Pegasus 4.3 User Guide

Pegasus 4.3 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 environment 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 summaries 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.

Chapter 2. Tutorial

Introduction

This tutorial will take you through the steps of creating and running a simple workflow using Pegasus. This tutorial is intended for new users who want to get a quick overview of Pegasus concepts and usage. The tutorial covers the creating, planning, submitting, monitoring, debugging, and generating statistics for a simple diamond-shaped workflow. 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.

Getting Started

In order to reduce the amount of work required to get started we have provided several virtual machines that contain all of the software required for this tutorial. Virtual machine images are provided for VirtualBox, Amazon EC2 and FutureGrid. Information about deploying the tutorial VM on these platforms is in the appendix. 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.

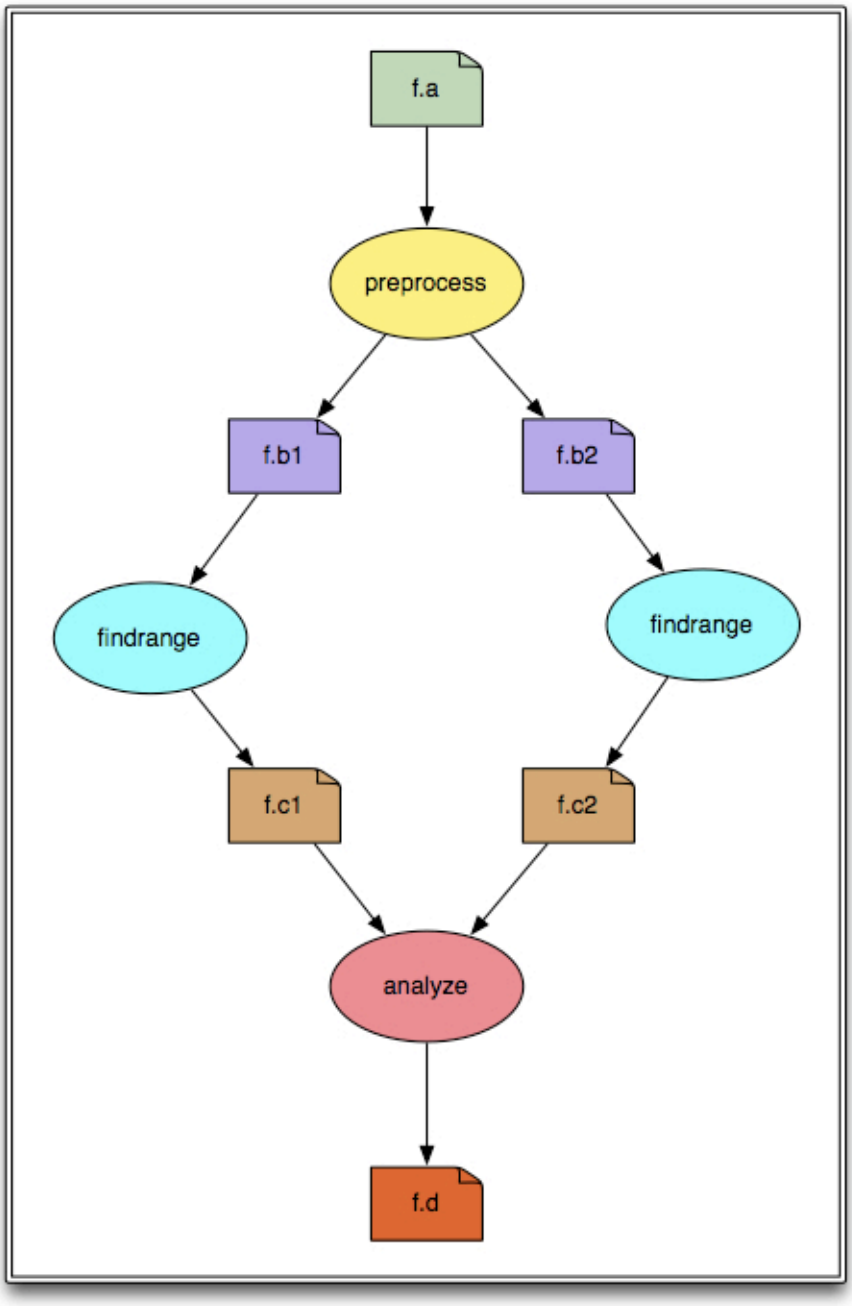
Advanced Users: In the case that you want to install Pegasus and Condor and go through the tutorial on your own machine instead of using one of the virtual machines, the tutorial files are available in the `doc/tutorial` directory of the Pegasus source distribution. These files will need to be modified in several places to fix the paths to the users home directory (which is assumed to be `/home/tutorial`). It is assumed that Pegasus was installed from the RPM, so the path to the Pegasus install is assumed to be `/usr`. Condor should be installed in the "Personal Condor" configuration. You will also need a passwordless ssh key to enable SCP file transfers to/from localhost. Getting everything set up correctly can be tricky, so we recommend getting started with one of the VMs if you are not familiar with Condor and UNIX.

The remainder of this tutorial will assume that you have a terminal open to the directory where the tutorial files are installed. If you are using one of the tutorial VMs these files are located in the tutorial user's home directory `/home/tutorial`.

Generating the Workflow

We will be creating and running a simple diamond-shaped workflow that looks like this:

Figure 2.1. Diamond Workflow



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

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.

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 diamond workflow is in the file `generate_dax.py`. Look at the file by typing:

```
$ more generate_dax.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 5 sections:

1. A few system libraries and the Pegasus.DAX3 library are imported. The search path is modified to include the directory with the Pegasus Python library.

2. The name for the DAX output file is retrieved from the arguments.

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

4. Jobs and files are added. The 4 jobs in the diagram above are added and the 6 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”.

5. 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.

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

```
$ ./generate_dax.py diamond.dax
Creating ADAG...
Adding preprocess job...
Adding left Findrange job...
Adding right Findrange job...
Adding Analyze job...
Adding control flow dependencies...
Writing diamond.dax
```

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

```
$ more diamond.dax
...
```

Information Catalogs

There are three information catalogs that Pegasus uses when planning the workflow. These are the Site Catalog, Transformation Catalog, and Replica Catalog.

The Site Catalog

The site catalog describes the sites where the workflow jobs are to be executed. Typically the sites in the site catalog describe remote clusters, such as PBS clusters or Condor pools. 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 is in `sites.xml`:

```
$ more sites.xml
...
```

There are two sites defined in the site catalog: “local” and “PegasusVM”. The “local” site is used by Pegasus to learn about the submit host where the workflow management system runs. The “PegasusVM” site is the personal Condor pool running on your (virtual) machine. In this case, the local site and the PegasusVM site refer to the same machine, but they are logically separate as far as Pegasus is concerned.

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”.

The PegasusVM site is configured with a “scratch” file system accessible via SCP (indicated by the scp:// URL). 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 “PegasusVM”.

The local site also has an environment variable called SSH_PRIVATE_KEY that tells Pegasus where to find the private key to use for SCP transfers. If you are running this tutorial on your own machine you will need to set up a passwordless ssh key and add it to authorized_keys. If you are using the tutorial VM this has already been set up for you.

Pegasus supports many different file transfer protocols. In this case the site catalog is set up so that input and output files are transferred to/from the PegasusVM site using SCP. Since both the local site and the PegasusVM site are actually the same machine, this configuration will just SCP files to/from localhost, which is just a complicated way to copy the files.

Finally, the PegasusVM 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.

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.dat`:

```
$ more tc.dat
...
```

The `tc.dat` file contains information about three transformations: preprocess, findrange, and analyze. These three transformations are referenced in the diamond DAX. The transformation catalog indicates that all three transformations are installed on the PegasusVM site, and are compiled for x86_64 Linux.

The actual executable files are located in the `bin` directory. All three executables are actually symlinked to the same Python script. This script is just an example transformation that sleeps for 30 seconds, and then writes its own name and the contents of all its input files to all of its output files.

The Replica Catalog

The final catalog is 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.dat` file:

```
$ more rc.dat
# This is the replica catalog. It lists information about each of the
# input files used by the workflow.

# The format is:
# LFN     PFN    pool="SITE"

f.a    file:///home/tutorial/input/f.a    pool="local"
```

This replica catalog contains only one entry for the diamond workflow’s only input file. This entry has an LFN of “f.a” with a PFN of “file:///home/tutorial/input/f.a” and the file is stored on the local site, which implies that it will need to be transferred to the PegasusVM site when the workflow runs. The Replica Catalog uses the keyword "pool" to refer to the site. Don't be confused by this: the value of the pool variable should be the name of the site where the file is located from the Site Catalog.

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.conf` file:

```
$ more pegasus.conf
# This tells Pegasus where to find the Site Catalog
pegasus.catalog.site=XML3
pegasus.catalog.site.file=sites.xml

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

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

Planning the Workflow

The planning stage is where Pegasus maps the abstract DAX to one or more execution sites. The planning step includes:

1. Adding a job to create the remote working directory
2. Adding stage-in jobs to transfer input data to the remote working directory
3. Adding cleanup jobs to remove data from the remote working directory when it is no longer needed
4. Adding stage-out jobs to transfer data to the final output location as it is generated
5. Adding registration jobs to register the data in a replica catalog
6. Task clustering to combine several short-running jobs into a single, longer-running job. This is done to make short-running jobs more efficient.
7. Adding wrappers to the jobs to collect provenance information so that statistics and plots can be created when the workflow is finished

The `pegasus-plan` command is used to plan a workflow. This command takes quite a few arguments, so we created a `plan_dax.sh` wrapper script that has all of the arguments required for the diamond workflow:

```
$ more plan_dax.sh
...
```

The script invokes the `pegasus-plan` command with arguments for the configuration file (`--conf`), the DAX file (`-d`), the submit directory (`--dir`), the execution site (`--sites`), the output site (`-o`) and two extra arguments that prevent Pegasus from removing any jobs from the workflow (`--force`) and that prevent Pegasus from adding cleanup jobs to the workflow (`--nocleanup`).

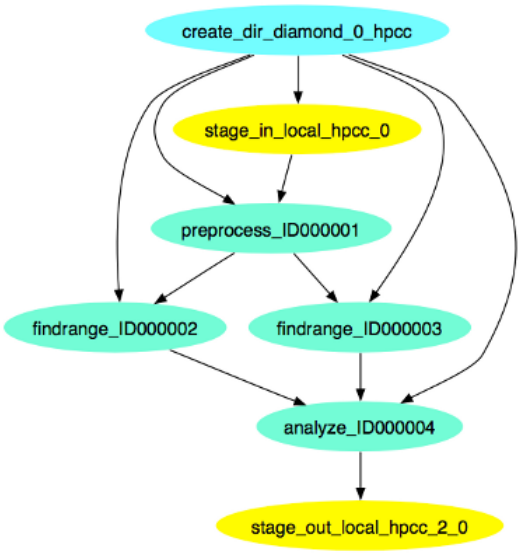
Top plan the diamond workflow invoke the `plan_dax.sh` script with the path to the DAX file:

```
$ ./plan_dax.sh diamond.dax
2012.07.24 21:11:03.256 EDT:

I have concretized your abstract workflow. The workflow has been entered
into the workflow database with a state of "planned". The next step is to
start or execute your workflow. The invocation required is:

pegasus-run  /home/tutorial/submit/tutorial/pegasus/diamond/run0001


2012.07.24 21:11:03.257 EDT:   Time taken to execute is 1.103 seconds
```

Note the line in the output that starts with `pegasus-run`. That is the command that we will use to submit 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 diamond workflow looks like after Pegasus has finished planning the DAX:

Figure 2.2. Diamond DAG



For this workflow the only jobs Pegasus needs to add are a directory creation job, a stage-in job (for f.a), and a stage-out job (for f.d). No registration jobs are added because all the files in the DAX are marked register="false", and no cleanup jobs are added because we passed the `--nocleanup` argument to `pegasus-plan`.

Submitting the Workflow

Once the workflow has been planned, the next step is to submit it to DAGMan/Condor for execution. This is done using the `pegasus-run` command. This command takes the path to the submit directory as an argument. Run the command that was printed by the `plan_dax.sh` script:

```
$ pegasus-run submit/tutorial/pegasus/diamond/run0001
------------------------------------------------------------------------
File for submitting this DAG to Condor       : diamond-0.dag.condor.sub
Log of DAGMan debugging messages             : diamond-0.dag.dagman.out
Log of Condor library output                 : diamond-0.dag.lib.out
Log of Condor library error messages         : diamond-0.dag.lib.err
Log of the life of condor_dagman itself      : diamond-0.dag.dagman.log

Submitting job(s).
1 job(s) submitted to cluster 19.
------------------------------------------------------------------------

Your Workflow has been started and runs in base directory given below

cd submit/tutorial/pegasus/diamond/run0001

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

pegasus-status -l submit/tutorial/pegasus/diamond/run0001

*** To remove your workflow run ***
pegasus-remove submit/tutorial/pegasus/diamond/run0001
```

Monitoring the Workflow

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

```
$ pegasus-status submit/tutorial/pegasus/diamond/run0001
STAT  IN_STATE  JOB
Run      01:48  diamond-0
Run      00:05   |-findrange_ID0000002
Run      00:05   \_findrange_ID0000003
Summary: 3 Condor jobs total (R:3)

UNREADY   READY     PRE  QUEUED    POST SUCCESS FAILURE %DONE
      2       0       0       3       0       3       0  37.5
Summary: 1 DAG total (Running:1)
```

This command shows the workflow (diamond-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` command to continuously monitor the workflow:

```
$ watch pegasus-status submit/tutorial/pegasus/diamond/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)
UNREADY   READY     PRE  QUEUED    POST SUCCESS FAILURE %DONE
      0       0       0       0       0       8       0 100.0
Summary: 1 DAG total (Success:1)
```

That means the workflow is finished successfully. You can type `ctrl-c` to terminate the `watch` command.

If the workflow finished successfully you should see the output file `f.d` in the `output` directory. This file was created by the various transformations in the workflow and shows all of the executables that were invoked by the workflow:

```
$ more output/f.d
/home/tutorial/bin/analyze:
/home/tutorial/bin/findrange:
/home/tutorial/bin/preprocess:
This is the input file of the diamond workflow
/home/tutorial/bin/findrange:
/home/tutorial/bin/preprocess:
This is the input file of the diamond workflow
```

Remember that the example transformations in this workflow just print their name to all of their output files and then copy all of their input files to their output files.

Debugging the 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/diamond/run0001
pegasus-analyzer: initializing...

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

 Total jobs         :      7 (100.00%)
 # jobs succeeded   :      7 (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/diamond/run0002
pegasus-analyzer: initializing...

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

 Total jobs         :      7 (100.00%)
 # jobs succeeded   :      2 (28.57%)
```

```
 # jobs failed      :      1 (14.29%)
 # jobs unsubmitted :      4 (57.14%)

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

====================preprocess_ID0000001==============================

 last state: POST_SCRIPT_FAILED
       site: PegasusVM
submit file: preprocess_ID0000001.sub
output file: preprocess_ID0000001.out.003
 error file: preprocess_ID0000001.err.003

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

site        : PegasusVM
hostname    : ip-10-252-31-58.us-west-2.compute.internal
executable  : /home/tutorial/bin/preprocess
arguments   : -i f.a -o f.b1 -o f.b2
exitcode    : -128
working dir : -

-------------Task #1 - preprocess - ID0000001 - stderr---------------

FATAL: The main job specification is invalid or missing.
```

In this example I removed the `bin/preprocess` executable and re-planned/re-submitted the workflow (that is why the command has run0002). The output of `pegasus-analyzer` indicates that the preprocess task failed with an error message that indicates that the executable could not be found.

Collecting Statistics

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/diamond/run0001

************************SUMMARY*****************************
# legends
# 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 Run - 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 walltime 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 walltime 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 walltime value includes jobs from the sub workflows
#       as well.
# Cumulative job walltime as seen from submit side:
#       The sum of the walltime of all jobs as reported by DAGMan.
#       This is similar to the regular cumulative job walltime, 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
```

```
#       walltime value includes jobs from the sub workflows as well.

------------------------------------------------------------------------------
Type            Succeeded  Failed  Incomplete  Total     Retries  Total Run
Tasks           4          0       0           4     ||  0        4
Jobs            7          0       0           7     ||  0        7
Sub Workflows   0          0       0           0     ||  0        0
------------------------------------------------------------------------------

Workflow wall time:                               3 mins, 25 secs, (205 s)
Workflow cumulative job wall time:                2 mins, 0 secs, (120 s)
Cumulative job walltime as seen from submit side: 2 mins, 0 secs, (120 s)

Summary: submit/tutorial/pegasus/diamond/run0001/statistics/summary.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/diamond/run0001/statistics/
breakdown.csv  jobs.txt          summary.txt         time.txt
breakdown.txt  summary-time.csv  time-per-host.csv   workflow.csv
jobs.csv       summary.csv       time.csv            workflow.txt
```

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

```
$ more submit/tutorial/pegasus/diamond/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.

# a1f5ba03-a827-4d0a-8d59-9941cbfbd83d (diamond)
Transformation   Count  Succeeded  Failed  Min     Max     Mean     Total
analyze          1      1          0       30.008  30.008  30.008   30.008
dagman::post     7      7          0       5.0     6.0     5.143    36.0
findrange        2      2          0       30.009  30.014  30.011   60.023
pegasus::dirmanager 1   1          0       0.194   0.194   0.194    0.194
pegasus::transfer 2     2          0       0.248   0.411   0.33     0.659
preprocess       1      1          0       30.025  30.025  30.025   30.025

# All
Transformation   Count  Succeeded  Failed  Min     Max     Mean     Total
analyze          1      1          0       30.008  30.008  30.008   30.008
dagman::post     7      7          0       5.0     6.0     5.143    36.0
findrange        2      2          0       30.009  30.014  30.011   60.023
pegasus::dirmanager 1   1          0       0.194   0.194   0.194    0.194
pegasus::transfer 2     2          0       0.248   0.411   0.33     0.659
preprocess       1      1          0       30.025  30.025  30.025   30.025
```

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.

Conclusion

Congratulations! You have completed the tutorial.

If you used Amazon EC2 or FutureGrid 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

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 2.4 or higher**. Check with:

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

- **HTCondor (formerly Condor) 7.6 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 4.0 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. Check Globus Installation

```
# echo $GLOBUS_LOCATION
/path/to/globus/install
```

  Make sure you source the Globus environment

```
# GLOBUS_LOCATION/etc/globus-user-env.sh
```

  Check the setup by running:#

```
# globus-version
5.2.0
```

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-4.3.0/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 environtment 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`
```

Native Packages (RPM/DEB)

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

RHEL / CentOS / Scientific Linux

Add the Pegasus repository to yum downloading the Pegasus repos file and adding it to **`/etc/yum.repos.d/`**. For RHEL 5 based systemes:

```
# wget -O /etc/yum.repos.d/pegasus.repo http://download.pegasus.isi.edu/wms/download/rhel/5/
pegasus.repo
```

For RHEL 6 based systems:

```
# wget -O /etc/yum.repos.d/pegasus.repo http://download.pegasus.isi.edu/wms/download/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   :4.3.0-1


Complete!
```

Debian

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:

```
# gpg --keyserver pgp.mit.edu --recv-keys 81C2A4AC
# gpg -a --export 81C2A4AC | apt-key add -
```

Add the Pegasus apt repository to your `/etc/apt/sources.list` file:

```
deb http://download.pegasus.isi.edu/wms/download/debian squeeze main
```

Install Pegasus with **apt-get :**

```
# apt-get update
...
# apt-get install pegasus
```

Pegasus from Tarballs

The Pegasus prebuild tarballs can be downloaded from the *Pegasus Download Page* [http://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-4.3.0:$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 [http://pegasus.isi.edu/wms/docs/schemas/dax-3.4/dax-3.4.xsd] The documentation of the schema and its elements can be found in dax-3.4.html [http://pegasus.isi.edu/wms/docs/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.

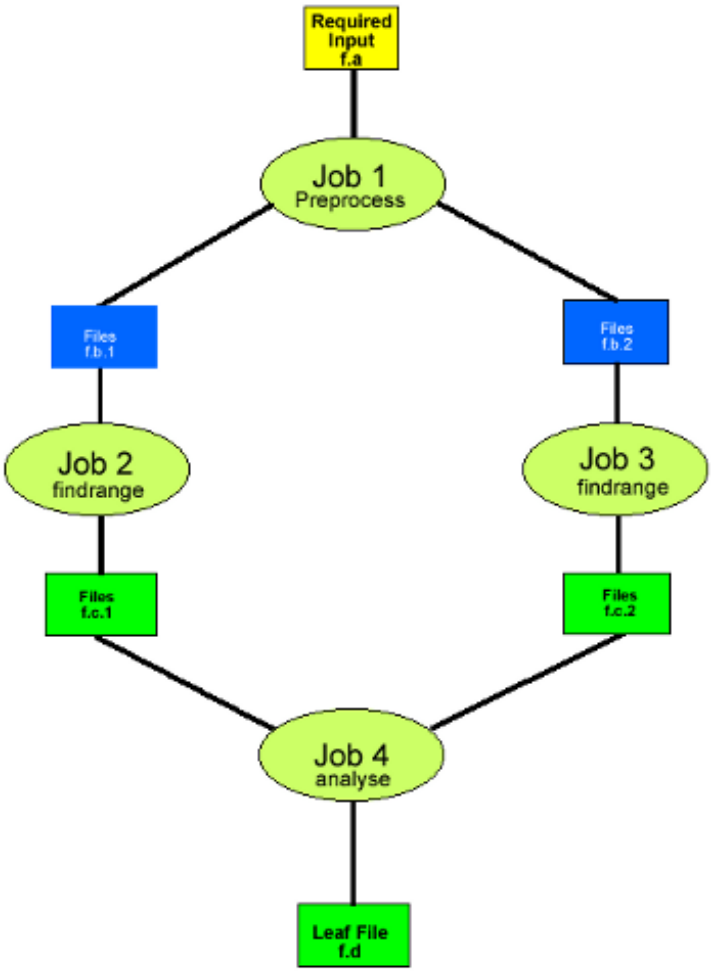
- **fan-in**, **gather**, **join**, and **converge** describe how multiple siblings are merged into fewer dependent child nodes.

  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: 2010-11-22T22:55:08Z -->
<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-3.2.xsd"
      version="3.2" name="diamond" index="0" count="1">
  <!-- 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="false" />
    <uses name="f.b1" link="output" register="false" transfer="false" />
    <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="false" />
    <uses name="f.c1" link="output" register="false" transfer="false" />
  </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.c2" link="output" register="false" transfer="false" />
    <uses name="f.b2" link="input" register="false" transfer="false" />
  </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="false" />
    <uses name="f.d" link="output" register="false" transfer="true" />
    <uses name="f.c1" link="input" register="false" transfer="false" />
  </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 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. **Replica Location Service**
   - **RLS**

- **LRC**

6. **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 **"pool".**

```
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 **"pool".**

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.

```
#1
f.a file:///Volumes/data/input/f.a pool="local"
#2
f.a file:///Volumes/data/input/f.a pool="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] pool="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://$input/f.1  pool="local"
f.2 file://$input/f.2  pool="local"
D1/f.3 file://$input/D1/f.3 pool="local"
```

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

1. **pegasus.catalog.replica.directory.site** to specify a site attribute other than local to associate with the mappings.
2. **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.
3. **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. The sql schema’s for this catalog can be found at **$PEGASUS_HOME/sql** directory. You will have to install the schema into either PostgreSQL or MySQL by running the appropriate commands to load the two scheams **create-XX-init.sql** and **create-XX-rc.sql** where **XX** is either **my** (for MySQL) or **pg** (for PostgreSQL)

To use JDBCRC, the user additionally needs to set the following properties

1. **pegasus.catalog.replica.db.url=*<jdbc url to the databse>***
2. **pegasus.catalog.replica.db.user=*<database user>***
3. **pegasus.catalog.replica.db.password=*<database password>***

Replica Location Service

Replica Location Service (RLS) is a distributed replica catalog, that ships with Globus. There is an index service called Replica Location Index (RLI) to which 1 or more Local Replica Catalog (LRC) report. Each LRC can contain all or a subset of mappings.

Details about RLS can be found at http://www.globus.org/toolkit/data/rls/

RLS

In this mode, Pegasus queries the central RLI to discover in which LRC’s the mappings for a LFN reside. It then queries the individual LRC’s for the PFN’s. To use this mode the following properties need to be set:

1. **pegasus.catalog.replica=RLS**

2. **pegasus.catalog.replica.url=*<url to the globus LRC>***

LRC

This mode is availabe If the user does not want to query the RLI (Replica Location Index), but instead wishes to directly query a single Local Replica Catalog. To use the LRC mode the follow properties need to be set

1. **pegasus.catalog.replica=*LRC***

2. **pegasus.catalog.replica.url=*<url to the globus LRC>***

Details about Globus Replica Catalog and LRC can be found at http://www.globus.org/toolkit/data/rls/

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 two lrcs at the same time specify the following:

- **pegasus.catalog.replica.mrc.lrc1=LRC**
- **pegasus.catalog.replica.mrc.lrc1.url=*<url to the 1st globus LRC>***
- **pegasus.catalog.replica.mrc.lrc2=LRC**
- **pegasus.catalog.replica.mrc.lrc2.url=<url to the 2nd globus LRC>**

In the above example,**lrc1** and **lrc2** are any valid identifier names and **url** is the property key that needed to be specified.

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.

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

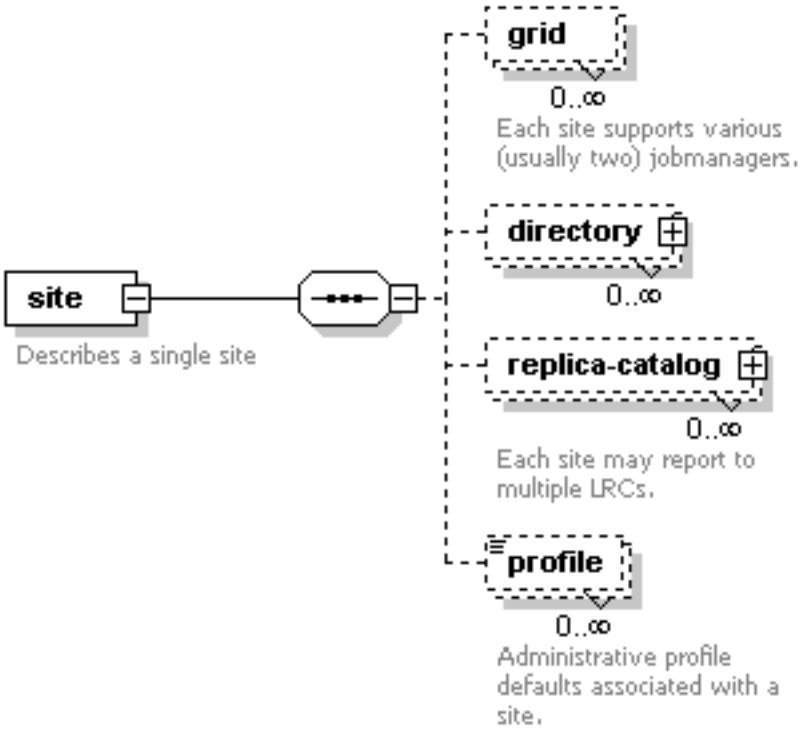
1. **XML4**(Default) Corresponds to the schema described here [http://pegasus.isi.edu/wms/docs/schemas/sc-4.0/sc-4.0.html].

2. **XML3**(Deprecated) Corresponds to the schema described here [http://pegasus.isi.edu/wms/docs/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 4.2. 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

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>***

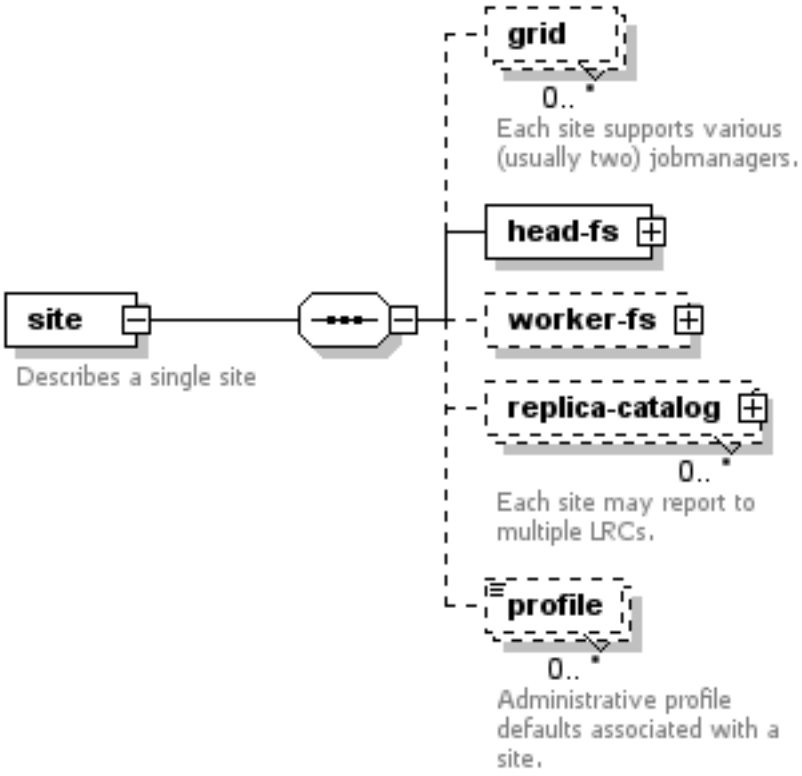
XML3

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 4.3. 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

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 Client pegasus-sc-client

The pegasus-sc-client can be used to generate a site catalog for Open Science Grid (OSG) by querying their Monitoring Interface likes VORS or OSGMM. See pegasus-sc-client --help for more details

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 [http://pegasus.isi.edu/wms/docs/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 two implementations of the Transformation Catalog

1. **Text:** A multiline text based Transformation Catalog (DEFAULT)
2. **File:** A simple multi column text based Transformation Catalog
3. **Database:** A database backend (MySQL or PostgreSQL) via JDB

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 Iinstalled ("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>***

Singleline Text based TC (File)

Warning

This format is now deprecated in favor of the multiline TC. If you are still using the single line TC you should convert it to multiline using the tc-converter client.

The format of the this TC is as follows.

```
#site  logicaltr   physicaltr   type  system  profiles(NS::KEY="VALUE")

site1 sys::date:1.0 /usr/bin/date  INSTALLED INTEL32::LINUX:FC4.2:3.6 ENV::PATH="/usr/
bin";PEGASUS_HOME="/usr/local/pegasus"
```

The system and profile entries are optional and will use default values if not specified. The entries in the file format have the following meaning:

1. **site** - A site identifier.
2. **logicaltr** - The logical transformation name. The format is NAMESPACE::NAME:VERSION where NAMESPACE and NAME are optional.
3. **physicaltr** - The physical transformation path or URL.

   If the transformation type is INSTALLED then it needs to be an absolute path to the executable. If the type is STAGEABLE then the path needs to be a HTTP, FTP or gsiftp URL

4. **type -** The type of transformation. Can have on of two values
   - **INSTALLED**: This means that the transformation is installed on the remote site
   - **STAGEABLE**: This means that the transformation is available as a static binary and can be staged to a remote site.
5. **system -** The system for which the transformation is compiled.

   The formation of the sytem is ARCH::OS:OSVERSION:GLIBC where the GLIBC and OS VERSION are optional. ARCH can have one of the following values INTEL32, INTEL64, SPARCV7, SPARCV9, AIX, AMD64. OS can have one of the following values LINUX,SUNOS. The default value for system if none specified is INTEL32::LINUX
6. **Profiles -** The profiles associated with the transformation. For indepth information about profiles and their priorities read the Profile Guide.

   The format for profiles is NS::KEY="VALUE" where NS is the namespace of the profile e.g. Pegasus,condor,DAGMan,env,globus. The key and value can be any strings. Remember to quote the value with double quotes. If you need to specify several profiles you can do it in several ways
   - NS1::KEY1="VALUE1",KEY2="VALUE2";NS2::KEY3="VALUE3",KEY4="VALUE4"

     This is the most optimized form. Multiple key values for the same namespace are separated by a comma "," and different namespaces are separated by a semicolon ";"
   - NS1::KEY1="VALUE1";NS1::KEY2="VALUE2";NS2::KEY3="VALUE3";NS2::KEY4="VALUE4"

     You can also just repeat the triple of NS::KEY="VALUE" separated by semicolons for a simple format;

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

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

Database TC (Database)

The database TC alows you to use a relational database. To use the database TC you need to have installed a MySQL or PostgreSQL server. The schema for the database is available in $PEGASUS_HOME/sql directory. You will have to install the schema into either PostgreSQL or MySQL by running the appropriate commands to load the two scheams **create-XX-init.sql** and **create-XX-tc.sql** where XX is either **my** (for MySQL) or **pg** (for PostgreSQL)

To use the Database TC you need to set the following properties

1. **pegasus.catalog.transformation.db.driver=MySQL | Postgres**
2. **pegasus.catalog.transformation.db.url=*<jdbc url to the databse>***
3. **pegasus.catalog.transformation.db.user=*<database user>***
4. **pegasus.catalog.transformation.db.password=*<database password>***

TC Client pegasus-tc-client

We need to map our declared transformations (preprocess, findranage, 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
```

TC Converter Client pegasus-tc-converter

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

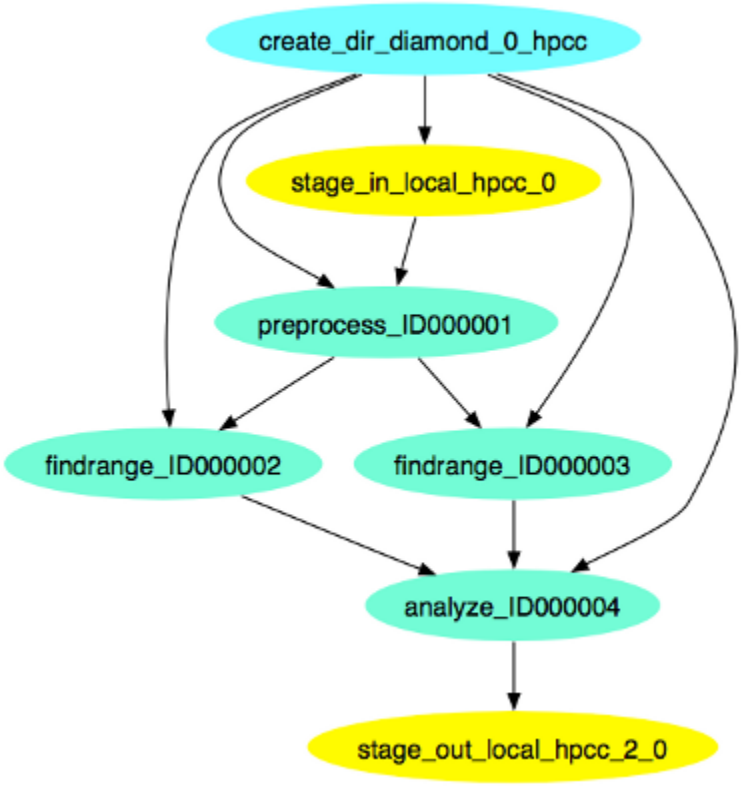
Chapter 5. Running Workflows

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 concatination 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 " Reference Manual" 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.

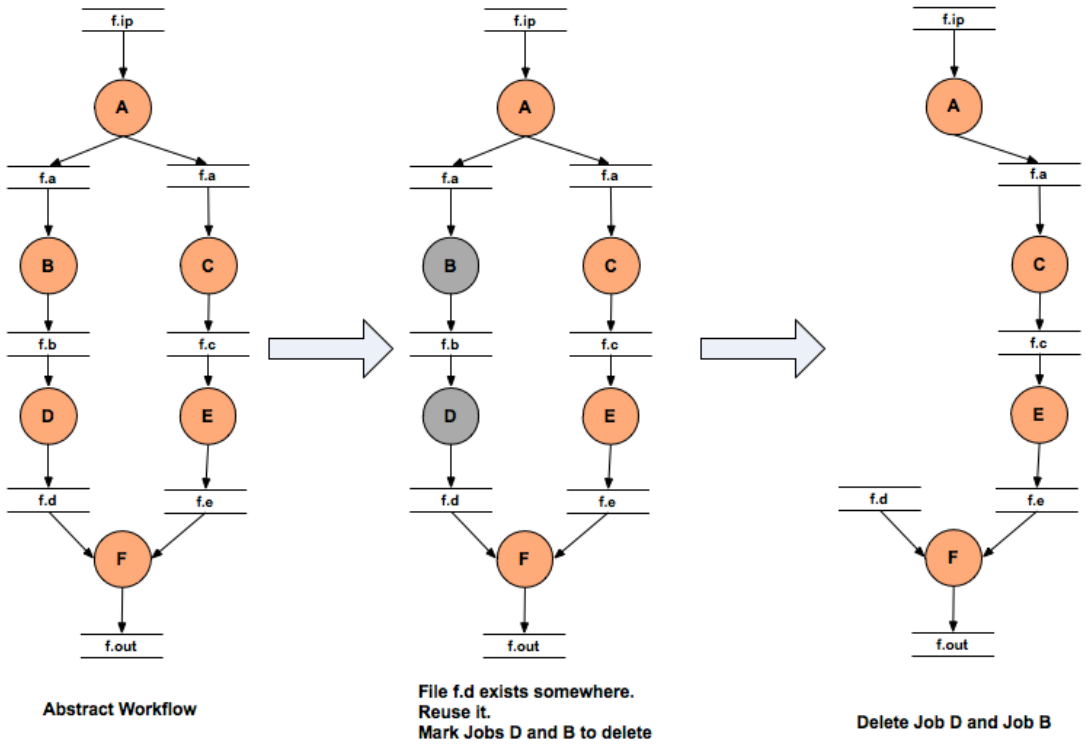
Second Pass - The algorithm removes the job whose output files exist in the Replica Catalog and tries to cascade the deletion upwards to the parent jobs. We start the breadth first traversal of the workflow bottom up.

```
( 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
       )
 )
```

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 NonJavaCallout.

Table 5.1. Table 1: Key Value Pairs that are currently generated for the site selector temporary file that is generated in the NonJavaCallout.

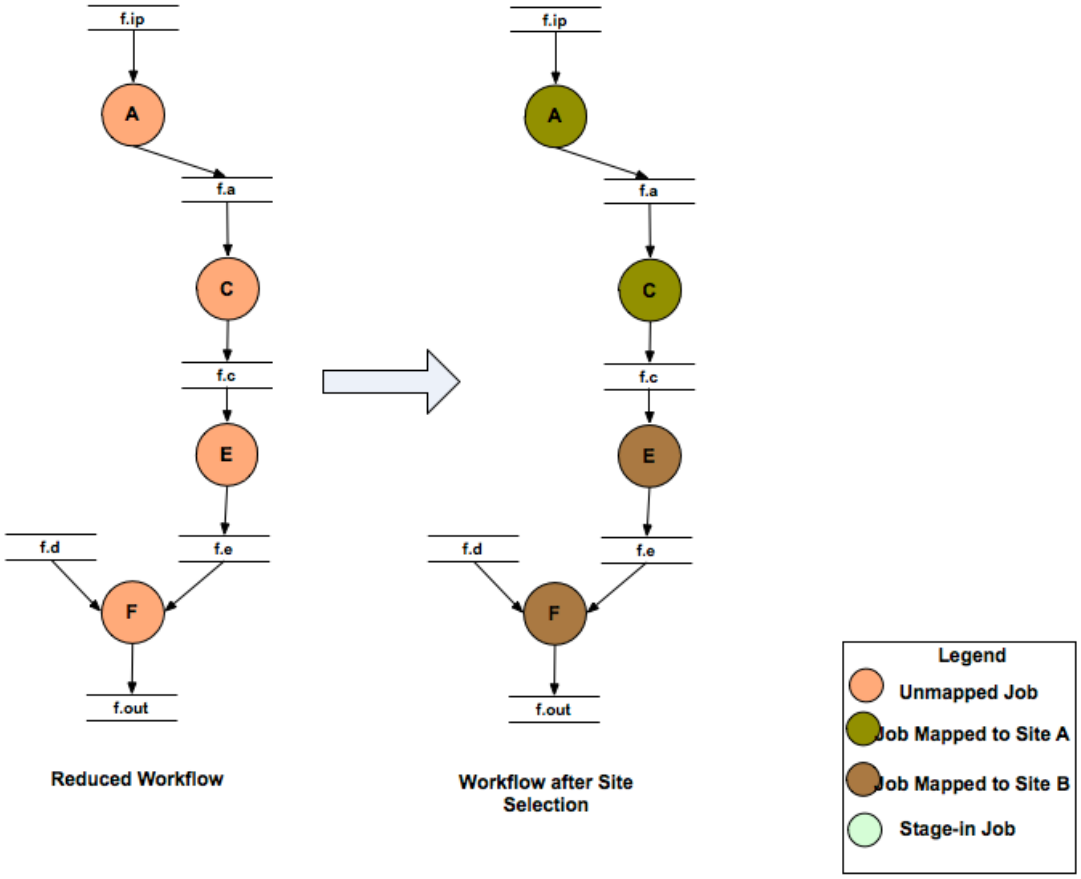
Key	Value

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 Reference Manual chapter.

Tip

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

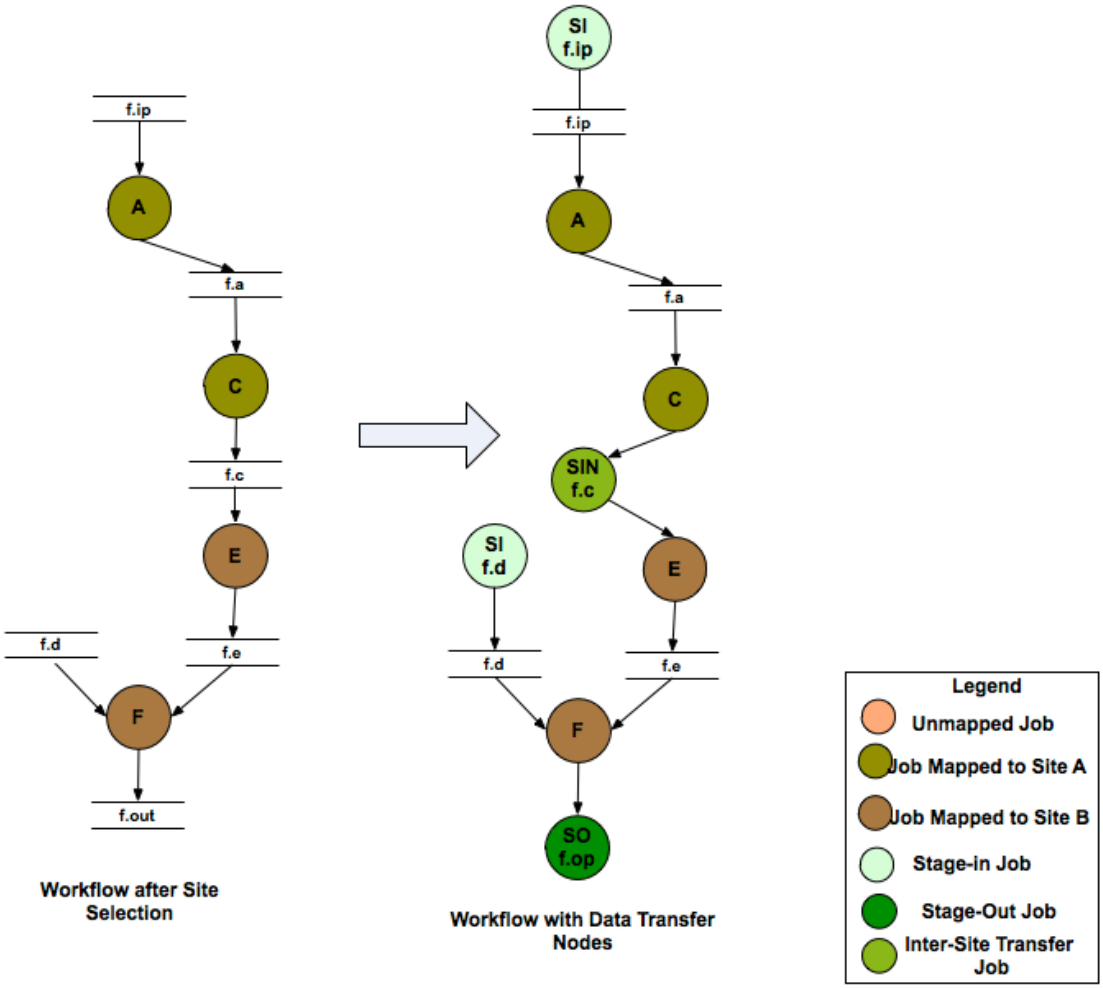
Addition of Data Transfer and Registration Nodes

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 Reference Manual. 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

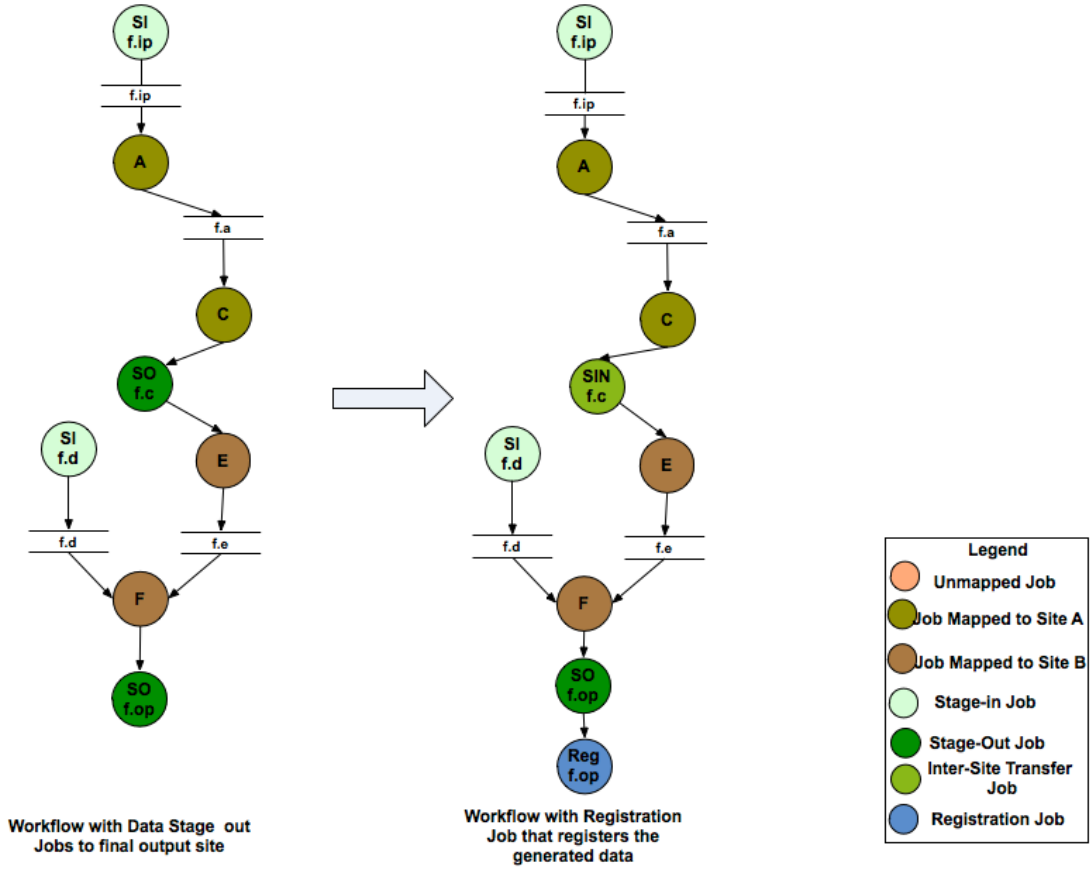
figures out automatically which underlying client to use. pegasus-transfer is distributed with the PEGASUS and can be found in the bin subdirectory . Pegasus Transfer Refiners are are described in the detail in the Transfers section of the Reference Manual. The default transfer refiner that is used in Pegasus is the **Bundle** Transfer Refiner, that bundles data stage-in nodes and data stage-out nodes 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.

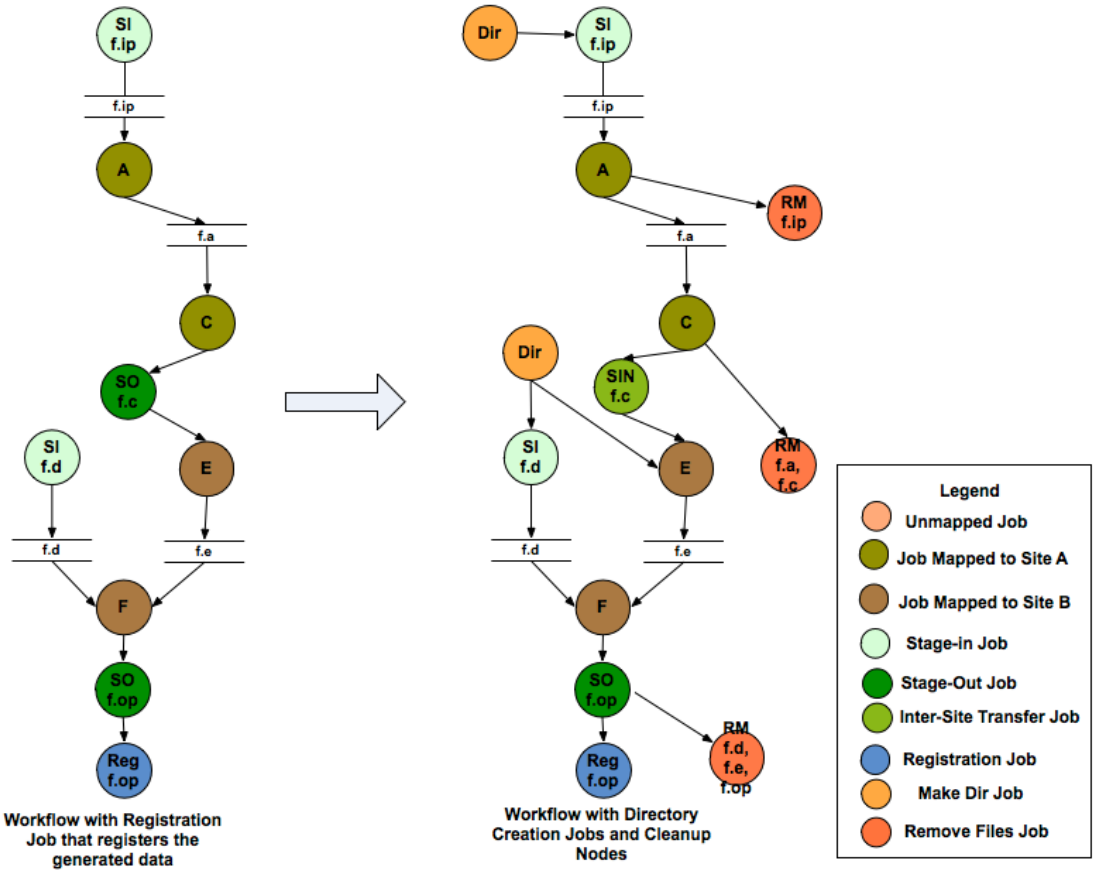
The staging site for a job is the execution site if running in a sharedfs mode, else it is the one specified by **--staging-site** option to the planner. More details about staging site can be found in the data staging configuration chapter.

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.

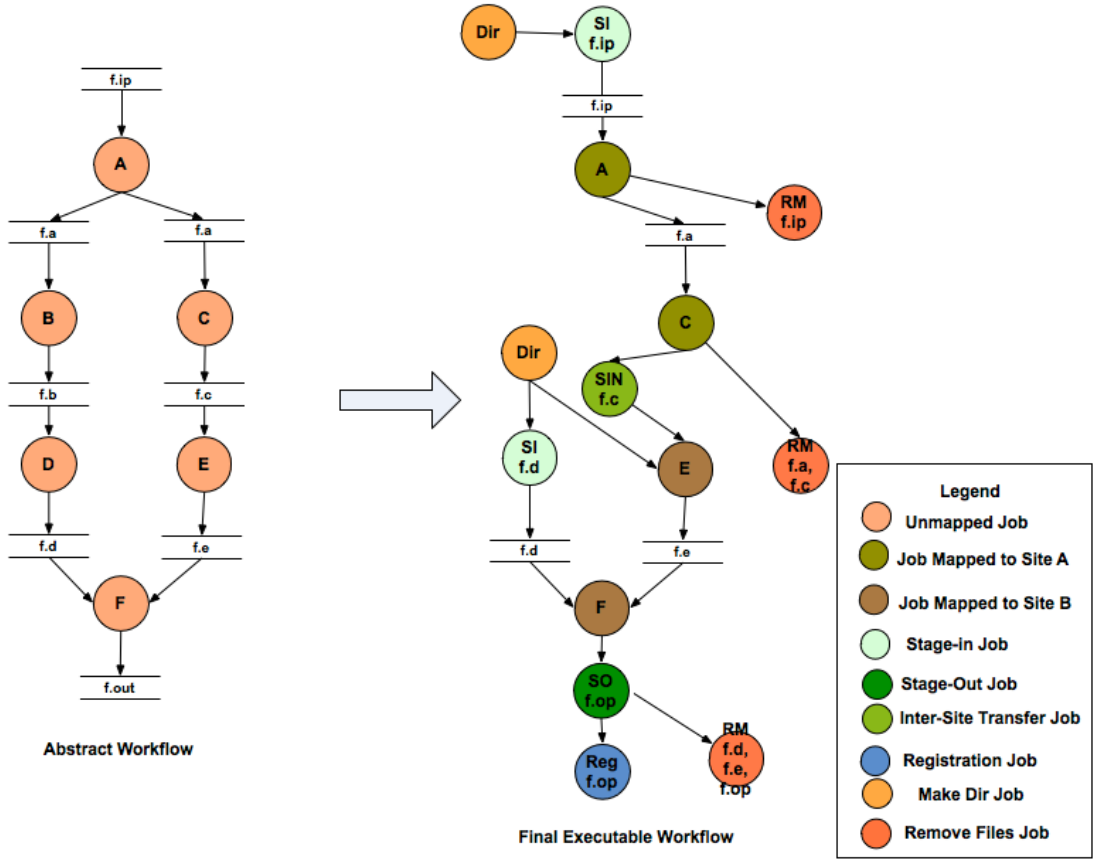
2. **Shell**

   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

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.

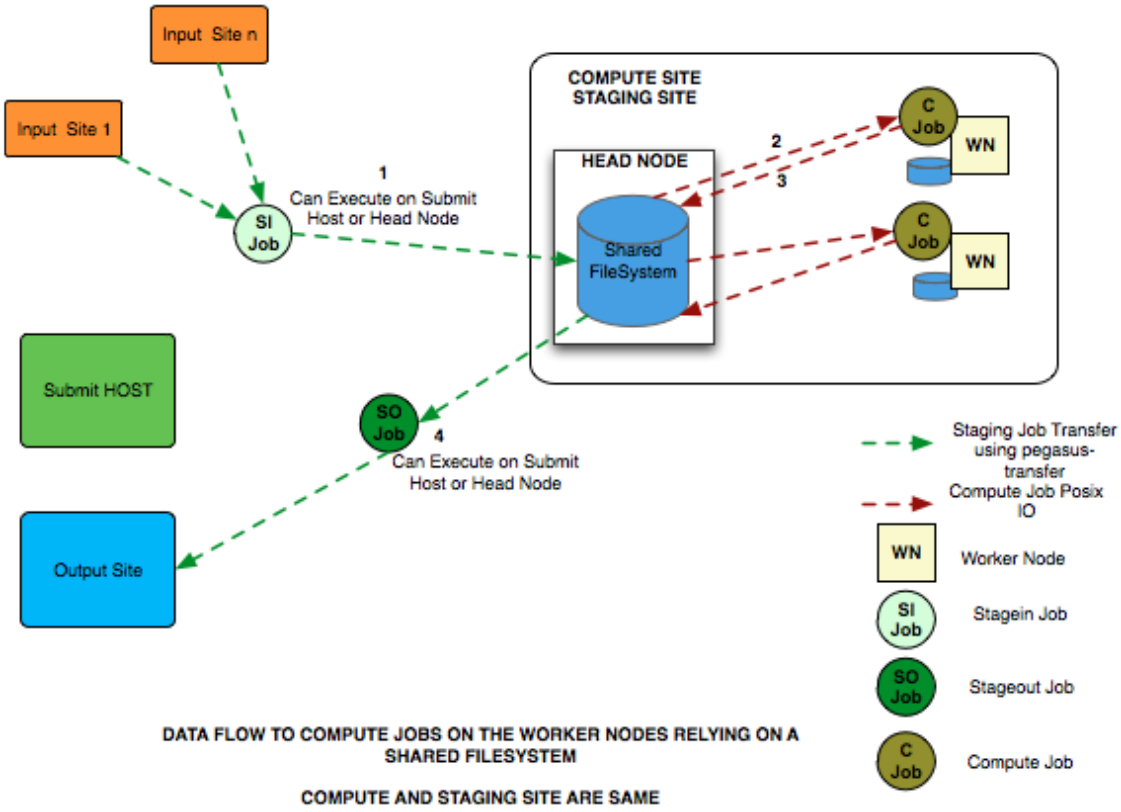
7. **Worker Node Directory**

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

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



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.

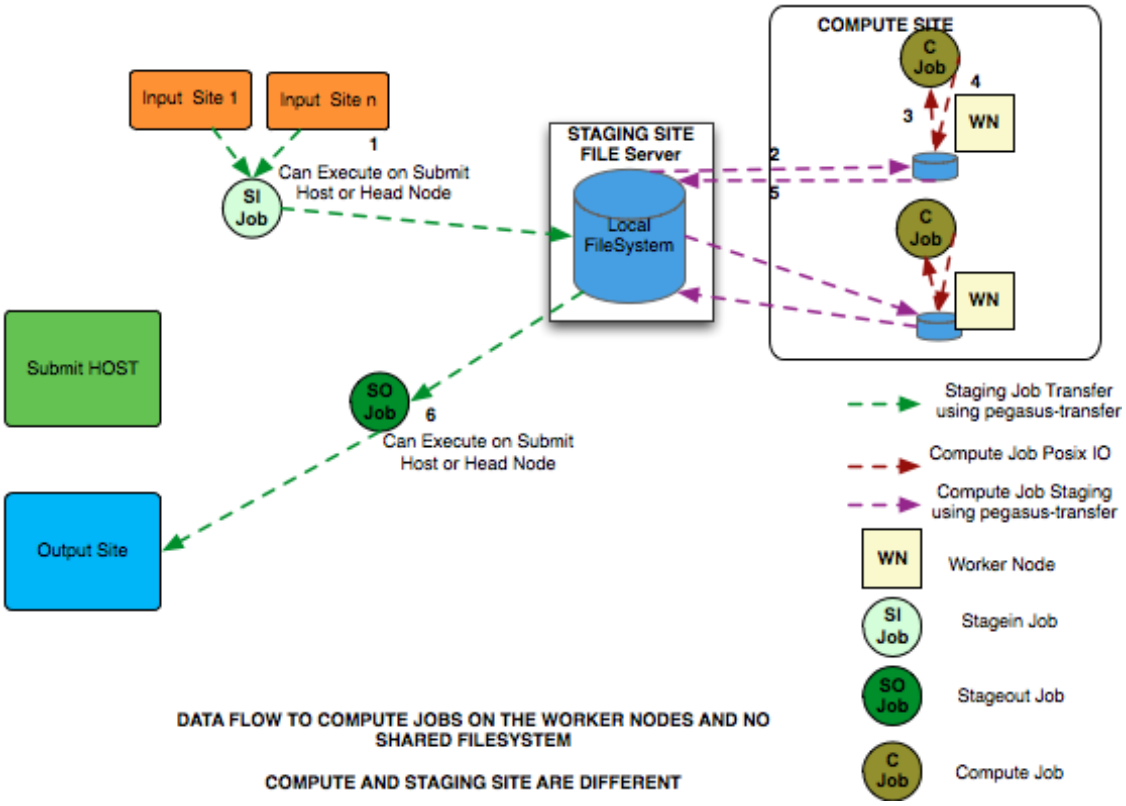
Setup

- 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.

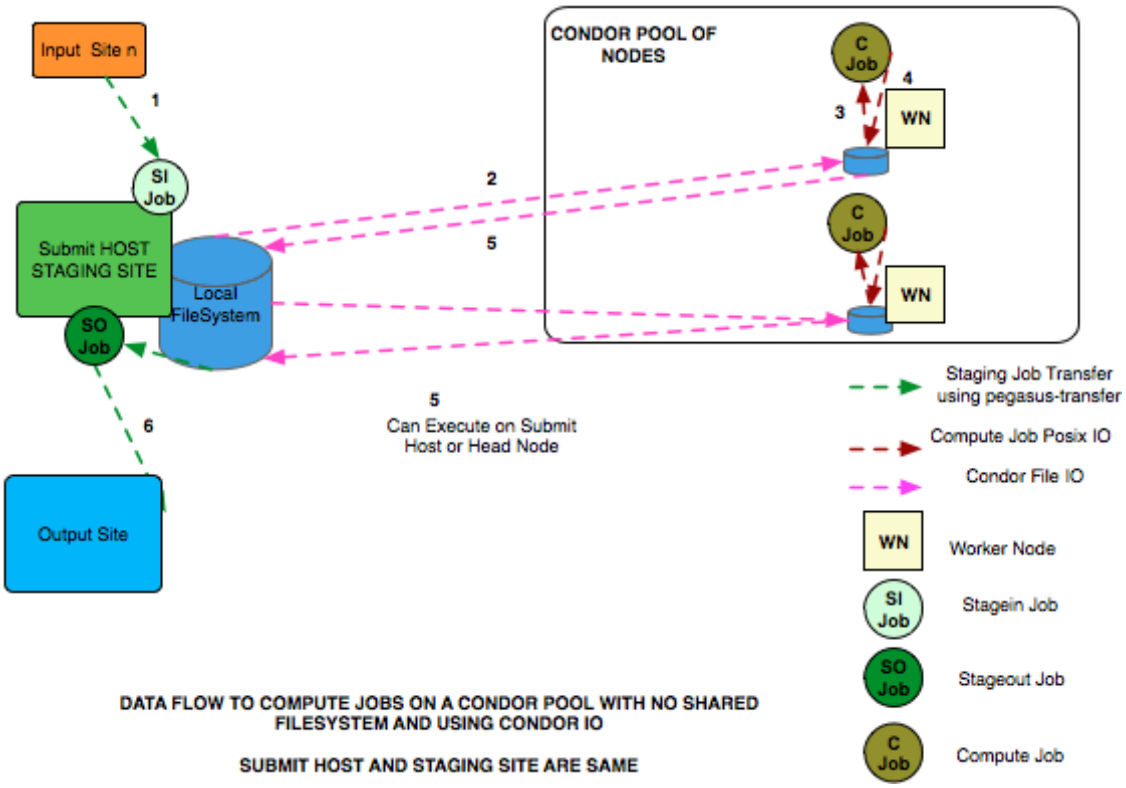
Condor Pool Without a Shared Filesystem

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.

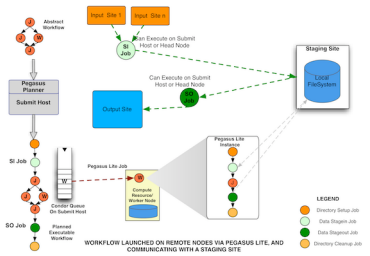
Tip

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 theReference chapter.

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

Table 5.2. Table2: 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

This is the reference guide to the basic properties regarding the Pegasus Workflow Planner, and their respective default values. Please refer to the advanced properties guide to know about all the properties that a user can use to configure the Pegasus Workflow Planner. 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.replica              Regex
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 Properties

Replica Catalog

pegasus.catalog.replica

System:	Pegasus
Since:	2.0
Type:	enumeration
Value[0]:	RLS
Value[1]:	LRC
Value[2]:	JDBCRC
Value[3]:	File
Value[4]:	Directory
Value[5]:	MRC
Value[6]:	Regex

Default:	RLS

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.

RLS
: RLS (Replica Location Service) is a distributed replica catalog, which ships with GT4. There is an index service called Replica Location Index (RLI) to which 1 or more Local Replica Catalog (LRC) report. Each LRC can contain all or a subset of mappings. In this mode, Pegasus queries the central RLI to discover in which LRC's the mappings for a LFN reside. It then queries the individual LRC's for the PFN's. To use RLS, the user additionally needs to set the property pegasus.catalog.replica.url to specify the URL for the RLI to query. Details about RLS can be found at http://www.globus.org/toolkit/data/rls/

LRC
: If the user does not want to query the RLI, but directly a single Local Replica Catalog. To use LRC, the user additionally needs to set the property pegasus.catalog.replica.url to specify the URL for the LRC to query. Details about RLS can be found at http://www.globus.org/toolkit/data/rls/

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.url
  2. pegasus.catalog.replica.db.user
  3. pegasus.catalog.replica.db.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 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 "pool".

  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 "pool".

  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:///Volumes/data/input/f.a pool="local"
f.a file:///Volumes/data/input/f.a pool="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:///Volumes/data/input/[1]/[0] pool="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://$input/f.1  pool="local"
f.2 file://$input/f.2  pool="local"
D1/f.3 file://$input/D2/f.3 pool="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
```

For example, if a user wants to query two lrc's at the same time he/she can specify as follows

```
pegasus.catalog.replica.mrc.lrc1 LRC
pegasus.catalog.replica.mrc.lrc2.url rls://sukhna
pegasus.catalog.replica.mrc.lrc2 LRC
pegasus.catalog.replica.mrc.lrc2.url rls://smarty
```

In the above example, lrc1, lrc2 are any valid identifier names and url is the property key that needed to be specified.

pegasus.catalog.replica.url

System:	Pegasus
Since:	2.0
Type:	URI string
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.

Site Catalog

pegasus.catalog.site.file

System:	Site Catalog
Since:	2.0
Type:	file location string
Default:	${pegasus.home.sysconfdir}/sites.xml

Running things on the grid requires an extensive description of the capabilities of each compute cluster, commonly termed "site". This property describes the location of the file that contains such a site description. As the format is currently in flow, please refer to the userguide and Pegasus for details which format is expected.

Transformation Catalog

pegasus.catalog.transformation

System:	Transformation Catalog
Since:	2.0
Type:	enumeration
Value[0]:	Text
Value[1]:	File
Default:	Text
See also:	pegasus.catalog.transformation.file

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

```
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"
```

File THIS FORMAT IS DEPRECATED. WILL BE REMOVED IN COMING VERSIONS. USE pegasus-tc-converter to convert File format to Text Format. In this mode, a file 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 new TC file format uses 6 columns:

1. The resource ID is represented in the first column.
2. The logical transformation uses the colonized format ns::name:vs.
3. The path to the application on the system
4. The installation type is identified by one of the following keywords - all upper case: INSTALLED, STAGEABLE. If not specified, or **NULL** is used, the type defaults to INSTALLED.
5. The system is of the format ARCH::OS[:VER:GLIBC]. The following arch types are understood: "INTEL32", "INTEL64", "SPARCV7", "SPARCV9". The following os types are understood: "LINUX", "SUNOS", "AIX". If unset or **NULL**, defaults to INTEL32::LINUX.
6. Profiles are written in the format NS::KEY=VALUE,KEY2=VALUE;NS2::KEY3=VALUE3 Multiple key-values for same namespace are seperated by a comma "," and multiple namespaces are seperated by a semicolon ";". If any of your profile values contains a comma you must not use the namespace abbreviator.

pegasus.catalog.transformation.file

Systems:	Transformation Catalog
Type:	file location string
Default:	${pegasus.home.sysconfdir}/tc.text | ${pegasus.home.sysconfdir}/tc.data
See also:	pegasus.catalog.transformation

This property is used to set the path to the textual transformation catalogs of type File or Text. If the transformation catalog is of type Text then tc.text file is picked up from sysconfdir, else tc.data

Data Staging Configuration

pegasus.data.configuration

System:	Pegasus
Since:	3.1
Type:	enumeration
Value[0]:	sharedfs
Value[1]:	nonsharedfs
Value[2]:	condorio
Default:	sharedfs

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 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. Internally, if this is set the following properties are set.

```
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 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. On loading this property, internally the following properies are set

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

Chapter 6. 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 Condor pool, and shell planner. The former is preferred as the latter does not support all Pegasus' features (such as notifications).

Running on a local Condor 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 condor universe local. The site catalog can be left very simple in this case:

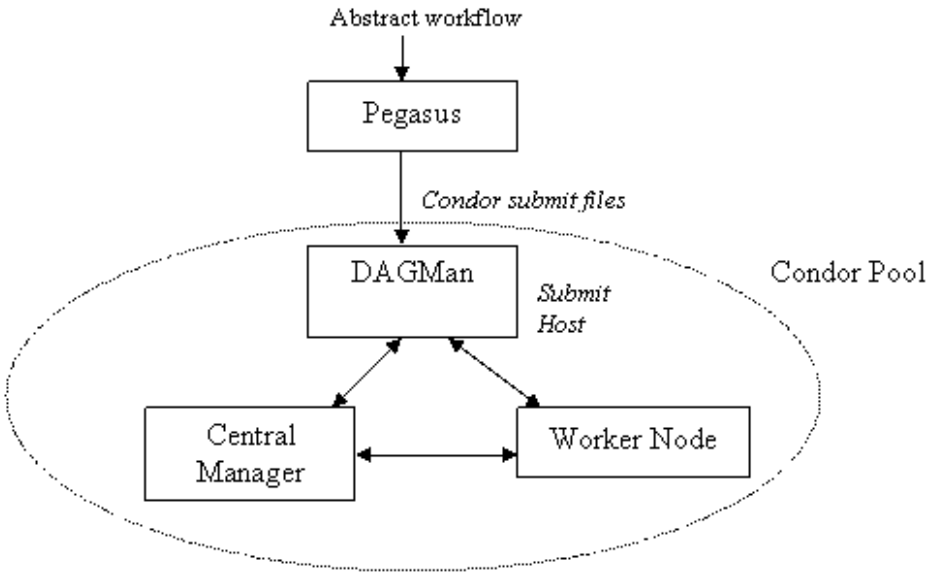
```
<?xml version="1.0" encoding="UTF-8"?>
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog http://pegasus.isi.edu/
schema/sc-4.0.xsd"
             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 Condor. 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 Condor pool is a set of machines that use Condor for resource management. A Condor 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 Condor pool.

Figure 6.1. The distributed resources appear to be part of a Condor pool.



The workflow is submitted using DAGMan from one of the job submission machines in the Condor 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 Condor 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 Condor 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 Condor pool, where each resource identifies a subset of machine (worker nodes) in the pool.

When running on a Condor 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 - Condor will do the transfers using the existing Condor daemons. For an example of this mode see the example workflow in `share/pegasus/examples/condor-blackdiamond-condorio/`. In condorio mode, the site catalog for the execution site is very simple as storage is provided by Condor:

```
<?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 Condor 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 Condor pool:

```
  <!-- Change the style to Condor for jobs to be executed in the Condor 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 Condor 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 Condor pool for certain timeframe. These machines join the Condor pool temporarily and can be used to execute jobs in a non preemptive manner. This functionality is achieved using a Condor feature called **glideins** (see http://cs.wisc.edu/condor/glidein) . The startd daemon is the Condor 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 configued 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 Condor 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 Condor.
- 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 CondorC users can submit workflows to remote condor pools. CondorC is a condor specific solution for remote submission that does not involve the setting up a GRAM on the headnode. To enable CondorC submission to a site, user needs to associate pegasus profile key named style with value as condorc. In case, the remote Condor pool does not have a shared filesytem between the nodes making up the pool, users should use pegasus in the condorio data configuration. In this mode, all the data is staged to the remote node in the Condor pool using Condor File transfers and is executed using PegasusLite.

A sample site catalog for submission to a CondorC 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"
```

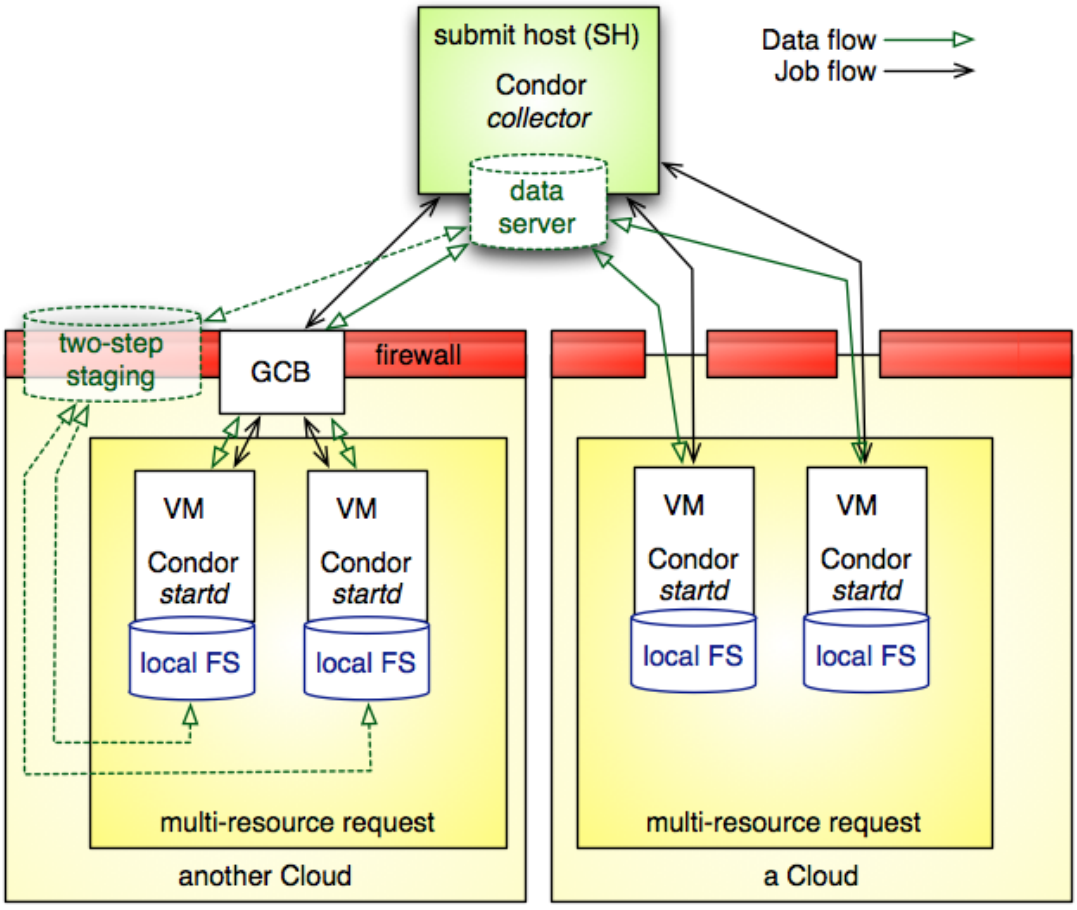
```
             version="4.0">

    <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 CondorC 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 condorc -->
        <profile namespace="pegasus" key="style">condorc</profile>

        <!-- specify which condor 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 CondorIO mode, users should set the following in their properties

```
# pegasus properties
pegasus.data.configuration    condorio
```

Infrastructure Clouds

Figure 6.2. Cloud Sample Site Layout



This figure shows a sample environment for executing Pegasus on multiple clouds (known as "sky computing"). At this point, it is up to the user to provision the remote resources with a proper VM image that includes a Condor worker that is configured to report back to a Condor master outside the cloud.

In this discussion, the *submit host* (SH) is located logically external to the cloud provider(s). The SH is the point where a user submits Pegasus workflows for execution. This site typically runs a Condor collector to gather resource announcements, or is part of a larger Condor pool that collects these announcements. Condor makes the remote resources available to the submit host's Condor 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 provisioning broker -- Nimbus, Eucalyptus or EC2 -- at the remote site is responsible to allocate and set up the resources. For a multi-node request, the worker nodes often require access to a form of shared data storage. Concretely, either a POSIX-compliant shared file system (e.g. NFS, PVFS) is available to the nodes, or can be brought up for the lifetime of the application workflow. The task steps of the application workflow facilitate shared file systems to exchange intermediary results between tasks on the same cloud site. Pegasus also supports an S3 data mode for the application workflow data staging.

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.

Once resources begin appearing in the pool managed by the submit machine’s Condor collector, the application workflow can be submitted to Condor. A Condor 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 Condor 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.

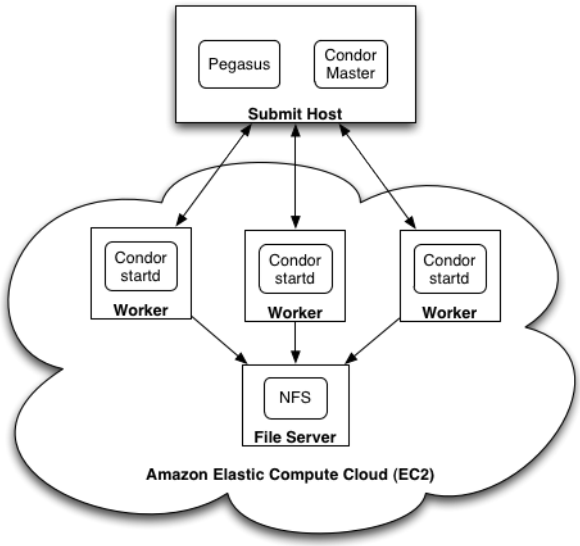
The left side shows a more difficult setup where the connectivity is fully firewalled without any connectivity except to in-site nodes. In this case, a proxy server process, the *generic connection broker* (GCB), needs to be set up in the DMZ of the cloud site to facilitate Condor I/O between the submit 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 6.3. Amazon EC2



The submit machine runs Pegasus and a Condor master (collector, schedd, negotiator). The workers run a Condor 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 Condor for this environment can be found at http://www.isi.edu/~gideon/condor-ec2 [http://www.isi.edu/~gideon/condor-ec2/].

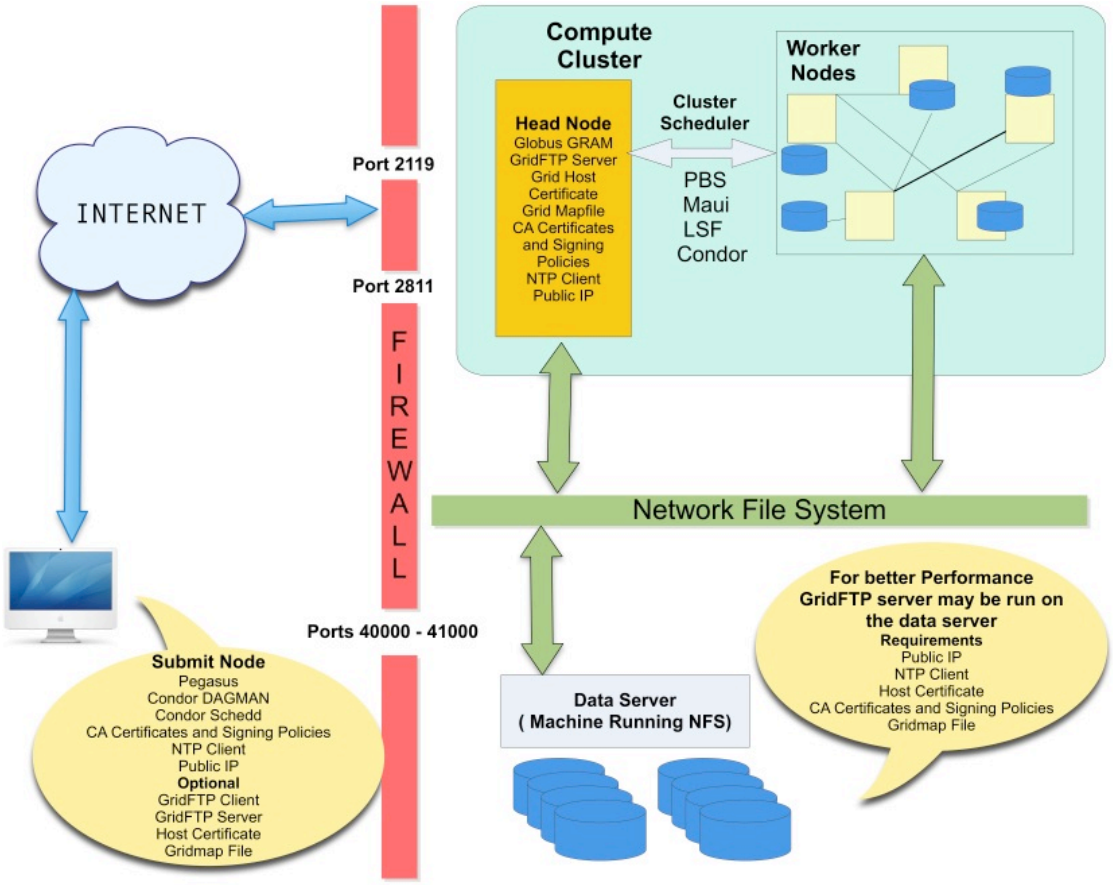
The site catalog entry for this configuration is similar to what you would create for running on a local Condor pool with a shared file system.

FutureGrid

FutureGrid [https://portal.futuregrid.org/] is a distributed testbed for cloud computing. There is a tutorial on how to run Pegasus on FutureGrid using the Nimbus cloud management system here: http://pegasus.isi.edu/futuregrid/tutorials [http://pegasus.isi.edu/futuregrid/tutorials/]

Remote Cluster using Globus GRAM

Figure 6.4. 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, Condor 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, Condor wraps the job into Globus calls via Condor-G. Globus uses GRAM to manage jobs on remote sites. In terms of a software stack, Pegasus wraps the job into Condor. Condor 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 Condor 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" />

        <directory type="shared-scratch" path="/home/virgo034">
            <file-server operation="all" url="gsiftp://ce01-lcg.cr.cnaf.infn.it/home/virgo034"/>
        </directory>



        <profile namespace="pegasus" key="style">cream</profile>
        <profile namespace="globus" key="queue">virgo</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.italiangrid.it/twiki/bin/view/CREAM/UserGuide#1_1_Before_starting_get_your_use] .

Local Cluster Using Glite

This section describes the various changes required in the site catalog for Pegasus to generate an executable workflow that uses gLite blahp to directly submit to PBS on the local machine. This mode of submission should only be used when the condor on the submit host can directly talk to scheduler running on the cluster. It is recommended that the cluster that gLite talks to is designated as a separate compute site in the Pegasus site catalog. To tag a site as a gLite site the following two profiles need to be specified for the site in the site catalog

1. **pegasus** profile **style** with value set to **glite**.

2. **condor** profile **grid_resource** with value set to **pbs|lsf**

An example site catalog entry for a glite site looks as follows 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="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-pbs" 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-pbs/shared-
scratch">
            <file-server operation="all" url="file:///lfs/glite-sharedfs-example/local-pbs/shared-
scratch"/>
        </directory>

        <profile namespace="env" key="PEGASUS_HOME">/lfs/software/pegasus/pegasus-4.2.0</profile>

        <profile namespace="pegasus" key="style" >glite</profile>
        <profile namespace="pegasus" key="change.dir">true</profile>

        <profile namespace="condor" key="grid_resource">pbs</profile>
        <profile namespace="condor" key="batch_queue">batch</profile>
        <profile namespace="globus" key="maxwalltime">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

Changes to Jobs

As part of applying the style to the job, this style adds the following classads expressions to the job description.

1. +remote_queue - value picked up from globus profile queue
2. +remote_cerequirements - See below

Remote CE Requirements

The remote CE requirements are constructed from the following profiles associated with the job. The profiles for a job are derived from various sources

1. transformation catalog
2. site catalog
3. DAX
4. user properties

The following globus profiles if associated with the job are picked up and translated to corresponding glite key

1. hostcount -> PROCS
2. count -> NODES
3. maxwalltime -> WALLTIME

The following condor profiles if associated with the job are picked up and translated to corresponding glite key

1. priority -> PRIORITY

All the env profiles are translated to MYENV

The remote_cerequirements expression is constructed on the basis of the profiles associated with job . An example +remote_cerequirements classad expression in the submit file is listed below

```
+remote_cerequirements = "PROCS==18 && NODES==1 && PRIORITY==10 && WALLTIME==3600 \
   && PASSENV==1 && JOBNAME==\"TEST JOB\" && MYENV ==\"JAVA_HOME=/bin/java,APP_HOME=/bin/app\""
```

Specifying directory for the jobs

gLite blahp does not follow the remote_initialdir or initialdir classad directives. Hence, all the jobs that have this style applied don't have a remote directory specified in the submit directory. Instead, Pegasus relies on kickstart to change to the working directory when the job is launched on the remote node.

Remote Cluster using BOSCO and SSH submissions

BOSCO [http://bosco.opensciencegrid.org/about/] enables users to submit jobs to remote clusters using SSH. This description describes how to specify a site catalog entry for a site to which jobs can be submitted over SSH. 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 as follows 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="USC_HPCC_Bosco" arch="x86_64" os="LINUX">

        <!-- Specify the service information as grid gateways. This should match what Bosco provided
 when the cluster
             was set up. -->
        <grid type="batch" contact="username@hpc-login2.usc.edu" scheduler="PBS" jobtype="compute"/>
        <grid type="batch" contact="username@hpc-login2.usc.edu" scheduler="PBS"
 jobtype="auxillary"/>

        <!-- Scratch directory on the cluster -->
        <directory type="shared-scratch" path="/home/rcf-40/tmp">
            <file-server operation="all" url="scp://username@hpc-login2.usc.edu/home/rcf-40/tmp"/>
        </directory>

        <!-- SSH is the style to use for Bosco SSH submits -->
        <profile namespace="pegasus" key="style">ssh</profile>

        <!-- Bosco is using the grid universe, which means the globus
             namespace can be used to control the jobs -->
        <profile namespace="globus" key="queue">default</profile>
        <profile namespace="globus" key="maxwalltime">30</profile>

    </site>


</sitecatalog>
```

Note

It is recommended to have a submit node configured either as a BOSCO submit node or a vanilla Condor node. You cannot have Condor configured both as a BOSCO install and a traditional condor submit node.

Starting 4.3 there is a bosco-shared-fs example in the examples directory of the distribution.

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 Condor, please see the Condor 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 Condor based, so standard Condor 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 7. 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 **<daxlabel-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">7F454C4602010100000000000000000</
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 DAGMan.

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.

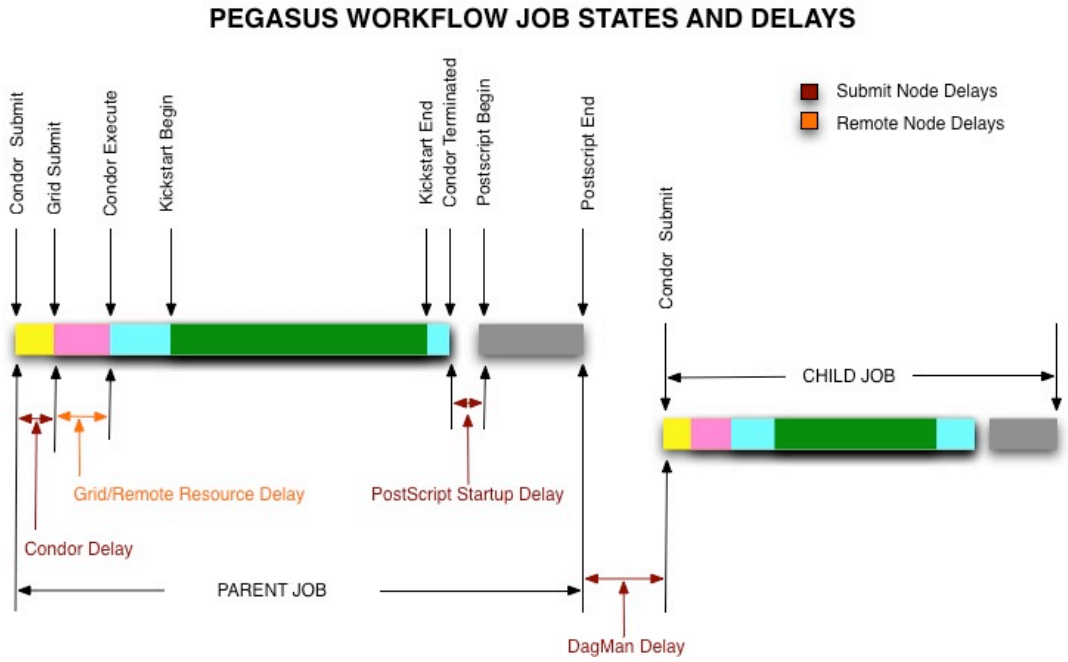
Table 7.1. Table 1: The job lifecycle when executed as part of the workflow

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 7.2. Table 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 8. 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. These tools internally parse the Condor log files and have a similar interface. 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>
```

All these utilities query a database (usually a sqllite in the workflow submit directory) that is populated by the monitoring daemon **pegasus-monitord** .

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 unsubmitted :      0 (0.00%)

*****************************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. 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 walltime 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 walltime 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 walltime value includes jobs from the sub workflows as well.
# Cumulative job walltime as seen from submit side:
#   The sum of the walltime of all jobs as reported by DAGMan.
#   This is similar to the regular cumulative job walltime, 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 walltime 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 walltime as seen from submit side : 4 mins, 10 secs
```

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 walltime 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 walltime 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 walltime value includes jobs from the sub workflows as well.
# Cumulative job walltime as seen from submit side:
#   The sum of the walltime of all jobs as reported by DAGMan.
#   This is similar to the regular cumulative job walltime, 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 walltime 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 walltime as seen from submit side : 8 mins, 35 secs
```

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 walltime 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 cummulate job wall time** - The sum of the walltime 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 walltime value includes jobs from the sub workflows as well. This value is multiplied by the multiplier_factor in the job instance table.
- **Cumulative job walltime as seen from submit side** - The sum of the walltime of all jobs as reported by DAGMan. This is similar to the regular cumulative job walltime, 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 walltime 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 8.1. Workflow Statistics

#	Type	Succeeded	Failed	Incomplete	Total	Retries	Total Run	Workflow Retries
2a6df11b-9972-4ba0-b4ba-4fd39c357af4								0
	Tasks	4	0	0	4	0	4	
	Jobs	13	0	0	13	0	13	
	Sub Workflows	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 8.2. Job statistics

	Job	Try	Site	Kick-start	Mult	Kickstart_Mult	CPU-Time	Post	Con-dorQ-Time	Re-source	Run-time	Se-qexec	Seqex-ec-De-lay
analyze_ID0000004		4	local	60.002	1	60.002	59.843	5.0	0.0	-	62.0	-	-
create_dir_diamond_01_local		1	local	0.027	1	0.027	0.003	5.0	5.0	-	0.0	-	-
findrange_ID0000002		2	local	60.001	10	600.01	59.921	5.0	0.0	-	60.0	-	-
findrange_ID0000003		3	local	60.002	10	600.02	59.912	5.0	10.0	-	61.0	-	-
preprocess_ID0000001		1	local	60.002	1	60.002	59.898	5.0	5.0	-	60.0	-	-
register_local_1_0		1	local	0.459	1	0.459	0.432	6.0	5.0	-	0.0	-	-
register_local_1_1		1	local	0.338	1	0.338	0.331	5.0	5.0	-	0.0	-	-
register_local_2_0		1	local	0.348	1	0.348	0.342	5.0	5.0	-	0.0	-	-
stage_in_local_local		1_0	local	0.39	1	0.39	0.032	5.0	5.0	-	0.0	-	-
stage_out_local_local		10_0	local	0.165	1	0.165	0.108	5.0	10.0	-	0.0	-	-
stage_out_local_local		11_0	local	0.147	1	0.147	0.098	7.0	5.0	-	0.0	-	-
stage_out_local_local		11_1	local	0.139	1	0.139	0.089	5.0	6.0	-	0.0	-	-
stage_out_local_local		12_0	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 multipler_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 8.3. Transformation Statistics

Transformation	Count	Succeeded	Failed	Min	Max	Mean	Total
dagman::post	13	13	0	5.0	7.0	5.231	68.0
diamond::analyze	1	1	0	60.002	60.002	60.002	60.002
diamond::findrange	2	2	0	600.01	600.02	600.02	1200.03
diamond::preprocess	1	1	0	60.002	60.002	60.002	60.002
pegasus::dirmanager	1	1	0	0.027	0.027	0.027	0.027
pegasus::pegasus-transfer	5	5	0	0.139	0.39	0.197	0.986
pegasus::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 8.4. 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 show below.

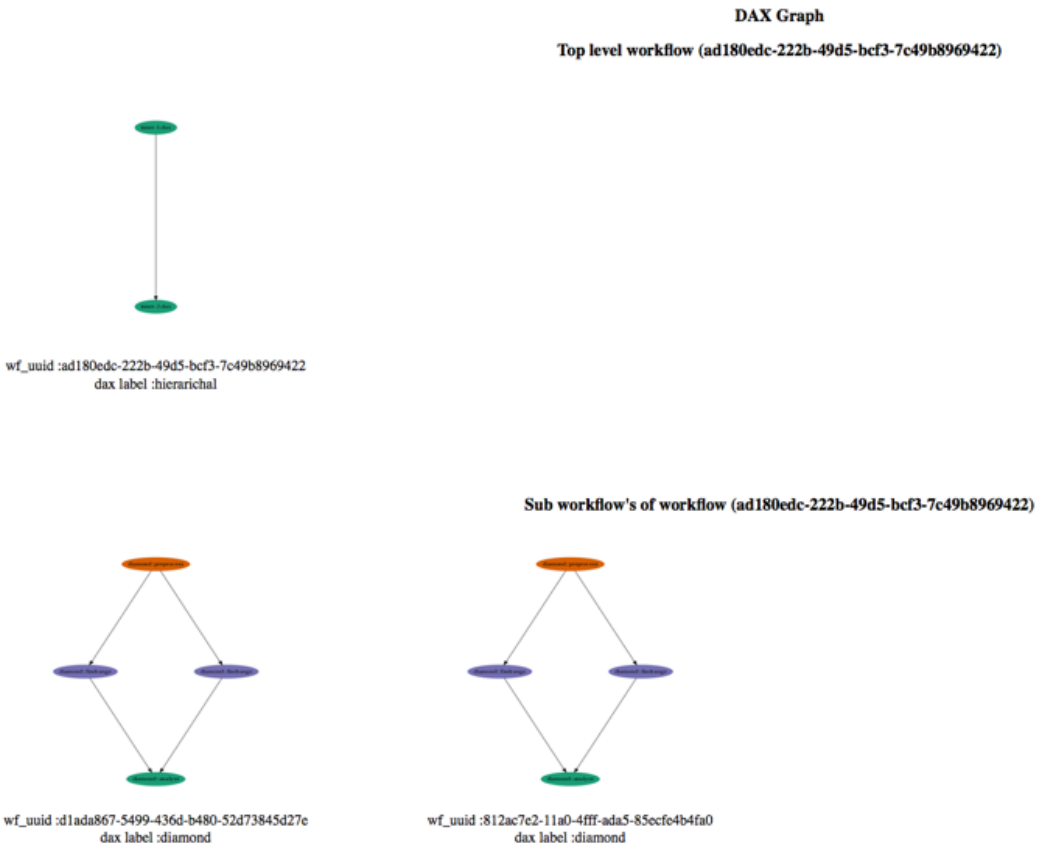
Figure 8.1. pegasus-plot index page

pegasus-plots generates the following plots and charts.

Dax Graph

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

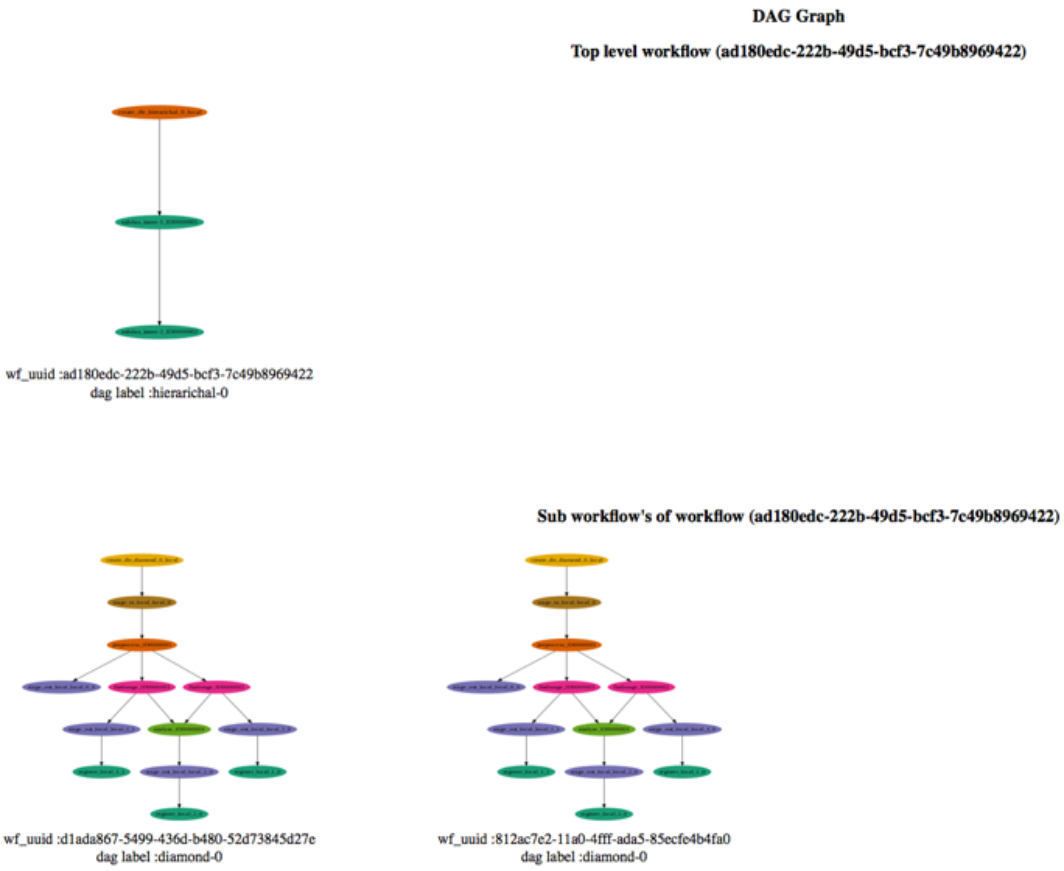
Figure 8.2. DAX Graph



Dag Graph

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

Figure 8.3. DAG Graph



Gantt workflow execution chart

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

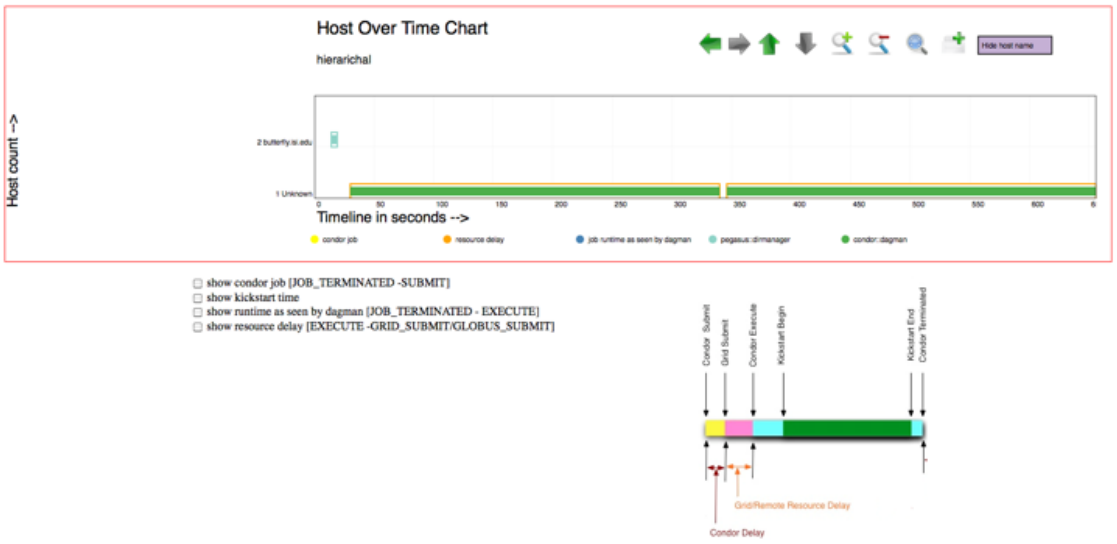
Figure 8.4. Gantt Chart

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 8.5. Host over time chart

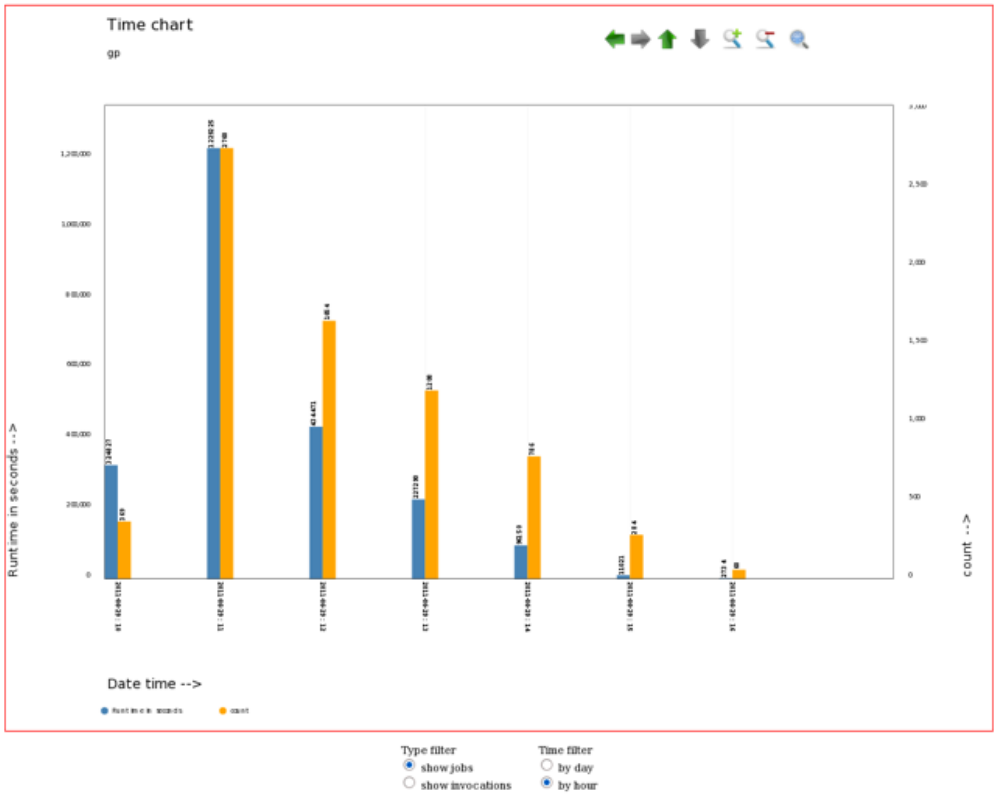


The toolbar at the top provides zoom in/out , pan left/right/top/bottom and show/hide host name functionality.The toolbar at the bottom can be used to show/hide job states. Failed job instances are shown in red border in the chart. Clicking on a sub workflow job instance will take you to the corresponding 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 8.6. Time chart

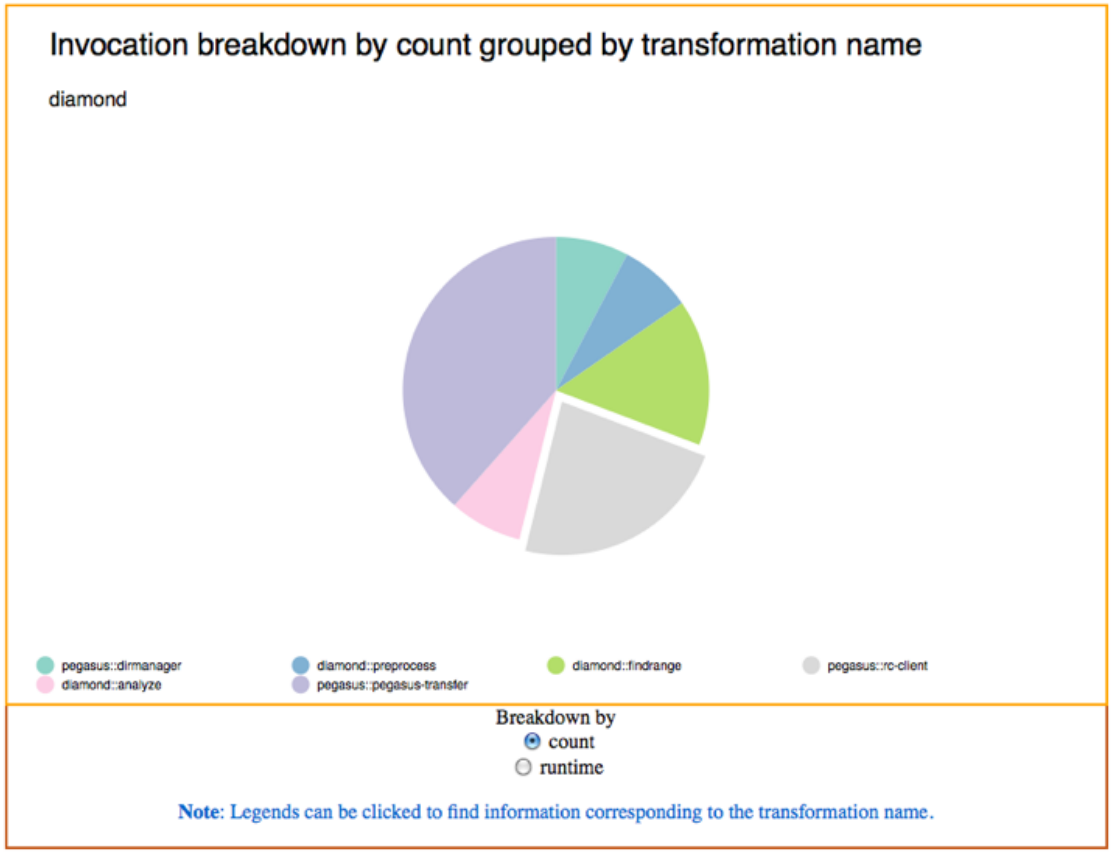


The toolbar at the top provides zoom in/out and pan left/right/top/bottom functionality. The toolbar at the bottom can be used to switch between job instances/ invocations and day/hour filtering.

Breakdown chart

Breakdown chart shows invocation count and runtime of the workflow run grouped by transformation name. A sample page is shown below.

Figure 8.7. 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.

pegasus-dashboard

pegasus-dashboard is a python based utility which starts a web server. The users can then connect to this server using a browser to monitor/debug workflows. The pegasus-dashboard utility uses Flask framework to implement it's functionality. The pegasus-dashboard command displays a message about how to install Flask and it's dependedncies, if and only if it is unable to locate an existing Flask installation.

Note

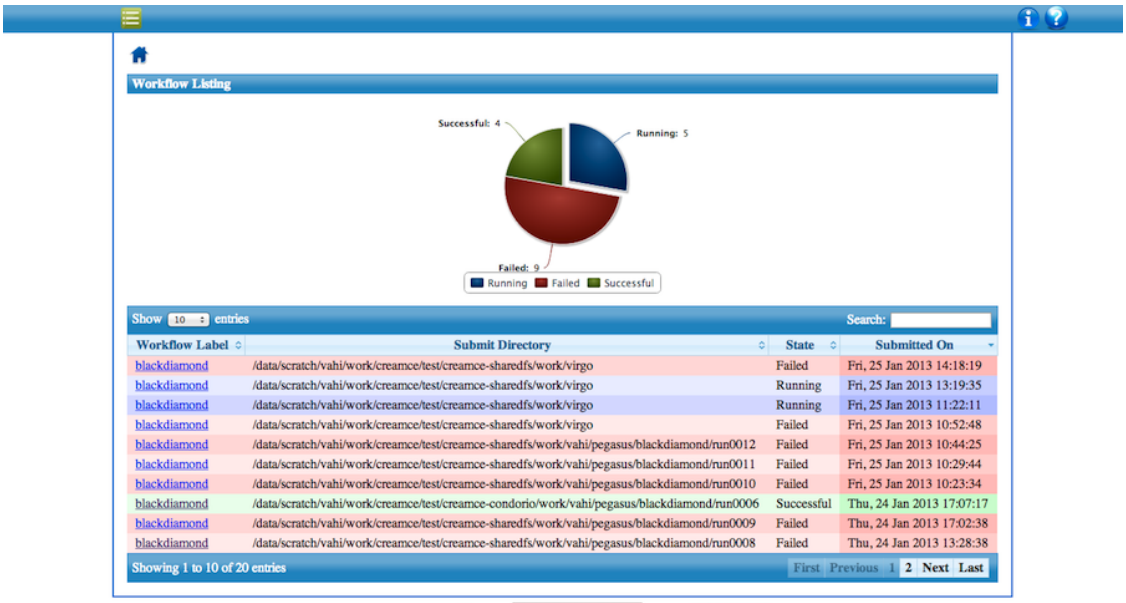
pegasus-dashboard can only monitor workflows which have been executed using Pegasus 4.2.0 and above.

By default, the server is configured to listen on all network interfaces on port 8000. A user can view the dashboard on http://<IP_ADDRESS>:8000/

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

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. The home page lists only the top level workflows (Pegasus supports hierarchical workflows i.e. workflows within a workflow).

Figure 8.8. Dashboard Home Page

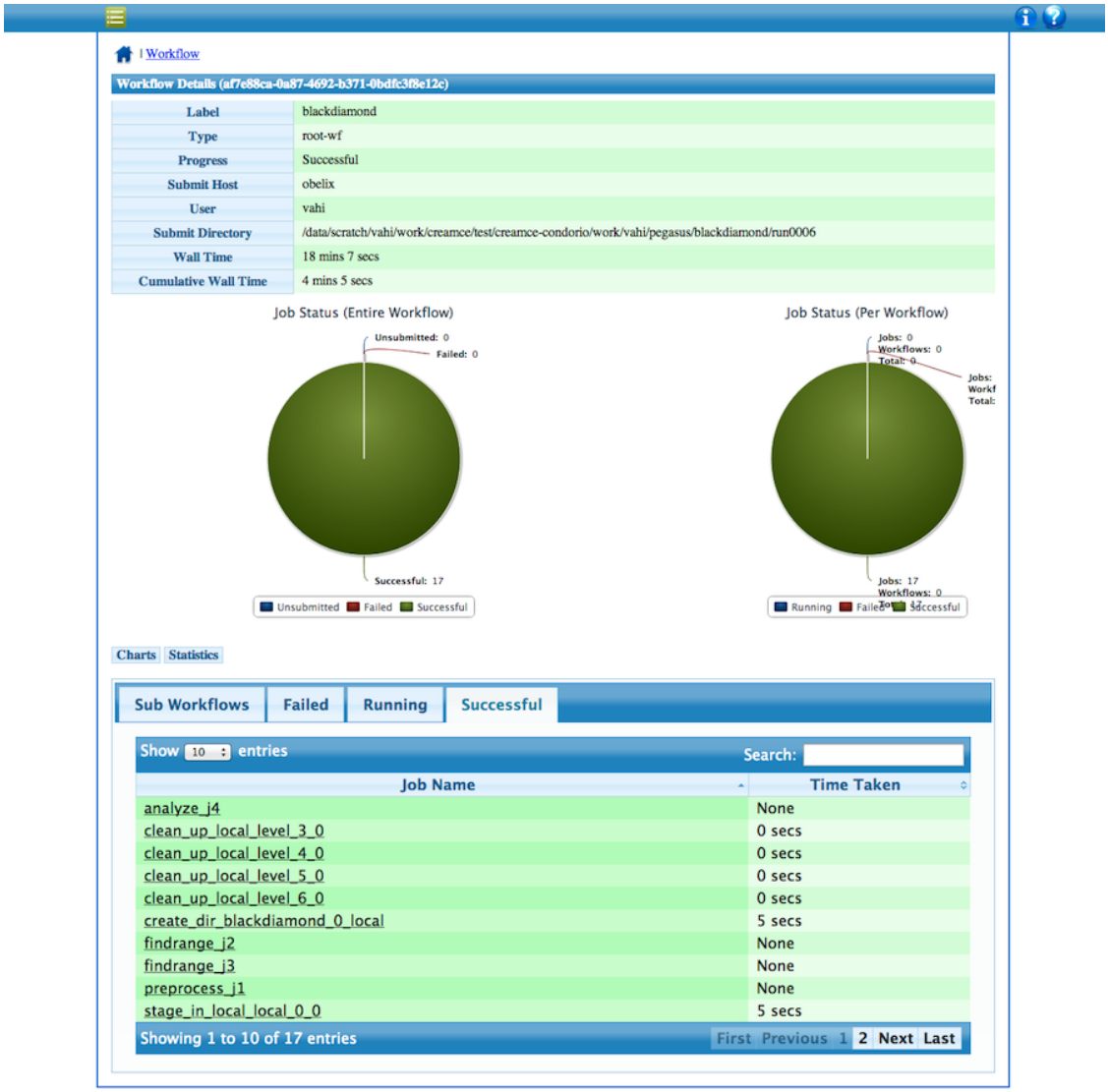


To view details specific to a workflow, the user can click on corresponding workflow label. The workflow details page lists workflow specific information like workflow label, workflow status, location of the submit directory, etc. The details page also displays pie charts showing the distribution of 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, and successful 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.

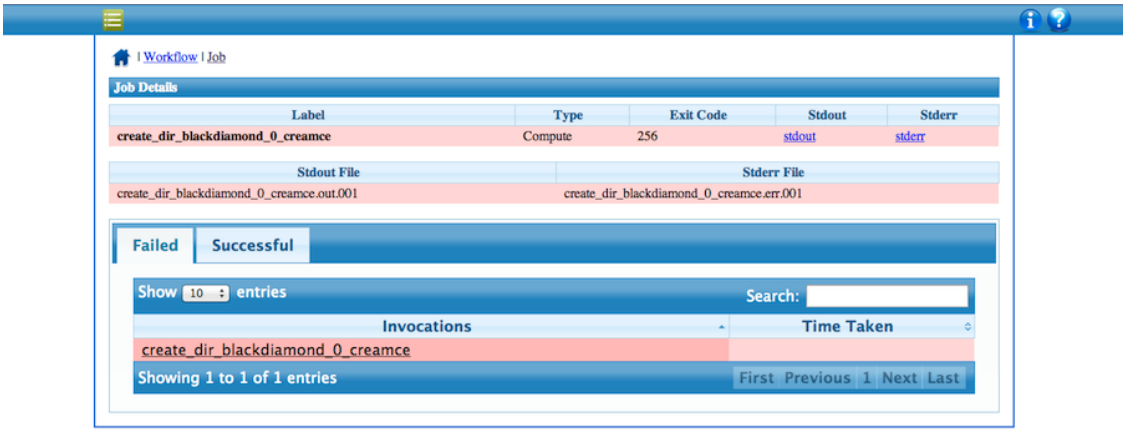
Figure 8.9. Dashboard Workflow Page



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.

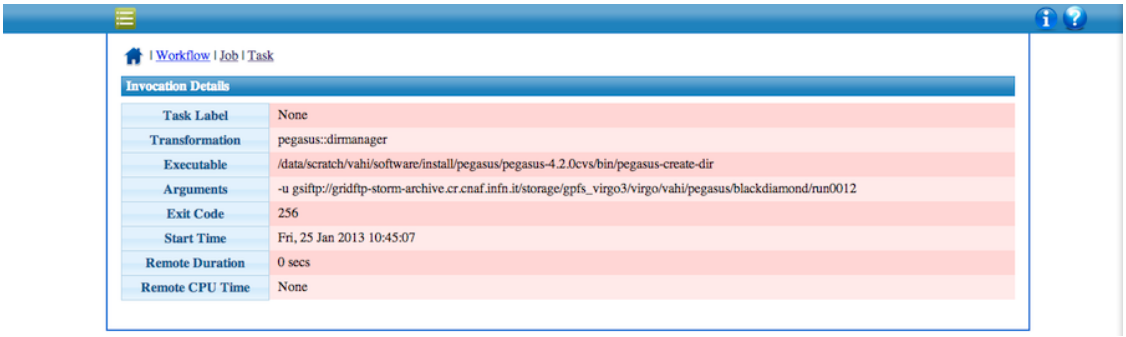
The job details page also shows tab's for failed, and successful task invocations (Pegasus allows users to group multiple smaller task's into a single job i.e. a job may consist of one or more tasks)

Figure 8.10. Dashboard Job Description Page



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 8.11. 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.

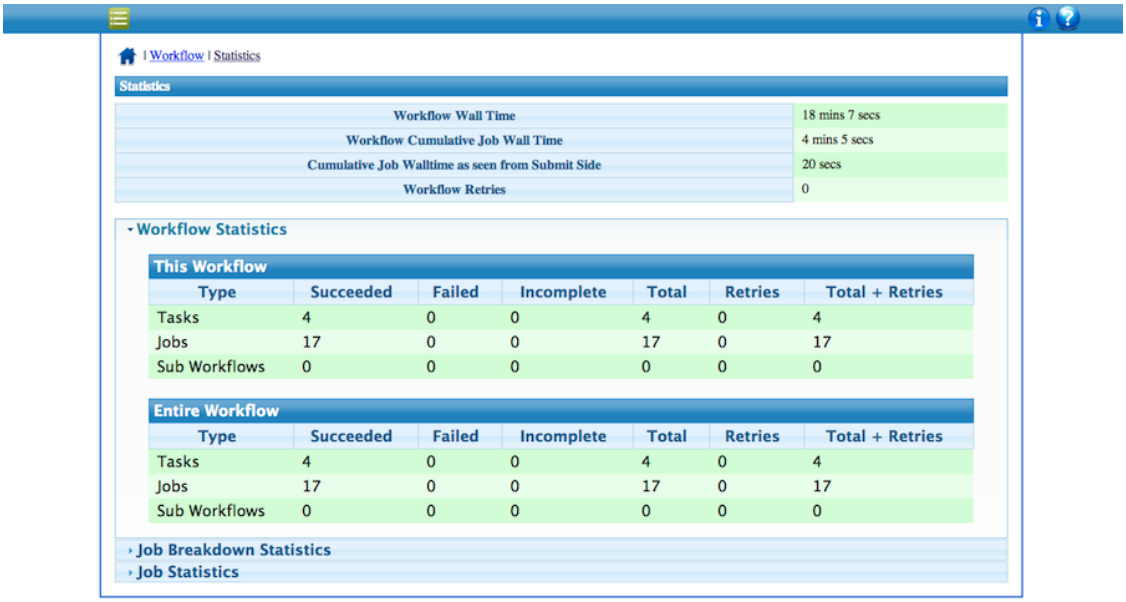
The Statistics page shows the following statistics.

1. Workflow level statistics

2. Job breakdown statistics

3. Job specific statistics

Figure 8.12. Dashboard Statistics Page



| Workflow | Statistics

Statistics

Workflow Wall Time	18 mins 7 secs
Workflow Cumulative Job Wall Time	4 mins 5 secs
Cumulative Job Walltime as seen from Submit Side	20 secs
Workflow Retries	0

Workflow Statistics

This Workflow

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

Entire Workflow

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

Job Breakdown Statistics

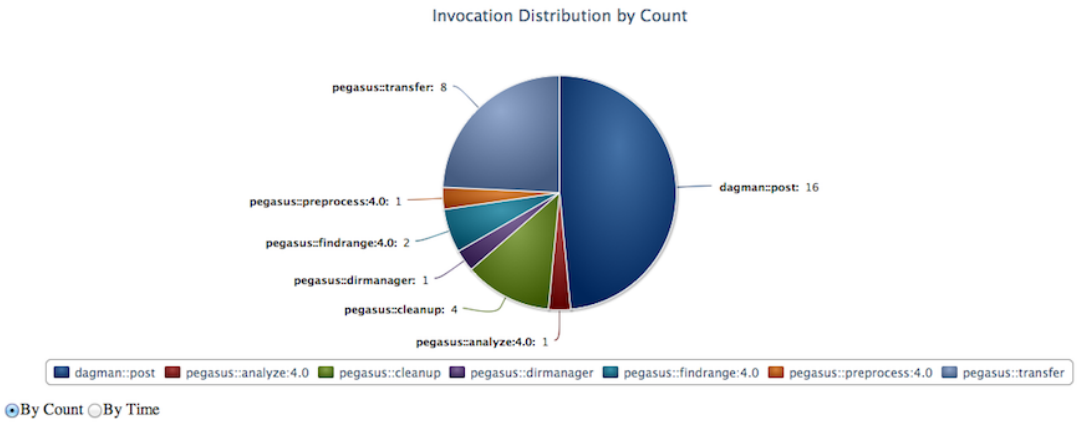
Job Statistics

The Charts page shows the following charts.

1. Job Distribution by Count/Time
2. Time Chart by Job/Invocation
3. Workflow Execution Gantt Chart

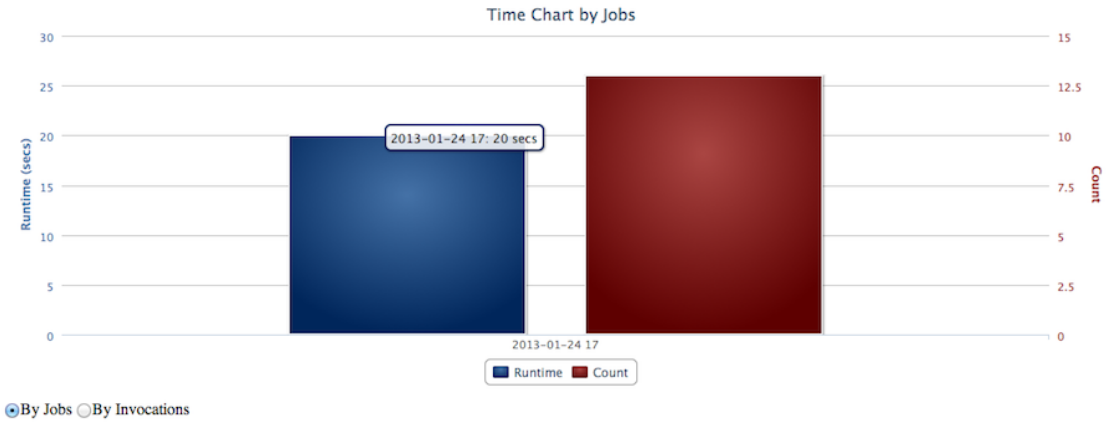
The chart below shows the invocation distribution by count or time.

Figure 8.13. Dashboard Plots - Job Distribution



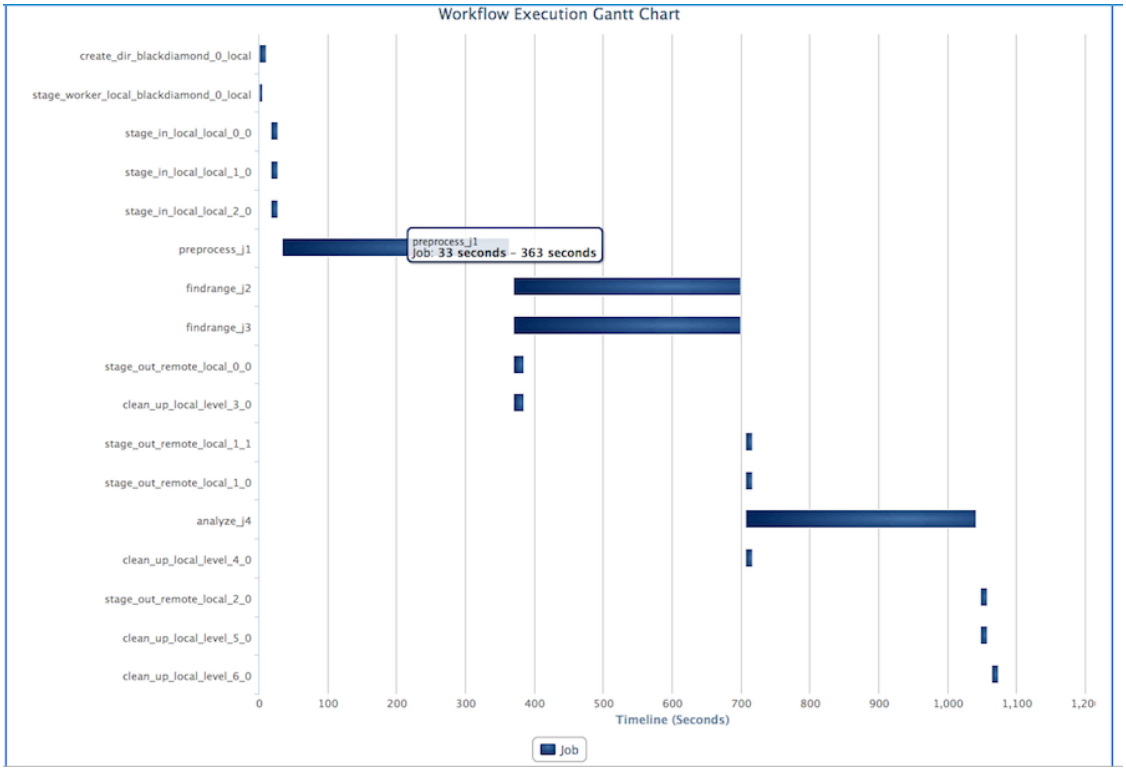
The time chart shown below shows the number of jobs/invocations in the workflow and their total runtime

Figure 8.14. Dashboard Plots - Time Chart



The workflow gantt chart lays out the execution of the jobs in the workflow over time.

Figure 8.15. Dashboard Plots - Workflow Gantt Chart



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

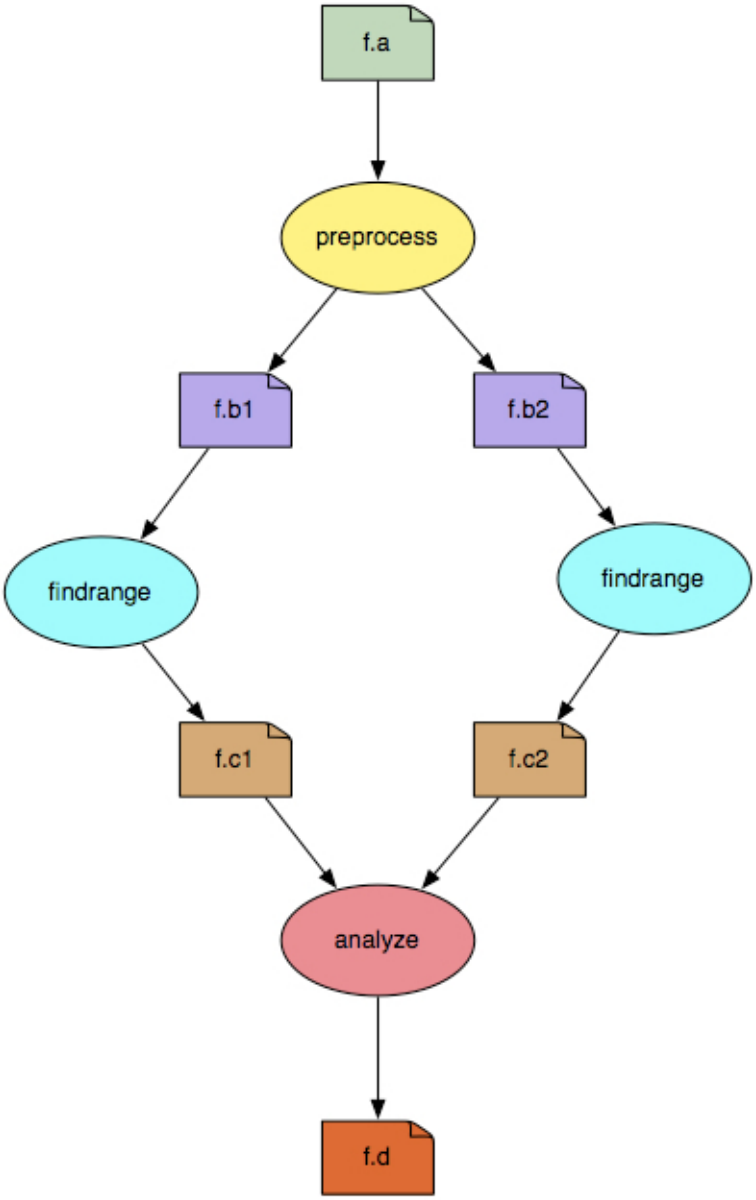
These examples assumes you have access to a cluster with Globus installed. A pre-ws gatekeeper and gridftp server is required. You also need Globus and Pegasus installed, both on the machine you are submitting from, and the cluster.

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
```

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. Reference Manual

Properties

This is the reference guide to all properties regarding the Pegasus Workflow Planner, and their respective default values. Please refer to the user guide for a discussion when and which properties to use to configure various components. 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.

Local Directories

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.

pegasus.home.datadir

Systems:	all
Type:	directory location string
Default:	${pegasus.home}/share

The datadir directory contains broadly visiable and possilby exported configuration files that rarely change. This directory is currently unused.

pegasus.home.sysconfdir

Systems:	all
Type:	directory location string
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.

pegasus.home.sharedstatedir

Systems:	all
Type:	directory location string
Default:	${pegasus.home}/com

Frequently changing files that are broadly visible are stored in the shared state directory. This is currently unused.

pegasus.home.localstatedir

Systems:	all
Type:	directory location string
Default:	${pegasus.home}/var

Frequently changing files that are specific to a machine and/or installation are stored in the local state directory. This directory is being used for the textual transformation catalog, and the file-based replica catalog.

pegasus.dir.submit.logs

System:	Pegasus

Since:	2.4
Type:	directory location string
Default:	false

By default, Pegasus points the condor logs for the workflow to /tmp directory. This is done to ensure that the logs are created in a local directory even though the submit directory maybe on NFS. In the submit directory the symbolic link to the appropriate log file in the /tmp exists.

However, since /tmp is automatically purged in most cases, users may want to preserve their condor logs in a directory on the local filesystem other than /tmp

Site Directories

The site directory properties modify the behavior of remotely run jobs. In rare occasions, it may also pertain to locally run compute jobs.

pegasus.dir.useTimestamp

System:	Pegasus
Since:	2.1
Type:	Boolean
Default:	false

While creating the submit directory, Pegasus employs a run numbering scheme. Users can use this property to use a timestamp based numbering scheme instead of the runxxxx scheme.

pegasus.dir.exec

System:	Pegasus
Since:	2.0
Type:	remote directory location string
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.

pegasus.dir.storage.mapper

System:	Pegasus
Since:	4.3
Type:	enumeration
Value[0]:	Flat
Value[1]:	Hashed
Value[2]:	Replica
Default:	Flat
See Also:	pegasus.dir.storage.deep

This property modifies determines how the output files are mapped on the output site storage location.

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.

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

```
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
```

pegasus.dir.storage.deep

System:	Pegasus
Since:	2.1
Type:	Boolean
Default:	false
See Also:	pegasus.dir.storage.mapper

This 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.

pegasus.dir.create.strategy

System:	Pegasus
Since:	2.2
Type:	enumeration
Value[0]:	HourGlass
Value[1]:	Tentacles
Value[2]:	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.

pegasus.dir.create.impl

System:	Pegasus
Since:	2.2
Type:	enumeration
Value[0]:	DefaultImplementation
Value[1]:	S3
Default:	DefaultImpelmentation

This property is used to select the executable that is used to create the working directory on the compute sites.

DefaultImplementation — The default executable that is used to create a directory is the dirmanager executable shipped with Pegasus. It is found at $PEGASUS_HOME/bin/dirmanager 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.

S3 — This option is used to create buckets in S3 instead of a directory. This should be set when running workflows on Amazon EC2. This implementation relies on s3cmd command line client to create the bucket. An entry for transformation amazon::s3cmd needs to exist in the Transformation Catalog for this to work.

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.

pegasus.schema.dax

Systems:	Pegasus
Since:	2.0
Type:	XML schema file location string
Value[0]:	${pegasus.home.sysconfdir}/dax-3.2.xsd

Default:	${pegasus.home.sysconfdir}/dax-3.2.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 GriPhyN website dynamically.

pegasus.schema.sc

Systems:	Pegasus
Since:	2.0
Type:	XML schema file location string
Value[0]:	${pegasus.home.sysconfdir}/sc-3.0.xsd
Default:	${pegasus.home.sysconfdir}/sc-3.0.xsd

This file is a copy of the XML schema that describes the xml description of the site catalog, that is generated as a result of using genpoolconfig command. 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.

pegasus.schema.ivr

Systems:	all
Type:	XML schema file location string
Value[0]:	${pegasus.home.sysconfdir}/iv-2.0.xsd
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 GriPhyN website dynamically.

Database Drivers For All Relational Catalogs

pegasus.catalog.*.db.driver

System:	Pegasus
Type:	Java class name
Value[0]:	Postgres
Value[1]:	MySQL
Value[2]:	SQLServer2000 (not yet implemented!)
Value[3]:	Oracle (not yet implemented!)
Default:	(no default)
See also:	pegasus.catalog.provenance

The database driver class is dynamically loaded, as required by the schema. Currently, only PostGreSQL 7.3 and MySQL 4.0 are supported. Their respective JDBC3 driver is provided as part and parcel of the PEGASUS.

A user may provide their own implementation, derived from org.griphyn.vdl.dbdriver.DatabaseDriver, to talk to a database of their choice.

For each schema in PTC, a driver is instantiated separately, which has the same prefix as the schema. This may result in multiple connections to the database backend. As fallback, the schema "*" driver is attempted.

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
provenance
```

pegasus.catalog.*.db.url

System:	PTC, ...
Type:	JDBC database URI string
Default:	(no default)
Example:	jdbc:postgresql:${user.name}

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.

```
Postgres : jdbc:postgresql:[//hostname[:port]/]database
MySQL    : jdbc:mysql://hostname[:port]]/database
SQLServer: jdbc:microsoft:sqlserver://hostname:port
Oracle   : jdbc:oracle:thin:[user/password]@//host[:port]/service
```

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
provenance
```

pegasus.catalog.*.db.user

System:	PTC, ...
Type:	string
Default:	(no default)
Example:	${user.name}

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
provenance
```

pegasus.catalog.*.db.password

System:	PTC, ...
Type:	string
Default:	(no default)
Example:	${user.name}

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 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
provenance
```

pegasus.catalog.*.db.*

System:	PTC, RC

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 7.3 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 4.0 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
provenance
```

Catalog Properties

Replica Catalog

pegasus.catalog.replica

System:	Pegasus
Since:	2.0
Type:	enumeration
Value[0]:	RLS
Value[1]:	LRC
Value[2]:	JDBCRC
Value[3]:	File
Value[4]:	Directory
Value[5]:	MRC
Value[6]:	Regex
Default:	RLS

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.

RLS
: RLS (Replica Location Service) is a distributed replica catalog, which ships with GT4. There is an index service called Replica Location Index (RLI) to which 1 or more Local Replica Catalog (LRC) report. Each LRC can contain all or a subset of mappings. In this mode, Pegasus queries the central RLI to discover in which LRC's the mappings for a LFN reside. It then queries the individual LRC's for the PFN's. To use RLS, the user additionally needs to set the property pegasus.catalog.replica.url to specify the URL for the RLI to query. Details about RLS can be found at http://www.globus.org/toolkit/data/rls/

LRC
: If the user does not want to query the RLI, but directly a single Local Replica Catalog. To use LRC, the user additionally needs to set the property pegasus.catalog.replica.url to specify the URL for the LRC to query. Details about RLS can be found at http://www.globus.org/toolkit/data/rls/

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.url

2. pegasus.catalog.replica.db.user

3. pegasus.catalog.replica.db.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 "pool".

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 "pool".

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:///Volumes/data/input/f.a pool="local"
f.a file:///Volumes/data/input/f.a pool="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:///Volumes/data/input/[1]/[0] pool="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://$input/f.1  pool="local"
f.2 file://$input/f.2  pool="local"
D1/f.3 file://$input/D2/f.3 pool="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://$input/f.1  pool="local"
f.2 file://$input/f.2  pool="local"
f.3 file://$input/D2/f.3 pool="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
```

For example, if a user wants to query two lrc's at the same time he/she can specify as follows

```
pegasus.catalog.replica.mrc.lrc1 LRC
pegasus.catalog.replica.mrc.lrc2.url rls://sukhna
pegasus.catalog.replica.mrc.lrc2 LRC
pegasus.catalog.replica.mrc.lrc2.url rls://smarty
```

In the above example, lrc1, lrc2 are any valid identifier names and url is the property key that needed to be specified.

pegasus.catalog.replica.url

System:	Pegasus

Since:	2.0
Type:	URI string
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.

pegasus.catalog.replica.chunk.size

System:	Pegasus, rc-client
Since:	2.0
Type:	Integer
Default:	1000

The 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.

pegasus.catalog.replica.lrc.ignore

System:	Replica Catalog - RLS
Since:	2.0
Type:	comma separated list of LRC urls
Default:	(no default)
See also:	pegasus.catalog.replica.lrc.restrict

Certain users may like to skip some LRCs while querying for the physical locations of a file. If some LRCs need to be skipped from those found in the rli then use this property. You can define either the full URL or partial domain names that need to be skipped. E.g. If a user wants rls://smarty.isi.edu and all LRCs on usc.edu to be skipped then the property will be set as pegasus.rls.lrc.ignore=rls://smarty.isi.edu,usc.edu

pegasus.catalog.replica.lrc.restrict

System:	Replica Catalog - RLS
Since:	1.3.9
Type:	comma separated list of LRC urls
Default:	(no default)
See also:	pegasus.catalog.replica.lrc.ignore

This property applies a tighter restriction on the results returned from the LRCs specified. Only those PFNs are returned that have a pool attribute associated with them. The property "pegasus.rc.lrc.ignore" has a higher priority than "pegasus.rc.lrc.restrict". For example, in case a LRC is specified in both properties, the LRC would be ignored (i.e. not queried at all instead of applying a tighter restriction on the results returned).

pegasus.catalog.replica.lrc.site.[site-name]

System:	Replica Catalog - RLS
Since:	2.3.0
Type:	LRC url
Default:	(no default)

This property allows for the LRC url to be associated with site handles. Usually, a pool attribute is required to be associated with the PFN for Pegasus to figure out the site on which PFN resides. However, in the case where an LRC is responsible for only a single site's mappings, Pegasus can safely associate LRC url with the site. This association can be used to determine the pool attribute for all mappings returned from the LRC, if the mapping does not have a pool attribute associated with it.

The site_name in the property should be replaced by the name of the site. For example

```
pegasus.catalog.replica.lrc.site.isi  rls://lrc.isi.edu
```

tells Pegasus that all PFNs returned from LRC rls://lrc.isi.edu are associated with site isi.

The [site_name] should be the same as the site handle specified in the site catalog.

pegasus.catalog.replica.cache.asrc

System:	Pegasus
Since:	2.0
Type:	Boolean
Value[0]:	false
Value[1]:	true
Default:	false
See also:	pegasus.catalog.replica

This 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.

Site Catalog

pegasus.catalog.site

System:	Site Catalog
Since:	2.0
Type:	enumeration
Value[0]:	XML4
Value[1]:	XML3
Default:	XML4

The site catalog file format is now automatically detected, so there should be no need to use the property anymore.

pegasus.catalog.site.file

System:	Site Catalog

Since:	2.0
Type:	file location string
Default:	${pegasus.home.sysconfdir}/sites.xml
See also:	pegasus.catalog.site

Running things on the grid requires an extensive description of the capabilities of each compute cluster, commonly termed "site". This property describes the location of the file that contains such a site description. As the format is currently in flow, please refer to the userguide and Pegasus for details which format is expected.

Transformation Catalog

pegasus.catalog.transformation

System:	Transformation Catalog
Since:	2.0
Type:	enumeration
Value[0]:	Text
Value[1]:	File
Default:	Text
See also:	pegasus.catalog.transformation.file

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

```
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"
```

File THIS FORMAT IS DEPRECATED. WILL BE REMOVED IN COMING VERSIONS. USE pegasus-tc-converter to convert File format to Text Format. In this mode, a file 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 new TC file format uses 6 columns:

1. The resource ID is represented in the first column.
2. The logical transformation uses the colonized format ns::name:vs.
3. The path to the application on the system
4. The installation type is identified by one of the following keywords - all upper case: INSTALLED, STAGEABLE. If not specified, or **NULL** is used, the type defaults to INSTALLED.
5. The system is of the format ARCH::OS[:VER:GLIBC]. The following arch types are understood: "INTEL32", "INTEL64", "SPARCV7", "SPARCV9". The following os types are understood: "LINUX", "SUNOS", "AIX". If unset or **NULL**, defaults to INTEL32::LINUX.
6. Profiles are written in the format NS::KEY=VALUE,KEY2=VALUE;NS2::KEY3=VALUE3 Multiple key-values for same namespace are seperated by a comma "," and multiple namespaces are seperated by a semicolon ";". If any of your profile values contains a comma you must not use the namespace abbreviator.

pegasus.catalog.transformation.file

Systems:	Transformation Catalog
Type:	file location string
Default:	${pegasus.home.sysconfdir}/tc.text | ${pegasus.home.sysconfdir}/tc.data
See also:	pegasus.catalog.transformation

This property is used to set the path to the textual transformation catalogs of type File or Text. If the transformation catalog is of type Text then tc.text file is picked up from sysconfdir, else tc.data

Provenance Catalog

pegasus.catalog.provenance

System:	Provenance Tracking Catalog (PTC)
Since:	2.0
Type:	Java class name
Value[0]:	InvocationSchema
Value[1]:	NXDInvSchema
Default:	(no default)
See also:	pegasus.catalog.*.db.driver

This property denotes the schema that is being used to access a PTC. The PTC is usually not a standard installation. If you use a database backend, you most likely have a schema that supports PTCs. By default, no PTC will be used.

Currently only the InvocationSchema is available for storing the provenance tracking records. Beware, this can become a lot of data. The values are names of Java classes. If no absolute Java classname is given, "org.griphyn.vdl.dbschema." is prepended. Thus, by deriving from the DatabaseSchema API, and implementing the PTC interface, users can provide their own classes here.

Alternatively, if you use a native XML database like eXist, you can store data using the NXDInvSchema. This will avoid using any of the other database driver properties.

pegasus.catalog.provenance.refinement

System:	PASOA Provenance Store

Since:	2.0.1
Type:	Java class name
Value[0]:	Pasoa
Value[1]:	InMemory
Default:	InMemory
See also:	pegasus.catalog.*.db.driver

This property turns on the logging of the refinement process that happens inside Pegasus to the PASOA store. Not all actions are currently captured. It is still an experimental feature.

The PASOA store needs to run on localhost on port 8080 https://localhost:8080/prserv-1.0

Replica Selection Properties

pegasus.selector.replica

System:	Replica Selection
Since:	2.0
Type:	URI string
Default:	default
See also:	pegasus.replica.*.ignore.stagein.sites
See also:	pegasus.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.

Default
: If a PFN that is a file URL (starting with file:///) and has a pool attribute matching to the site handle of the site where the compute is to be run is found, then that is returned. Else,a random PFN is selected amongst all the PFN's that have a pool attribute matching to the site handle of the site where a compute job is to be run. Else, a random pfn is selected amongst all the PFN's.

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.

Regex — This replica selector allows the user allows the user to specific regex expressions that can be used to rank various PFN's returned from the Replica Catalog for a particular LFN. This replica selector selects the highest ranked PFN i.e the replica with the lowest rank value.

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 is an integer value that denotes the rank of an expression with a rank value of 1 being the highest rank.

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.

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.

pegasus.selector.replica.*.ignore.stagein.sites

System:	Replica Selection
Type:	comma separated list of sites
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.

pegasus.selector.replica.*.prefer.stagein.sites

System:	Replica Selection
Type:	comma separated list of sites
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.

pegasus.selector.replica.regex.rank.[value]

System:	Replica Selection
Type:	Regex Expression
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

pegasus.selector.site

System:	Pegasus
Since:	2.0
Type:	enumeration
Value[0]:	Random
Value[1]:	RoundRobin
Value[2]:	NonJavaCallout
Value[3]:	Group
Value[4]:	Heft
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 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 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 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.

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 communica-

tion 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.

pegasus.selector.site.path

System:	Site Selector
Since:	2.0
Type:	String

If one calls out to an external site selector using the NonJavaCallout 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.

pegasus.site.selector.env.*

System:	Pegasus
Since:	1.2.3
Type:	String

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.

pegasus.selector.site.timeout

System:	Site Selector
Since:	2.0
Type:	non negative integer
Default:	60

It sets the number of seconds Pegasus waits to hear back from an external site selector using the NonJavaCallout interface before timing out.

pegasus.selector.site.keep.tmp

System:	Pegasus
Since:	2.0
Type:	enumeration
Value[0]:	onerror
Value[1]:	always
Value[2]:	never
Default:	onerror

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

pegasus.data.configuration

System:	Pegasus
Since:	4.0
Type:	enumeration
Value[0]:	sharedfs
Value[1]:	nonsharedfs
Value[2]:	condorio
Default:	sharedfs

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 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. Internally, if this is set the following properties are set.

```
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 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. On loading this property, internally the following properies are set

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

pegasus.transfer.bypass.input.staging

System:	Pegasus
Since:	4.3
Type:	Boolean
Default:	(no default)
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

pegasus.transfer.*.impl

System:	Pegasus
Type:	enumeration
Value[0]:	Transfer
Value[1]:	GUC
Default:	Transfer
See also:	pegasus.transfer.refiner
Since:	2.0

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 pool 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.

pegasus.transfer.refiner

System:	Pegasus
Type:	enumeration
Value[0]:	Basic
Value[1]:	Cluster
Default:	Cluster
Since:	2.0
See also:	pegasus.transfer.*.impl

This property determines how the transfer nodes are added to the workflow. The various refiners differ in the how they link the various transfer jobs, and the number of transfer jobs that are created per compute jobs.

Basic
: This is a basic refinement strategy that adds a stage-in job per compute job and a stage-out per compute jobs. It is not recommended to use this , especially for large workflows where lots of stage-in jobs maybe created for a workflow. This is only recommended for experimental setups.

Cluster
: In this refinement strategy, clusters of stage-in and stageout jobs are created per level of the workflow. This workflow allows you to control the number of stagein and stageout jobs by associating pegasus profiles stagein.clusters and stageout.clusters with the jobs or in the site catalog for the staging sites.

pegasus.transfer.sls.*.impl

System:	Pegasus

Type:	enumeration
Value[0]:	Transfer
Value[1]:	Condor
Default:	Transfer
Since:	2.2.0
See also:	pegasus.data.configuration
See also:	pegasus.execute.*.filesystem.local

This property specifies the transfer tool to be used for Second Level Staging (SLS) of input and output data between the head node and worker node filesystems.

Currently, the * in the property name CANNOT be replaced to achieve finer grained control to dictate what type of SLS transfers need to be managed with which grid transfer tool.

The various grid transfer tools that can be used to manage SLS 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

Condor
: This results in Condor file transfer mechanism to be used to transfer the input data files from the submit host directly to the worker node directories. This is used when running in pure Condor mode or in a Condor pool that does not have a shared filesystem between the nodes.

  When setting the SLS transfers to Condor make sure that the following properties are also set

```
pegasus.gridstart          PegasusLite
pegasus.execute.*.filesystem.local  true
```

  Alternatively, you can set

```
pegasus.data.configuration          condorio
```

  in lieu of the above 3 properties.

  Also make sure that pegasus.gridstart is not set.

  Please refer to the section on "Condor Pool Without a Shared Filesystem" in the chapter on Planning and Submitting.

pegasus.transfer.arguments

System:	Pegasus
Since:	2.0
Type:	String
Default:	(no default)
See also:	pegasus.transfer.sls.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.

pegasus.transfer.sls.arguments

System:	Pegasus
Since:	2.4
Type:	String
Default:	(no default)
See also:	pegasus.transfer.arguments
See also:	pegasus.transfer.sls.*.impl

This determines the extra arguments with which the SLS transfer implementation is invoked. The transfer executable that is invoked is dependant upon the SLS transfer implementation that has been selected.

pegasus.transfer.stage.sls.file

System:	Pegasus
Since:	3.0
Type:	Boolean
Default:	(no default)
See also:	pegasus.gridstart
See also:	pegasus.execute.*.filesystem.local

For executing jobs on the local filesystem, Pegasus creates sls files for each compute jobs. These sls files list the files that need to be staged to the worker node and the output files that need to be pushed out from the worker node after completion of the job. By default, pegasus will stage these SLS files to the shared filesystem on the head node as part of first level data stagein jobs. However, in the case where there is no shared filesystem between head nodes and the worker nodes, the user can set this property to false. This will result in the sls files to be transferred using the Condor File Transfer from the submit host.

pegasus.transfer.worker.package

System:	Pegasus
Type:	boolean
Default:	false
Since:	3.0
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.

pegasus.transfer.links

System:	Pegasus
Type:	boolean
Default:	false
Since:	2.0
See also:	pegasus.transfer
See also:	pegasus.transfer.force

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 pool attribute associated with the PFN that matches the execution pool 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 pools 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 pool.

pegasus.transfer.*.remote.sites

System:	Pegasus
Type:	comma separated list of sites
Default:	no default
Since:	2.0

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 pool 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.

pegasus.transfer.staging.delimiter

System:	Pegasus
Since:	2.0
Type:	String
Default:	:

See also:	pegasus.transformation.selector

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.

pegasus.transfer.disable.chmod.sites

System:	Pegasus
Since:	2.0
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.

pegasus.transfer.setup.source.base.url

System:	Pegasus
Type:	URL
Default:	no default
Since:	2.3

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.

Gridstart And Exitcode Properties

pegasus.gridstart

System:	Pegasus
Since:	2.0
Type:	enumeration
Value[0]:	Kickstart
Value[1]:	None
Value[2]:	PegasusLite
Default:	Kickstart
See also:	pegasus.execute.*.filesystem.local

Jobs that are launched on the grid maybe wrapped in a wrapper executable/script that enables information about about the execution, resource consumption, and - most importantly - the exitcode of the remote application. At present, a

job scheduled on a remote site is launched with a gridstart if site catalog has the corresponding gridlaunch attribute set and the job being launched is not MPI.

Users can explicitly decide what gridstart to use for a job, by associating the pegasus profile key named gridstart with the job.

Kickstart	In this mode, all the jobs are lauched via kickstart. The kickstart executable is a light-weight program which connects the stdin,stdout and stderr filehandles for PEGASUS jobs on the remote site. Kickstart is an executable distributed with PEGASUS that can generally be found at ${pegasus.home.bin}/kickstart.
None	In this mode, all the jobs are launched directly on the remote site. Each job's stdin,stdout and stderr are connected to condor commands in a manner to ensure that they are sent back to the submit host.
PegasusLite	In this mode, the compute jobs are wrapped by PegasusLite instances. PegasusLite instance is a bash script, that is launced on the compute node. It determins at runtime the directory a job needs to execute in, pulls in data from the staging site , launches the job, pushes out the data and cleans up the directory after execution.

pegasus.gridstart.kickstart.set.xbit

System:	Pegasus
Since:	2.4
Type:	Boolean
Default:	false
See also:	pegasus.transfer.disable.chmod.sites

Kickstart has an option to set the X bit on an executable before it launches it on the remote site. In case of staging of executables, by default chmod jobs are launched that set the x bit of the user executables staged to a remote site.

On setting this property to true, kickstart gridstart module adds a -X option to kickstart arguments. The -X arguments tells kickstart to set the x bit of the executable before launching it.

User should usually disable the chmod jobs by setting the property pegasus.transfer.disable.chmod.sites , if they set this property to true.

pegasus.gridstart.kickstart.stat

System:	Pegasus
Since:	2.1
Type:	Boolean
Default:	false
See also:	pegasus.gridstart.generate.lof

Kickstart has an option to stat the input files and the output files. The stat information is collected in the XML record generated by kickstart. Since stat is an expensive operation, it is not turned on by on. Set this property to true if you want to see stat information for the input files and output files of a job in it's kickstart output.

pegasus.gridstart.generate.lof

System:	Pegasus
Since:	2.1
Type:	Boolean
Default:	false

See also:	pegasus.gridstart.kickstart.stat

For the stat option for kickstart, we generate 2 lof (list of filenames) files for each job. One lof file containing the input lfn's for the job, and the other containing output lfn's for the job. In some cases, it maybe beneficial to have these lof files generated but not do the actual stat. This property allows you to generate the lof files without triggering the stat in kickstart invocations.

pegasus.gridstart.invoke.always

System:	Pegasus
Since:	2.0
Type:	Boolean
Default:	false
See also:	pegasus.gridstart.invoke.length

Condor has a limit in it, that restricts the length of arguments to an executable to 4K. To get around this limit, you can trigger Kickstart to be invoked with the -I option. In this case, an arguments file is prepared per job that is transferred to the remote end via the Condor file transfer mechanism. This way the arguments to the executable are not specified in the condor submit file for the job. This property specifies whether you want to use the invoke option always for all jobs, or want it to be triggered only when the argument string is determined to be greater than a certain limit.

pegasus.gridstart.invoke.length

System:	Pegasus
Since:	2.0
Type:	Long
Default:	4000
See also:	pegasus.gridstart.invoke.always

Gridstart is automatically invoked with the -I option, if it is determined that the length of the arguments to be specified is going to be greater than a certain limit. By default this limit is set to 4K. However, it can overriden by specifying this property.

Interface To Condor And Condor Dagman

The Condor DAGMan facility is usually activate using the condor_submit_dag command. However, many shapes of workflows have the ability to either overburden the submit host, or overflow remote gatekeeper hosts. While DAGMan provides throttles, unfortunately these can only be supplied on the command-line. Thus,PEGASUS provides a versatile wrapper to invoke DAGMan, called pegasus-submit-dag. It can be configured from the command-line, from user- and system properties, and by defaults.

pegasus.condor.logs.symlink

System:	Condor
Type:	Boolean
Default:	true
Since:	3.0

By default pegasus has the Condor common log [dagname]-0.log in the submit file as a symlink to a location in /tmp . This is to ensure that condor common log does not get written to a shared filesystem. If the user knows for sure that the workflow submit directory is not on the shared filesystem, then they can opt to turn of the symlinking of condor common log file by setting this property to false.

pegasus.condor.arguments.quote

System:	Condor
Type:	Boolean
Default:	true
Since:	2.0
Old Name:	pegasus.condor.arguments.quote

This property determines whether to apply the new Condor quoting rules for quoting the argument string. The new argument quoting rules appeared in Condor 6.7.xx series. We have verified it for 6.7.19 version. If you are using an old condor at the submit host, set this property to false.

pegasus.dagman.notify

System:	DAGman wrapper
Type:	Case-insensitive enumeration
Value[0]:	Complete
Value[1]:	Error
Value[2]:	Never
Default:	Never
Document:	http://www.cs.wisc.edu/condor/manual/v6.9/condor_submit_dag.html
Document:	http://www.cs.wisc.edu/condor/manual/v6.9/condor_submit.html

The pegasus.dagman.nofity property has been deprecated in favor of the Pegasus notification framework. Please see the reference manual for details on how to get workflow notifications. pegasus.dagman.nofity will be removed in the an upcoming version of Pegasus.

pegasus.dagman.verbose

System:	DAGman wrapper
Type:	Boolean
Value[0]:	false
Value[1]:	true
Default:	false
Document:	http://www.cs.wisc.edu/condor/manual/v6.9/condor_submit_dag.html

The pegasus-submit-dag wrapper processes properties to set DAGMan commandline arguments. If set and true, the argument activates verbose output in case of DAGMan errors.

pegasus.dagman.[category].maxjobs

System:	DAGman wrapper
Type:	Integer
Since:	2.2
Default:	no default

Document:	http://vtcpc.isi.edu/pegasus/index.php/ChangeL-og\#Support_for_DAGMan_node_categories

DAGMan now allows for the nodes in the DAG to be grouped in category. The tuning parameters like maxjobs then can be applied per category instead of being applied to the whole workflow. To use this facility users need to associate the dagman profile key named category with their jobs. The value of the key is the category to which the job belongs to.

You can then use this property to specify the value for a category. For the above example you will set pegasus.dagman.short-running.maxjobs

Monitoring Properties

pegasus.monitord.events

System:	Pegasus-monitord
Type:	Boolean
Default:	true
Since:	3.0.2
See Also:	pegasus.monitord.output

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.

pegasus.monitord.output

System:	Pegasus-monitord
Type:	String
Since:	3.0.2
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

SQL Alchemy End Point	Example Value
Netlogger BP File	file:///submit/dir/myworkflow.bp
SQL Lite Database	sqlite:///submit/dir/myworkflow.db
MySQL Database	mysql://user:password@host:port/databasename

pegasus.dashboard.output

System:	Pegasus-monitord
Type:	String
Since:	4.2
See Also:	pegasus.monitord.output

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/databasename

pegasus.monitord.notifications

System:	Pegasus-monitord
Type:	Boolean
Default:	true
Since:	3.1
See Also:	pegasus.monitord.notifications.max
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.

pegasus.monitord.notifications.max

System:	Pegasus-monitord
Type:	Integer
Default:	10
Since:	3.1
See Also:	pegasus.monitord.notifications
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.

pegasus.monitord.notifications.timeout

System:	Pegasus-monitord
Type:	Integer
Default:	0
Since:	3.1
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.

pegasus.monitord.stdout.disable.parsing

System:	Pegasus-monitord
Type:	Boolean
Default:	False
Since:	3.1.1

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.

Job Clustering Properties

pegasus.clusterer.job.aggregator

System:	Job Clustering
Since:	2.0
Type:	String
Value[0]:	seqexec
Value[1]:	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 seqexec that runs each of the smaller jobs sequentially on the same node. The executable "seqexec" 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 mpiexec that runs the smaller jobs via mpi on n nodes where n is the nodecount associated with the merged job. The executable "mpiexec" is a PEGASUS

tool distributed in the PEGASUS worker package, and can be usually found at {pegasus.home}/bin/mpiexec.

pegasus.clusterer.job.aggregator.seqexec.log

System:	Job Clustering
Type:	Boolean
Default:	false
Since:	2.3
See also:	pegasus.clusterer.job.aggregator
See also:	pegasus.clusterer.job.aggregator.seqexec.log.global

Seqexec logs the progress of the jobs that are being run by it in a progress file on the remote cluster where it is executed.

This property sets the Boolean flag, that indicates whether to turn on the logging or not.

pegasus.clusterer.job.aggregator.seqexec.log.global

System:	Job Clustering
Type:	Boolean
Default:	true
Since:	2.3
See also:	pegasus.clusterer.job.aggregator
See also:	pegasus.clusterer.job.aggregator.seqexec.log
Old Name:	pegasus.clusterer.job.aggregator.seqexec.hasgloballog

Seqexec 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 seqexec 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 seqexec jobs on a particular cluster or progress log per job.

pegasus.clusterer.job.aggregator.seqexec.firstjobfail

System:	Job Clustering
Type:	Boolean
Default:	true
Since:	2.2
See also:	pegasus.clusterer.job.aggregator

By default seqexec does not stop execution even if one of the clustered jobs it is executing fails. This is because seqexec tries to get as much work done as possible.

This property sets the Boolean flag, that indicates whether to make seqexec stop on the first job failure it detects.

pegasus.clusterer.label.key

System:	Job Clustering

Type:	String
Default:	label
Since:	2.0
See also:	pegasus.partitioner.label.key

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

pegasus.log.manager

System:	Pegasus
Since:	2.2.0
Type:	Enumeration
Value[0]:	Default
Value[1]:	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
```

pegasus.log.manager.formatter

System:	Pegasus
Since:	2.2.0
Type:	Enumeration
Value[0]:	Simple
Value[1]:	Netlogger
Default:	Simple
See also:	pegasus.log.manager.formatter

This property sets the formatter to use for formatting the log messages while logging.

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
```

pegasus.log.*

System:	Pegasus
Since:	2.0
Type:	String
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.

pegasus.log.metrics

System:	Pegasus
Since:	2.1.0
Type:	Boolean
Default:	true
See also:	pegasus.log.metrics.file

This property enables the logging of certain planning and workflow metrics to a global log file. By default the file to which the metrics are logged is ${pegasus.home}/var/pegasus.log.

pegasus.log.metrics.file

System:	Pegasus
Since:	2.1.0
Type:	Boolean

Default:	${pegasus.home}/var/pegasus.log
See also:	pegasus.log.metrics

This property determines the file to which the workflow and planning metrics are logged if enabled.

pegasus.metrics.app

System:	Pegasus
Since:	4.3.0
Type:	String
See also:	pegasus.log.metrics

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

Miscellaneous Properties

pegasus.code.generator

System:	Pegasus
Since:	3.0
Type:	enumeration
Value[0]:	Condor
Value[1]:	Shell
Value[2]:	PMC
Default:	Condor

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.

pegasus.register

System:	Pegasus
Since:	4.1.0

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.

pegasus.job.priority.assign

System:	Pegasus
Since:	3.0.3
Type:	Boolean
Default:	true

This property can be used to turn off the default level based condor priorities that are assigned to jobs in the executable workflow.

pegasus.file.cleanup.strategy

System:	Pegasus
Since:	2.2
Type:	enumeration
Value[0]:	InPlace
Default:	InPlace

This property is used to select the strategy of how the the cleanup nodes are added to the executable workflow.

InPlace This is the only mode available .

pegasus.file.cleanup.impl

System:	Pegasus
Since:	2.2
Type:	enumeration
Value[0]:	Cleanup
Value[1]:	RM
Value[2]:	S3
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 dirmanager executable shipped with Pegasus. It is found at $PEGASUS_HOME/bin/dirmanager 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.

S3 This mode is used to delete files/objects from the buckets in S3 instead of a directory. This should be set when running workflows on Amazon EC2. This implementation relies on s3cmd command line client to create the bucket. An entry for transformation amazon::s3cmd needs to exist in the Transformation Catalog for this to work.

pegasus.file.cleanup.clusters.num

System:	Pegasus
Since:	4.2
Type:	Integer
Default:	2

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.

pegasus.file.cleanup.clusters.size

System:	Pegasus
Since:	4.2.1
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 parameters is only used if pegasus.file.cleanup.clusters.num is not set.

pegasus.file.cleanup.scope

System:	Pegasus
Since:	2.3.0
Type:	enumeration
Value[0]:	fullahead
Value[1]:	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 --nocleanup option to pegasus-plan.

pegasus.catalog.transformation.mapper

System:	Staging of Executables
Since:	2.0

Type:	enumeration
Value[0]:	All
Value[1]:	Installed
Value[2]:	Staged
Value[3]:	Submit
Default:	All
See also:	pegasus.transformation.selector

Pegasus now supports transfer of statically linked executables as part of the concrete 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 submit host (pool 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.

pegasus.selector.transformation

System:	Staging of Executables
Since:	2.0
Type:	enumeration
Value[0]:	Random
Value[1]:	Installed
Value[2]:	Staged
Value[3]:	Submit
Default:	Random
See also:	pegasus.catalog.transformation

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 pool, 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 pool.
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 pools.
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 (pool local), are selected as sources for staging the executables to the remote execution pools.

pegasus.execute.*.filesystem.local

System:	Pegasus
Type:	Boolean
Default:	false
Since:	2.1.0
See also:	pegasus.data.configuration

Normally, Pegasus transfers the data to and from a directory on the shared filesystem on the head node of a compute site. The directory needs to be visible to both the head node and the worker nodes for the compute jobs to execute correctly.

By setting this property to true, you can get Pegasus to execute jobs on the worker node filesystem. In this case, when the jobs are launched on the worker nodes, the jobs grab the input data from the workflow specific execution directory on the compute site and push the output data to the same directory after completion. The transfer of data to and from the worker node directory is referred to as Second Level Staging (SLS).

pegasus.parser.dax.preserver.linebreaks

System:	Pegasus
Type:	Boolean
Default:	false
Since:	2.2.0

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.

Profiles

The Pegasus Workflow Mapper uses the concept of profiles to encapsulate configurations for various aspects of dealing with the Grid infrastructure. Profiles 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.

This document describes various types of profiles, levels of priorities for intersecting profiles, and how to specify profiles in different contexts.

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.

Profile Namespaces

Each namespace refers to a different aspect of a job’s runtime settings. A profile’s representation in the concrete plan (e.g. the Condor submit files) depends 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.

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 10.1. Table 1: Useful Environment Settings

Environment Variable	Description
PEGASUS_HOME	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
GLOBUS_LOCATION	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
LD_LIBRARY_PATH	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 RLS 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

Table 2 shows some commonly used RSL instructions. For an authoritative list of all possible RSL instructions refer to the Globus RSL specification.

Table 10.2. Table 2: Useful Globus RSL Instructions

Key	Description
count	the number of times an executable is started.
jobtype	specifies how the job manager should start the remote job. While Pegasus defaults to single, use mpi when running MPI jobs.
maxcputime	the max cpu time for a single execution of a job.
maxmemory	the maximum memory in MB required for the job
maxtime	the maximum time or walltime for a single execution of a job.
maxwalltime	the maximum walltime for a single execution of a job.
minmemory	the minumum amount of memory required for this job
project	associates an account with a job at the remote end.
queue	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 10.3. Table 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 10.4. Table 4: Useful Condor Commands

Key	Description
universe	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.
periodic_release	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.
periodic_remove	is the number of times a job is allowed to get into HOLD state before being removed from the queue. Pegasus defaults to 3.
filesystemdomain	Useful for Condor glide-ins to pin a job to a remote site.
stream_error	boolean to turn on the streaming of the stderr of the remote job back to submit host.
stream_output	boolean to turn on the streaming of the stdout of the remote job back to submit host.
priority	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.
request_cpus	New in Condor 7.8.0 . Number of CPU's a job requires.
request_memory	New in Condor 7.8.0 . Amount of memory a job requires.
request_disk	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. Table 5 shows prohibited Condor commands.

Table 10.5. Table 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.

Table 6 shows the keys in the dagman profile domain that are understood by Pegasus and can be associated at a per job basis.

Table 10.6. Table 6: Useful dagman Commands that can be associated at a per job basis

Key	Description
PRE	is the path to the pre-script. DAGMan executes the pre-script before it runs the job.
PRE.ARGUMENTS	are command-line arguments for the pre-script, if any.
POST	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
POST.PATH.* (where * is replaced by the value of the POST Profile)	the path to the post script on the submit host.
POST.ARGUMENTS	are the command line arguments for the post script, if any.
RETRY	is the number of times DAGMan retries the full job cycle from pre-script through post-script, if failure was detected.
CATEGORY	the DAGMan category the job belongs to.
PRIORITY	the priority to apply to a job. DAGMan uses this to select what jobs to release when MAXJOBS is enforced for the DAG.

Table 7 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 10.7. Table 7: Useful dagman Commands that can be specified in the properties file.

Key	Description
MAXPRE	sets the maximum number of PRE scripts within the DAG that may be running at one time
MAXPOST	sets the maximum number of PRE scripts within the DAG that may be running at one time
MAXJOBS	sets the maximum number of jobs within the DAG that will be submitted to Condor at one time.
MAXIDLE	sets the maximum number of idle jobs within the DAG that will be submitted to Condor at one time.
[CATEGORY-NAME].MAXJOBS	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.
POST.SCOPE	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.

Table 8 shows some of the useful configuration option Pegasus understands.

Table 10.8. Table 8: Useful pegasus Profiles.

Key	Description
workdir	Sets the remote initial dir for a Condor-G job. Overrides the work directory algorithm that uses the site catalog and properties.
clusters.num	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.
clusters.size	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.
cores	The number of cores, associated with the job. This is solely used for accounting purposes in the database while generating statistics. It corresponds to the multiplier_factor in the job_instance table described here.
runtime	Please refer to the Pegasus Clustering Guide for detailed description. This profile specifies the expected runtime of a job.
clusters.maxruntime	Please refer to the Pegasus Clustering Guide for detailed description. This profile specifies the maximum runtime of a job.
job.aggregator	Indicates the clustering executable that is used to run the clustered job on the remote site.
gridstart	Determines the executable for launching a job. Possible values are **Kickstart | NoGridStart** at the moment.
gridstart.path	Sets the path to the gridstart . This profile is best set in the Site Catalog.
gridstart.arguments	Sets the arguments with which GridStart is used to launch a job on the remote site.
stagein.clusters	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 *Bundle* Transfer Refiner, which is the Default Refiner used in Pegasus. This profile is best set in the Site Catalog or in the Properties file
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. This profile is best set in the Site Catalog or in the Properties file
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. This profile is best set in the Site Catalog or in the Properties file
stageout.clusters	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 *Bundle* Transfer Refiner, , which is the Default Refiner used in Pegasus.

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. This profile is best set in the Site Catalog or in the Properties file
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. This profile is best set in the Site Catalog or in the Properties file
group	Tags a job with an arbitrary group identifier. The group site selector makes use of the tag.
change.dir	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.
create.dir	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.
transfer.proxy	If true, tells Pegasus to explicitly transfer the proxy for transfer jobs to the remote site. This is useful, when you want to use a full proxy at the remote end, instead of the limited proxy that is transferred by CondorG.
transfer.arguments	Allows the user to specify the arguments with which the transfer executable is invoked. However certain options are always generated for the transfer executable(base-uri se-mount-point).
style	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.
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.

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 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>
```

The XML schema requires that profiles are the first children of a pool element. If the element ordering is wrong, the XML parser will produce errors and warnings:

```
<pool handle="isi_condor" gridlaunch="/home/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>
  <lrc url="rls://sukhna.isi.edu" />
  &mldr;
</pool>
```

The multi-line textual version of the site catalog uses the following syntax:

```
profile namespace "key" "value"
```

The order within the textual pool definition is not important. Profiles can appear anywhere:

```
pool isi_condor {
  gridlaunch "/home/shared/pegasus/bin/kickstart"
  profile env "GLOBUS_LOCATION" "/home/shared/globus"
  profile env "LD_LIBRARY_PATH" "/home/shared/globus/lib"
  &mldr;
}
```

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.

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.

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. Restricted
3. Regex

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

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.

This selector looks at each PFN returned for a LFN and checks to see if

1. the PFN is a file URL (starting with file:///)
2. the PFN has a pool attribute matching to the site handle of the site where the compute job that requires the input file is to be run.

If a PFN matching the conditions above exists then that is returned by the selector .

Else, a random PFN is selected amongst all the PFN’s that have a pool attribute matching to the site handle of the site where a compute job is to be run.

Else, a random pfn is selected amongst all the PFN’s

To use this replica selector set the following property

```
pegasus.selector.replica                  Default
```

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 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
```

Regex

This replica selector allows the user allows the user to specific regex expressions that can be used to rank various PFN’s returned from the Replica Catalog for a particular LFN. This replica selector selects the highest ranked PFN i.e the replica with the lowest rank value.

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
```

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                  Default
```

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)
- 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.

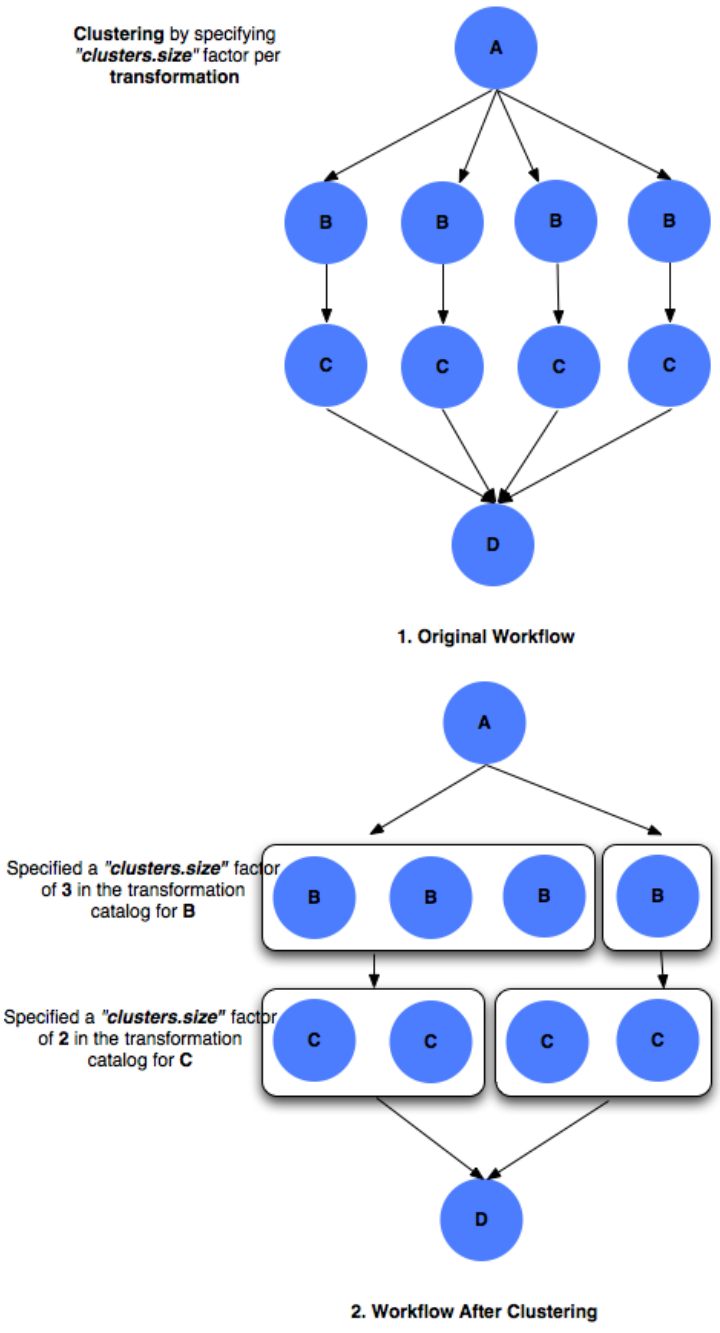
- **clusters.size factor**

  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

  ```
  #site   transformation   pfn           type              architecture  profiles
  ```

```
siteX    B     /shared/PEGASUS/bin/jobB INSTALLED       INTEL32::LINUX  PEGASUS::clusters.size=3
siteX    C     /shared/PEGASUS/bin/jobC INSTALLED       INTEL32::LINUX  PEGASUS::clusters.size=2
```

Figure 10.1. 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

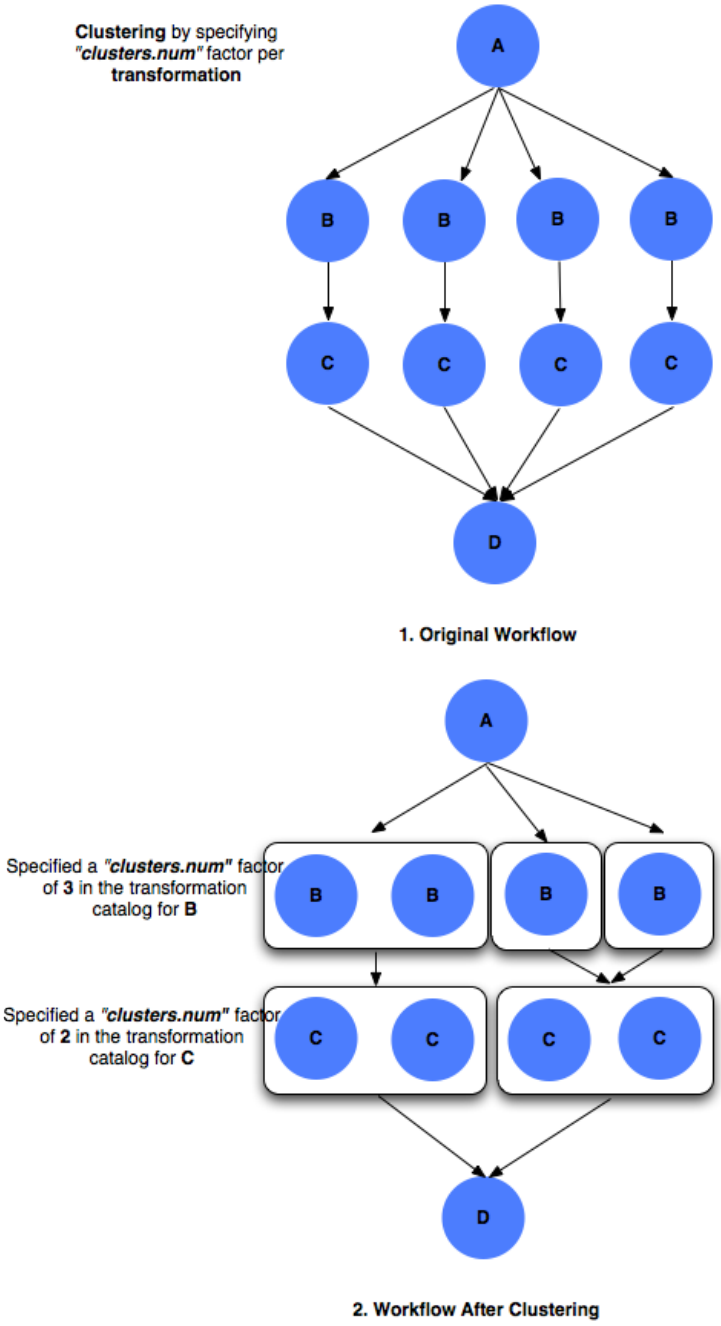
```
#site  transformation      pfn           type            architecture    profiles

siteX    B     /shared/PEGASUS/bin/jobB INSTALLED       INTEL32::LINUX  PEGASUS::clusters.num=3
siteX    C     /shared/PEGASUS/bin/jobC INSTALLED       INTEL32::LINUX  PEGASUS::clusters.num=2
```

In the case, where both the factors are associated with the job, the clusters.num value supersedes the clusters.size value.

```
#site  transformation   pfn             type             architecture   profiles

siteX    B     /shared/PEGASUS/bin/jobB INSTALLED       INTEL32::LINUX
 PEGASUS::clusters.size=3,clusters.num=3
```

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 10.2. 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**.

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 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

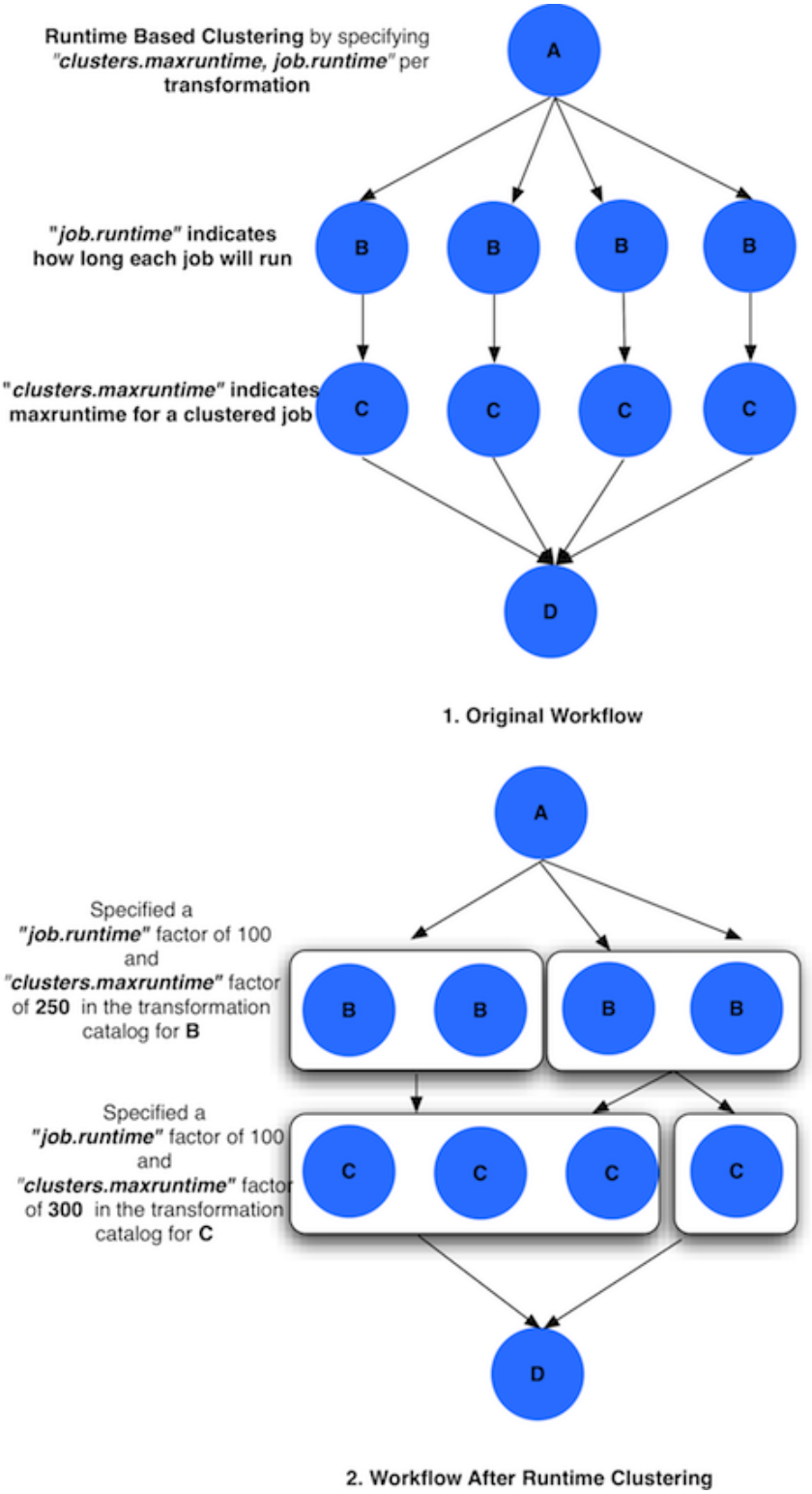
- **clusters.maxruntime**

  maxruntime for the clustered job

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.

```
#site  transformation   pfn             type             architecture   profiles

siteX    B     /shared/PEGASUS/bin/jobB INSTALLED       INTEL32::LINUX
 PEGASUS::clusters.maxruntime=250,runtime=100
siteX    C     /shared/PEGASUS/bin/jobC INSTALLED       INTEL32::LINUX
 PEGASUS::clusters.maxruntime=300,runtime=100
```

Figure 10.3. Clustering by runtime



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.

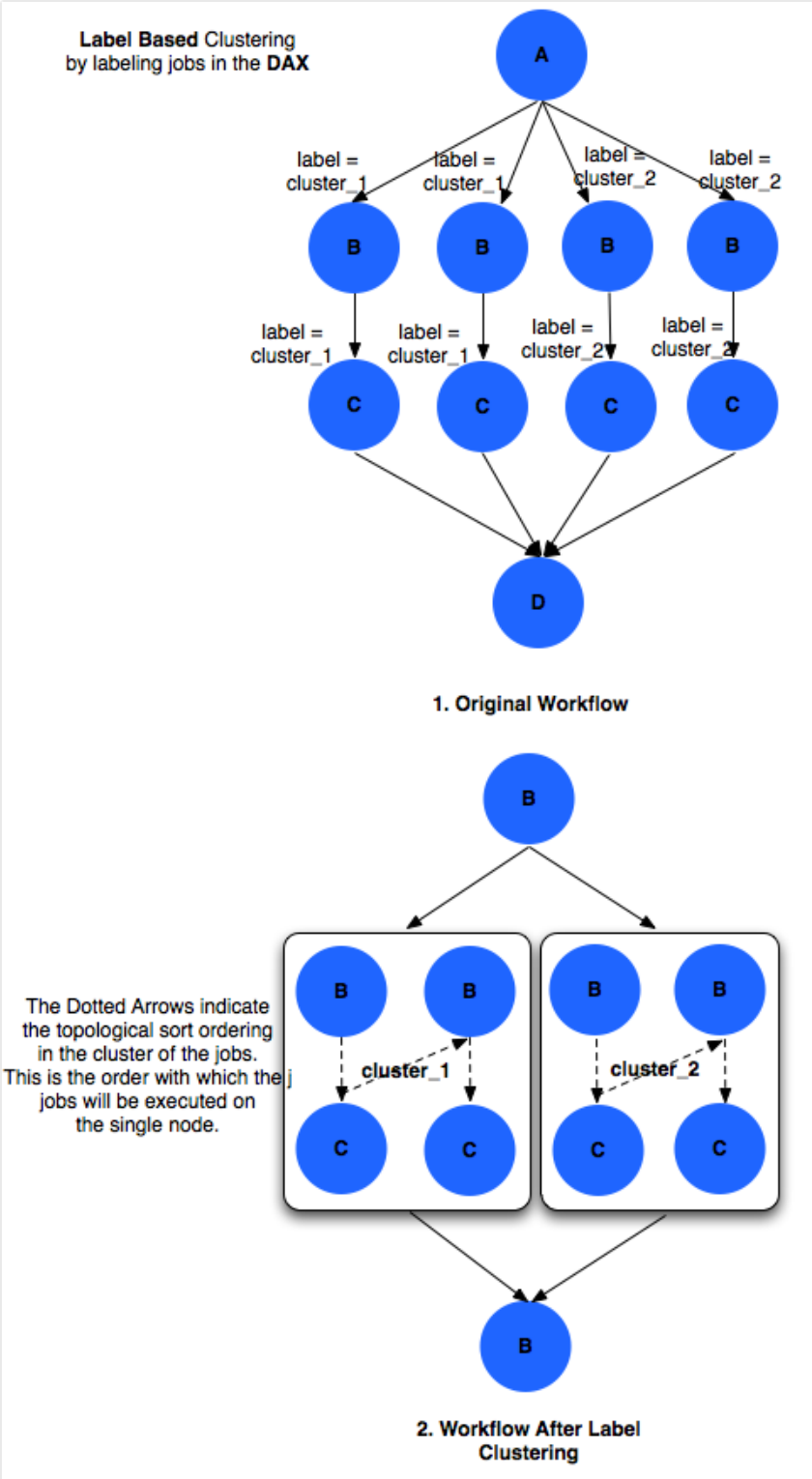
In addition to the above two profiles, we need to inform pegasus-plan to use runtime clustering. This is done by setting the following property .

```
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 10.4. 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" dontRegister="false" dontTransfer="false"/>
    <uses file="user.f.c2" link="input" dontRegister="false" dontTransfer="false"/>
    <uses file="user.f.d" link="output" dontRegister="false" dontTransfer="false"/>
  </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.

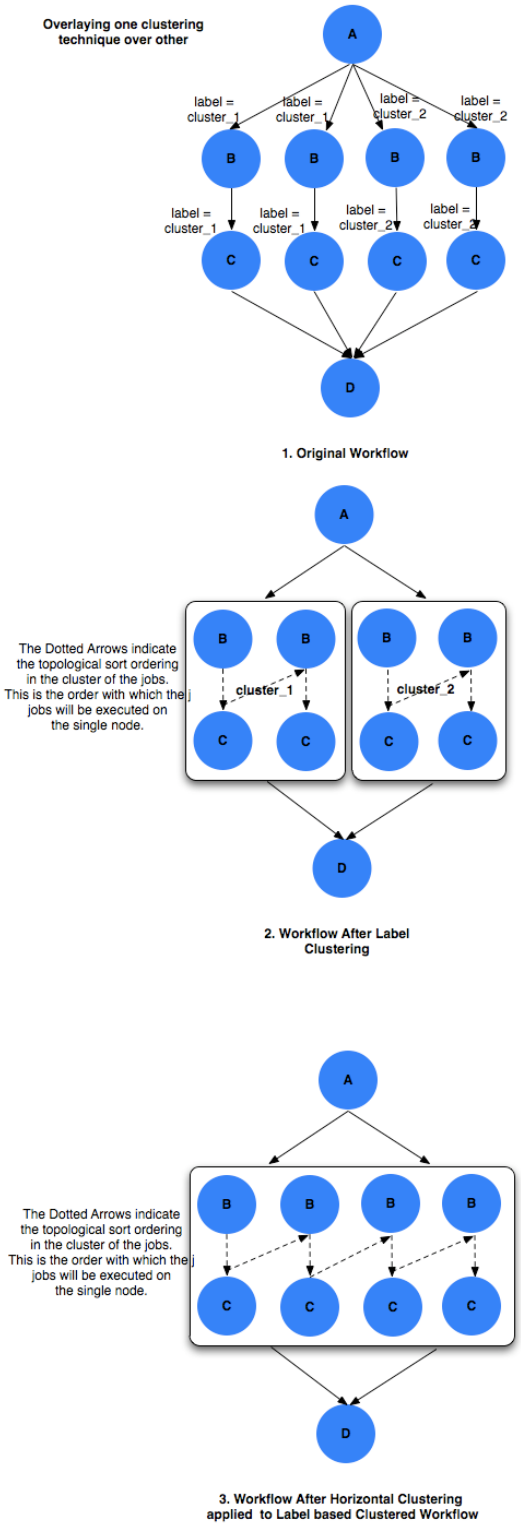
Recursive Clustering

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 10.5. Recursive clustering



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 10.9. Table : 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.

```
#site  transformation   pfn            type                 architecture    profiles

siteX    pegasus::mpiexec     /usr/pegasus/bin/pegasus-mpi-cluster INSTALLED       INTEL32::LINUX
 globus::xcount=32;globus::host_xcount=1
```

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.

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**

   ```
   #site  transformation   pfn            type                architecture profiles

   siteX    B     /shared/PEGASUS/bin/jobB INSTALLED       INTEL32::LINUX
    pegasus::clusters.size=3,job.aggregator=mpiexec
   ```

   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.

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 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.

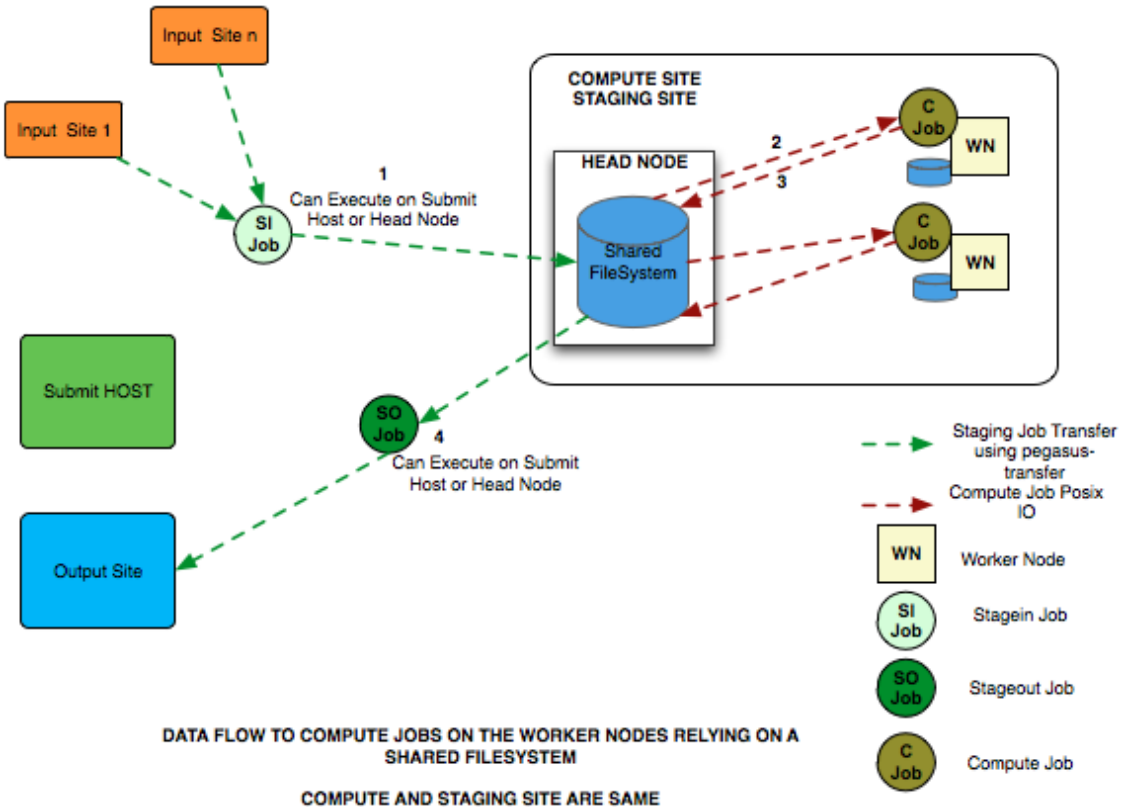
7. **Worker Node Directory**

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

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 10.6. Shared File System Setup



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.

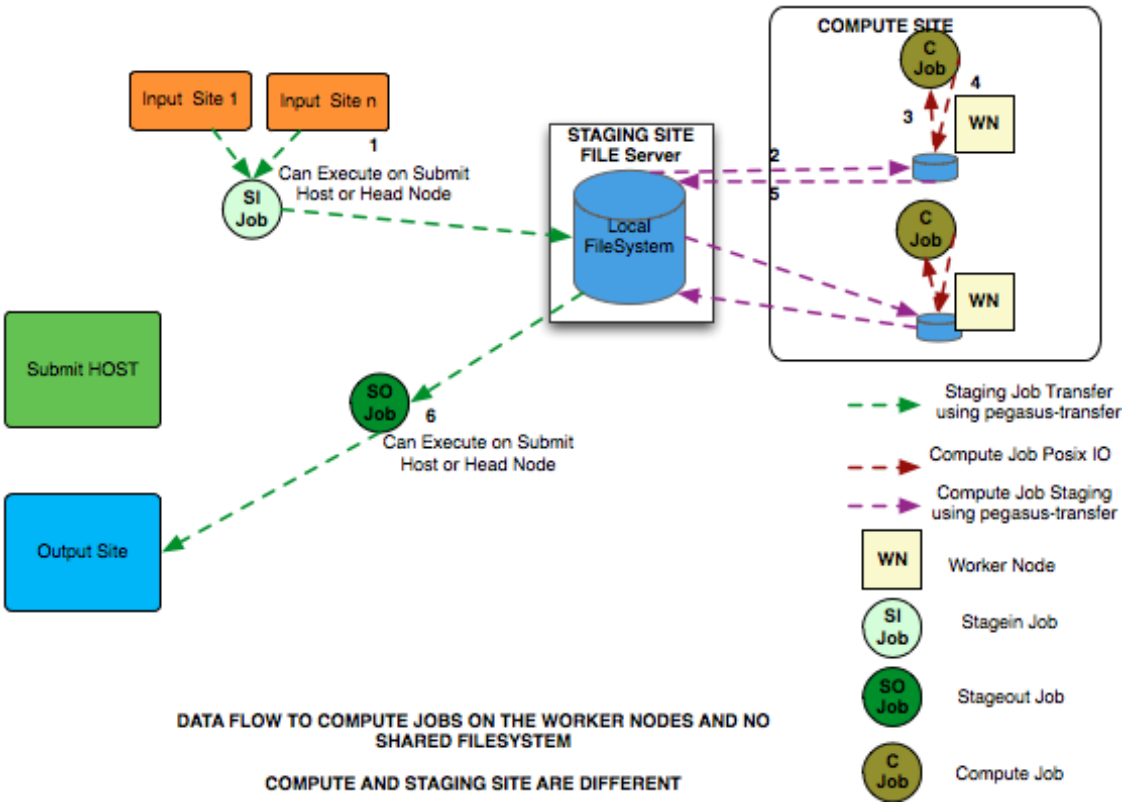
Non Shared Filesystem

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 10.7. 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.

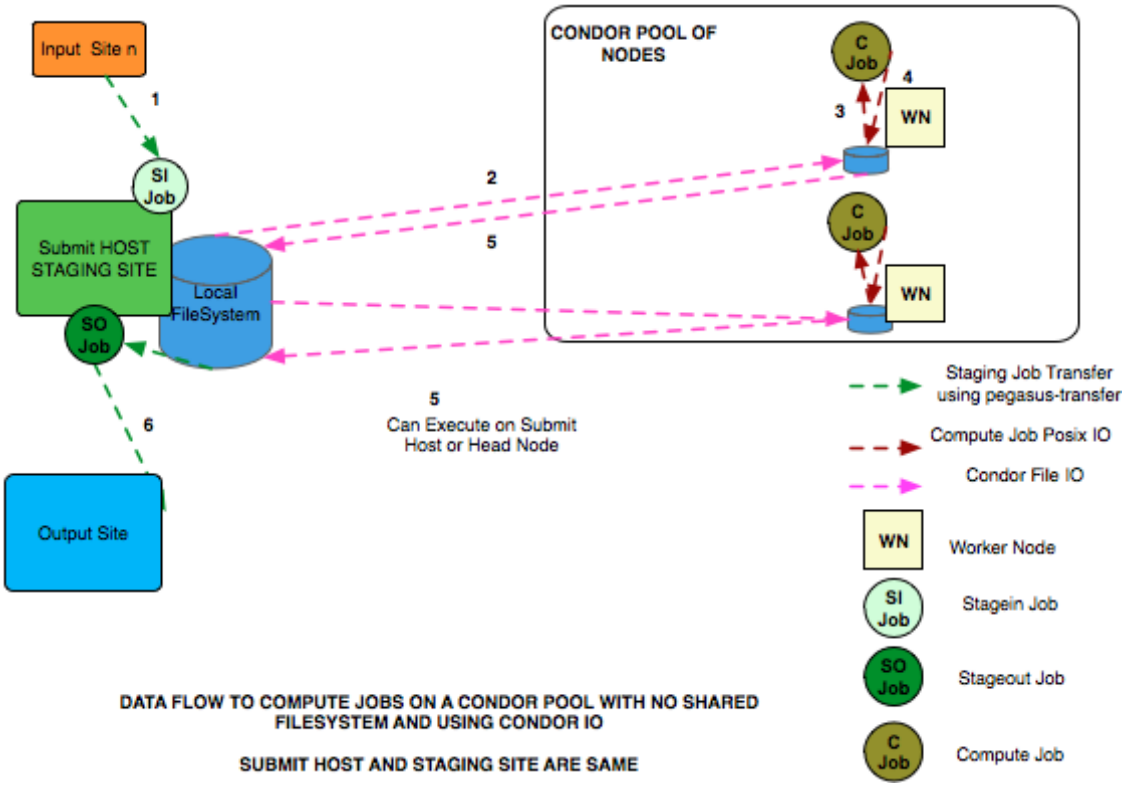
Condor Pool Without a Shared Filesystem

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 10.8. 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. For e.g if the file server specified for the staging site/compute site is a file server, then Pegasus will schedule all the stage in data movement jobs on the compute site to stage-in the input data for the workflow. Another case would be if 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).

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 10.10. Property Variations for pegasus.transfer.*.remote.sites

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	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]_** .

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 pool attribute associated with the URL in the Replica Catalog. If the pool 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 pool 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 pool="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.

The default Transfer Refiner used in Pegasus is the Bundle Refiner that allows the user to specify how many local|remote stagein|stageout jobs are created per execution site.

The behavior of the refiner is controlled by specifying certain pegasus profiles

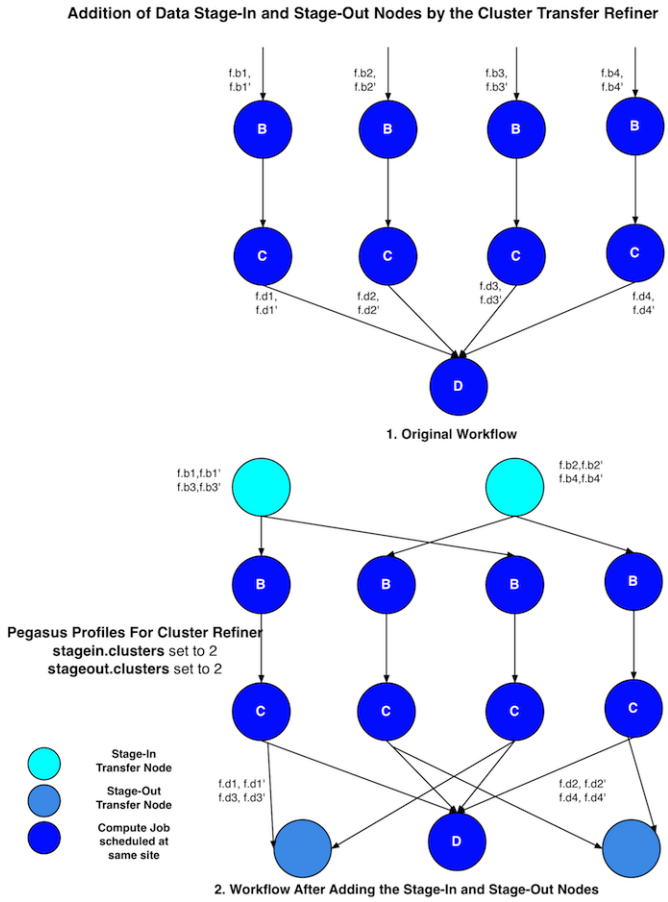
1. either with the execution sites in the site catalog

2. OR globally in the properties file

Table 10.11. 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.

Figure 10.9. Default Transfer Case : Input Data To Workflow Specific Directory on Shared File System



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.

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. **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

3. **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 properties

   - pegasus.dir.storage.mapper.replica Regex|File

   - pegasus.dir.storage.mapper.replica.file the RC file at the backend to use

Tip

The mappers can be configured by setting the property **pegasus.dir.storage.mapper**

Executable Used for Transfer Jobs

Pegasus refers to a python script called **pegasus-transfer** as the executable in the transfer jobs to transfer the data. pegasus-transfer is a python based wrapper around various transfer clients . 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 10.12. Transfer Clients interfaced to by pegasus-transfer

Transfer Client	Used For
globus-url-copy	staging files to and from a gridftp server.
lcg-copy	staging files to and from a SRM server.
wget	staging files from a HTTP server.
cp	copying files from a POSIX filesystem .
ln	symlinking against input files.
pegasus-s3/s3cmd	staging files to and from s3 bucket in the amazon cloud
scp	staging files using scp
iget	staging files to and from a irods server.

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**

Executables used for Directory Creation and Cleanup Jobs

Starting 4.0, Pegasus has changed the way how the scratch directories are created on the staging site. The planner now prefers to schedule the directory creation and cleanup jobs locally. The jobs refer to python based tools, that call out to protocol specific clients to determine what client is picked up. For protocols, where specific remote cleanup and directory creation clients don't exist (for example gridftp), the python tools rely on the corresponding transfer tool to create a directory by initiating a transfer of an empty file. The python clients used to create directories and remove files are called

- pegasus-create-dir
- pegasus-cleanup

Both these clients inspect the URL's to to determine what underlying client to pick up.

Table 10.13. Clients interfaced to by pegasus-create-dir

Client	Used For
globus-url-copy	to create directories against a gridftp/ftp server
srm-mkdir	to create directories against a SRM server.
mkdir	to create a directory on the local filesystem
pegasus-s3	to create a s3 bucket in the amazon cloud
scp	staging files using scp
imkdir	to create a directory against an IRODS server

Table 10.14. Clients interfaced to by pegasus-cleanup

Client	Used For
globus-url-copy	to remove a file against a gridftp/ftp server. In this case a zero byte file is created
srm-rm	to remove files against a SRM server.
rm	to remove a file on the local filesystem
pegasus-s3	to remove a file from the s3 bucket.
scp	to remove a file against a scp server. In this case a zero byte file is created.
irm	to remove a file against an IRODS server

The only case, where the create dir and cleanup jobs are scheduled to run remotely is when for the staging site, a file server is specified.

Credentials Staging

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 datat 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, iRods password and SSH keys.

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="env" 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` environment profile to the location of the file. Site catalog example:

```
<profile namespace="env" key="S3CFG" >/home/user/.s3cfg</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:

```
# iRODS personal configuration file.
#
# iRODS server host name:
irodsHost 'iren.renci.org'
# iRODS server port number:
irodsPort 1259

# Default storage resource name:
irodsDefResource 'renResc'
# Home directory in iRODS:
irodsHome '/tip-renci/home/mats'
# Current directory in iRODS:
irodsCwd '/tip-renci/home/mats'
# Account name:
irodsUserName 'mats'
# Zone:
irodsZone 'tip-renci'
```

```
# this is used with Pegasus
irodsPassword 'somesecretpassword'
```

The location of the file can be given to the workflow using the `irodsEnvFile` environment profile. Site catalog example:

```
<profile namespace="env" 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="env" key="SSH_PRIVATE_KEY" >/home/user/wf/wfsshkey</profile>
```

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 10.15. 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 (pool 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 Pegasus Worker Package

Pegasus can optionally stage the pegasus worker package as part of the executable workflow to remote workflow specific execution directory. The pegasus worker package contains the pegasus auxillary executables that are required on the remote site. If the worker package is not staged as part of the executable workflow, then Pegasus relies on the installed version of the worker package on the remote site. To determine the location of the installed version of the worker package on a remote site, Pegasus looks for an environment profile PEGASUS_HOME for the site in the Site Catalog.

Users can set the following property to true to turn on worker package staging

```
pegasus.transfer.worker.package          true
```

By default, when worker package staging is turned on pegasus pulls the compatible worker package from the Pegasus Website. To specify a different worker package location, users 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.

Worker Package Staging in Non Shared Filesystem setup

Worker package staging is automatically set to true , when workflows are setup to run in a non shared filesystem setup i.e. **pegasus.data.configuration** is set to **nonsharedfs** or **condorio** . In these configurations, a stage_worker job is created that brings in the worker package to the submit directory of the workflow. For each job, the worker package is then transferred with the job using Condor File Transfers (**transfer_input_files**) . This transfer always happens unless, PEGASUS_HOME is specified in the site catalog for the site on which the job is scheduled to run.

Users can explicitly set the following property to false, to turn off worker package staging by the Planner. This is applicable , when running in the cloud and virtual machines / worker nodes already have the pegasus worker tools installed.

```
pegasus.transfer.worker.package          false
```

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>
```

Hierarchical Workflows

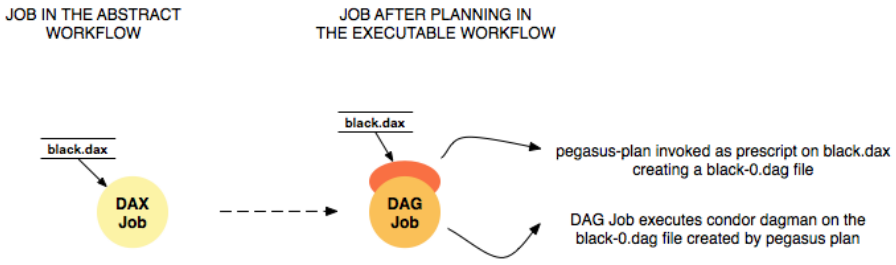
Introduction

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

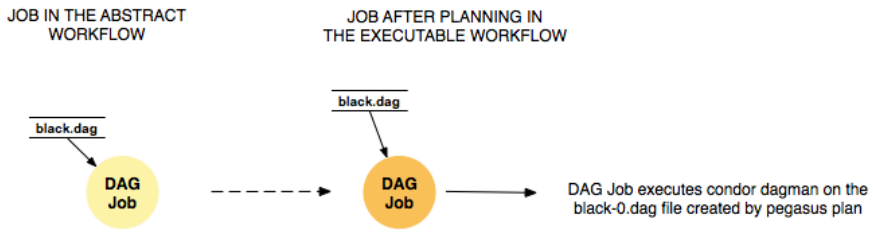
1. daxjob - refers to a sub workflow represented as a DAX. During the planning of a workflow, the DAX jobs are mapped to condor dagman jobs that have pegasus plan invocation on the dax (referred to in the DAX job) as the prescript.

Figure 10.10. 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 10.11. 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 10.16. Options inherited from parent workflow

Option Name	Description
--sites	list of execution sites.

It is highly recommended that users **dont 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 user 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 of 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
```

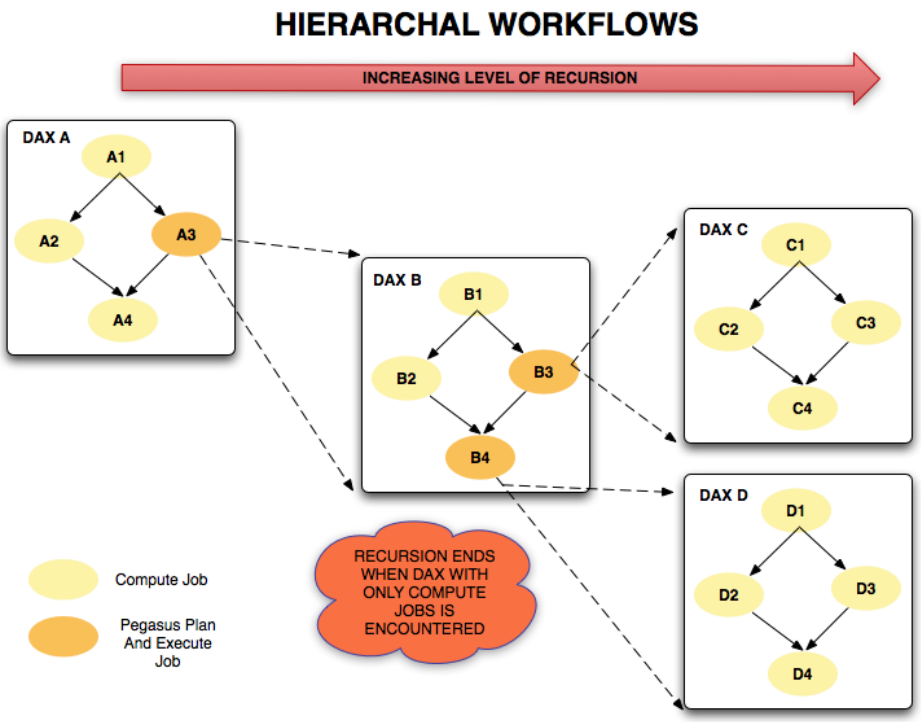
The above is useful in the case, where you are staging out the output files to a storage site, and you want the child sub workflow to stage these files from the storage output site instead of the workflow execution directory where the files were originally created.

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) .

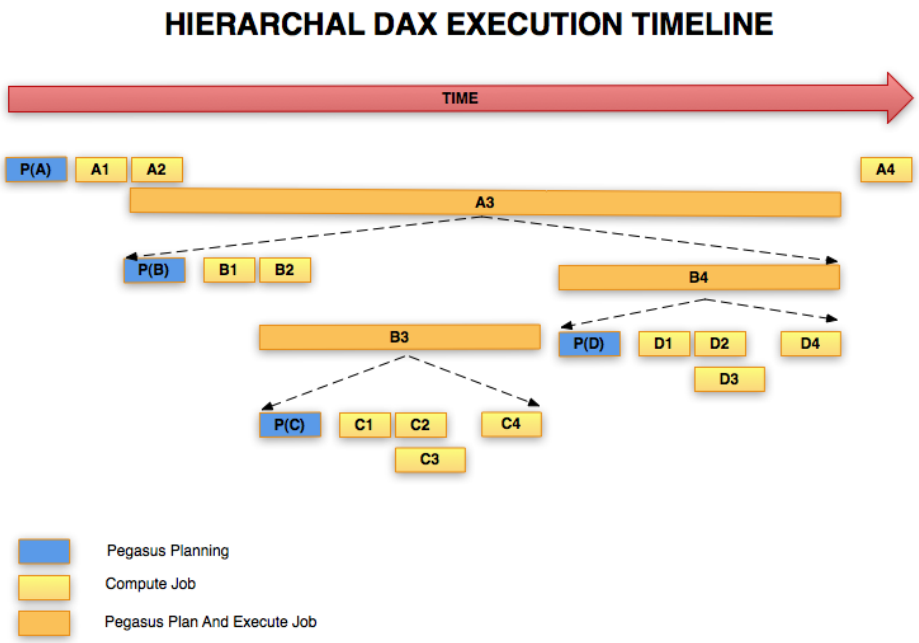
The figure below illustrates an example with recursion 2 levels deep.

Figure 10.12. Recursion in Hierarchal Workflows



The execution time-line of the various jobs in the above figure is illustrated below.

Figure 10.13. 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.

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 10.17. Table 1. 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, on_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 Table 1
  - 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

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:

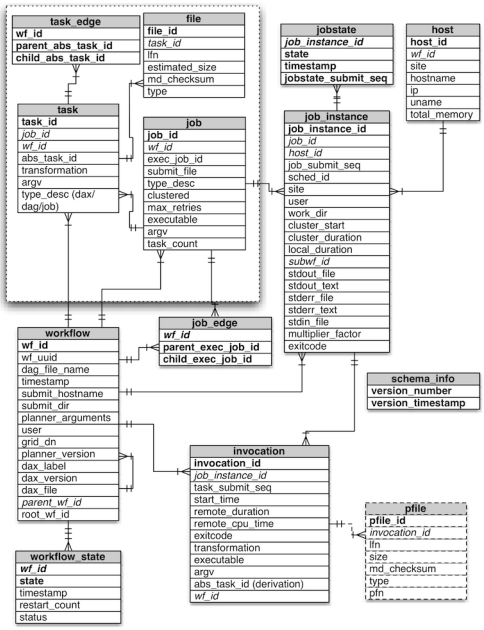
- **jobstate.log**: contains a summary of workflow and job execution.
- **monitord.log**: contains any log messages generated by **pegasus-monitord**. It is not overwritten when it restarts. This file is not generated in replay mode, as all log messages from **pegasus-monitord** are output to the console. 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 Stampede 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 10.14. Stampede 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 walltime
- In the summary, the cumulative job walltime 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.

API Reference

DAX XML Schema

The DAX format is described by the XML schema instance document dax-3.3.xsd [http://pegasus.isi.edu/wms/docs/schemas/dax-3.3/dax-3.3.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-3.3.html [http://pegasus.isi.edu/wms/docs/schemas/dax-3.3/dax-3.3.html] as well as locally in `doc/schemas/dax-3.3/dax-3.3.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 Notifications

   Very simple workflow-level notifications. These are defined in the Notification section.

2. 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.

3. 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.

4. 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-3.3.xsd"
      version="3.3"
      name="diamond"
      index="0"
```

```
count="1">
```

The following attributes are supported for the root element *adag*.

Table 10.18.

attribute	optional?	type	meaning
version	required	*VersionPattern*	Version number of DAX instance document. Must be 3.3.
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 Notifications

(something to be said here.)

```
<!-- 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 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 is currently NOT SUPPORTED -->
  <metadata key="timestamp" type="int">/* ISO-8601 *or* 20100417134523:int */</metadata>
  <metadata key="origin" type="string">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 is currently NOT SUPPORTED! -->
  <metadata key="timestamp" type="int">/* see above */</metadata>
  <metadata key="origin" type="string">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 10.19.

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 look-up 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"  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, on_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="on_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 10.20.

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 10.21.

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.
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 DAGMan 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:NMTOKEN` 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-3.3.xsd"
 version="3.3" 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 and Python. 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 Workflow notification elements
3. 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.
4. (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.
5. Create a new *Job* object.
6. Add arguments, files, profiles, notifications and other information to the *Job* object
7. Add the job object to the *ADAG* object
8. Repeat step 4-6 as necessary.
9. Add all dependencies to the *ADAG* object.

10.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 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("blackdiamond");
        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");
        File fa = new File("f.a");
        fa.addPhysicalFile("file://" + cwd + "/f.a", "local");
        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);

        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.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.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.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.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.

```
#!/usr/bin/env python

from Pegasus.DAX3 import *
import sys
import os

if len(sys.argv) != 2:
        print "Usage: %s PEGASUS_HOME" % (sys.argv[0])
        sys.exit(1)

# Create a abstract dag
diamond = ADAG("diamond")

# Add input file to the DAX-level replica catalog
a = File("f.a")
a.addPFN(PFN("file://" + os.getcwd() + "/f.a", "local"))
diamond.addFile(a)

# Add executables to the DAX-level replica catalog
# In this case the binary is keg, which is shipped with Pegasus, so we use
# the remote PEGASUS_HOME to build the path.
e_preprocess = Executable(namespace="diamond", name="preprocess", version="4.0", os="linux",
 arch="x86_64")
e_preprocess.addPFN(PFN("file://" + sys.argv[1] + "/bin/keg", "TestCluster"))
diamond.addExecutable(e_preprocess)

e_findrange = Executable(namespace="diamond", name="findrange", version="4.0", os="linux",
 arch="x86_64")
e_findrange.addPFN(PFN("file://" + sys.argv[1] + "/bin/keg", "TestCluster"))
diamond.addExecutable(e_findrange)

e_analyze = Executable(namespace="diamond", name="analyze", version="4.0", os="linux",
 arch="x86_64")
e_analyze.addPFN(PFN("file://" + sys.argv[1] + "/bin/keg", "TestCluster"))
diamond.addExecutable(e_analyze)

# Add a preprocess job
preprocess = Job(namespace="diamond", name="preprocess", version="4.0")
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)
preprocess.uses(b2, link=Link.OUTPUT)
diamond.addJob(preprocess)

# Add left Findrange job
frl = Job(namespace="diamond", name="findrange", version="4.0")
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)
diamond.addJob(frl)

# Add right Findrange job
frr = Job(namespace="diamond", name="findrange", version="4.0")
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)
diamond.addJob(frr)

# Add Analyze job
analyze = Job(namespace="diamond", name="analyze", version="4.0")
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, register=True)
diamond.addJob(analyze)

# Add control-flow 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)

# Add notification for analyze job
analyze.invoke(When.ON_ERROR, '/home/user/bin/email -s "Analyze job failed" user@example.com')

# Add notification for workflow
diamond.invoke(When.AT_END, '/home/user/bin/email -s "Workflow finished" user@example.com')
diamond.invoke(When.ON_SUCCESS, '/home/user/bin/publish_workflow_result')

# Write the DAX to stdout
diamond.writeXML(sys.stdout)
```

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 [http://pegasus.isi.edu/wms/docs/3.0/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.

```
#!/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 );
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' ) );
$adag->addFile($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' ) );
            $adag->addExecutable($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 );
$adag->addJob($job1);

my $fnc1 = newFilename( name => 'f.c1', %hash );
$fnb1->link( LINK_IN );
$job2->addArgument( '-a', $job2->name, '-T60', '-i', $fnb1,
                    '-o', $fnc1 );
$adag->addJob($job2);

my $fnc2 = newFilename( name => 'f.c2', %hash );
$fnb2->link( LINK_IN );
$job3->addArgument( '-a', $job3->name, '-T60', '-i', $fnb2,
                    '-o', $fnc2 );
$adag->addJob($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 );
$adag->addJob($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 );
```

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.

Command Line Tools

<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>

<xi:include></xi:include>

Chapter 11. Useful Tips

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).

New Clients for directory creation and file cleanup

Pegasus 4.0 has new clients for directory creation and cleanup.

- pegasus-create-dir
- pegasus-cleanup

Both these clients are python based wrapper scripts around various protocol specific clients that are used to determine what client to pick up.

Table 11.1. Clients interfaced to by pegasus-create-dir

Client	Used For
globus-url-copy	to create directories against a gridftp/ftp server
srm-mkdir	to create directories against a SRM server.
mkdir	to create a directory on the local filesystem
pegasus-s3	to create a s3 bucket in the amazon cloud
scp	staging files using scp
imkdir	to create a directory against an IRODS server

Table 11.2. Clients interfaced to by pegasus-cleanup

Client	Used For
globus-url-copy	to remove a file against a gridftp/ftp server. In this case a zero byte file is created
srm-rm	to remove files against a SRM server.
rm	to remove a file on the local filesystem

Client	Used For
pegasus-s3	to remove a file from the s3 bucket.
scp	to remove a file against a scp server. In this case a zero byte file is created.
irm	to remove a file against an IRODS server

With Pegasus 4.0, the planner will prefer to run the create dir and cleanup jobs locally on the submit host. The only case, where these jobs are scheduled to run remotely is when for the staging site, a file server is specified.

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 CLASS-PATH 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 [http://pegasus.isi.edu/wms/docs/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 [http://pegasus.isi.edu/wms/docs/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 Profileschapter. 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 11.3. Table 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 11.4. Table 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 11.5. Table 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.remote.clusters
bundle.stageout	stageout.clusters | stageout.local.clusters | stageout.remote.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.transformation.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 11.6. Table 1: 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
cleanup	pegasus-cleanup
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.

Chapter 12. 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 12.1. 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 12.2. 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)
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.
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

JSON KEY	DESCRIPTION
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 12.3. 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 13. 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 and FutureGrid.

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

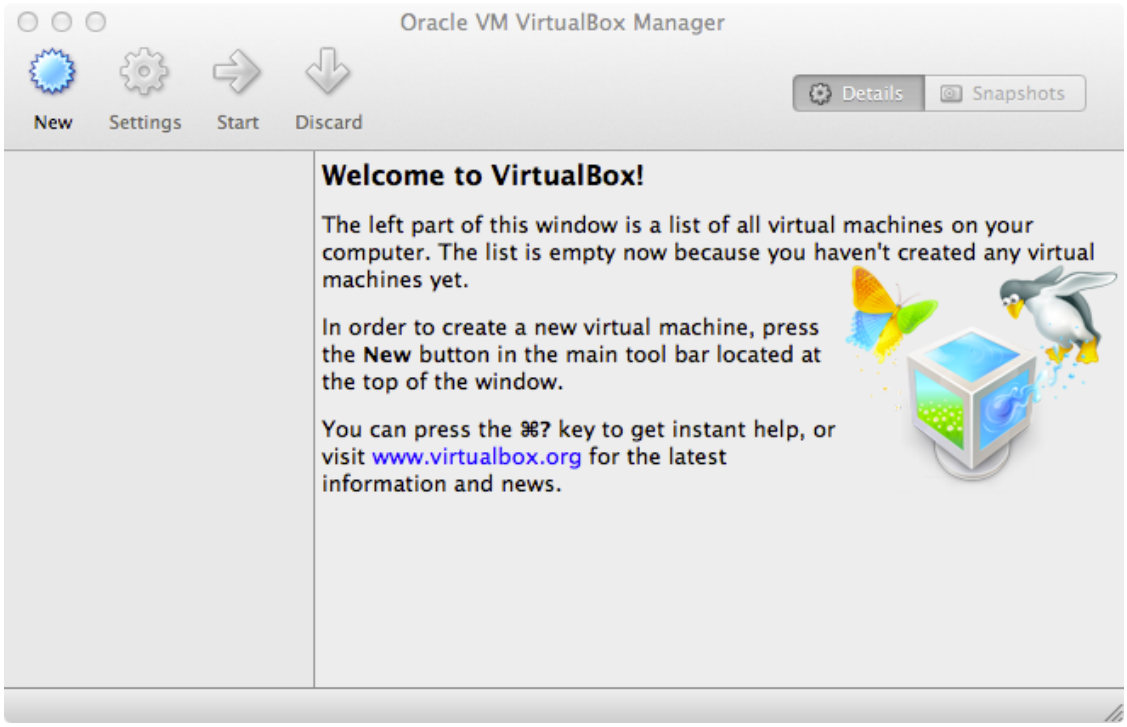
Next, download the Pegasus Tutorial VM from the Pegasus download page: http://pegasus.isi.edu/downloads

Unzip the downloaded file and move the `.vmdk` file it contains to somewhere that you can find it later.

Create Virtual Machine

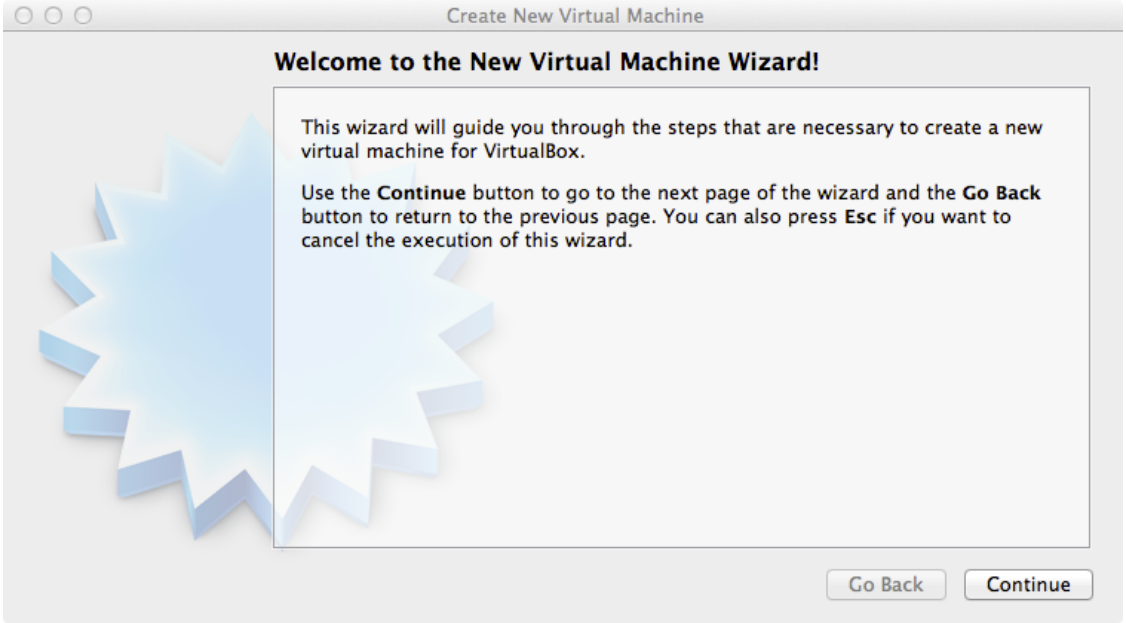
Start VirtualBox. You should get a screen that looks like this:

Figure A.1. VirtualBox Welcome Screen



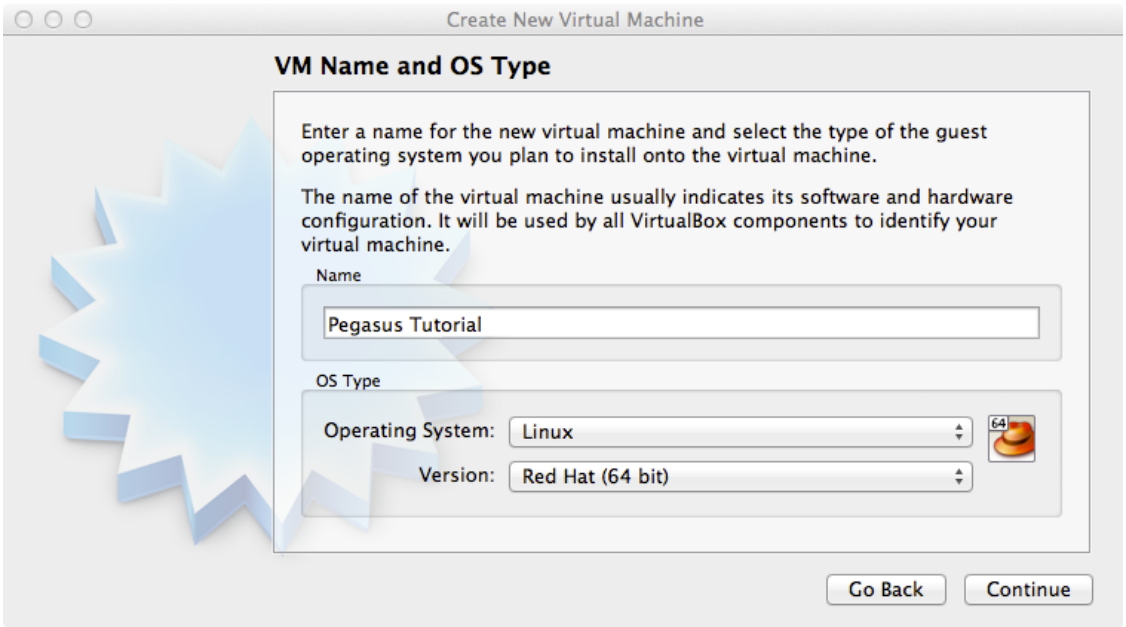
Click on the "New" button. The "Create New Virtual Machine Wizard" will appear:

Figure A.2. Create New Virtual Machine Wizard



Click "Continue" to get to the VM Name and OS Type step:

Figure A.3. VM Name and OS Type



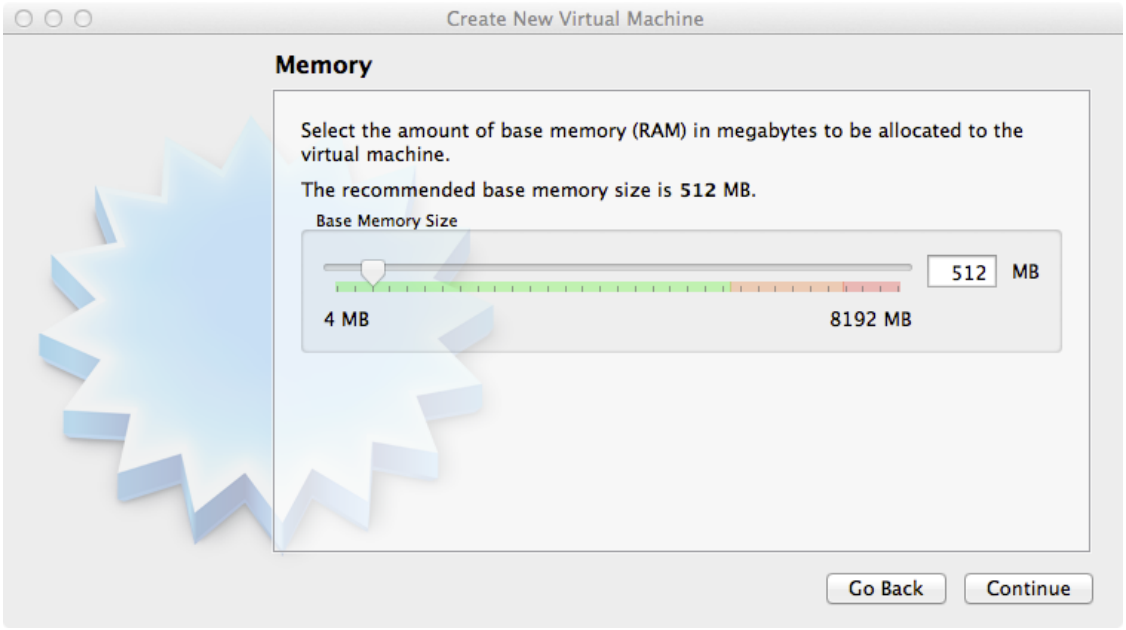
In the Name field type "Pegasus Tutorial". Set the Operating System to "Linux" and the Version to "Red Hat (64 bit)".

Warning

Make sure to select "Red Hat (64 bit)" as the Version. If this is incorrect the virtual machine may not be able to start.

Click "Continue" to get to the Memory step. You can leave this at the default of 512 MB.

Figure A.4. Memory



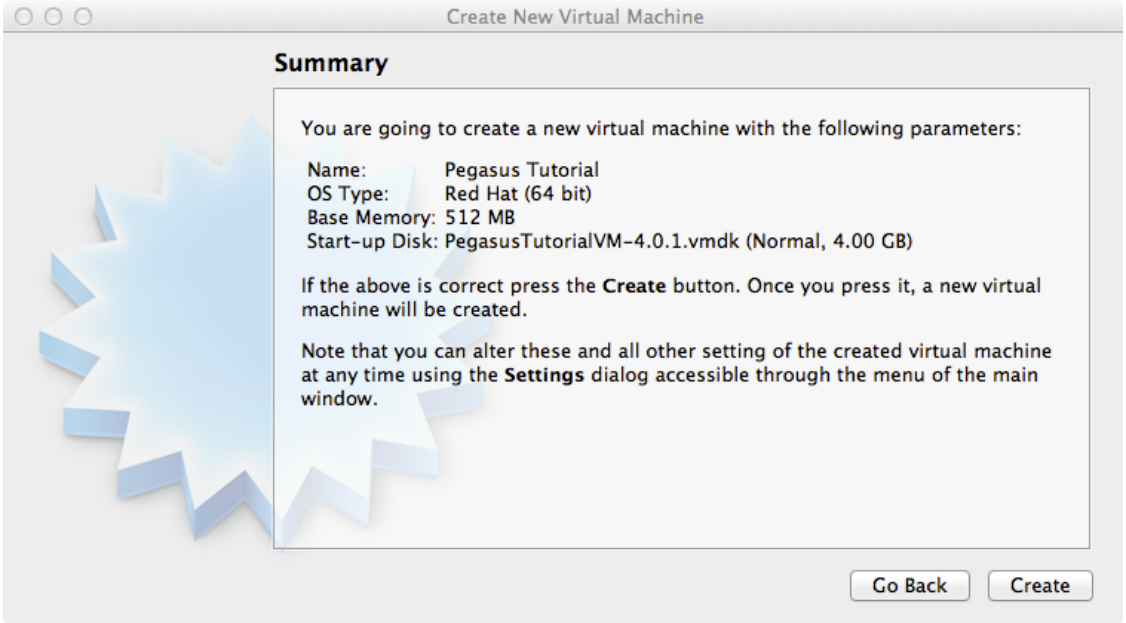
Click "Continue" again to get to the "Virtual Hard Disk" step:

Figure A.5. Virtual Hard Disk

Leave "Start-up Disk" checked. Choose "Use existing hard disk". Click the folder icon and locate the .vmdk file that you downloaded earlier.

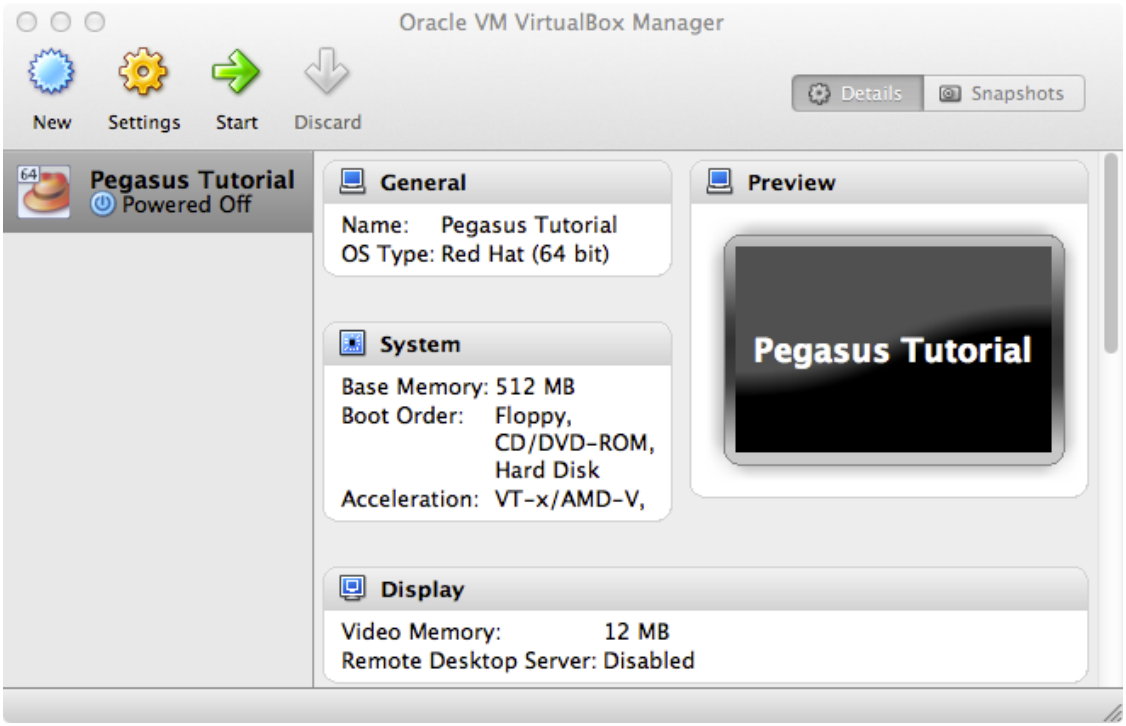
When you have selected the .vmdk file, choose "Open" and then click "Continue" to get to the Summary page:

Figure A.6. Summary



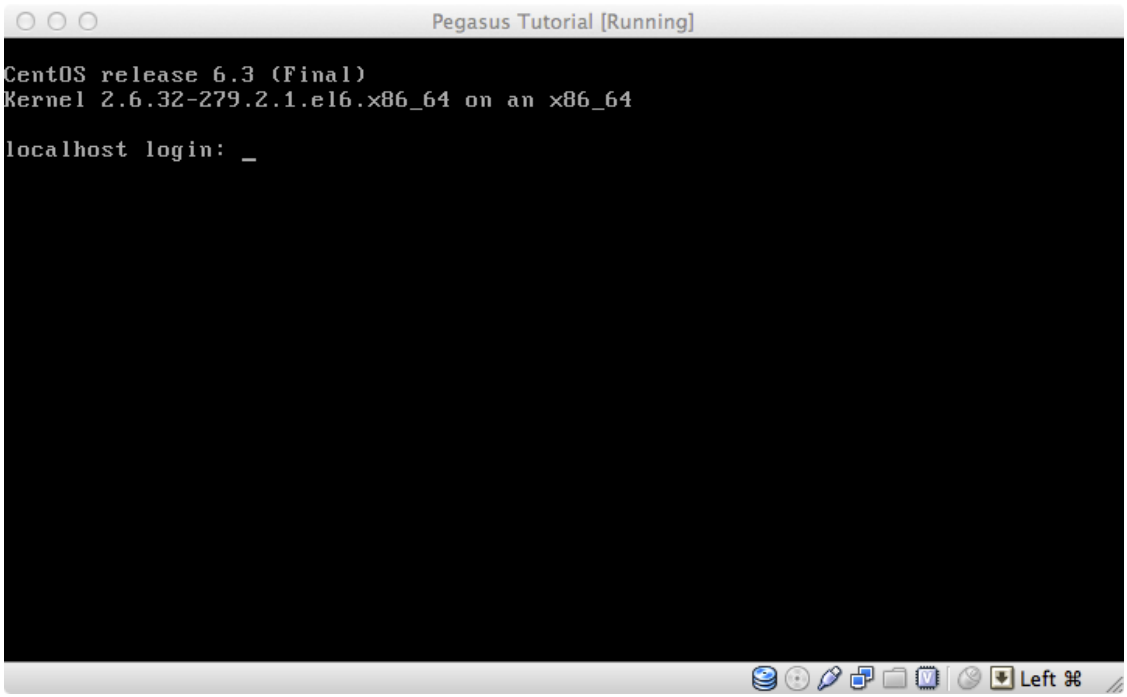
Click "Create". You will get back to the welcome screen showing the new virtual machine:

Figure A.7. Welcome Screen with new virtual machine



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.8. Login Screen



Log in as user "**tutorial**" with password "**pegasus**".

After you log in 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".

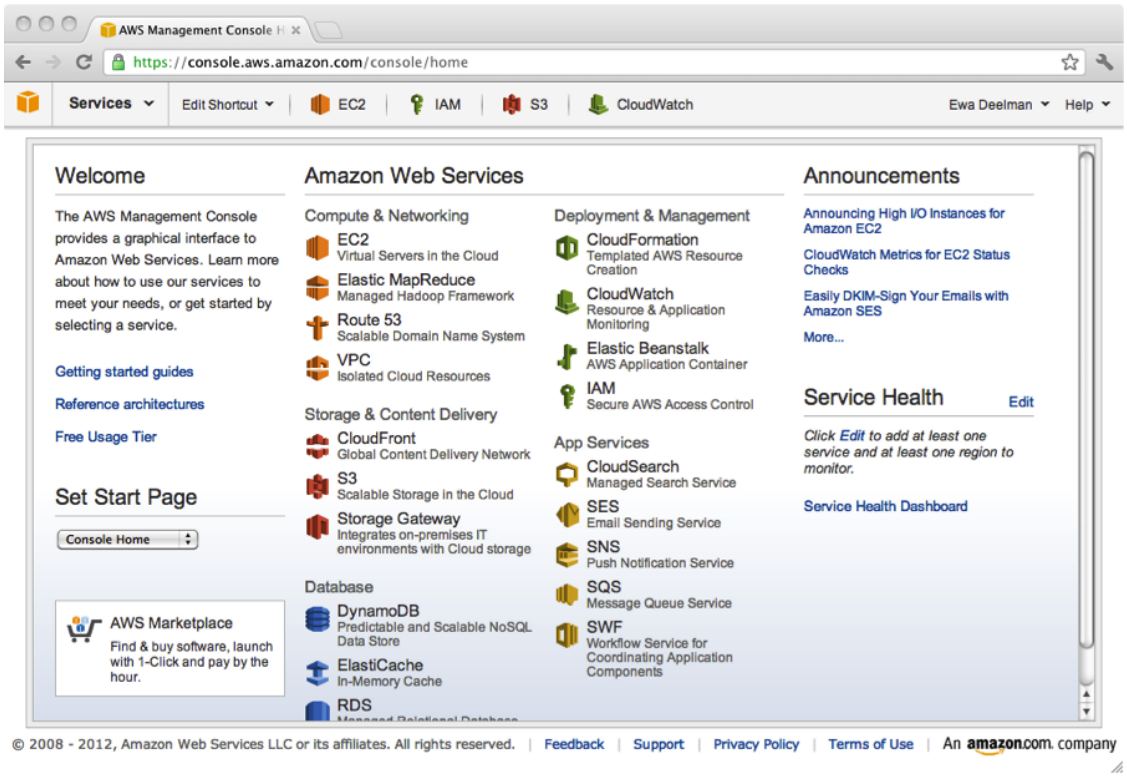
Amazon EC2

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

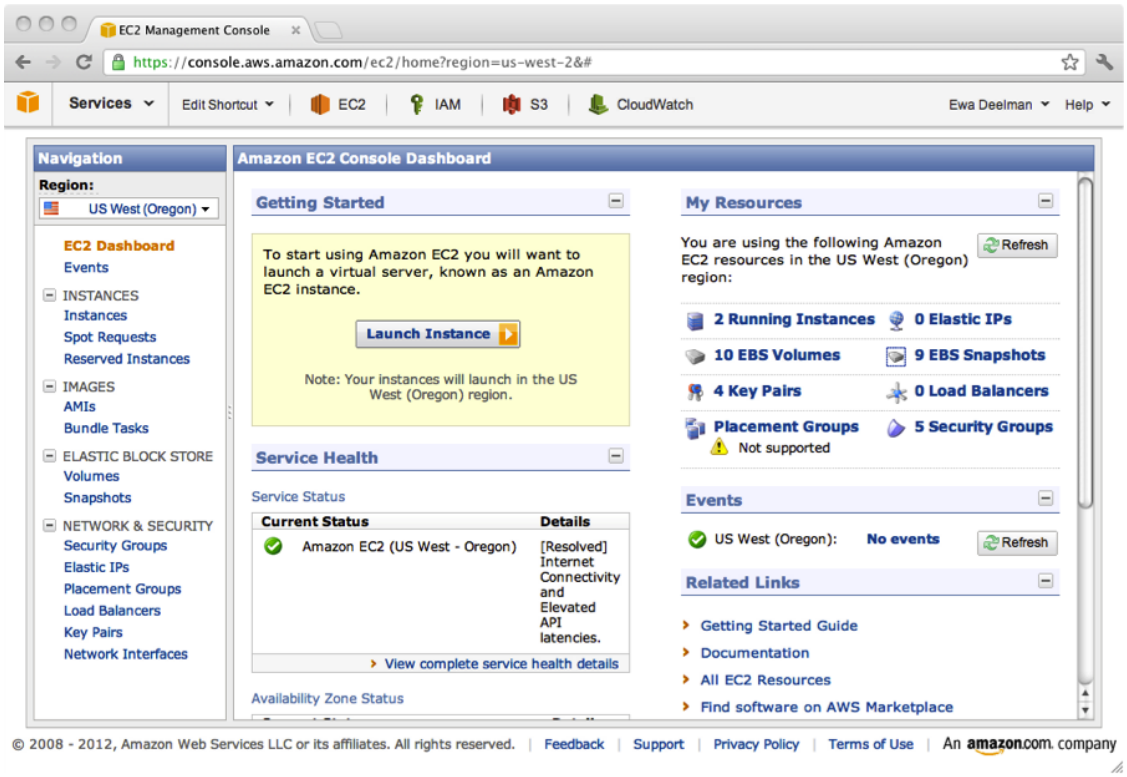
Once you have an account, sign into the AWS Management Console at this URL: http://console.aws.amazon.com. You will get a page that looks like this:

Figure A.9. AWS Management Console



Choose the "EC2" icon under "Amazon Web Services". You will get this page:

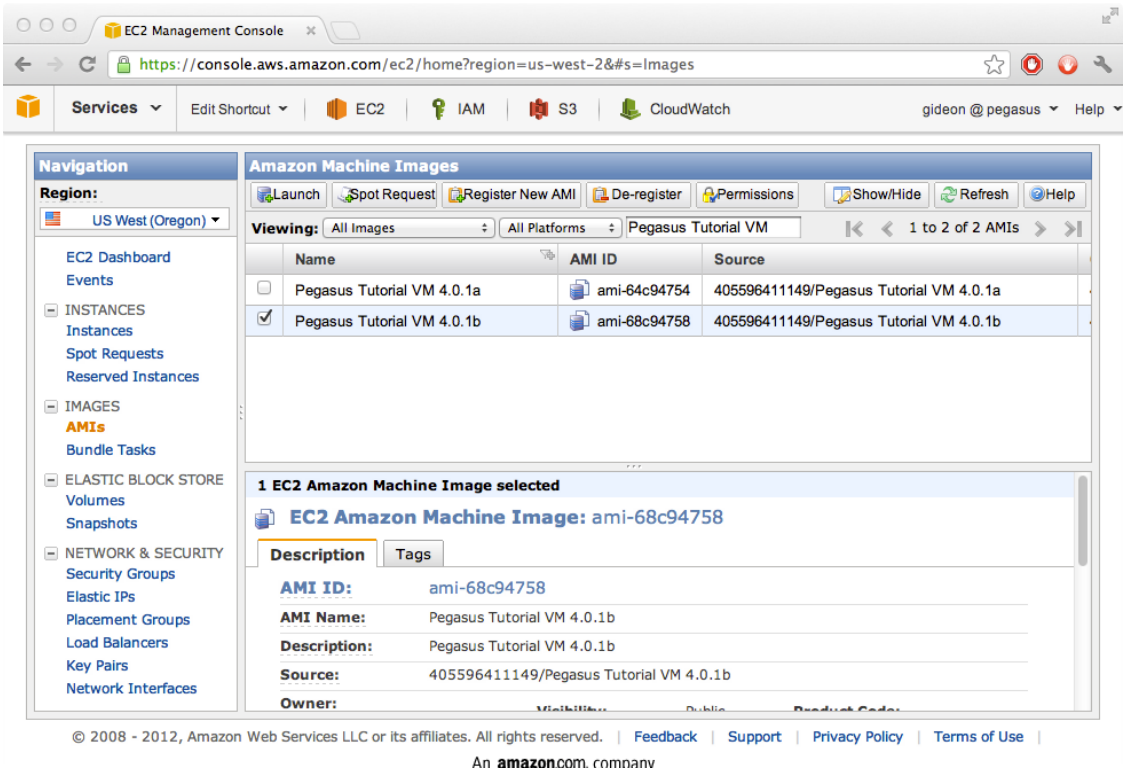
Figure A.10. EC2 Management Console



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:

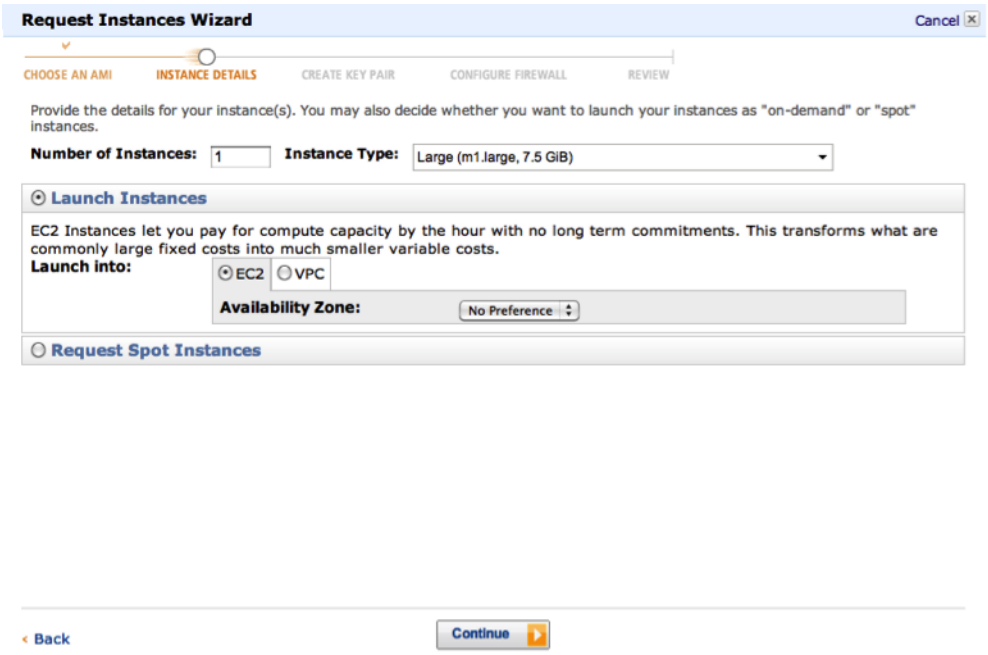
Figure A.11. Locating the Tutorial VM



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.

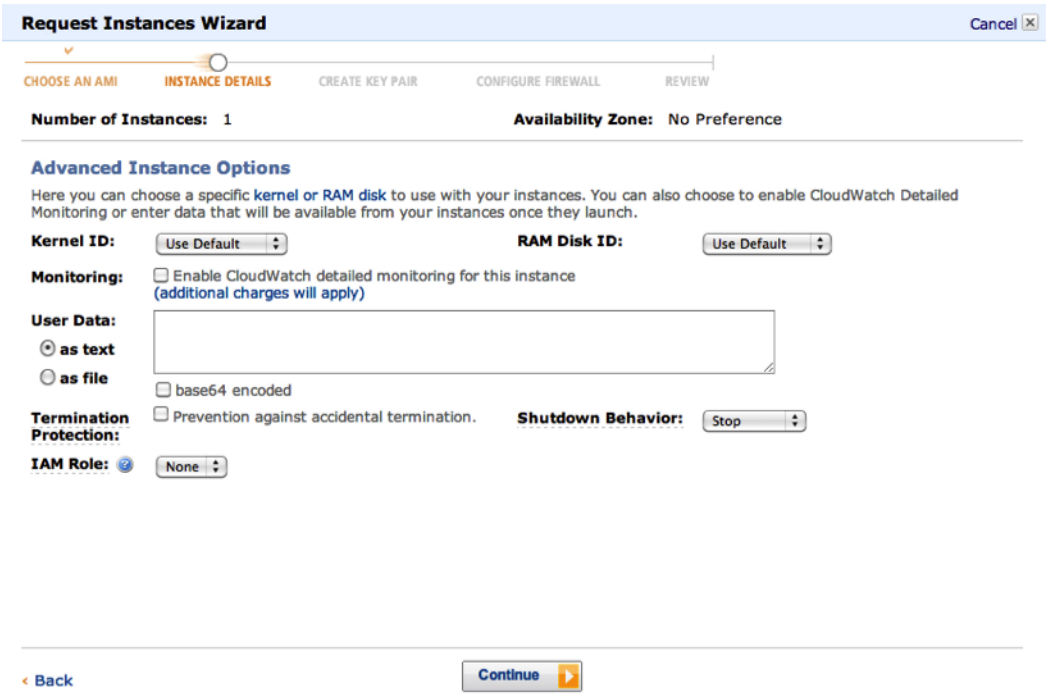
Check the check box next to the latest Pegasus Tutorial VM and click the “Launch” button. The "Request Instances Wizard" will pop up:

Figure A.12. Request Instances Wizard: Step 1



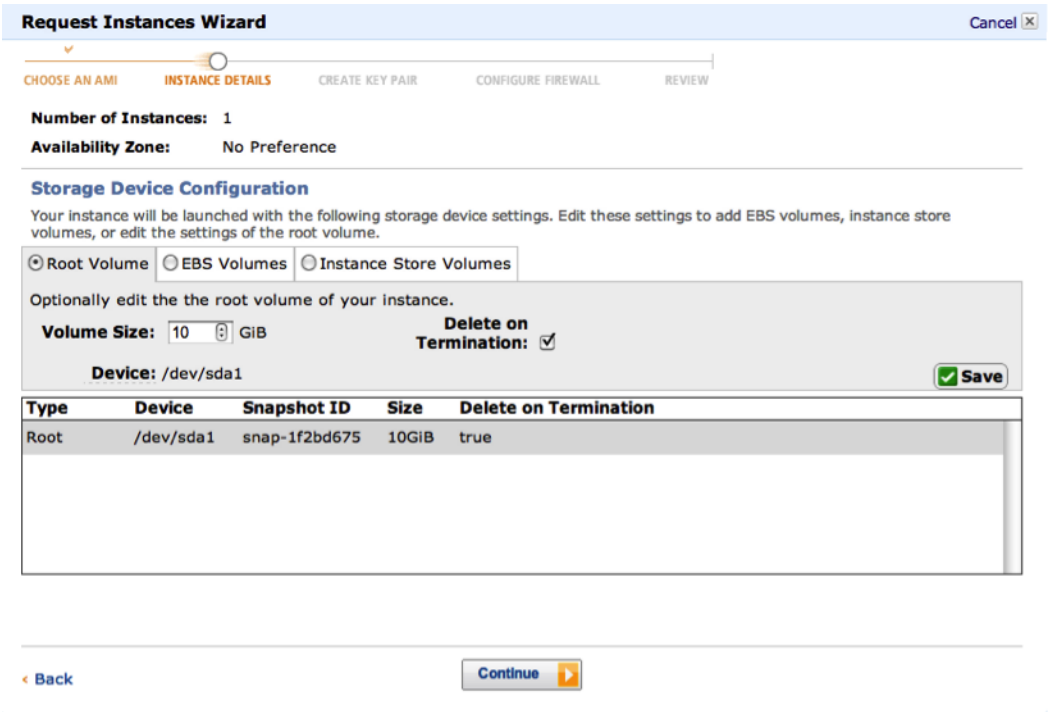
In the first step of the Request Instances Wizard choose the “Large” instance type and click “Continue”:

Figure A.13. Request Instances Wizard: Step 2



Don’t change anything on the “Advanced Instance Options” step and click “Continue”:

Figure A.14. Request Instances Wizard: Step 3



On the “Storage Device Configuration” step make sure “Delete on Termination” is set to "true", then click “Continue”:

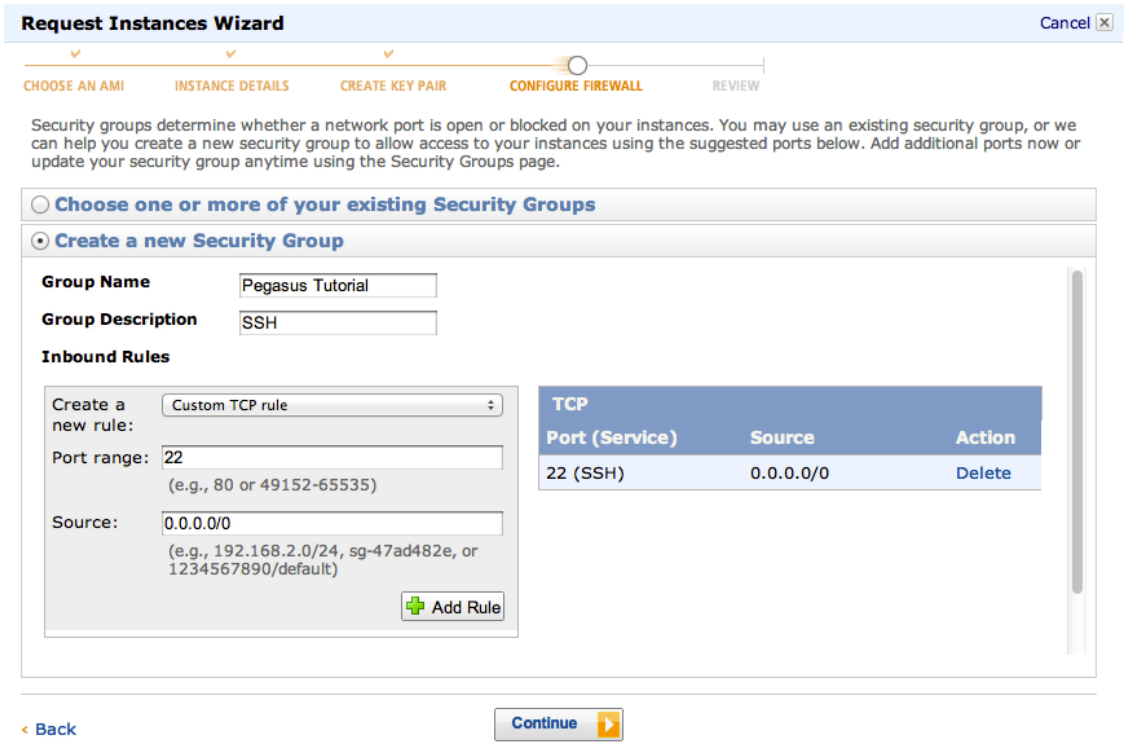
Figure A.15. Request Instances Wizard: Step 4

On the next step type “Pegasus Tutorial” into the “Value” field and click “Continue”:

Figure A.16. 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.17. Request Instances Wizard: Step 6



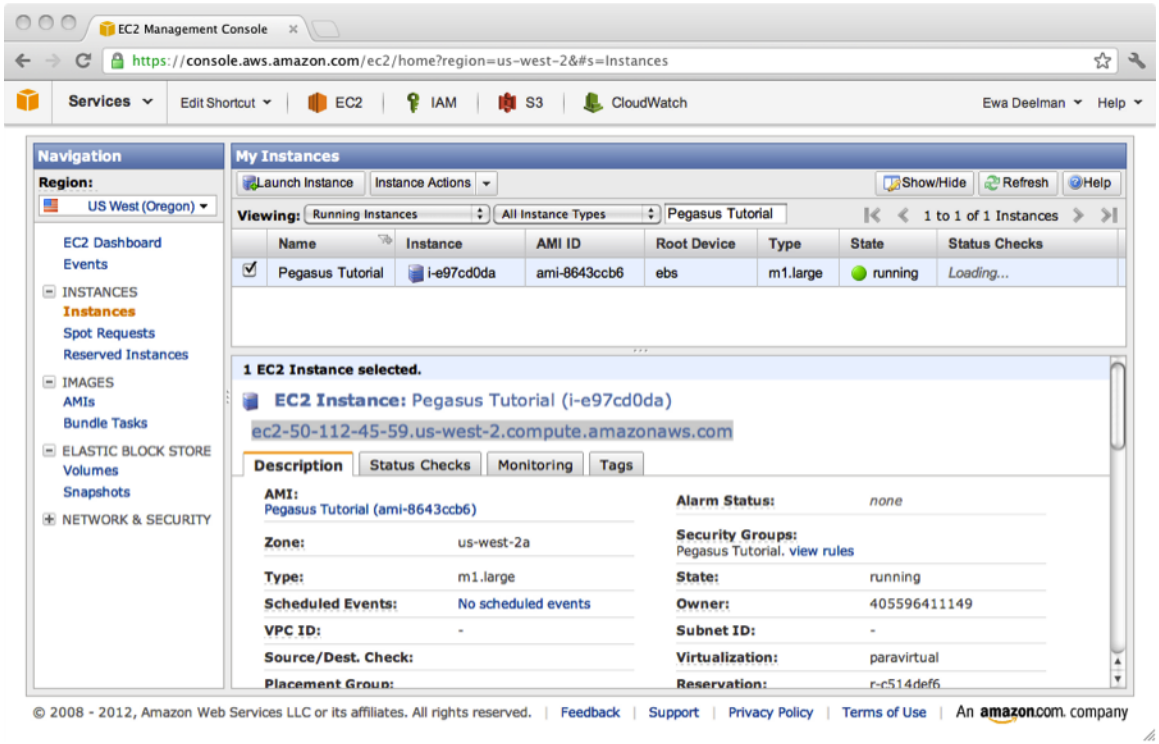
On the next page choose “Create a new Security Group”. Name the security group “Pegasus Tutorial” and give it a description. Create an inbound TCP rule to allow connections on port 22 (SSH) from source 0.0.0.0/0 and click "Add Rule". Then click “Continue”.

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.18. Request Instances Wizard: Step 7

On the last step of the wizard validate your selections and click “Launch”.

Figure A.19. 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's password: pegasus
[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.

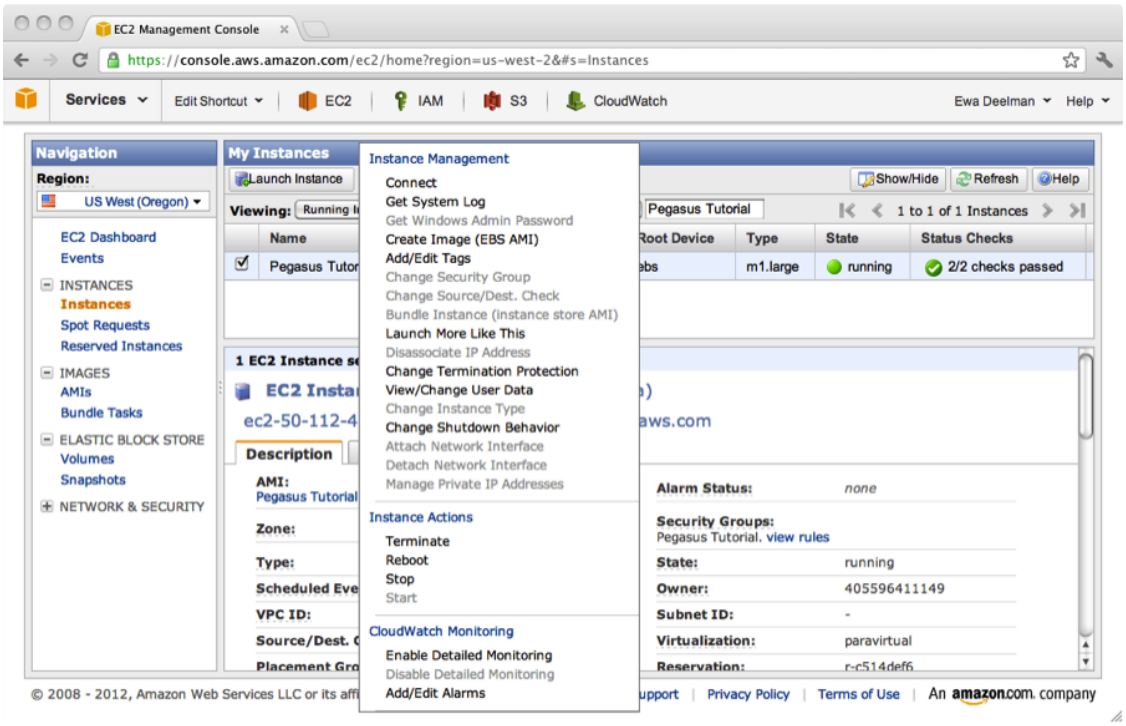
If you are on Windows you will need to install an SSH client. You can download the PuTTY SSH client and find documentation for how to configure it here: http://www.chiark.greenend.org.uk/~sgtatham/putty

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.

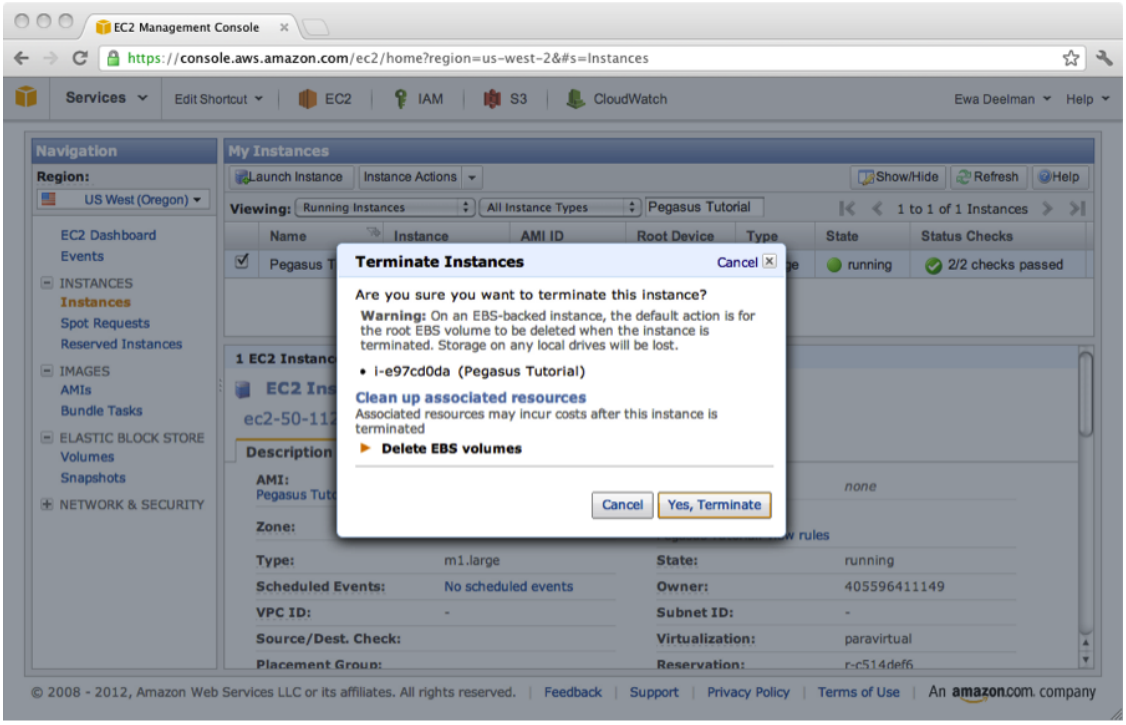
To terminate the VM click on “Instances” link on the left side of the AWS Management Console, check the box next to the “Pegasus Tutorial” VM, and click “Instance Actions”-->“Terminate”:

Figure A.20. Terminate Instance



Then click "Yes, terminate":

Figure A.21. Yes, Terminate Instance



FutureGrid

The FutureGrid Project (https://portal.futuregrid.org) is a cloud computing testbed supported by the National Science Foundation. It consists of a collection of computational, networking, and storage resources located throughout the United States. The goal of the FutureGrid Project is to study the behavior and usefulness of cloud computing technologies. It provides a platform where researchers can experiment with different cloud technologies.

Getting Started

The Pegasus Tutorial VM has been deployed to the FutureGrid "India" site using OpenStack. In order to launch the VM, you will need to have a FutureGrid account. You can get one by going to https://portal.futuregrid.org and either joining an existing project, or starting a new project.

If you are not familiar with using OpenStack on FutureGrid, we recommend that you review the "Using OpenStack on FutureGrid" tutorial found here: https://portal.futuregrid.org/tutorials/openstack.

Launching the VM

First, log into the India site using your FutureGrid username and password:

```
$ ssh USERNAME@india.futuregrid.org
```

If you have not already done so, source the `novarc` file that contains your OpenStack credentials:

```
$ source ~/.futuregrid/openstack/novarc
```

Also, load the euca2ools module to add them to your environment:

```
$ module load euca2ools
```

Next, query OpenStack to find the latest Pegasus Tutorial VM image:

```
$ euca-describe-images | grep PegasusTutorialVM
IMAGE ami-0000003e juve/PegasusTutorialVM-4.0.1.fg.manifest.xml available public x86_64 ...
```

Find the image ID (ami-0000003e in the example above). If you get multiple results, use the latest version.

Launch the tutorial VM using the `euca-run-instances` command with the image ID you found in the previous step:

```
$ euca-run-instances ami-0000003e
RESERVATION    r-y9ue0rs7    461884eef90047fbb4eb9ec92f22a1e3   default
INSTANCE    i-00000c38    ami-0000003e    server-3128    server-3128    pending    ...
```

Note the instance ID (i-00000c38 in the example). Monitor the status of your VM by invoking the `euca-describe-instances` command periodically with the instance ID until the VM status changes from "pending" to "running":

```
$ euca-describe-instances i-00000c38
RESERVATION    r-y9ue0rs7    461884eef90047fbb4eb9ec92f22a1e3    default
INSTANCE    i-00000c38    ami-0000003e    server-3128    server-3128    pending   ...
$ ...
$ euca-describe-instances i-00000c38
RESERVATION     r-y9ue0rs7     461884eef90047fbb4eb9ec92f22a1e3     default
INSTANCE    i-00000c38    ami-0000003e    149.165.158.123    server-3128    running    ...
```

Note down the IP address of the instance (149.165.158.123 in the example). Log into the VM as the tutorial user:

```
$ ssh tutorial@149.165.158.123
```

The password is "pegasus".

At this point you should return to the tutorial chapter and complete the tutorial.

Terminating the VM

Log out of the VM by typing:

```
$ exit
```

Using the instance ID you found in the last section (NOT the image ID), terminate the VM by typing:

```
$ euca-terminate-instances i-00000c38
INSTANCE    i-00000c38
```

When you invoke `euca-describe-instances` you should no longer see the VM running (you should not get any output):

```
$ euca-describe-instances i-00000c38
$
```