the man page.

Migration from Pegasus 4.6 to 4.7

In addition to the database changes, in Pegasus 4.7 the default submit directory layout was changed from a flat structure where all submit files independent of the number of jobs in the workflow appeared in a single directory. For 4.7, the default is a hierarchal directory structure two levels deep. To use the earlier layout, set the following property

```
pegasus.dir.submit.mapper      Flat
```

Migrating From Pegasus <4.5 to Pegasus 4.5.X

Since Pegasus 4.5 all databases are managed by a single tool: **pegasus-db-admin**. Databases will be automatically updated when **pegasus-plan** is invoked, but it may require manually invocation of the **pegasus-db-admin** for other Pegasus tools.

The **check** command verifies if the database is compatible with the Pegasus' latest version. If the database is not compatible, it will print the following message:

```
$ pegasus-db-admin check
```

```
Your database is NOT compatible with version 4.5.0
```

If you are running the **check** command for the first time, the tool will prompt the following message:

```
Missing database tables or tables are not updated:
    dbversion
Run 'pegasus-db-admin update <path_to_database>' to create/update your database.
```

To update the database, run the following command:

```
$ pegasus-db-admin update
Your database has been updated.
Your database is compatible with Pegasus version: 4.5.0
```

The **pegasus-db-admin** tool can operate directly over a database URL, or can read configuration parameters from the properties file or a submit directory. In the later case, a database type should be provided to indicate which properties should be used to connect to the database. For example, the tool will seek for *pegasus.catalog.replica.db.** properties to connect to the JDBCRC database; or seek for *pegasus.catalog.master.url* (or *pegasus.dashboard.output*, which is deprecated) property to connect to the MASTER database; or seek for the *pegasus.catalog.workflow.url* (or *pegasus.monitord.output*, which is deprecated) property to connect to the WORKFLOW database. If none of these properties are found, the tool will connect to the default database in the user's home directory (sqlite:///${HOME}/.pegasus/workflow.db).

Example: connection by providing the URL to the database:

```
$ pegasus-db-admin create sqlite:///${HOME}/.pegasus/workflow.db
$ pegasus-db-admin update sqlite:///${HOME}/.pegasus/workflow.db
```

Example: connection by providing a properties file that contains the information to connect to the database. Note that a database type (MASTER, WORKFLOW, or JDBCRC) should be provided:

```
$ pegasus-db-admin update -c pegasus.properties -t MASTER
$ pegasus-db-admin update -c pegasus.properties -t JDBCRC
$ pegasus-db-admin update -c pegasus.properties -t WORKFLOW
```

Example: connection by providing the path to the submit directory containning the *braindump.txt* file, where information to connect to the database can be extracted. Note that a database type (MASTER, WORKFLOW, or JDBCRC) should also be provided:

```
$ pegasus-db-admin update -s /path/to/submitdir -t WORKFLOW
$ pegasus-db-admin update -s /path/to/submitdir -t MASTER
$ pegasus-db-admin update -s /path/to/submitdir -t JDBCRC
```

Note that, if no URL is provided, the tool will create/use a SQLite database in the user's home directory: *${HOME}/.pegasus/workflow.db*.

For complete description of pegasus-db-admin, see the man page.

Migrating From Pegasus 3.1 to Pegasus 4.X

With Pegasus 4.0 effort has been made to move the Pegasus installation to be FHS compliant, and to make workflows run better in Cloud environments and distributed grid environments. This chapter is for existing users of Pegasus who use Pegasus 3.1 to run their workflows and walks through the steps to move to using Pegasus 4.0

Move to FHS layout

Pegasus 4.0 is the first release of Pegasus which is Filesystem Hierarchy Standard (FHS) [http://www.pathname.com/fhs/] compliant. The native packages no longer installs under /opt. Instead, pegasus-* binaries are in /usr/bin/ and example workflows can be found under /usr/share/pegasus/examples/.

To find Pegasus system components, a pegasus-config tool is provided. pegasus-config supports setting up the environment for

- Python

- Perl

- Java

- Shell

For example, to find the PYTHONPATH for the DAX API, run:

```
export PYTHONPATH=`pegasus-config --python`
```

For complete description of pegasus-config, see the man page.

Stampede Schema Upgrade Tool

Starting Pegasus 4.x the monitoring and statistics database schema has changed. If you want to use the pegasus-statistics, pegasus-analyzer and pegasus-plots against a 3.x database you will need to upgrade the schema first using the schema upgrade tool /usr/share/pegasus/sql/schema_tool.py or /path/to/pegasus-4.x/share/pegasus/sql/schema_tool.py

Upgrading the schema is required for people using the MySQL database for storing their monitoring information if it was setup with 3.x monitoring tools.

If your setup uses the default SQLite database then the new databases run with Pegasus 4.x are automatically created with the correct schema. In this case you only need to upgrade the SQLite database from older runs if you wish to query them with the newer clients.

To upgrade the database

```
For SQLite Database
```

cd /to/the/workflow/directory/with/3.x.monitord.db

```
Check the db version
```

/usr/share/pegasus/sql/schema_tool.py -c connString=sqlite:////to/the/workflow/directory/with/
workflow.stampede.db
```
2012-02-29T01:29:43.330476Z INFO    netlogger.analysis.schema.schema_check.SchemaCheck.init |
2012-02-29T01:29:43.330708Z INFO
 netlogger.analysis.schema.schema_check.SchemaCheck.check_schema.start |
2012-02-29T01:29:43.348995Z INFO    netlogger.analysis.schema.schema_check.SchemaCheck.check_schema
                                    | Current version set to: 3.1.
2012-02-29T01:29:43.349133Z ERROR   netlogger.analysis.schema.schema_check.SchemaCheck.check_schema
                                    | Schema version 3.1 found - expecting 4.0 - database admin will
 need to run upgrade tool.
```

```
Convert the Database to be version 4.x compliant
```

/usr/share/pegasus/sql/schema_tool.py -u connString=sqlite:////to/the/workflow/directory/with/
workflow.stampede.db
```
2012-02-29T01:35:35.046317Z INFO    netlogger.analysis.schema.schema_check.SchemaCheck.init |
2012-02-29T01:35:35.046554Z INFO
 netlogger.analysis.schema.schema_check.SchemaCheck.check_schema.start |
2012-02-29T01:35:35.064762Z INFO    netlogger.analysis.schema.schema_check.SchemaCheck.check_schema
                                    | Current version set to: 3.1.
2012-02-29T01:35:35.064902Z ERROR   netlogger.analysis.schema.schema_check.SchemaCheck.check_schema
                                    | Schema version 3.1 found - expecting 4.0 - database admin will
 need to run upgrade tool.
2012-02-29T01:35:35.065001Z INFO    netlogger.analysis.schema.schema_check.SchemaCheck.upgrade_to_4_0
                                    | Upgrading to schema version 4.0.
```

```
Verify if the database has been converted to Version 4.x
```

/usr/share/pegasus/sql/schema_tool.py -c connString=sqlite:////to/the/workflow/directory/with/
workflow.stampede.db
```
2012-02-29T01:39:17.218902Z INFO    netlogger.analysis.schema.schema_check.SchemaCheck.init |
```

```
2012-02-29T01:39:17.219141Z INFO
 netlogger.analysis.schema.schema_check.SchemaCheck.check_schema.start |
2012-02-29T01:39:17.237492Z INFO   netlogger.analysis.schema.schema_check.SchemaCheck.check_schema |
 Current version set to: 4.0.
2012-02-29T01:39:17.237624Z INFO   netlogger.analysis.schema.schema_check.SchemaCheck.check_schema |
 Schema up to date.

For upgrading a MySQL database the steps remain the same. The only thing that changes is the
 connection String to the database
E.g.
```

/usr/share/pegasus/sql/schema_tool.py -u connString=mysql://username:password@server:port/dbname

After the database has been upgraded you can use either 3.x or 4.x clients to query the database with **pegasus-statistics**, as well as **pegasus-plots** and **pegasus-analyzer.**

Existing users running in a condor pool with a non shared filesystem setup

Existing users that are running workflows in a cloud environment with a non shared filesystem setup have to do some trickery in the site catalog to include placeholders for local/submit host paths for execution sites when using CondorIO. In Pegasus 4.0, this has been rectified.

For example, for a 3.1 user, to run on a local-condor pool without a shared filesystem and use Condor file IO for file transfers, the site entry looks something like this

```
<site  handle="local-condor" arch="x86" os="LINUX">
      <grid  type="gt2" contact="localhost/jobmanager-fork" scheduler="Fork" jobtype="auxillary"/>
      <grid  type="gt2" contact="localhost/jobmanager-condor" scheduler="unknown"
jobtype="compute"/>
      <head-fs>

        <!-- the paths for scratch filesystem are the paths on local site as we execute create dir
 job
            on local site. Improvements planned for 4.0 release.-->
          <scratch>
             <shared>
                <file-server protocol="file" url="file:///" mount-point="/submit-host/scratch"/>
                <internal-mount-point mount-point="/submit-host/scratch"/>
             </shared>
          </scratch>
          <storage>
             <shared>
                <file-server protocol="file" url="file:///" mount-point="/glusterfs/scratch"/>
                <internal-mount-point mount-point="/glusterfs/scratch"/>
             </shared>
          </storage>
      </head-fs>
      <replica-catalog  type="LRC" url="rlsn://dummyValue.url.edu" />
      <profile namespace="env" key="PEGASUS_HOME" >/cluster-software/pegasus/2.4.1</profile>
      <profile namespace="env" key="GLOBUS_LOCATION" >/cluster-software/globus/5.0.1</profile>

      <!-- profies for site to be treated as condor pool -->
      <profile namespace="pegasus" key="style" >condor</profile>
      <profile namespace="condor" key="universe" >vanilla</profile>


      <!-- to enable kickstart staging from local site-->
      <profile namespace="condor" key="transfer_executable">true</profile>


   </site>
```

With Pegasus 4.0 the site entry for a local-condor pool can be as concise as the following

```
<site  handle="condorpool" arch="x86" os="LINUX">
      <head-fs>
          <scratch />
          <storage />
      </head-fs>
      <profile namespace="pegasus" key="style" >condor</profile>
      <profile namespace="condor" key="universe" >vanilla</profile>
```

```
                </site>
```

The planner in 4.0 correctly picks up the paths from the local site entry to determine the staging location for the condor io on the submit host.

Users should read pegasus data staging configuration chapter and also look in the examples directory (share/pegasus/examples).

Migrating From Pegasus 2.X to Pegasus 3.X

With Pegasus 3.0 effort has been made to simplify configuration. This chapter is for existing users of Pegasus who use Pegasus 2.x to run their workflows and walks through the steps to move to using Pegasus 3.0

PEGASUS_HOME and Setup Scripts

Earlier versions of Pegasus required users to have the environment variable PEGASUS_HOME set and to source a setup file $PEGASUS_HOME/setup.sh | $PEGASUS_HOME/setup.csh before running Pegasus to setup CLASSPATH and other variables.

Starting with Pegasus 3.0 this is no longer required. The above paths are automaticallly determined by the Pegasus tools when they are invoked.

All the users need to do is to set the PATH variable to pick up the pegasus executables from the bin directory.

```
$ export PATH=/some/install/pegasus-3.0.0/bin:$PATH
```

Changes to Schemas and Catalog Formats

DAX Schema

Pegasus 3.0 by default now parses DAX documents conforming to the DAX Schema 3.2 available here [schemas/dax-3.2/dax-3.2.xsd] and is explained in detail in the chapter on API references.

Starting Pegasus 3.0 , DAX generation API's are provided in Java/Python and Perl for users to use in their DAX Generators. The use of API's is highly encouraged. Support for the old DAX schema's has been deprecated and will be removed in a future version.

For users, who still want to run using the old DAX formats i.e 3.0 or earlier, can for the time being set the following property in the properties and point it to dax-3.0 xsd of the installation.

```
pegasus.schema.dax  /some/install/pegasus-3.0/etc/dax-3.0.xsd
```

Site Catalog Format

Pegasus 3.0 by default now parses Site Catalog format conforming to the SC schema 3.0 (XML3) available here [schemas/dax-3.2/dax-3.2.xsd] and is explained in detail in the chapter on Catalogs.

Pegasus 3.0 comes with a pegasus-sc-converter that will convert users old site catalog (XML) to the XML3 format. Sample usage is given below.

```
$ pegasus-sc-converter -i sample.sites.xml -I XML -o sample.sites.xml3 -O XML3
```

```
2010.11.22 12:55:14.169 PST:   Written out the converted file to sample.sites.xml3
```

To use the converted site catalog, in the properties do the following

1. unset pegasus.catalog.site or set pegasus.catalog.site to XML3

2. point pegasus.catalog.site.file to the converted site catalog

Transformation Catalog Format

Pegasus 3.0 by default now parses a file based multiline textual format of a Transformation Catalog. The new Text format is explained in detail in the chapter on Catalogs.

Pegasus 3.0 comes with a pegasus-tc-converter that will convert users old transformation catalog (File) to the Text format. Sample usage is given below.

```
$ pegasus-tc-converter -i sample.tc.data -I File -o sample.tc.text -O Text
```

```
2010.11.22 12:53:16.661 PST:   Successfully converted Transformation Catalog from File to Text
2010.11.22 12:53:16.666 PST:   The output transfomation catalog is in file  /lfs1/software/install/
pegasus/pegasus-3.0.0cvs/etc/sample.tc.text
```

To use the converted transformation catalog, in the properties do the following

1. unset pegasus.catalog.transformation or set pegasus.catalog.transformation to Text

2. point pegasus.catalog.transformation.file to the converted transformation catalog

Properties and Profiles Simplification

Starting with Pegasus 3.0 all profiles can be specified in the properties file. Profiles specified in the properties file have the lowest priority. Profiles are explained in the detail in the configuration chapter. As a result of this a lot of existing Pegasus Properties were replaced by profiles. The table below lists the properties removed and the new profile based names.

Table 18.1. Property Keys removed and their Profile based replacement

Old Property Key	New Property Key
pegasus.local.env	no replacement. Specify env profiles for local site in the site catalog
pegasus.condor.release	condor.periodic_release
pegasus.condor.remove	condor.periodic_remove
pegasus.job.priority	condor.priority
pegasus.condor.output.stream	pegasus.condor.output.stream
pegasus.condor.error.stream	condor.stream_error
pegasus.dagman.retry	dagman.retry
pegasus.exitcode.impl	dagman.post
pegasus.exitcode.scope	dagman.post.scope
pegasus.exitcode.arguments	dagman.post.arguments
pegasus.exitcode.path.*	dagman.post.path.*
pegasus.dagman.maxpre	dagman.maxpre
pegasus.dagman.maxpost	dagman.maxpost
pegasus.dagman.maxidle	dagman.maxidle
pegasus.dagman.maxjobs	dagman.maxjobs
pegasus.remote.scheduler.min.maxwalltime	globus.maxwalltime
pegasus.remote.scheduler.min.maxtime	globus.maxtime
pegasus.remote.scheduler.min.maxcputime	globus.maxcputime
pegasus.remote.scheduler.queues	globus.queue

Profile Keys for Clustering

The pegasus profile keys for job clustering were **renamed**. The following table lists the old and the new names for the profile keys.

Table 18.2. Old and New Names For Job Clustering Profile Keys

Old Pegasus Profile Key	New Pegasus Profile Key

collapse	clusters.size
bundle	clusters.num

Transfers Simplification

Pegasus 3.0 has a new default transfer client pegasus-transfer that is invoked by default for first level and second level staging. The pegasus-transfer client is a python based wrapper around various transfer clients like globus-url-copy, lcg-copy, wget, cp, ln . pegasus-transfer looks at source and destination url and figures out automatically which underlying client to use. pegasus-transfer is distributed with the PEGASUS and can be found in the bin subdirectory .

Also, the Bundle Transfer refiner has been made the default for pegasus 3.0. Most of the users no longer need to set any transfer related properties. The names of the profiles keys that control the Bundle Transfers have been changed . The following table lists the old and the new names for the Pegasus Profile Keys and are explained in details in the Profiles Chapter.

Table 18.3. Old and New Names For Transfer Bundling Profile Keys

Old Pegasus Profile Key	New Pegasus Profile Keys
bundle.stagein	stagein.clusters | stagein.local.clusters | stagein.re-mote.clusters
bundle.stageout	stageout.clusters | stageout.local.clusters | stageout.re-mote.clusters

Worker Package Staging

Starting Pegasus 3.0 there is a separate boolean property **pegasus.transfer.worker.package** to enable worker package staging to the remote compute sites. Earlier it was bundled with user executables staging i.e if **pegasus.catalog.trans-formation.mapper** property was set to Staged .

Clients in bin directory

Starting with Pegasus 3.0 the pegasus clients in the bin directory have a pegasus prefix. The table below lists the old client names and new names for the clients that replaced them

Table 18.4. Old Client Names and their New Names

Old Client	New Client
rc-client	pegasus-rc-client
tc-client	pegasus-tc-client
pegasus-get-sites	pegasus-sc-client
sc-client	pegasus-sc-converter
tailstatd	pegasus-monitord
genstats and genstats-breakdown	pegasus-statistics
show-job	pegasus-plots
dirmanager	pegasus-dirmanager
exitcode	pegasus-exitcode
rank-dax	pegasus-rank-dax
transfer	pegasus-transfer

Best Practices For Developing Portable Code

This document lists out issues for the algorithm developers to keep in mind while developing the respective codes. Keeping these in mind will alleviate a lot of problems while trying to run the codes on the Grid through workflows.

Supported Platforms

Most of the hosts making a Grid run variants of Linux or in some case Solaris. The Grid middleware mostly supports UNIX and it's variants.

Running on Windows

The majority of the machines making up the various Grid sites run Linux. In fact, there is no widespread deployment of a Windows-based Grid. Currently, the server side software of Globus does not run on Windows. Only the client tools can run on Windows. The algorithm developers should not code exclusively for the Windows platforms. They must make sure that their codes run on Linux or Solaris platforms. If the code is written in a portable language like Java, then porting should not be an issue.

If for some reason the code can only be executed on windows platform, please contact the pegasus team at pegasus aT isi dot edu . In certain cases it is possible to stand up a linux headnode in front of a windows cluster running Condor as it's scheduler.

Packaging of Software

As far as possible, binary packages (preferably statically linked) of the codes should be provided. If for some reason the codes, need to be built from the source then they should have an associated makefile (for C/C++ based tools) or an ant file (for Java tools). The building process should refer to the standard libraries that are part of a normal Linux installation. If the codes require non-standard libraries, clear documentation needs to be provided, as to how to install those libraries, and make the build process refer to those libraries.

Further, installing software as root is not a possibility. Hence, all the external libraries that need to be installed can only be installed as non-root in non-standard locations.

MPI Codes

If any of the algorithm codes are MPI based, they should contact the Grid group. MPI can be run on the Grid but the codes need to be compiled against the installed MPI libraries on the various Grid sites. The pegasus group has some experience running MPI code through PBS.

Maximum Running Time of Codes

Each of the Grid sites has a policy on the maximum time for which they will allow a job to run. The algorithms catalog should have the maximum time (in minutes) that the job can run for. This information is passed to the Grid sites while submitting a job, so that Grid site does not kill a job before that published time expires. It is OK, if the job runs only a fraction of the max time.

Codes cannot specify the directory in which they should be run

Codes are installed in some standard location on the Grid Sites or staged on demand. However, they are not invoked from directories where they are installed. The codes should be able to be invoked from any directory, as long as one can access the directory where the codes are installed.

This is especially relevant, while writing scripts around the algorithm codes. At that point specifying the relative paths do not work. This is because the relative path is constructed from the directory where the script is being invoked. A suggested workaround is to pick up the base directory where the software is installed from the environment or by using the **dirname** cmd or api. The workflow system can set appropriate environment variables while launching jobs on the Grid.

No hard-coded paths

The algorithms should not hard-code any directory paths in the code. All directories paths should be picked up explicitly either from the environment (specifying environment variables) or from command line options passed to the algorithm code.

Wrapping legacy codes with a shell wrapper

When wrapping a legacy code in a script (or another program), it is necessary that the wrapper knows where the executable lives. This is accomplished using an environmental variable. Be sure to include this detail in the component description when submitting a component for use on the Grid -- include a brief descriptive name like GDA_BIN.

Propogating back the right exitcode

A job in the workflow is only released for execution if its parents have executed successfully. Hence, it is very important that the algorithm codes exit with the correct error code in case of success and failure. The algorithms should exit with a status of 0 in case of success, and a non zero status in case of error. Failure to do so will result in erroneous workflow execution where jobs might be released for execution even though their parents had exited with an error.

The algorithm codes should catch all errors and exit with a non zero exitcode. The successful execution of the algorithm code can only be determined by an exitcode of 0. The algorithm code should not rely upon something being written to the stdout to designate success for e.g. if the algorithm code writes out to the stdout SUCCESS and exits with a non zero status the job would be marked as failed.

In *nix, a quick way to see if a code is exiting with the correct code is to execute the code and then execute echo $?.

```
$ component-x input-file.lisp
... some output ...
$ echo $?
0
```

If the code is not exiting correctly, it is necessary to wrap the code in a script that tests some final condition (such as the presence or format of a result file) and uses exit to return correctly.

Static vs. Dynamically Linked Libraries

Since there is no way to know the profile of the machine that will be executing the code, it is important that dynamically linked libraries are avoided or that reliance on them is kept to a minimum. For example, a component that requires libc 2.5 may or may not run on a machine that uses libc 2.3. On *nix, you can use the **ldd** command to see what libraries a binary depends on.

If for some reason you install an algorithm specific library in a non standard location make sure to set the LD_LIBRARY_PATH for the algorithm in the transformation catalog for each site.

Temporary Files

If the algorithm codes create temporary files during execution, they should be cleared by the codes in case of errors and success terminations. The algorithm codes will run on scratch file systems that will also be used by others. The scratch directories get filled up very easily, and jobs will fail in case of directories running out of free space. The temporary files are the files that are not being tracked explicitly through the workflow generation process.

Handling of stdio

When writing a new application, it often appears feasible to use *stdin* for a single file data, and *stdout* for a single file output data. The *stderr* descriptor should be used for logging and debugging purposes only, never to put data on it. In the *nix world, this will work well, but may hiccup in the Windows world.

We are suggesting that you avoid using stdio for data files, because there is the implied expectation that stdio data gets magically handled. There is no magic! If you produce data on *stdout*, you need to declare to Pegasus that your *stdout* has your data, and what LFN Pegasus can track it by. After the application is done, the data product will be a remote file just like all other data products. If you have an input file on *stdin*, you must track it in a similar manner. If you produce logs on *stderr* that you care about, you must track it in a similar manner. Think about it this way: Whenever you are redirecting stdio in a *nix shell, you will also have to specify a file name.

Most execution environments permit to connect *stdin*, *stdout* or *stderr* to any file, and Pegasus supports this case. However, there are certain very specific corner cases where this is not possible. For this reason, we recommend that in new code, you avoid using stdio for data, and provide alternative means on the commandline, i.e. via **--input *fn*** and **--output *fn*** commandline arguments instead relying on *stdin* and *stdout*.

Configuration Files

If your code requires a configuration file to run and the configuration changes from one run to another, then this file needs to be tracked explicitly via the Pegasus WMS. The configuration file should not contain any absolute paths to any data or libraries used by the code. If any libraries, scripts etc need to be referenced they should refer to relative paths starting with a `./xyz` where `xyz` is a tracked file (defined in the workflow) or as $ENV-VAR/xyz where `$ENV-VAR` is set during execution time and evaluated by your application code internally.

Code Invocation and input data staging by Pegasus

Pegasus will create one temporary directory per workflow on each site where the workflow is planned. Pegasus will stage all the files required for the execution of the workflow in these temporary directories. This directory is shared by all the workflow components that executed on the site. You will have no control over where this directory is placed and as such you should have no expectations about where the code will be run. The directories are created per workflow and not per job/alogrithm/task. Suppose there is a component component-x that takes one argument: input-file.lisp (a file containing the data to be operated on). The staging step will bring input-file.lisp to the temporary directory. In *nix the call would look like this:

```
$ /nfs/software/component-x input-file.lisp
```

Note that Pegasus will call the component using the full path to the component. If inside your code/script you invoke some other code you cannot assume a path for this code to be relative or absolute. You have to resovle it either using a dirname $0 trick in shell assuming the child code is in the same directory as the parent or construct the path by expecting an enviornment variable to be set by the workflow system. These env variables need to be explicitly published so that they can be stored in the transformation catalog.

Now suppose that internally, component-x writes its results to /tmp/component-x-results.lisp. This is not good. Components should not expect that a /tmp directory exists or that it will have permission to write there. Instead, component-x should do one of two things: 1. write component-x-results.lisp to the directory where it is run from or 2. component-x should take a second argument output-file.lisp that specifies the name and path of where the results should be written.

Logical File naming in DAX

The logical file names used by your code can be of two types.

- Without a directory path e.g. `f.a`, `f.b` etc

- With a directory path e.g. `a/1/f.a`, `b/2/f.b`

Both types of files are supported. We will create any directory structure mentioned in your logical files on the remote execution site when we stage in data as well as when we store the output data to a permanent location. An example invocation of a code that consumes and produces files will be

```
$/bin/test --input f.a --output f.b
```

OR

```
$/bin/test --input a/1/f.a --output b/1/f.b
```

> **Note**
>
> A logical file name should never be an absolute file path, e.g. /a/1/f.a In other words, there should not be a starting slash (/) in a logical filename.

Slot Partitioning and CPU Affinity in Condor

By default, Condor will evenly divide the resources in a machine (such as RAM, swap space and disk space) among all the CPUs, and advertise each CPU as its own slot with an even share of the system resources. If you want to have your custom configuration, you can use the following setting to define the maximum number of different slot types:

```
MAX_SLOT_TYPES = 2
```

For each slot type, you can divide system resources unevenly among your CPUs. The **N** in the name of the macro listed below must be an integer from 1 to **MAX_SLOT_TYPES** (defined above).

```
SLOT_TYPE_1 = cpus=2, ram=50%, swap=1/4, disk=1/4
SLOT_TYPE_N = cpus=1, ram=20%, swap=1/4, disk=1/8
```

Slots can also be partitioned to accommodate actual needs by accepted jobs. A partitionable slot is always unclaimed and dynamically splitted when jobs are started. Slot partitioning can be enable as follows:

```
SLOT_TYPE_1_PARTITIONABLE = True
SLOT_TYPE_N_PARTITIONABLE = True
```

Condor can also bind cores to each slot through CPU affinity:

```
ENFORCE_CPU_AFFINITY = True
SLOT1_CPU_AFFINITY=0,2
SLOT2_CPU_AFFINITY=1,3
```

Note that CPU numbers may vary from machines. Thus you need to verify what is the association for your machine. One way to accomplish this is by using the **lscpu** command line tool. For instance, the output provided from this tool may look like:

```
NUMA node0 CPU(s):     0,2,4,6,8,10
NUMA node1 CPU(s):     1,3,5,7,9,11
```

The following example assumes a machine with 2 sockets and 6 cores per socket, where even cores belong to socket 1 and odd cores to socket 2:

```
NUM_SLOTS_TYPE_1 = 1
NUM_SLOTS_TYPE_2 = 1
SLOT_TYPE_1_PARTITIONABLE = True
SLOT_TYPE_2_PARTITIONABLE = True

SLOT_TYPE_1 = cpus=6
SLOT_TYPE_2 = cpus=6

ENFORCE_CPU_AFFINITY = True

SLOT1_CPU_AFFINITY=0,2,4,6,8,10
SLOT2_CPU_AFFINITY=1,3,5,7,9,11
```

Please read the section on "Configuring The Startd for SMP Machines" in the Condor Administrator's Manual for full details.

Chapter 19. Funding, citing, and anonymous usage statistics

Citing Pegasus in Academic Works

The preferred generic way to cite Pegasus is:

Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems, Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, Daniel S. Katz. Scientific Programming Journal, Vol 13(3), 2005, Pages 219-237.

Usage Statistics Collection

Purpose

Pegasus WMS is primarily a NSF funded project as part of the NSF SI2 [http://www.nsf.gov/funding/pgm_summ.jsp?pims_id=504817] track. The SI2 program focuses on robust, reliable, usable and sustainable software infrastructure that is critical to the CIF21 vision. As part of the requirements of being funded under this program, Pegasus WMS is required to gather usage statistics of Pegasus WMS and report it back to NSF in annual reports. The metrics will also enable us to improve our software as they will include errors encountered during the use of our software.

Overview

We plan to instrument and augment the following clients in our distribution to report the metrics.

- pegasus-plan

- pegasus-transfer

- pegasus-monitord

For the Pegasus WMS 4.2 release, only the pegasus-plan client has been instrumented to send metrics.

All the metrics are sent in JSON format to a server at USC/ISI over HTTP. The data reported is as generic as possible and is listed in detail in the section titled "Metrics Collected".

Configuration

By default, the clients will report usage metrics to a server at ISI. However, users have an option to configure the report by setting the following environment variables

- PEGASUS_METRICS

  A boolean value (true | false) indicating whether metrics reporting is turned ON/OFF

- PEGASUS_USER_METRICS_SERVER

  A comma separated list of URLs of the servers to which to report the metrics in addition to the default server.

Metrics Collected

All metrics are sent in JSON format and the metrics sent by the various clients include the following data

Table 19.1. Common Data Sent By Pegasus WMS Clients

JSON KEY	DESCRIPTION
client	the name of the client (e.g "pegasus-plan")
version	the version of the client
type	type of data - "metrics" | "error"
start_time	start time of the client (in epoch seconds with millisecond precision)
end_time	end time of the client (in epoch seconds with millisecond precision)
duration	the duration of the client
exitcode	the exitcode with which the client exited
wf_uuid	the uuid of the executable workflow. It is generated by pegasus-plan at planning time.

Pegasus Planner Metrics

The metrics messages sent by the planner in addition include the following data

Table 19.2. Metrics Data Sent by pegasus-plan

JSON KEY	DESCRIPTION
root_wf_uuid	the root workflow uuid. For non hierarchal workflows the root workflow uuid is the same as the workflow uuid.
data_config	the data configuration mode of pegasus
compute_tasks	the number of compute tasks in the workflow
dax_tasks	the number of dax tasks in the abstract workflow (DAX)
dag_tasks	the number of dag tasks in the abstract workflow (DAX)
total_tasks	the number of the total tasks in the abstract workflow (DAX)
dax_input_files	the number of input files in the abstract workflow (DAX)
dax_inter_files	the number of intermediate files in the abstract workflow (DAX)
dax_output_files	the number of output files in the abstract workflow (DAX)
dax_total_files	the number of total files in the abstract workflow (DAX)
compute_jobs	the number of compute jobs in the executable workflow
clustered_jobs	the number of clustered jobs in the executable workflow.
si_tx_jobs	the number of data stage-in jobs in the executable workflow.
so_tx_jobs	the number of data stage-out jobs in the executable workflow.
inter_tx_jobs	the number of inter site data transfer jobs in the executable workflow.
reg_job	the number of registration jobs in the executable workflow.
cleanup_jobs	the number of cleanup jobs in the executable workflow.
create_dir_jobs	the number of create directory jobs in the executable workflow.

JSON KEY	DESCRIPTION
dax_jobs	the number of sub workflows corresponding to dax tasks in the executable workflow.
dag_jobs	the number of sub workflows corresponding to dag tasks in the executable workflow.
chmod_jobs	the number of jobs that set the xbit of the staged executables
total_jobs	the total number of jobs in the workflow

In addition if pegasus-plan encounters an error during the planning process the metrics message has an additional field in addition to the fields listed above.

Table 19.3. Error Message sent by pegasus-plan

JSON KEY	DESCRIPTION
error	the error payload is the stack trace of errors caught during planning

Note

pegasus-plan leaves a copy of the metrics sent in the workflow submit directory in the file ending with ".metrics" extension. As a user you will always have access to the metrics sent.

Chapter 20. Glossary

Glossary

A

Abstract Workflow	See DAX

C

Concrete Workflow	See Executable Workflow
Condor-G	A task broker that manages jobs to run at various distributed sites, using Globus GRAM to launch jobs on the remote sites.http://cs.wisc.edu/condor
Clustering	The process of clustering short running jobs together into a larger job. This is done to minimize the scheduling overhead for the jobs. The scheduling overhead is only incurred for the clustered job. For example if scheduling overhead is x seconds and 10 jobs are clustered into a larger job, the scheduling overhead for 10 jobs will be x instead of 10x.

D

DAGMan	The workflow execution engine used by Pegasus.
Directed Acyclic Graph (DAG)	A graph in which all the arcs (connections) are unidirectional, and which has no loops (cycles).
DAX	The workflow input in XML format given to Pegasus in which transformations and files are represented as logical names. It is an execution-independent specification of computations
Deferred Planning	Planning mode to set up Pegasus. In this mode, instead of mapping the job at submit time, the decision of mapping a job to a site is deferred till a later point, when the job is about to be run or near to run.

E

Executable Workflow	A workflow automatically genetared by Pegasus in which files are represented by physical filenames, and in which sites or hosts have been selected for running each task.

F

Full Ahead Planning	Planning mode to set up Pegasus. In this mode, all the jobs are mapped before submitting the workflow for execution to the grid.

G

Globus	The Globus Alliance is a community of organizations and individuals developing fundamental technologies behind the "Grid," which lets people share computing power, databases, instruments, and other on-line tools securely

across corporate, institutional, and geographic boundaries without sacrificing local autonomy.

See Globus Toolkit

Globus Toolkit   Globus Toolkit is an open source software toolkit used for building Grid systems and applications.

GRAM   A Globus service that enable users to locate, submit, monitor and cancel remote jobs on Grid-based compute resources. It provides a single protocol for communicating with different batch/cluster job schedulers.

Grid   A collection of many compute resources , each under different administrative domains connected via a network (usually the Internet).

GridFTP   A high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks. It is based upon the Internet FTP protocol, and uses basic Grid security on both control (command) and data channels.

Grid Service   A service which uses standardized web service mechanisms to model and access stateful resources, perform lifecycle management and query resource state. The Globus Toolkit includes core grid services for execution management, data management and information management.

L

Logical File Name   The unique logical identifier for a data file. Each LFN is associated with a set of PFN's that are the physical instantiations of the file.

M

Metadata   Any attributes of a dataset that are explicitly represented in the workflow system. These may include provenance information (e.g., which component was used to generate the dataset), execution information (e.g., time of creation of the dataset), and properties of the dataset (e.g., density of a node type).

Monitoring and Discovery Service   A Globus service that implements a site catalog.

P

Physical File Name   The physical file name of the LFN.

Partitioner   A tool in Pegasus that slices up the DAX into smaller DAX's for deferred planning.

Pegasus   A system that maps a workflow instance into an executable workflow to run on the grid.

R

Replica Catalog   A catalog that maps logical file names on to physical file names.

Replica Location Service   A Globus service that implements a replica catalog

S

Site   A set of compute resources under a single administrative domain.

Site Catalog A catalog indexed by logical site identifiers that maintains information about the various grid sites. The site catalog can be populated from a static database or maybe populated dynamically by monitoring tools.

T

Transformation Any executable or code that is run as a task in the workflow.

Transformation Catalog A catalog that maps transformation names onto the physical pathnames of the transformation at a given grid site or local test machine.

W

Workflow Instance A workflow created in Wings and given to Pegasus in which workflow components and files are represented as logical names. It is an execution-independent specification of computations

Appendix A. Tutorial VM

Introduction

This appendix provides information on how to launch the Pegasus Tutorial VM. The VM is a quick way to get started using Pegasus. It comes pre-configured with Pegasus, DAGMan and Condor so that you can begin running workflows immediately.

In the following sections we will cover how to start, log into, and stop the tutorial VM locally, using the VirtualBox virtualization software, and remotely on Amazon EC2.

VirtualBox

VirtualBox is a free desktop virtual machine manager. You can use it to run the Pegasus Tutorial VM on your desktop or laptop.

Install VirtualBox

First, download and install the VirtualBox platform package from the VirtualBox website: https://www.virtualbox.org

Download VM Image

Next, download the Pegasus Tutorial VM from the Pegasus download page: http://pegasus.isi.edu/downloads

Move the downloaded file somewhere that you can find later.

Create Virtual Machine

Start VirtualBox. You should get a screen that looks like this:

Figure A.1. VirtualBox Welcome Screen
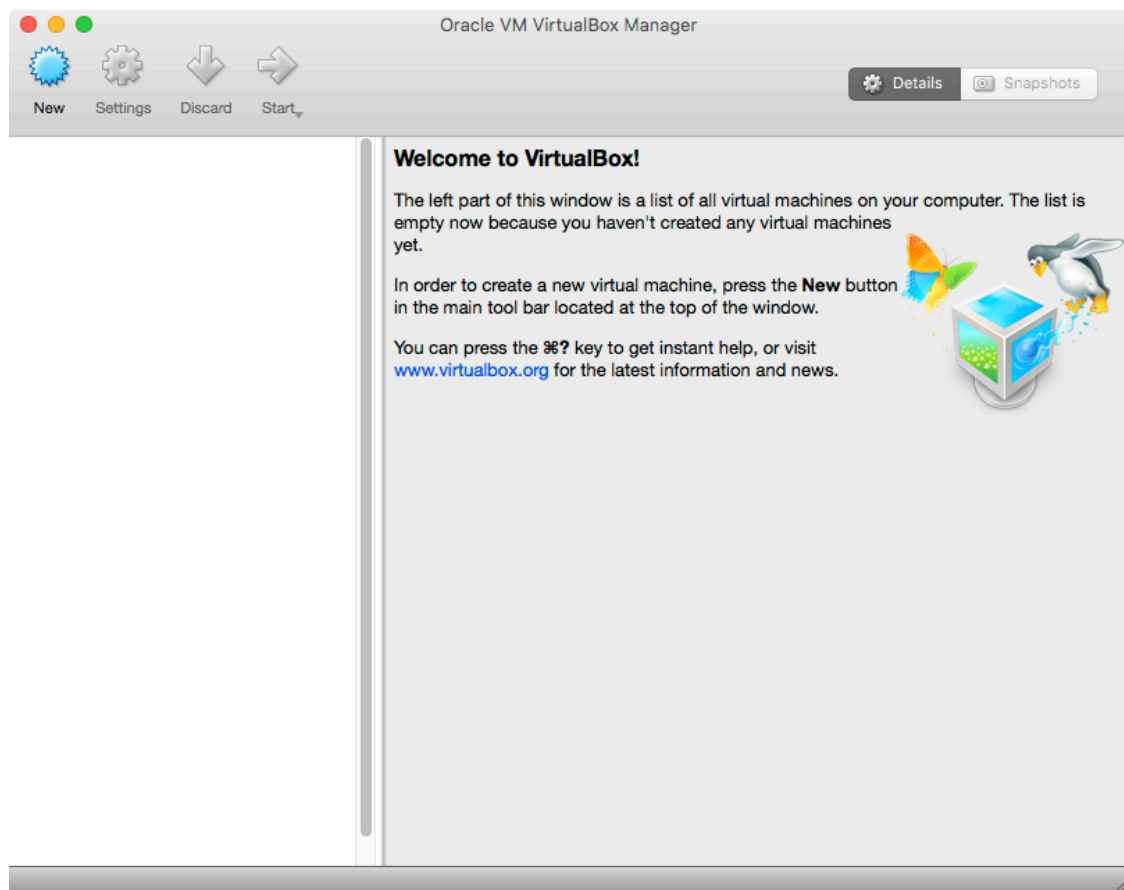
Click on File > Import Appliance, and Appliance Import Wizard will appear:

Figure A.2. Create New Virtual Machine Wizard
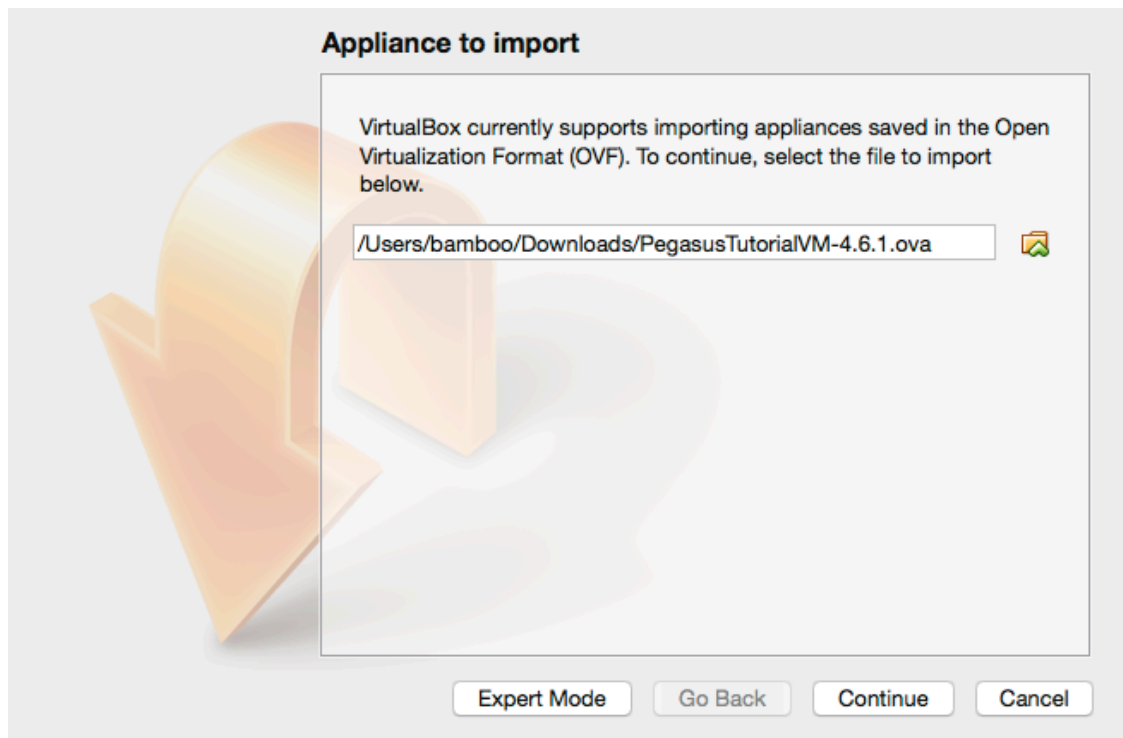
Click the folder icon and locate the .ova file that you downloaded earlier.

Click "Continue" to get to the "Appliance Settings" Page:

Figure A.3. VM Name and OS Type
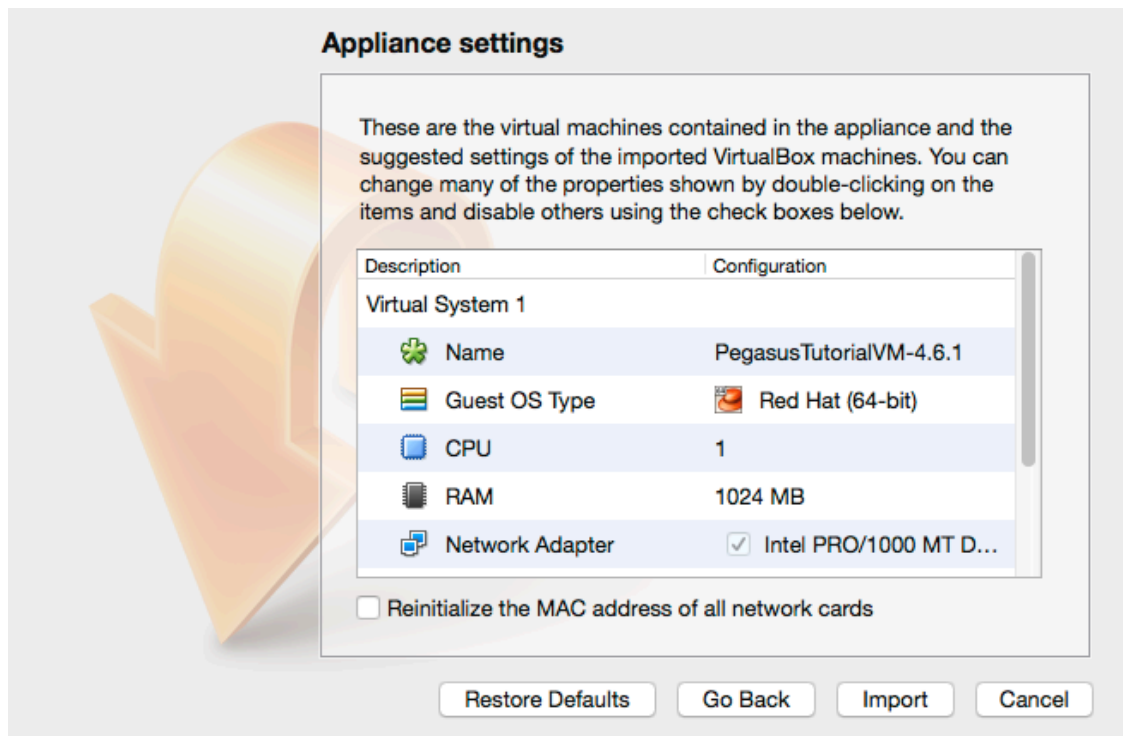
Click "Import". You will get back to the welcome screen showing the new virtual machine:

Figure A.4. Memory
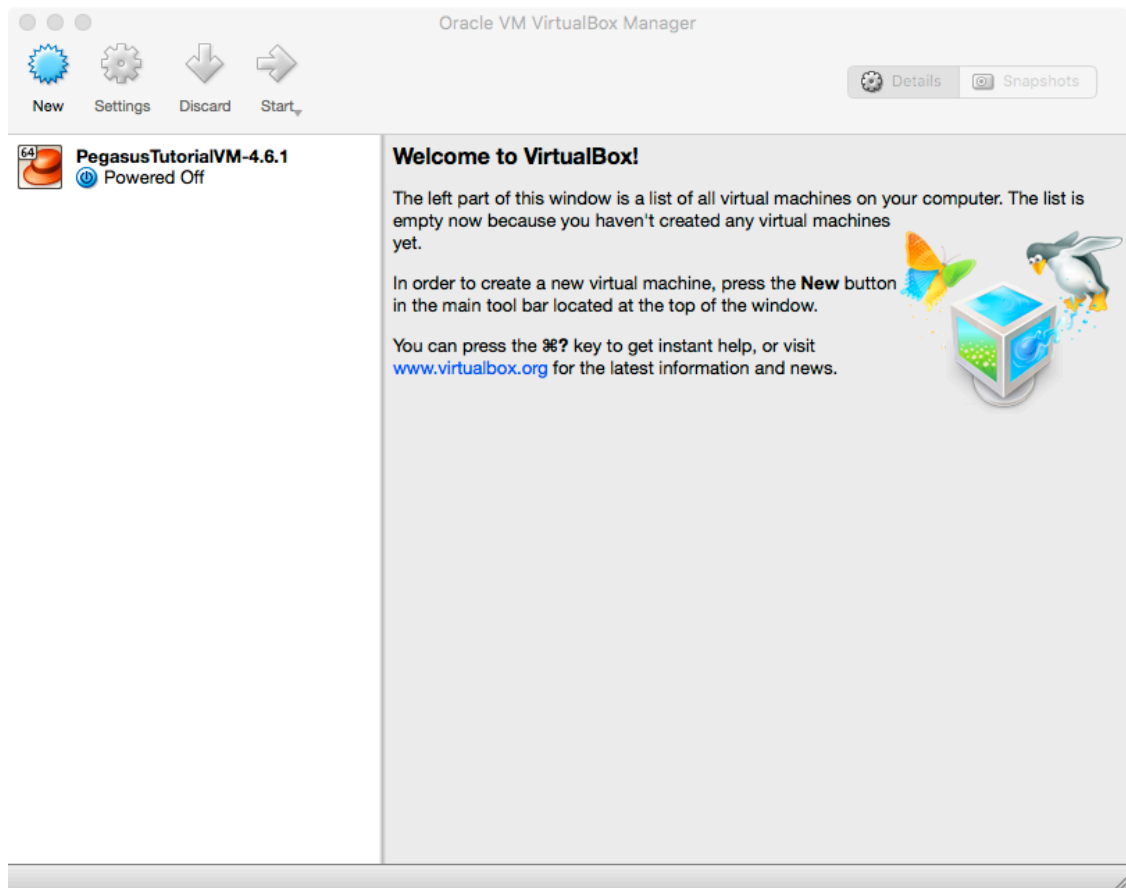
Click on the name of the virual machine and then click "Start". After a few seconds you should get to the login screen:

Figure A.5. Login Screen
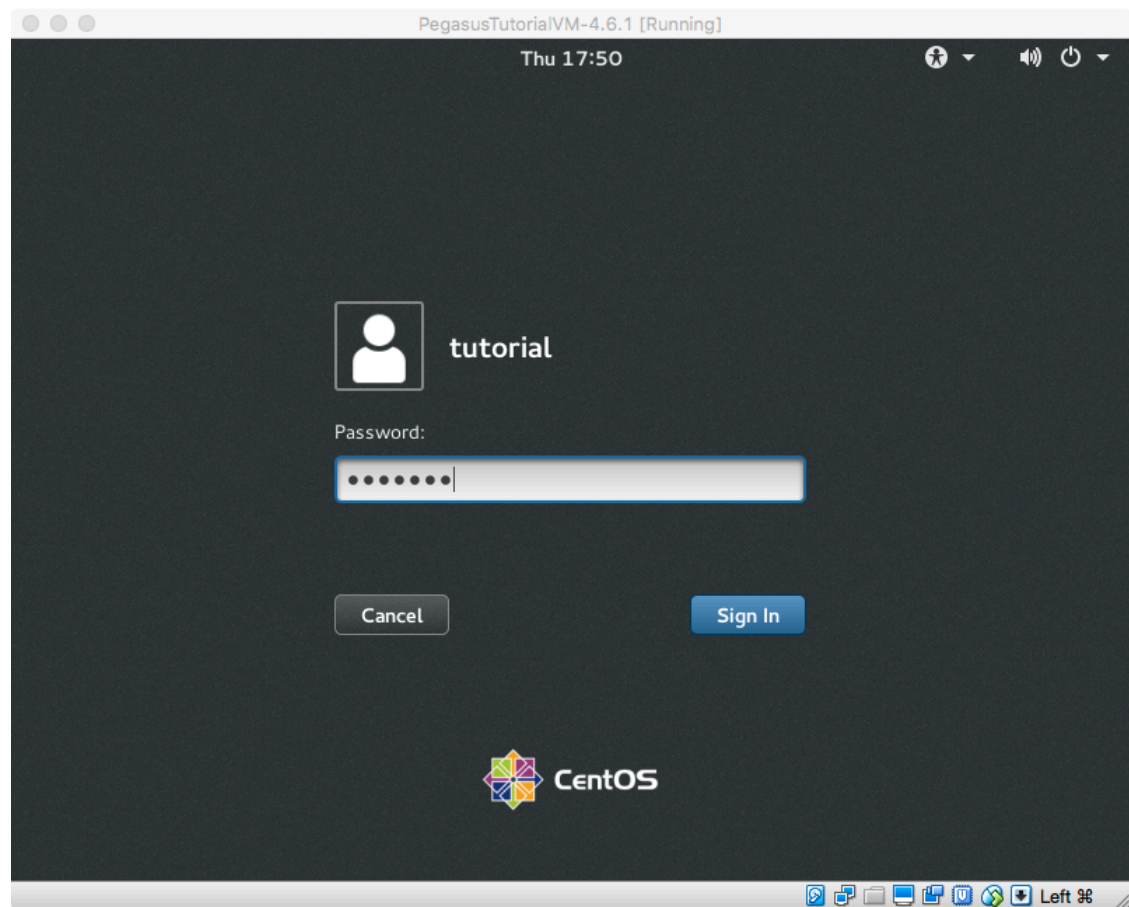
Log in as user "**tutorial**" with password "**pegasus**".

After you log in, Click the Terminal Icon, to open a Terminal. You can return to the tutorial chapter to complete the tutorial.

Terminating the VM

When you are done with the tutorial you can shut down the VM by typing:

```
$ sudo /sbin/poweroff
```

at the prompt and then enter the tutorial user's password.

Alternatively, you can just close the window and choose "Power off the machine".

Amazon EC2

In order to launch the tutorial VM you need to sign up for an Amazon Web Services account here: http://aws.amazon.com

Launching the VM

Once you have an account, sign into the AWS Management Console at this URL: http://console.aws.amazon.com. You will get a page that looks like this:

Figure A.6. AWS Management Console
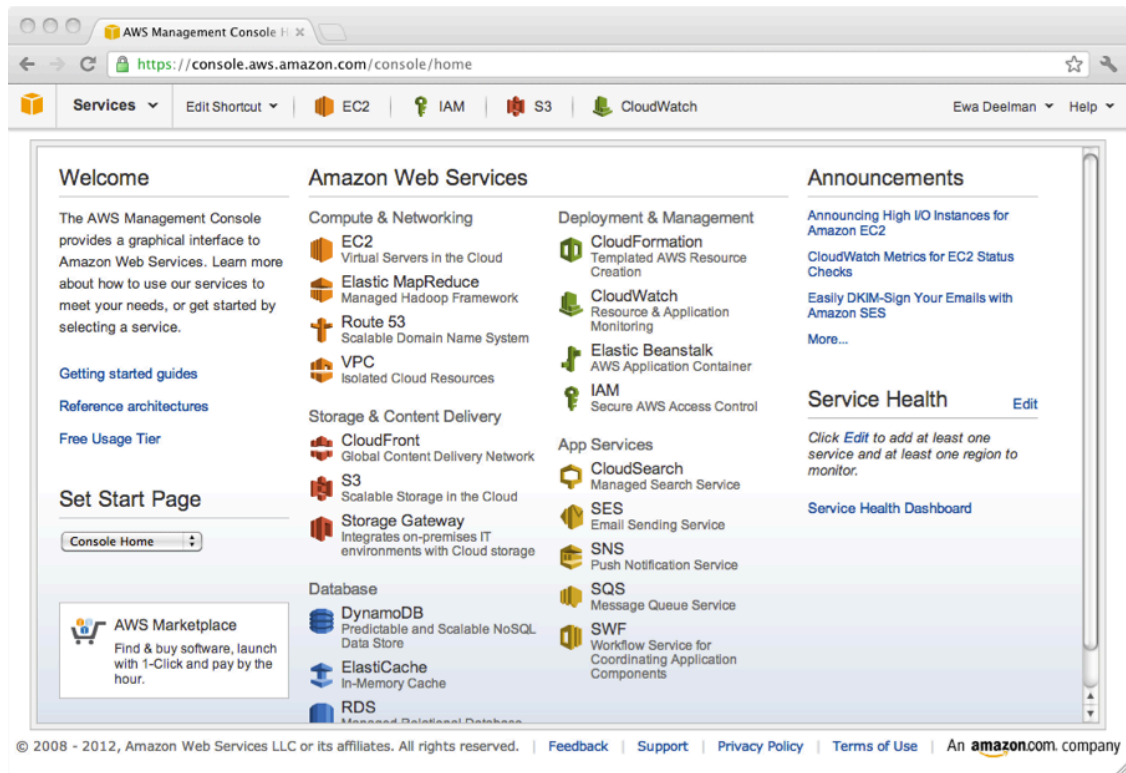
Choose the "EC2" icon under "Amazon Web Services". You will get this page:

Figure A.7. EC2 Management Console
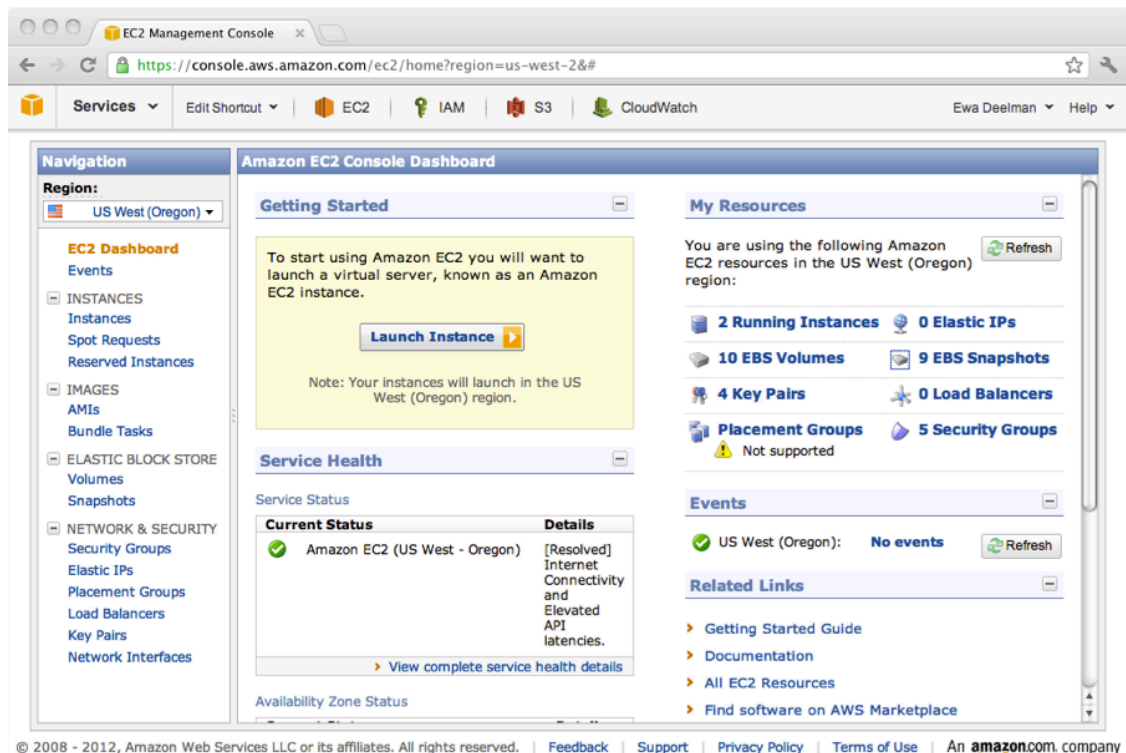
First, make sure the "Region:" drop-down in the upper left-hand corner is set to "US West (Oregon)".

Click on the "AMIs" link on the left side and set "Viewing:" to "All Images", "All Platforms", and type "Pegasus Tutorial VM" in the search box:

Figure A.8. Locating the Tutorial VM
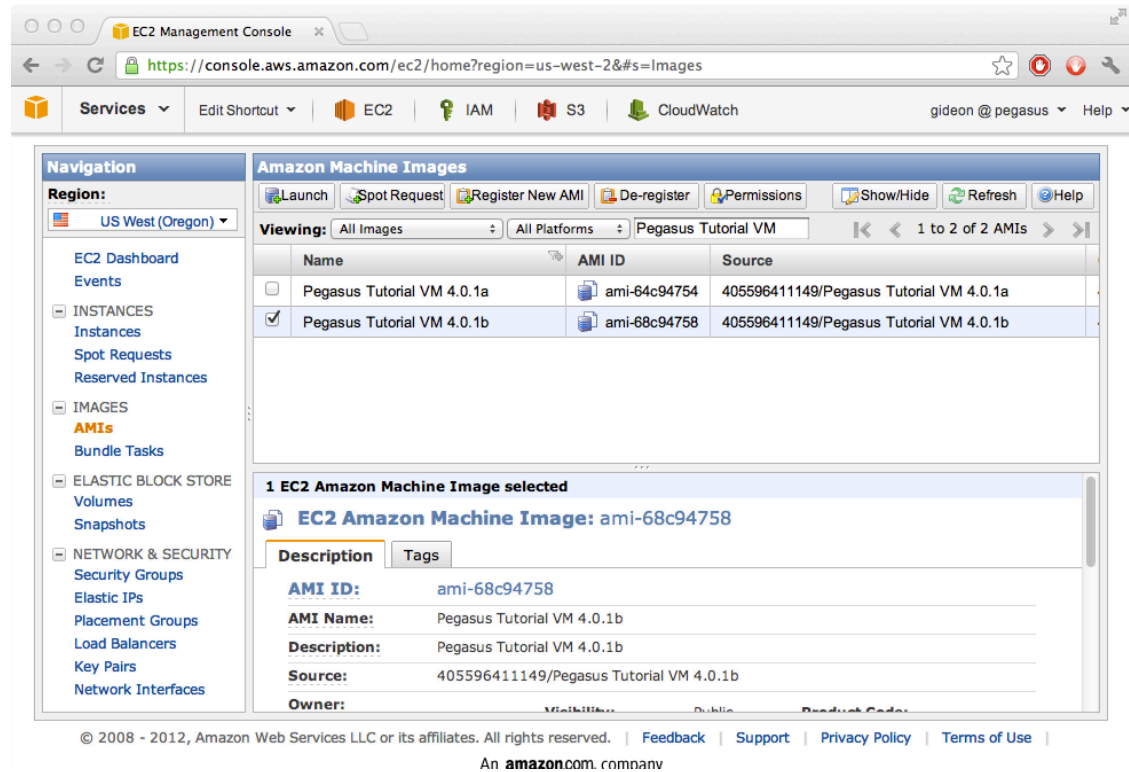
You will see several versions of the VM. If you don't see any AMIs named "Pegasus Tutorial VM" you may need to click the Refresh button. We update the VM regularly, so your search results will not match the picture above.

Check the check box next to the latest Pegasus Tutorial VM and click the "Launch" button. The "Request Instances Wizard" will pop up:

Figure A.9. Request Instances Wizard: Step 1

In the first step of the Request Instances Wizard choose the "Large" instance type and click "Continue":

Figure A.10. Request Instances Wizard: Step 2

Don't change anything on the "Advanced Instance Options" step and click "Continue":

Figure A.11. Request Instances Wizard: Step 3

On the "Storage Device Configuration" step make sure "Delete on Termination" is set to "true", then click "Continue":

Figure A.12. Request Instances Wizard: Step 4

On the next step type "Pegasus Tutorial" into the "Value" field and click "Continue":

Figure A.13. Request Instances Wizard: Step 5

On the next page choose one of your existing key pairs and click "Continue". If you don't have an existing key pair you can also choose "Proceed without a Key Pair" (you will log in with a username/password).

Figure A.14. Request Instances Wizard: Step 6

On the next page choose "Create a new Security Group". Name the security group "Pegasus Tutorial" and give it a description. Create an inbound TCP rule to allow connections on port 22 (SSH) from source 0.0.0.0/0 and click "Add Rule". This rule allows you to SSH into your EC2 instance. Create another TCP rule to allow connections on port 5000 from source 0.0.0.0/0 and click "Add Rule" again. This rule is for the Pegasus Dashboard web interface. Then click "Continue".

Note that you will only need to create this security group once. If you launch the Pegasus Tutorial VM again the security group should appear in the list of existing security groups.

Figure A.15. Request Instances Wizard: Step 7

On the last step of the wizard validate your selections and click "Launch".

Figure A.16. Running Instances
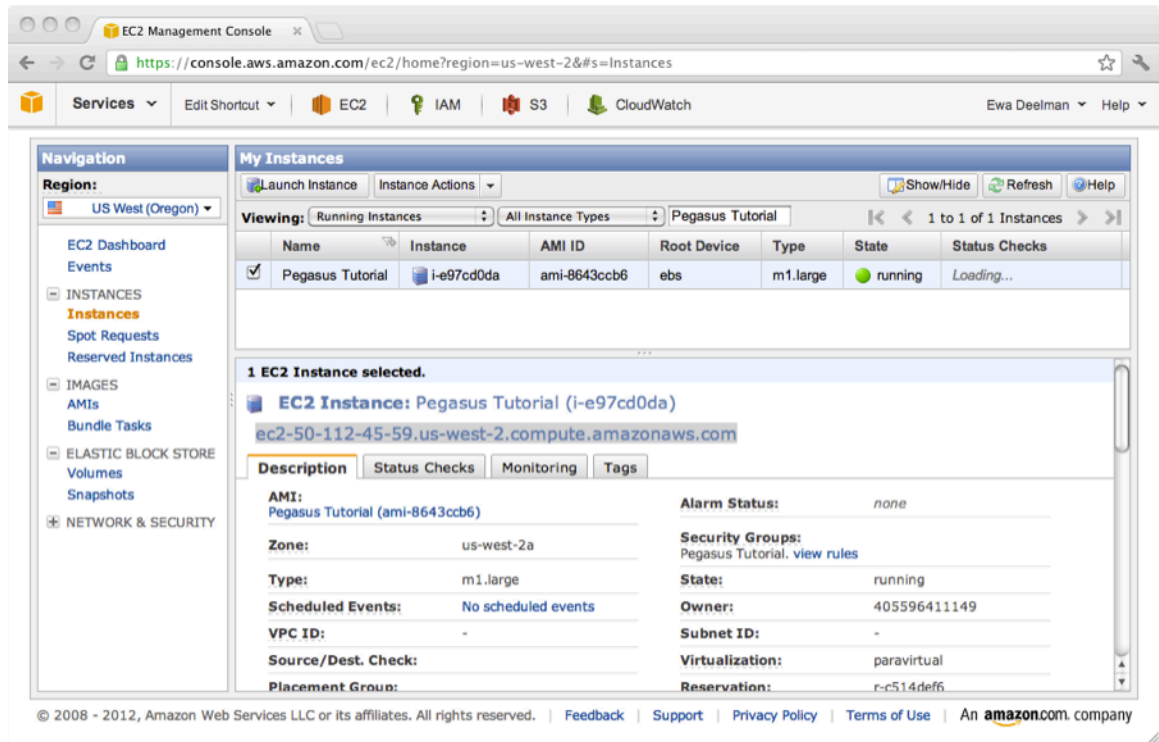
Finally, navigate to the "Instances" section and check the checkbox next to the "Pegasus Tutorial" instance. Copy the DNS name to the clipboard. In this example the name is: **ec2-50-112-45-59.us-west-2.compute.amazonaws.com**. Yours will almost surely be different.

At this point your VM will take a few minutes to boot. Wait until the "Status Checks" column reads: "2/2 checks passed" before continuing. You may need to click the Refresh button.

Logging into the VM

Log into the VM using SSH. The username is '**tutorial**' and the password is '**pegasus**'.

On UNIX machines such as Linux or Mac OS X you can log in via SSH by opening a terminal and typing:

```
$ ssh tutorial@ec2-50-112-45-59.us-west-2.compute.amazonaws.com
The authenticity of host 'ec2-50-112-45-59.us-west-2.compute.amazonaws.com (50.112.45.59)' can't be
 established.
RSA key fingerprint is 56:b0:11:ba:8f:98:ba:dd:75:f6:3c:09:ef:b9:2a:ac.
Are you sure you want to continue connecting (yes/no)? yes
[tutorial@localhost ~]$
```

where "ec2-50-112-45-59.us-west-2.compute.amazonaws.com" is the DNS name of your VM that you copied from the AWS Management Console.

If you are on Windows you will need to install an SSH client. You can download the PuTTY SSH client and find documentation for how to configure it here: http://www.chiark.greenend.org.uk/~sgtatham/putty

Shutting down the VM

When you are finished with the tutorial, make sure you terminate the VM. If you forget to do this you will be charged for all of the hours that the VM runs.

To terminate the VM click on "Instances" link on the left side of the AWS Management Console, check the box next to the "Pegasus Tutorial" VM, and click "Instance Actions"-->"Terminate":

Figure A.17. Terminate Instance
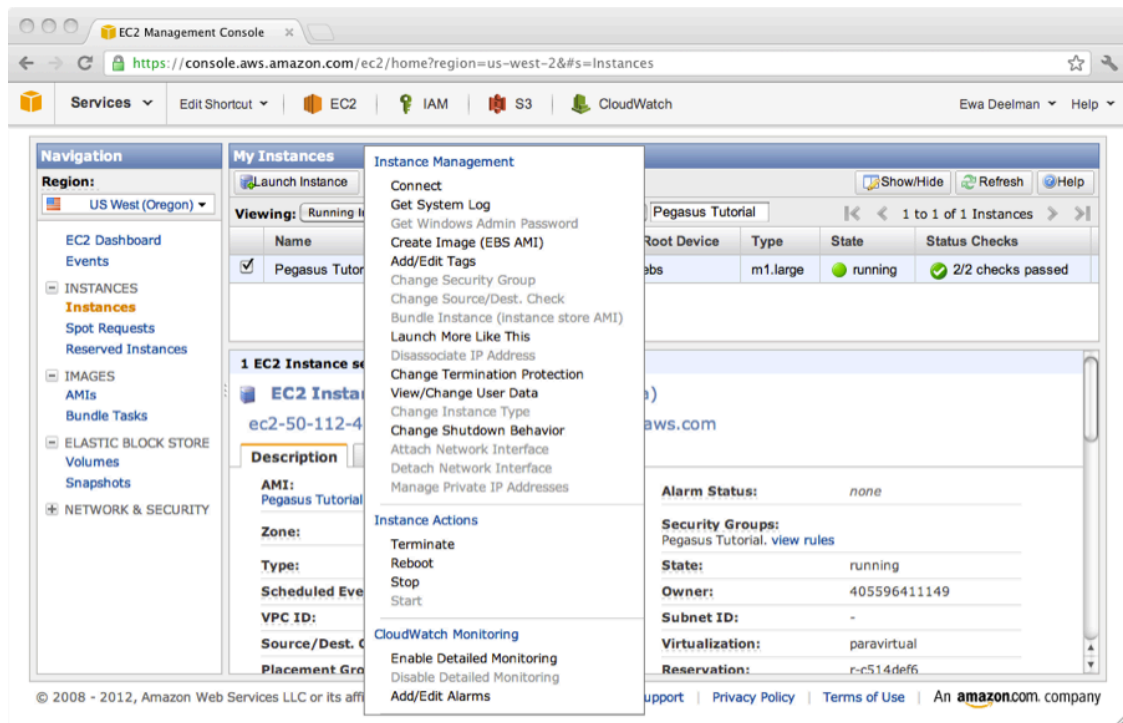
Then click "Yes, terminate":

Figure A.18. Yes, Terminate Instance