Pegasus 3.1 User Guide

Pegasus 3.1 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 New User Walkthrough 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://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator].

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 walkthrough 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.

- **Introduction - Chapter 1**

  (This chapter) An overview of Pegasus WMS and the features it offers, and a brief introduction to workflows.

- **New User Walkthrough - Chapter 2**

  We walk new users through a representative workflow project that includes all aspects of the Pegasus WMS. The project we use is is at a level of complexity that is needed for the vast majority of users.

- **Installation - Chapter 3**

  How to install Pegasus WMS

- **Creating Workflows - Chapter 4**

  Abstract workflow creation and Catalog Details

- **Running Workflows - Chapter 5**

  How to plan a DAX workflow and submit it to the DAG to create an executable workflow.

- **Submit Directory Details - Chapter 6**

  Description of the workflow afer it has been planned

- **Monitoring, Debugging, and Statistics - Chapter 7**

  How to monitor, debug, and view statistics about the execution of a workflow

- **Example Workflows - Chapter 8**

  Example workflows are included to assist users with becoming familiar with existing workflows to gain insight into creating their own

- **Reference Manual - Chapters 9**

  Advanced information of configuration details for use in designing advanced workflows for very large scale projects along with API details and a listing of all commands available

- **Useful Tips - Chapter 10**

  View useful tips, best practices, and Version 2.x upgrade instructions.

- **Glossary - Chapter 11**

  View a glossary of terms and concepts referenced in Pegasus WMS

Chapter 2. New User Walkthrough

Walkthrough Objectives

This walkthrough is intended for new users who want so get a quick overview of the Pegasus concepts and system. A preconfigured virtual machine is provided so that no software installation (except Virtal Box) is required. The walkthrough covers creating a workflow, submitting, monitoring, debugging, and generating run statistics. As concepts and tools are introduced, links to the main Pegasus documentation are provided.

Virtual Box Pegasus VM

Note

Virtual Box is required to run the virtual machine on your computer. If you do not already have it installed, download the binary version desired and install it from the Virtual Box Website [http://www.virtualbox.org/wiki/Downloads]

Download the corresponding disk image.

Virtual Box Pegasus Image [http://pegasus.isi.edu/wms/download/3.1/Pegasus-3.1.0-Debian-6-x86.vbox.zip]

It is around **600 MB** in size. The Image is in zip format. You will need to unzip it.

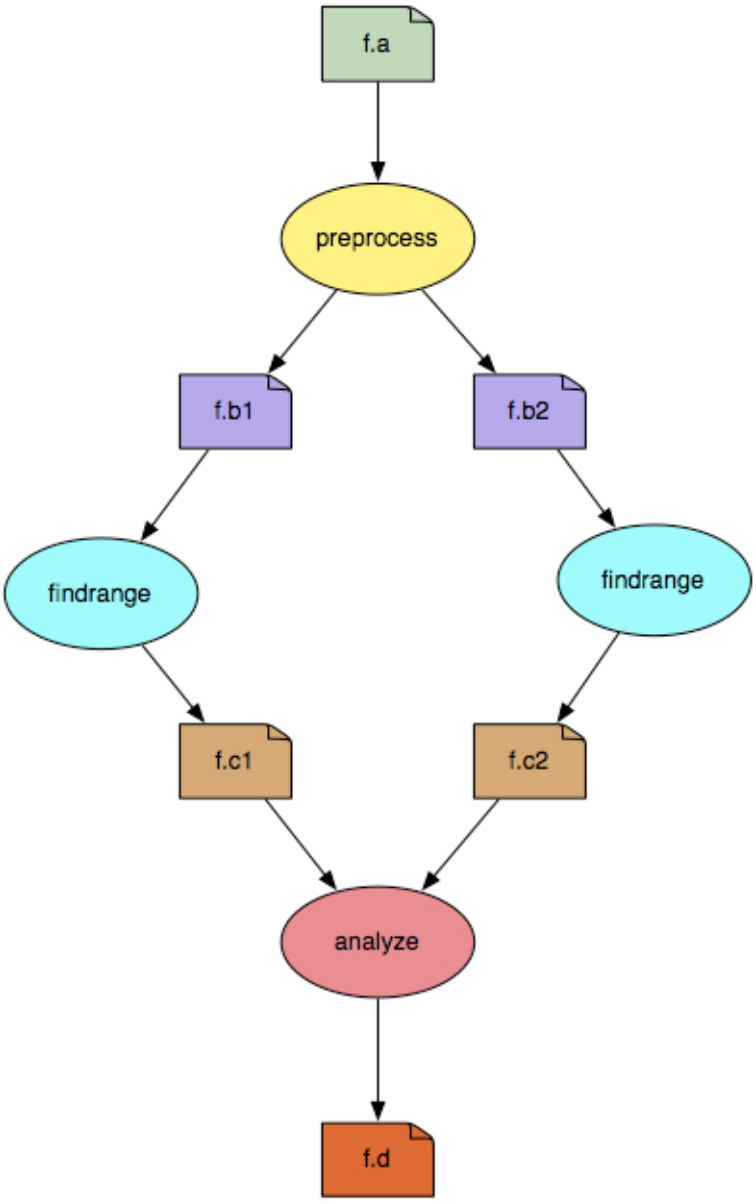
After untarring a folder named **Pegasus-3.1-Debian-6-x86.vbox** will be created that has the vmdk files for the VM. Load this VM using Virtual Box. Once you see the simple Linux desktop, move on to the next step.

Creating the Workflow (DAX)

Pegasus takes in an abstract workflow (DAX) and generates an executable workflow (DAG) that is run in an environment. For the purposes of this walkthrough, we will demonstrate the characteristics and structure of workflows by generating a workflow with a bit of Python code that uses the DAX API to generate a DAX. For a detailed description of workflows, how to create them, and how they are used in Pegasus see Creating Workflows

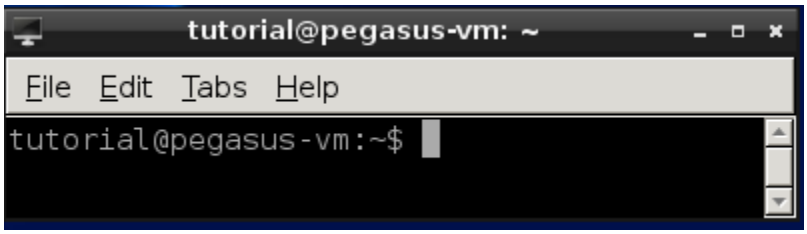
The workflow we will be creating is called Black Diamond because its shape. It is a made up workflow which has 4 jobs, files (f.*) passed between the jobs, and it is an interesting example as it shows data and job dependencies. The workflow looks like:

Figure 2.1. Figure: Black Diamond DAX



To create the DAX, open a new terminal:

Figure 2.2. Terminal Window



All the exercises in this Chapter will be run from the **$HOME/walkthrough/** directory . All the files that are required reside in this directory.

Change the directory to **$HOME/walkthrough:**

```
$ cd $HOME/walkthrough
```

The piece of code which generates the DAX is called a DAX generator, and in this case the code is written in Python. APIs are also available for Java and Perl, and if that does not fit in your tool chain, you can also write the DAX XML directly.

Open the file **create_diamond_dax.py**

```
$ nano create_diamond_dax.py
```

The code has 6 logical sections:

1. **Imports and Pegasus location setup.** We need to know the location of Pegasus because we need to import the DAX3 Pegasus Python module, and we need to know where on the file system the **keg** executable is.

2. **A new abstract dag (DAX) is created.** This is the main DAX object that we will add data, jobs and flow information to.

3. **A replica catalog is set up.** Replica catalogs tell Pegasus where to find data. In this example workflow, we only have one input, f.a, and thus the replica catalog only has one entry. For larger workflows, it is not uncommon to have thousands of entries in the replica catalog, and sometimes mulitple physical locations for the same logical filename so that Pegasus can pick which replica to use. Replica catalogs can either be included in the DAX (like in this example) or be standalone files or services. For more information about replica catalogs, see the Data Discovery chapter.

4. **Executables are added.** Just like we the replica catalog informs Pegasus on where to find data, the transformation catalog tells Pegasus where to find the executables for the workflow. The transformation catalog can exist inside the DAX (as in this example) or in a standalone file. You can list the same logical exectuable existing on multiple resources, and Pegasus will pick the appropiate one when the workflow is planned. More information can be found in the Executable Discovery chapter.

5. **Jobs are added.** The 4 jobs in the Black Diamond in the picture above are added. Arguments are defined, and **uses** clauses are added to list input and output files. This is an imporant step, as it allows Pegasus to track the files, and stage the data if necessary.

6. **Control flows are set up.** This is the edges in the picture, and defines parent/child relationships between the jobs. When the workflow is executing, this is the order the jobs will run in.

Close the file and execute it giving the location of the Pegasus install as argument:

```
$ ./create_diamond_dax.py /opt/pegasus/default
```

The output is the DAX XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated: 2011-07-19 12:59:14.617059 -->
<!-- generated by: tutorial -->
<!-- generator: python -->
<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">
        <file name="f.a">
                <pfn url="file:///home/tutorial/walkthrough/f.a" site="PegasusVM"/>
        </file>
        <executable name="preprocess" namespace="diamond" version="4.0"
                    arch="x86_64" os="linux" installed="true">
                <pfn url="file:///opt/pegasus/default/bin/keg" site="PegasusVM"/>
        </executable>
        <executable name="analyze" namespace="diamond" version="4.0"
                    arch="x86_64" os="linux" installed="true">
                <pfn url="file:///opt/pegasus/default/bin/keg" site="PegasusVM"/>
        </executable>
        <executable name="findrange" namespace="diamond" version="4.0"
```

```
                arch="x86_64" os="linux" installed="true">
                <pfn url="file:///opt/pegasus/default/bin/keg" site="PegasusVM"/>
        </executable>
        <job id="ID0000001" namespace="diamond" name="preprocess" version="4.0">
                <argument>-a preprocess -T60 -i <file name="f.a"/>
                        -o <file name="f.b1"/> <file name="f.b2"/></argument>
                <uses name="f.b1" link="output" executable="false"/>
                <uses name="f.a" link="input" executable="false"/>
                <uses name="f.b2" link="output" executable="false"/>
        </job>
        <job id="ID0000002" namespace="diamond" name="findrange" version="4.0">
                <argument>-a findrange -T60 -i <file name="f.b1"/>
                        -o <file name="f.c1"/></argument>
                <uses name="f.c1" link="output" executable="false"/>
                <uses name="f.b1" link="input" executable="false"/>
        </job>
        <job id="ID0000003" namespace="diamond" name="findrange" version="4.0">
                <argument>-a findrange -T60 -i <file name="f.b2"/>
                        -o <file name="f.c2"/></argument>
                <uses name="f.c2" link="output" executable="false"/>
                <uses name="f.b2" link="input" executable="false"/>
        </job>
        <job id="ID0000004" namespace="diamond" name="analyze" version="4.0">
                <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" executable="false"/>
                <uses name="f.d" link="output" register="true" executable="false"/>
                <uses name="f.c1" link="input" executable="false"/>
        </job>
        <child ref="ID0000002">
                <parent ref="ID0000001"/>
        </child>
        <child ref="ID0000003">
                <parent ref="ID0000001"/>
        </child>
        <child ref="ID0000004">
                <parent ref="ID0000002"/>
                <parent ref="ID0000003"/>
        </child>
</adag>
```

We need the DAX in a file to give to Pegasus, so run the same command again, but redirect it to file:

```
$ ./create_diamond_dax.py /opt/pegasus/default > diamond.xml
```

More information about creating workflows can be found in the Creating Workflows chapter.

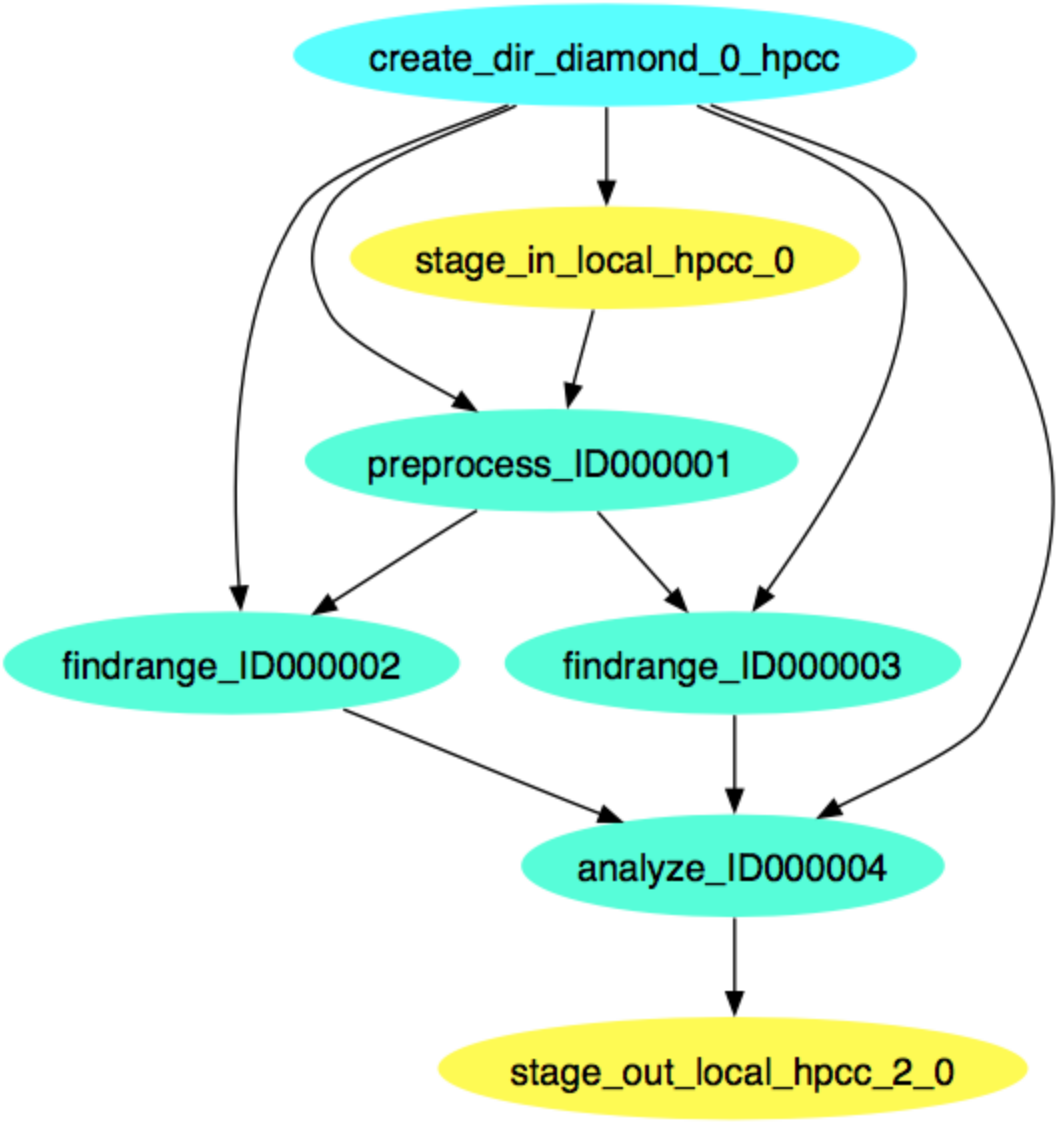
Submitting the Workflow

Submitting a Pegasus workflow consists of two steps, planning and submitting, but are often made by one single command for convenience. However, the planning stage is where Pegasus is doing powerful transformations to your workflow, so it is important to at least have an idea on what is happening under the covers. Planning includes, but is not limited to:

1. Adding remote create dir jobs
2. Adding stage in jobs to transfer data into the remote work directory
3. Adding cleanup jobs to clean up the work directory as the workflow progresses
4. Adding stage out jobs to transfer data to the final output location
5. Adding registration jobs to register the data in a replica catalog
6. Clusters job together - useful if you have many short tasks
7. Adds wrappers to the jobs to collect provenance information - this is so statistics and plots can be created after a run

In the case of our black diamond workflow, here is what it looks like after the Pegasus planner has processed the DAX:

Figure 2.3. Figure: Black Diamond DAG Image



To plan and submit the workflow, run:

```
$ pegasus-plan --conf pegasusrc --sites PegasusVM --dir runs --output local --dax diamond.xml --
submit
```

The output will look something like:

```
Submitting job(s).
1 job(s) submitted to cluster 18.

------------------------------------------------------------------------
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
------------------------------------------------------------------------

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

cd /home/tutorial/walkthrough/runs/tutorial/pegasus/diamond/run0001
```

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

pegasus-status -l /home/tutorial/walkthrough/runs/tutorial/pegasus/diamond/run0001

*** To remove your workflow run ***
pegasus-remove /home/tutorial/walkthrough/runs/tutorial/pegasus/diamond/run0001
```

Tip

The work directory created by Pegasus is where the concrete workflow exists, and the directory is also the handle for Pegasus commands acting on that instance. Using this handle will be covered in the next section.

Further information about planning and submitting workflows can be found in the Running Workflows chapter. Information about the files in the work directory can be found in the Submit Directory Details chapter.

Monitoring, Debugging and Statistics

Once the workflow has been submitted, you can check status of it with the pegasus-status tool. Use the directory handle (which is different for every workflow you submit) from the previous step and run it with the -l flag:

```
$ pegasus-status -l /home/tutorial/walkthrough/runs/tutorial/pegasus/diamond/run0001
STAT  IN_STATE  JOB
Run      02:59  diamond-0
Run      00:34   |-findrange_ID0000002
Run      00:32   |-findrange_ID0000003
Idle     00:19   \_clean_up_preprocess_ID0000001
Summary: 4 Condor jobs total (I:1 R:3)

UNRDY READY   PRE  IN_Q  POST  DONE  FAIL %DONE STATE   DAGNAME
    8     0     0     4     0     4     0  25.0 Running *diamond-0.dag
Summary: 1 DAG total (Running:1)
```

The first section shows jobs released to be handled by Condor. The second section shows a summary of the state of the workflow. Keep on checking the workflow with pegasus-status until it is 100% done.

Once the workflow has finished, you may use the pegasus-analyzer to debug the workflow. This is obviously most useful when workflows have failed for some reason. pegasus-analyzer will show you which jobs failed and the output of those jobs. Our simple black diamond should finish successfully, and pegasus-analyzer output should look like:

```
$ pegasus-analyzer --dir=/home/tutorial/walkthrough/runs/tutorial/pegasus/diamond/run0001
pegasus-analyzer: initializing...

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

 Total jobs         :     15 (100.00%)
 # jobs succeeded   :     15 (100.00%)
 # jobs failed      :      0 (0.00%)
 # jobs unsubmitted :      0 (0.00%)

************************************Done****************************************

pegasus-analyzer: end of status report
```

To get detailed run statistics, use the pegasus-statistics tool:

```
$ pegasus-statistics /home/tutorial/walkthrough/runs/tutorial/pegasus/diamond/run0001

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 workflow.

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             Unsubmitted         Total
Tasks              4                  0                  0                   4
Jobs               15                 0                  0                   15
Sub Workflows      0                  0                  0                   0
------------------------------------------------------------------------------
Workflow wall time                               : 5 mins, 35 secs, (total 335 seconds)

Workflow cumulative job wall time                : 4 mins, 1 sec,   (total 241 seconds)

Cumulative job walltime as seen from submit side : 4 mins, 2 secs,  (total 242 seconds)
```

More information about monitoring, debugging and statistics can be found in the Monitoring, Debugging and Statistics chapter.

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
```

- **Condor 7.4 or higher**. See http://www.cs.wisc.edu/condor/ 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
5.0.1
```

Environment

To use Pegasus, add the bin/ directory to your PATH.

```
Example for bourne shells:

  $ export PATH=/some/install/pegasus-3.0.0/bin:$PATH
```

Note

Pegasus 3.0 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.

Native Packages (RPM/DEB)

The preferred way to install Pegasus is with native (RPM/DEB) packages. Using this package, Pegasus is installed into `/opt/pegasus/{major_version}.{minor_version}`, and the packages have {major_version}.{minor_version} in the package name. This allows for multiple installed Pegasus versions installed. For example, if you installed Pegasus 2.4.3 and 3.0.0, you will have: with:

```
/opt/pegasus/2.4/
/opt/pegasus/3.0/
```

RHEL 5 / CentOS 5 / Scientific Linux 5

Add the Pegasus repository to yum by creating a file named **/etc/yum.repos.d/pegasus.repo** with the content:

```
# Pegasus 3.0
name=Pegasus30
baseurl=http://pegasus.isi.edu/wms/download/3.0/yum/rhel/$releasever/$basearch/
gpgcheck=0
enabled=1
```

Search for, and install Pegasus:

```
# yum search pegasus
pegasus-3.0 : Pegasus Workflow Management System
# yum install pegasus
Running Transaction
Installing     : pegasus-3.0

Installed:
pegasus-3.0 :3.0.0-1


Complete!
```

Debian 5 (Lenny)

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 keyring.debian.org --recv-keys 81C2A4AC
# gpg -a --export 81C2A4AC | apt-key add -
```

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

```
deb http://pegasus.isi.edu/wms/download/3.0/apt/debian lenny main
```

Install Pegasus with **apt-get :**

```
# apt-get update
...
# apt-get install pegasus-3.0
```

Pegasus from Tarballs

The Pegaus prebuild tarballs can be downloaded from the *Pegasus Download Page* [http://pegasus.isi.edu/wms/download.php].

Tarball without Condor

Use these tarballs if you already have Condor installed or prefer to keep the Condor 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-3.0.0:$PATH
  ```

Tarball with Included Condor

For convenience, there is Pegasus WMS tarballs which has Condor included. This way you can install Pegasus and Condor at the same time, with just minor configuration to get up and running.

- Untar the tarball

```
$ tar zxf pegasus-wms-*.tar.gz
```

- Edit the Condor Configuration file. The configuration file is currently configured to run only as a submit side (Runs schedd) supporting schedule, local and grid universe. If you want to use it for gt4, lsf, pbs or condor-c additional configuration changes may be required. Please check the Condor Manual for appropriate parameters.

  2 parameters need to change at this point. Change **!!PEGASUS_HOME!!** to the actual path where PEGASUS_WMS is installed and **CHANGE !!USER!!** to the user who will receive email in case of error. (This can be just your username)

  ```
  $ vim $PEGASUS_HOME/etc/condor_config


  RELEASE_DIR = !!PEGASUS_HOME!!    # CHANGE THIS TO PATH OF PEGASUS_WMS INSTALLATION
  WILL GET EMAIL IN CASE OF ERROR.
  ```

- Set up the environment:

  ```
  $ export PATH=PEGASUS_HOME/bin:PEGASUS_HOME/condor/bin:$PATH
  $ export CONDOR_CONFIG=PEGASUS_HOME/etc/condor_config
  ```

- Start Condor by running ./sbin/condor_master

  ```
  $ ./sbin/condor_master
  ```

- Verify that Condor is up by running the **condor-q** command

  ```
  $ condor_q

  -- Submitter: gmehta@smarty.isi.edu : <128.9.72.26:60126> : smarty.isi.edu
   ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD

  0 jobs; 0 idle, 0 running, 0 held
  ```

Basic Configuration Control

Basic system configuration is controlled through the properties settings selected for Pegasus Planner, Pegasus Home, and Catalogs.

Note

Values rely on proper capitalization, unless explicitly noted otherwise.

- Some default property values depend on the value of other properties.
- Curly braces refer to the value of the named property. For instance,**${pegasus.home}** means that the value depends on the value of the pegasus.home property plus any noted additions. You can use this notation to refer to other properties, though the extent of the subsitutions are limited. Usually, you want to refer to a set of the standard system properties.
- Nesting is not allowed.
- Substitutions shall only be done once.
- There is a priority to the order of reading and evaluating properties. Usually there is no need to worry about the priorities. However, it is good to know the details of when which property applies, and how one property is able to overwrite another.
- Property definitions in the system property file, usually found in **${pegasus.home.sysconfdir}/properties** , have the lowest priority. These properties are expected to be set up by the submit host's administrator.
- The properties defined in the user property file **${user.home}/.pegasusrc** have higher priority. These can overwrite settings found in the system's properties.
- Commandline properties have the highest priority. Each commandline property is introduced by a **-D** argument.

Note

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.

For more details see the **Basic Properties** section of the Running Workflows chapter

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-3.2.xsd [http://pegasus.isi.edu/wms/docs/schemas/dax-3.2/dax-3.2.xsd] The documentation of the schema and its elements can be found in dax-3.2.html [http://pegasus.isi.edu/wms/docs/schemas/dax-3.2/dax-3.2.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.

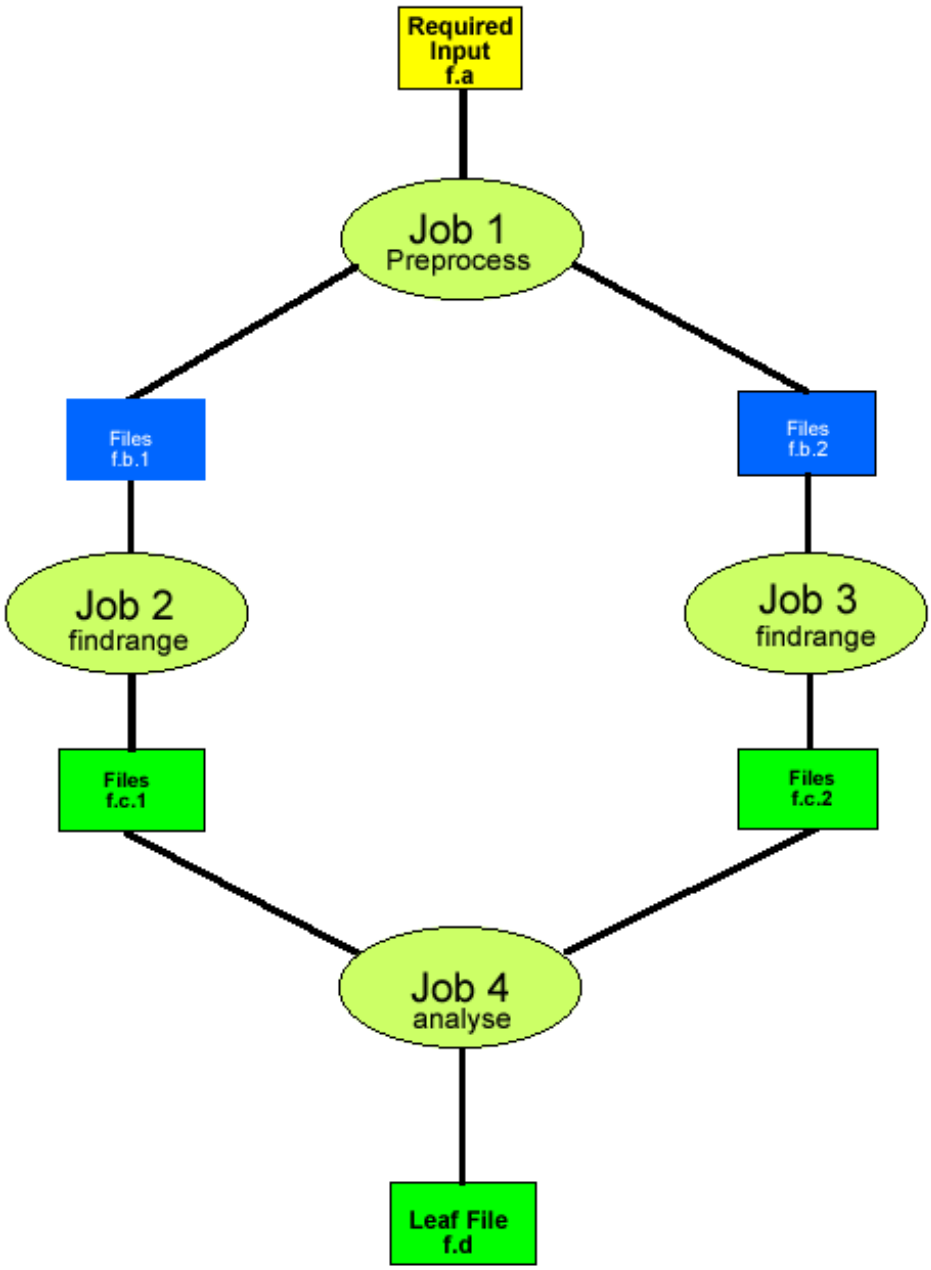
- **fan-in**, **gather**, **join**, and **converge** describe how multiple siblings are merged into fewer dependent child nodes.

  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 3 different implemenations of the Replica Catalog.

1. **File**(Default)

2. **Database via JDBC**
3. **Replica Location Service**
   - **RLS**
   - **LRC**
4. **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>***

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 sh$ cat $HOME ared 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.

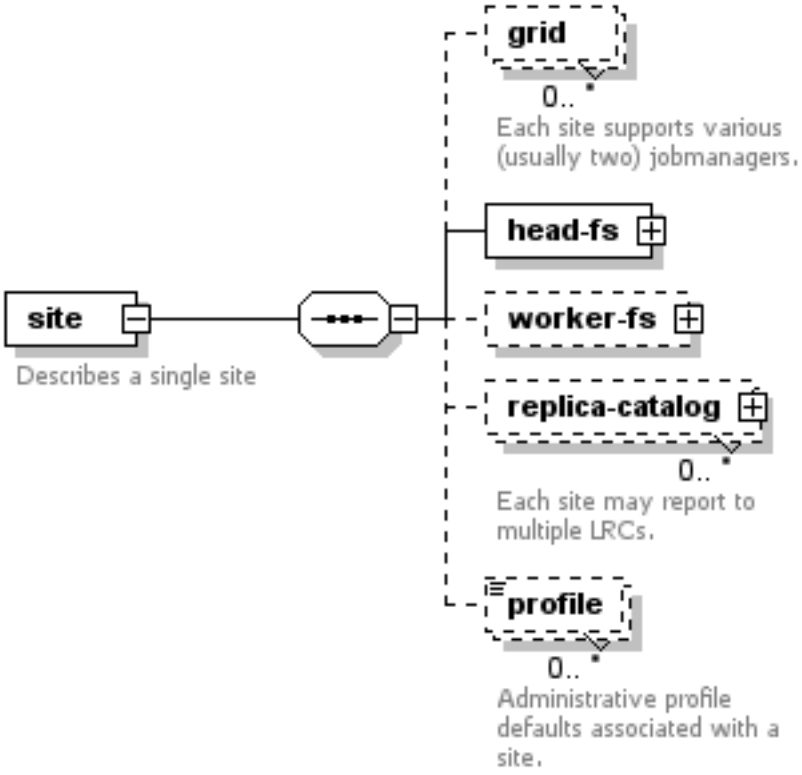
Pegasus currently supports two implementation of the Site Catalog:

1. **XML3**(Default)
2. **XML**(Deprecated)
3. **File**(Deprecated)

XML3

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 4.2. 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=XML3**
2. **pegasus.catalog.site.file=*<path to the site catalog file>***

XML

Warning

This format is now deprecated in favor of the XML3 format. If you are still using the XML or File format you should convert it to XML3 formation using the pegasus-sc-converter client

```
$ cat $HOME/sites.xml

<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog"
  xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog
  http://pegasus.isi.edu/schema/sc-2.0.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0">
  <site handle="local" gridlaunch="/nfs/vdt/pegasus/bin/kickstart"
   sysinfo="INTEL32::LINUX">
    <profile namespace="env" key="PEGASUS_HOME" >/nfs/vdt/pegasus</profile>
    <profile namespace="env" key="GLOBUS_LOCATION" >/vdt/globus</profile>
    <profile namespace="env" key="LD_LIBRARY_PATH" >/vdt/globus/lib</profile>
    <profile namespace="env" key="JAVA_HOME" >/vdt/java</profile>
    <lrc url="rlsn://localhost" />
    <gridftp  url="gsiftp://localhost" storage="/$HOME/storage" major="4" minor="0"
     patch="5">
    </gridftp>
    <jobmanager universe="transfer" url="localhost/jobmanager-fork" major="4" minor="0"
     patch="5" />
    <jobmanager universe="vanilla" url="localhost/jobmanager-fork" major="4" minor="0"
     patch="5" />
    <workdirectory >$HOME/workdir</workdirectory>
  </site>
  <site handle="clus1" gridlaunch="/opt/nfs/vdt/pegasus/bin/kickstart"
   sysinfo="INTEL32::LINUX">
    <profile namespace="env" key="PEGASUS_HOME" >/opt/nfs/vdt/pegasus</profile>
    <profile namespace="env" key="GLOBUS_LOCATION" >/opt/vdt/globus</profile>
    <profile namespace="env" key="LD_LIBRARY_PATH" >/opt/vdt/globus/lib</profile>
    <lrc url="rlsn://clus1.com" />
    <gridftp  url="gsiftp://clus1.com" storage="/jobmanager-fork" major="4" minor="0"
     patch="3">
    </gridftp>
    <jobmanager universe="transfer" url="clus1.com/jobmanager-fork" major="4" minor="0"
     patch="3" />
    <jobmanager universe="vanilla" url="clus1.com/jobmanager-pbs" major="4" minor="0"
     patch="3" />
    <workdirectory >$HOME/workdir-clus1</workdirectory>
  </site>
</sitecatalog>
```

1. **site** - A site identifier.
2. **lrc** - URL for a local replica catalog (LRC) to register your files in. Only used for RLS implementation of the RC
3. **workdirectory** - A remote working directory (Should be on a shared file system)
4. **gridftp** - A URL prefix for a remote storage location. and a path to the storage location
5. **jobmanager** - Url to the jobmanager entrypoints for the remote grid. Different universes are supported which map to different batch jobmanagers.

   "vanilla" for compute jobs and "transfer" for transfer jobs are mandatory. Generally a transfer universe should map to the fork jobmanager.

6. **gridlaunch** - Path to the remote kickstart tool (provenance tracking)

7. **sysinfo** - The arch/os/osversion/glibc of the site. The format is ARCH::OS:OSVER:GLIBC where OSVERSION and GLIBC 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 sysinfo if none specified is INTEL32::LINUX

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

   Profiles such as the environments to be set on a remote pool.

To use this format you need to set the following properties

1. **pegasus.catalog.site=XML**

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

Text

Warning

This format is now deprecated in favor of the XML3 format. If you are still using the File format you should convert it to XML3 format using the client pegasus-sc-converter

The format for the File is as follows

```
site site_id {
  #required. Can be a dummy value if using Simple File RC
  lrc "rls://someurl"

  #required on a shared file system
  workdir "path/to/a/tmp/shared/file/sytem/"

  #required one or more entries
  gridftp "gsiftp://hostname/mountpoint&rdquor; "GLOBUS VERSION"

  #required one or more entries
  universe transfer "hostname/jobmanager-<scheduler>" "GLOBUS VERSION"

  #reqired one or more entries
  universe vanilla "hostname/jobmanager-<scheduler>" "GLOBUS VERSION"

  #optional
  sysinfo  "ARCH::OS:OSVER:GLIBC"

  #optional
  gridlaunch "/path/to/gridlaunch/executable"

  #optional zero or more entries
  profile namespace "key" "value"
}
```

The gridlaunch and profile entries are optional. All the rest are required for each pool. Also the transfer and vanilla universe are mandatory. You can add multiple transfer and vanilla universe if you have more then one head node on the cluster. The entries in the Site Catalog have the following meaning:

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

2. **lrc** - URL for a local replica catalog (LRC) to register your files in. Only used for RLS implementation of the RC

3. **workdir** - A remote working directory (Should be on a shared file system)

4. **gridftp** gridftp - A URL prefix for a remote storage location.

5. **universe** - Different universes are supported which map to different batch jobmanagers.

"vanilla" for compute jobs and "transfer" for transfer jobs are mandatory. Generally a transfer universe should map to the fork jobmanager.

6. **gridlaunch** - Path to the remote kickstart tool (provenance tracking)

7. **sysinfo** - The arch/os/osversion/glibc of the site. The format is ARCH::OS:OSVER:GLIBC where OSVERSION and GLIBC are optiona.

   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 sysinfo if none specified is INTEL32::LINUX

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

   Profiles such as the environments to be set on a remote pool.

To use this format you need to set the following properties:

1. **pegasus.catalog.site=Text**

2. **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 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 Catalog Properties section of Running Workflows.

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

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

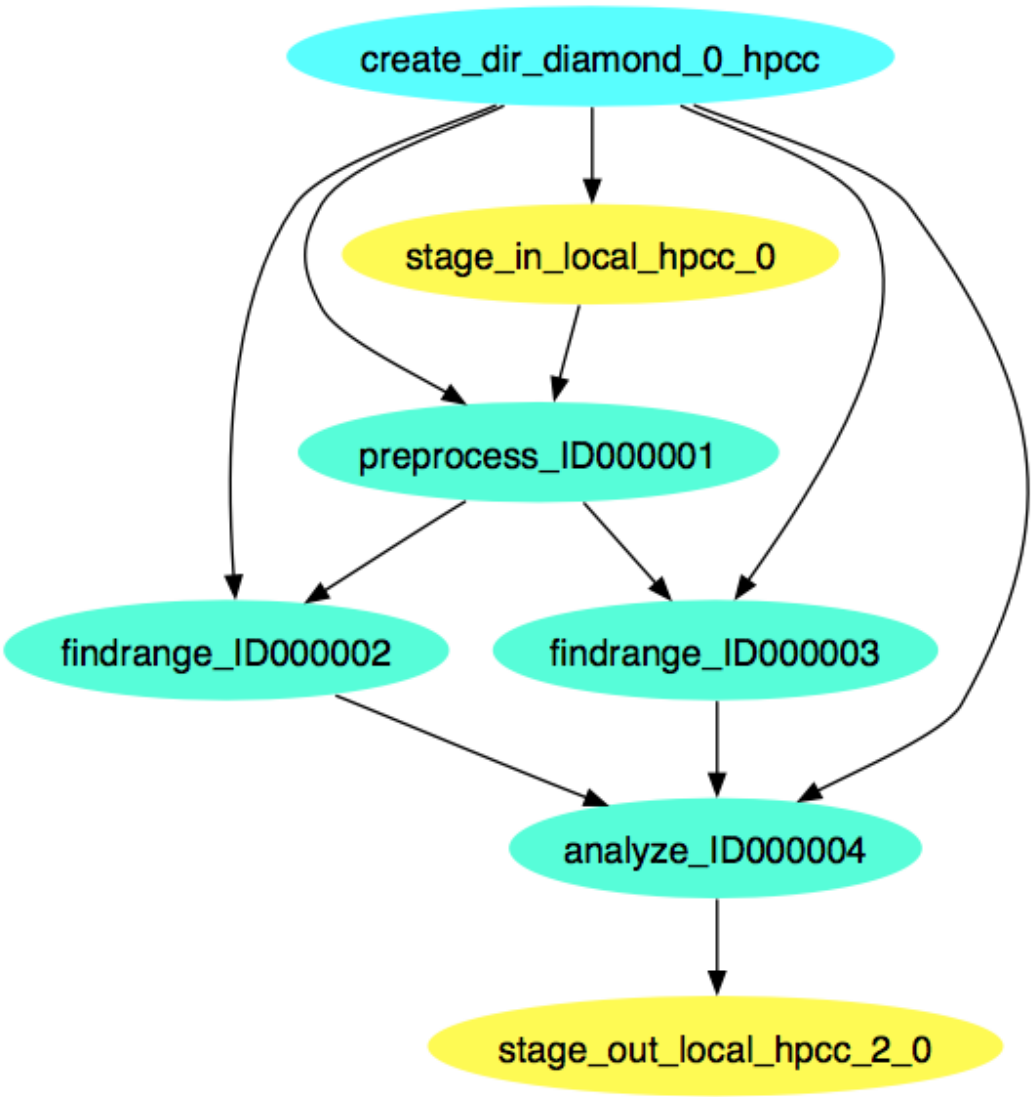
Chapter 5. Running Workflows

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.

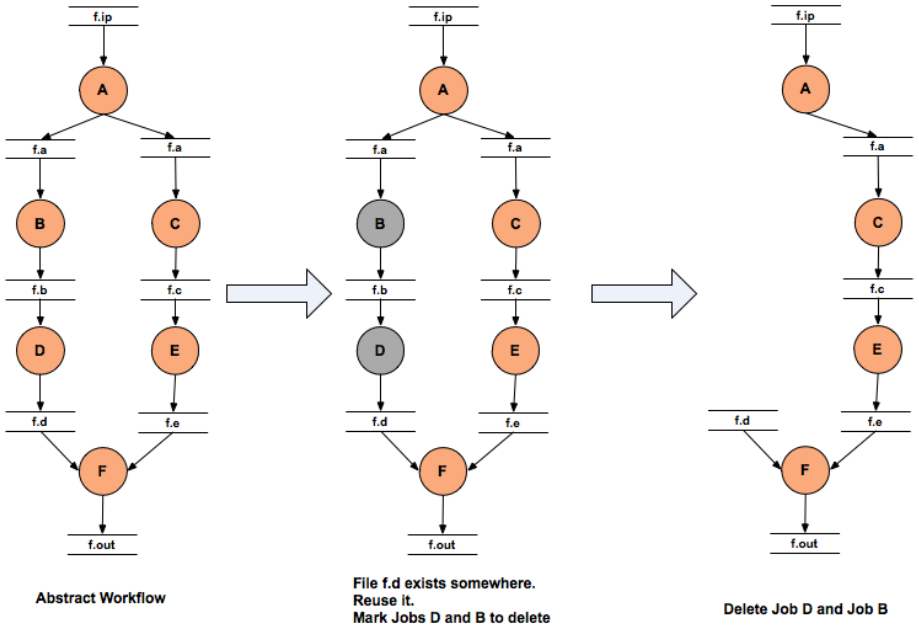
Second Pass - The algorithm removes the job whose output files exist in the Replica Catalog and tries to cascade the deletion upwards to the parent jobs. We start the breadth first traversal of the workflow bottom up.

```
( 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.

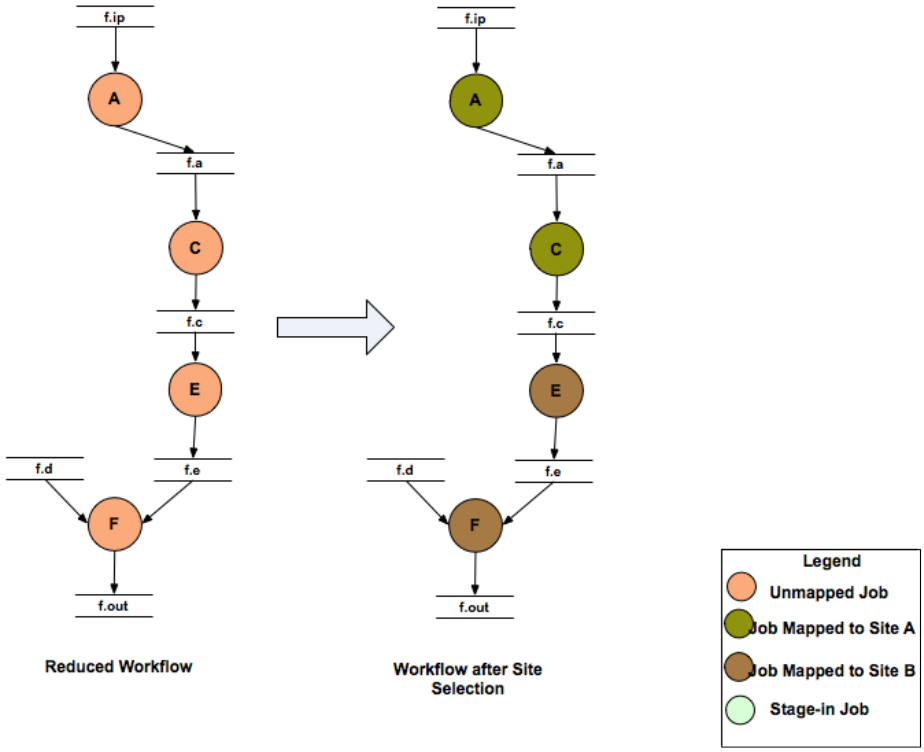
Table 5.1. Table 1: Key Value Pairs that are currently generated for the site selector temporary file that is generated in the NonJavaCallout.

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 Manualchapter.

Tip

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

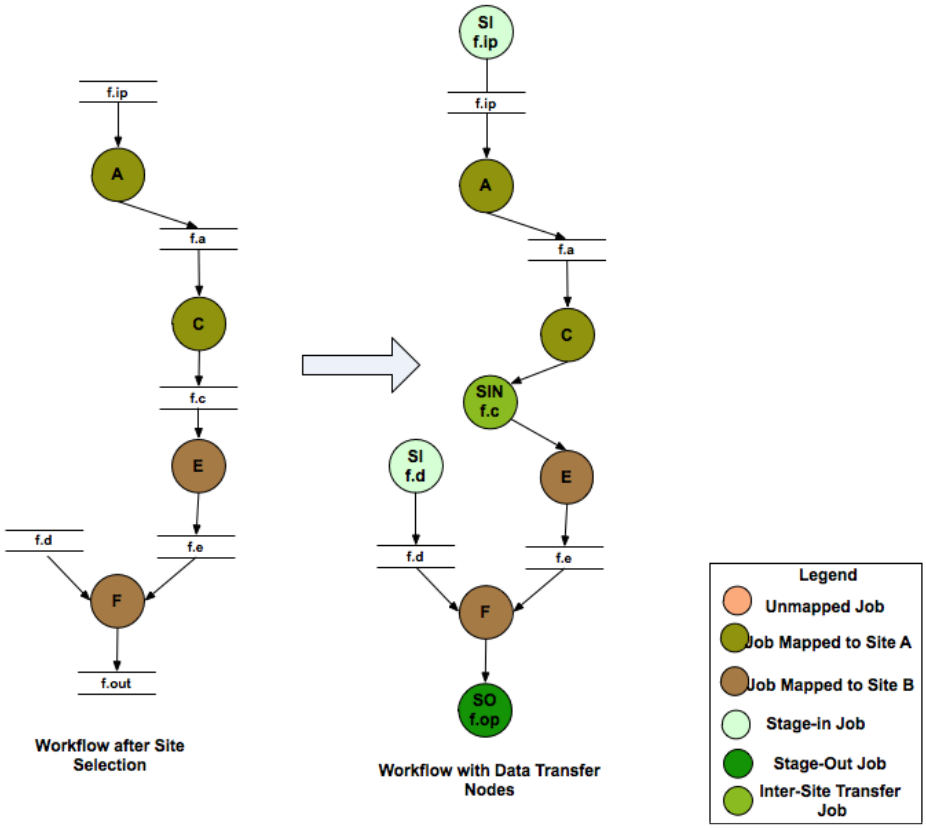
Addition of Data Transfer and Registration Nodes

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 execution site where the job executes. 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.

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

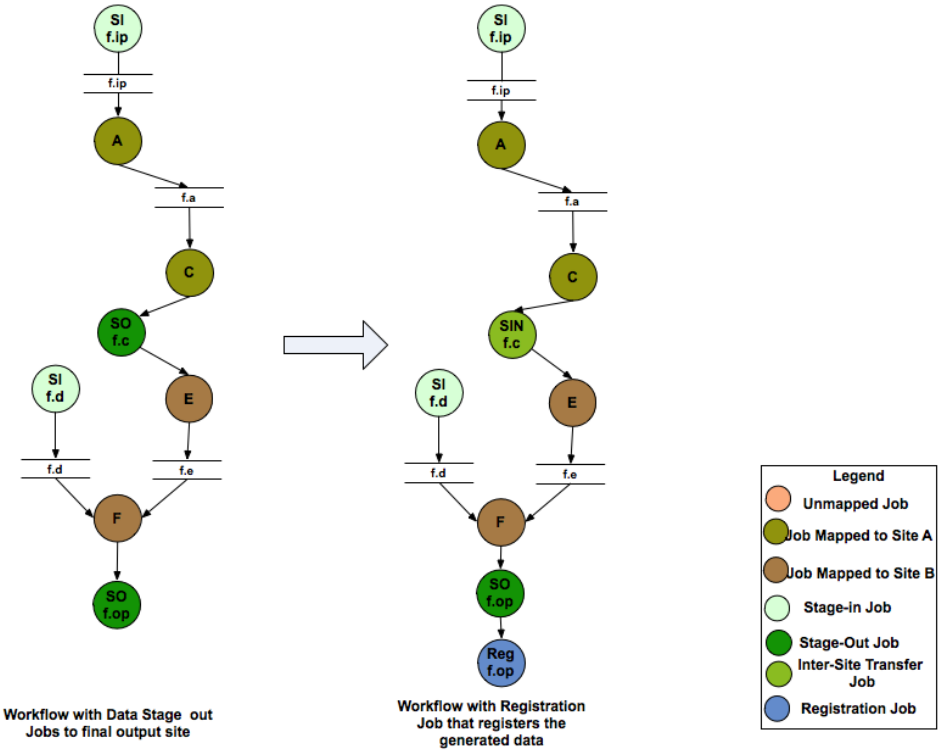
transfer clients like globus-url-copy, lcg-copy, wget, cp, ln . It looks at source and destination url and figures out automatically which underlying client to use. pegasus-transfer is distributed with the PEGASUS and can be found in the bin subdirectory . Pegasus Transfer Refiners are are described 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

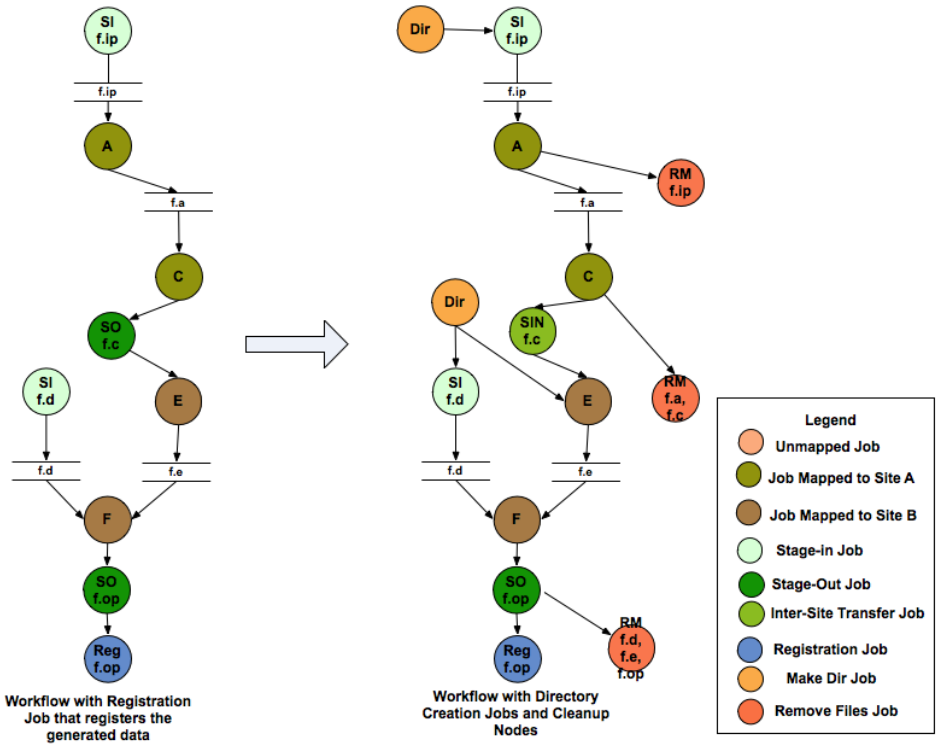
After the data transfer nodes have been added to the workflow, Pegasus adds a create dir jobs to the workflow. Pegasus usually , creates one workflow specific directory per compute site , that is on the shared filesystem of compute site. This directory is visible to all the worker nodes and that is where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

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



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.

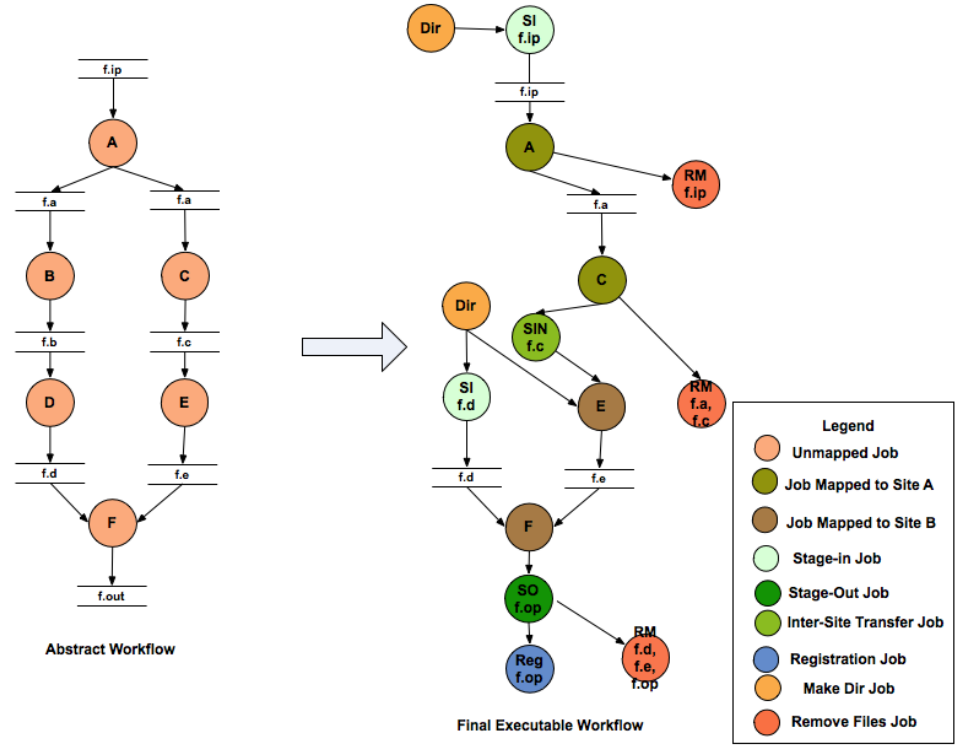
2. **Shell**

   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



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
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.transformation       Text
pegasus.catalog.transformation.file  ${pegasus.home}/etc/sample.tc.text
pegasus.catalog.site                 XML3
pegasus.catalog.site.file            ${pegasus.home}/etc/sample.sites.xml3
```

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]:	MRC
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.

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

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

The site catalog file is available in three major flavors: The Text and and XML formats for the site catalog are deprecated. Users can use pegasus-sc-converter client to convert their site catalog to the newer XML3 format.

1. THIS FORMAT IS DEPRECATED. WILL BE REMOVED IN COMING VERSIONS. USE pegasus-sc-converter to convert XML format to XML3 Format. The "XML" format is an XML-based file. The XML format reads site catalog conforming to the old site catalog schema available at http://pegasus.isi.edu/wms/docs/schemas/sc-2.0/sc-2.0.xsd
2. The "XML3" format is an XML-based file. The XML format reads site catalog conforming to the old site catalog schema available at http://pegasus.isi.edu/wms/docs/schemas/sc-3.0/sc-3.0.xsd

pegasus.catalog.site.file

System:	Site Catalog
Since:	2.0
Type:	file location string
Default:	${pegasus.home.sysconfdir}/sites.xml3 | ${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. The default value is dependant on the value specified for the property pegasus.catalog.site . If type of SiteCatalog used is XML3, then sites.xml3 is picked up from sysconfdir else sites.xml

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

Execution Environments

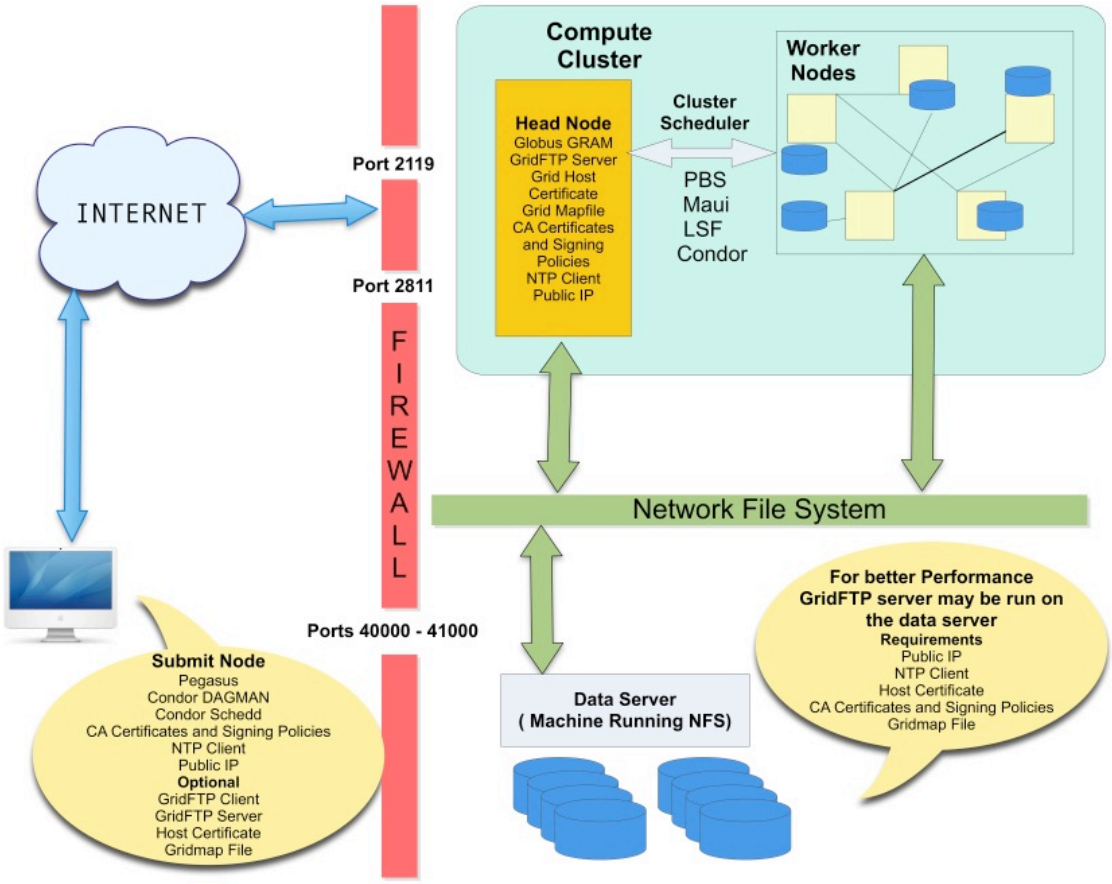
Pegasus supports a number of execution environments. An execution environment is a setup where jobs from a workflow are running.

- The simplest execution environment does not involve Condor. Pegasus is capable of planning small workflows for full-local execution using a shell planner. Please refer to the `examples` directory in your Pegasus installation, the shell planner's documentation section, or the tutorials, for details.
- A slighly more challenging setup is still all-local, but Condor manages queued jobs and Condor DAGMan the workflow. This setup permits limited parallelism with full scalability. With proper setup, Condor can scavenge cycles from unused computers in your department, enabling a pool of more than just a single machine.
- The vanilla setup are workflows to be executed in a grid environment. This setup requires a number of configurations, and an understanding of remote sites.
- Various advanced execution environment setups deal with minimally using Globus to obtain remote resources (Glide-ins), by-passing Globus completely (Condor-C), or supporting cloud computing (Nimbus, Eucalyptus, EC2). You should be familiar with the Grid Execution Environment, as some concept are borrowed.

The Grid Execution Environment

The rest of this chapter will focus on the vanilla grid execution environment.

Figure 5.8. Grid Sample Site Layout



The vanilla 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

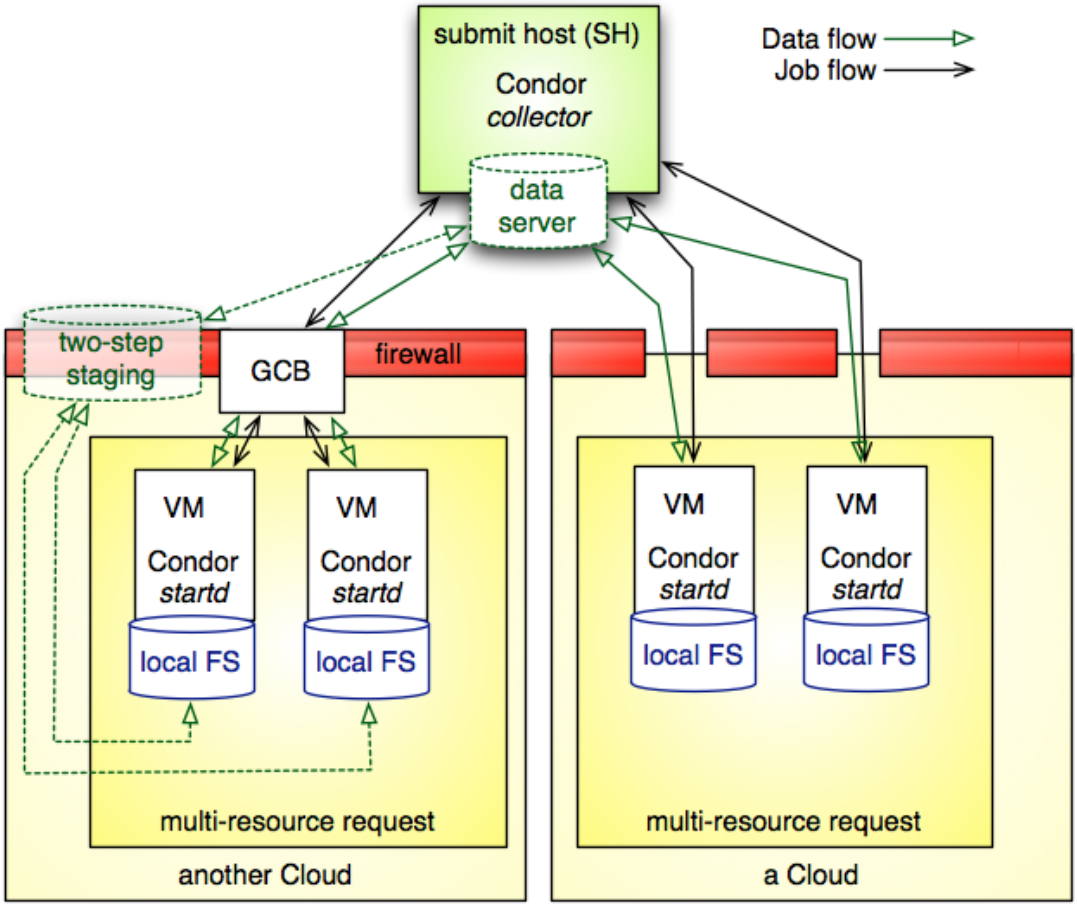
The shared filesystem requirement is imposed by Globus. Pegasus is capable of supporting advanced site layouts that do not require a shared filesystem. Please contact us for details, should you require such a setup.

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 mapfile, CA certificate chains and other setups. However, for performance purposes it is recommended that the GridFTP server has its own machine.

A Cloud Execution Environment

Figure 5.9. Cloud Sample Site Layout



The pevious figure shows a sample layout for sky computing (as in: multiple clouds) as supported by Pegasus. At this point, it is up to the user to provision the remote resources with a proper VM image that includes a Condor startd and proper Condor configuration to report back to a Condor collector that the Condor schedd has access to.

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 announcement. 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. We are working on expanding the current code to support a more generic set by Pegasus 3.1.

Shared File Systems

Ideally, any allocation of multiple resources shares a file system among them. This can be either a globally shared file system like NFS, or a per-request file system dynamically allocated like PVFS or NFS. Often, for the sake of speed, it is advisable that each resource has its own fast local non-shared disk space that can be transiently used by application workflow jobs.

Pegasus recommends the use of a shared file system, because it simplifies workflows and may even improve performance. Without shared file system, the same data may need to be staged to the same site but different resources multiple times. Without shared file systems, data products of a parent job need to be staged out, and staged back in to the child job, currently via an external data server, even if parent and child run on the same multi-resource allocation.

The Pegasus team explores the option of dynamically bringing up an NFS server on one resource of a multi-resource allocation for the duration of the allocation. This option is part of the configuration for an experiment’s workflow’s provisioning stage. We do not expect cloud sites to ban NFS and RPC traffic between resources of the same multi-resource allocation.

Resource Configurations

This section discusses the various resource configurations that can be used with Pegasus when Condor DAGMan is used as the workflow execution engine. It is assumed that there is a submit host where the workflow is submitted for execution. The following classification is done based the mechanism used for submitting jobs to the Grid resources and monitoring them. The classifications explored in this document are using Globus GRAM and using a Condor pool. Both of the configurations use Condor DAGMan to maintain the dependencies between the jobs, but differ in the manner as to how the jobs are launched. A combination of the above mentioned approach is also possible where some of the tasks in the workflow are executed in the Condor pool and the rest are executed on remote resources using Condor-G.

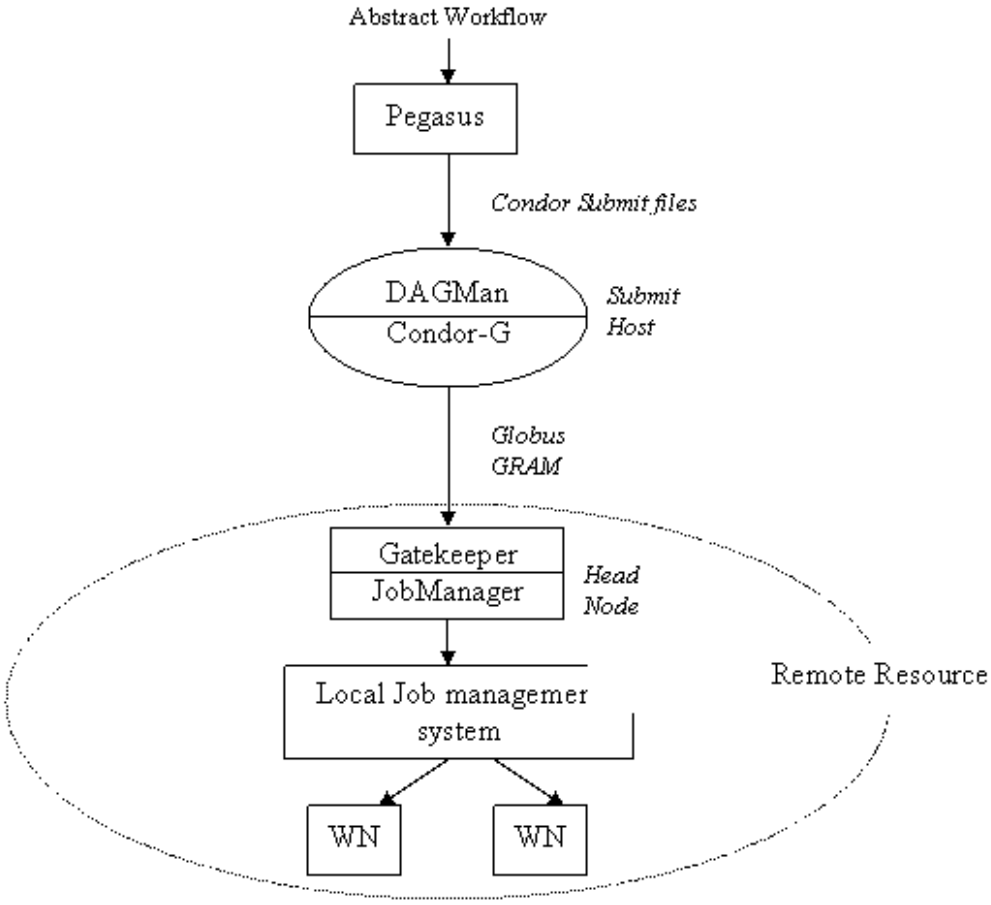
Locally on the Submit Host

In this configuration , Pegasus schedules the jobs to run locally on the submit host. This 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.

Using Globus GRAM

In this configuration, it is assumed that the target execution system consists of one or more Grid resources. These resources may be geographically distributed and under various administrative domains. Each resource might be a single desktop computer or a network of workstations (NOW) or a cluster of dedicated machines. However, each resource must provide a Globus GRAM interface which allows the users to submit jobs remotely. In case the Grid resource consists of multiple compute nodes, e.g. a cluster or a network of workstations, there is a central entity called the head node that acts as the single point of job submissions to the resource. It is generally possible to specify whether the submitted job should run on the head node of the resource or a worker node in the cluster or NOW. In the latter case, the head node is responsible for submitting the job to a local resource management system (PBS, LSF, Condor etc) which controls all the machines in the resource. Since, the head node is the central point of job submissions to the resource it should not be used for job execution since that can overload the head node delaying further job submissions. Pegasus does not make any assumptions about the configuration of the remote resource; rather it provides the mechanisms by which such distinctions can be made.

Figure 5.10. Resource Configuration using GRAM



In this configuration, Condor-G is used for submitting jobs to these resources. Condor-G is an extension to Condor that allows the jobs to be described in a Condor submit file and when the job is submitted to Condor for execution, it uses the Globus GRAM interface to submit the job to the remote resource and monitor its execution. The distinction is made in the Condor submit files by specify the universe as Grid and the grid_resource or globusscheduler attribute is used to indicate the location of the head node for the remote resource. Thus, Condor DAGMan is used for maintaining the dependencies between the jobs and Condor-G is used to launch the jobs on the remote resources using GRAM. The implicit assumption in this case is that all the worker nodes on a remote resource have access to a shared file system that can be used for data transfer between the tasks mapped on that resource. This data transfer is done using files.

Below is an example of a site configured to use Globus GRAM2| GRAM5

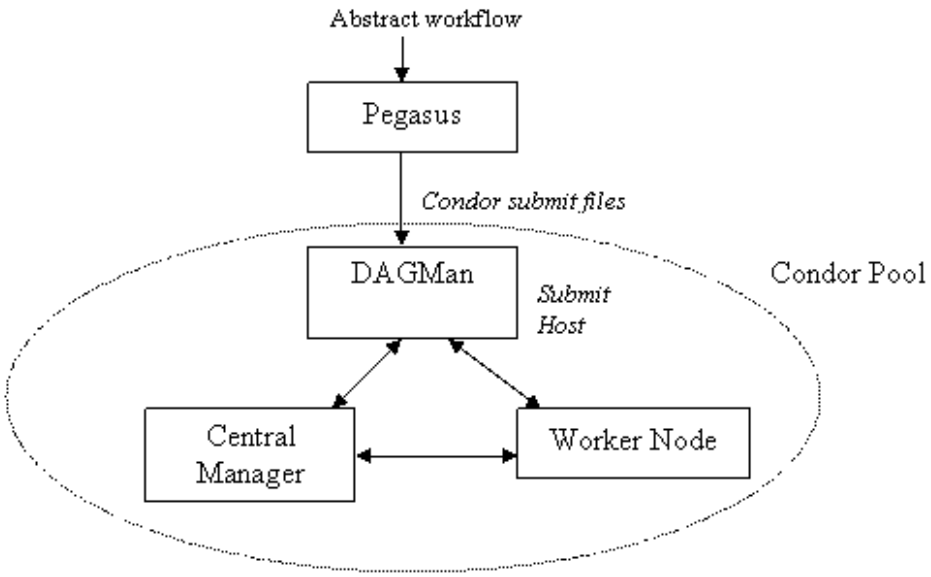
```
<site  handle="isi" arch="x86" os="LINUX" osrelease="" osversion="" glibc="">
      <grid  type="gt2" contact="smarty.isi.edu/jobmanager-pbs" scheduler="PBS"
jobtype="auxillary"/>
      <grid  type="gt2" contact="smarty.isi.edu/jobmanager-pbs" scheduler="PBS" jobtype="compute"/>
      <head-fs>
              <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" />
</site>
```

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 5.11. 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. Pegasus currently uses the FileSystemDomain classad[] attribute to restrict the set of machines that make up a single resource. To clarify this point, suppose there are certain execution machines in the Condor pool whose FileSystemDomain is set to “viz.isi.edu”. If the user wants to execute the workflow on these machines, then there should be a site, say “isi_viz” defined in the site catalog and the FileSystemDomain and universe attributes for this resource should be defined as “viz.isi.edu” and “vanilla” respectively. When invoking Pegasus, the user should select isi_viz as the compute resource.

```
<site  handle="isi_viz" 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://viz-login.isi.edu" \
                                                          mount-point="/nfs/scratch01"/>
                        <internal-mount-point mount-point="/nfs/scratch01"/>
                  </shared>
```

```
              </scratch>
              <storage>
                    <shared>
                          <file-server protocol="gsiftp" url="gsiftp://viz-login.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="condor" key="universe">vanilla</profile>
       <profile namespace="condor" key="FileSystemDomain">viz.isi.edu</profile>
 </site>
```

Condor Glideins

Pegasus WMS can execute workflows over Condor pool. This pool can contain machines managed by a single institution or department and belonging to a single administrative domain. This is the case for most of the Condor pools. 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 Glide-in http://cs.wisc.edu/condor/glidein that uses Globus GRAM interface for migrating machines from different domains to a Condor pool. The number of machines and the duration for which they are required can be specified

In this case, we use the abstraction of a local Condor pool to execute the jobs in the workflow over remote resources that have joined the pool for certain timeframe. Details about the use of this feature can be found in the condor manual (http://cs.wisc.edu/condor/manual/).

A basic step to migrate in a job to a local condor pool is described below.

```
$condor_glidein -count 10 gatekeeper.site.edu/jobmanager-pbs
```

GlideIn of Remote Globus Resources

The above step glides in 10 nodes to the user’s local condor pool, from the remote pbs scheduler running at gatekeeper.site.edu. By default, the glide in binaries are installed in the users home directory on the remote host.

It is possible that the Condor pool can contain resources from multiple Grid sites. It is normally the case that the resources from a particular site share the same file system and thus use the same FileSystemDomain attribute while advertising their presence to the Central Manager of the pool. If the user wants to run his jobs on machines from a particular Grid site, he has to specify the FileSystemDomain attribute in the requirements classad expression in the submit files with a value matching the FileSystemDomain of the machines from that site. For example, the user migrates nodes from the ISI cluster (with FileSystemDomain viz.isi.org) into a Condor pool and specifies FileSystemDomain == “viz.isi.edu”. Condor would then schedule the jobs only on the nodes from the ISI VIZ cluster in the local condor pool. The FileSystemDomain can be specified for an execution site in the site catalog with condor profile namespace as follows

```
 <site  handle="isi_viz" 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://viz-login.isi.edu" \
                                                             mount-point="/nfs/scratch01"/>
                         <internal-mount-point mount-point="/nfs/scratch01"/>
                    </shared>
              </scratch>
              <storage>
                     <shared>
                          <file-server protocol="gsiftp" url="gsiftp://viz-login.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="pegasus" key="style">glidein</profile>
      <profile namespace="condor" key="universe">vanilla</profile>
      <profile namespace="condor" key="FileSystemDomain">viz.isi.edu</profile>
 </site>
```

Specifying the FileSystemDomain key in condor namespace for a site, triggers Pegasus into generating the requirements classad expression in the submit file for all the jobs scheduled on that particular site. For example, in the above case all jobs scheduled on site isi_condor would have the following expression in the submit file.

```
requirements = FileSystemDomain == &ldquo;viz.isi.edu&rdquor;
```

Condor Glidein's using glideinWMS

glideinWMS [???] is a glidein system widely used on Open Science Grid. Pegasus has a convenience style named glideinWMS to make running workflows on top of glideinWMS easy. This style is similar to the Condor style, but provides some more defaults set up to work well with glideinWMS. All you have to do is specify the style profile in the site catalog:

```
 <site  handle="isi_viz" 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://viz-login.isi.edu" \
                                                              mount-point="/nfs/scratch01"/>
                         <internal-mount-point mount-point="/nfs/scratch01"/>
                    </shared>
              </scratch>
              <storage>
                     <shared>
                          <file-server protocol="gsiftp" url="gsiftp://viz-login.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="pegasus" key="style">glideinwms</profile>
 </site>
```

Planned jobs will have universe, requirements and rank set:

```
universe = vanilla
rank = DaemonStartTime
requirements = ((GLIDEIN_Entry_Name == "isi_iz") || (TARGET.Pegasus_Site == "isi_viz")) \
              && (IS_MONITOR_VM == False) && (Arch != "") && (OpSys != "") \
              && (Disk != -42) && (Memory > 1) && (FileSystemDomain != "")
```

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

```
 <site  handle="isi_viz_glite" 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://viz-login.isi.edu" \
                                                    mount-point="/nfs/scratch01"\>
                        <internal-mount-point mount-point="/nfs/scratch01"/>
                  </shared>
            </scratch>
            <storage>
                  <shared>
                        <file-server protocol="gsiftp" url="gsiftp://viz-login.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" />

      <!-- following profiles reqd for glite grid style-->
      <profile namespace="pegasus" key="style">glite</profile>
      <profile namespace="condor" key="grid_resource">pbs</profile>
</site>
```

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.

Condor Pools Without a Shared Filesystem and Using Condor File Transfers

It is possible to run on a condor pool without the worker nodes sharing a filesystem and relying on Condor to do the file transfers. However, for the Condor file to transfer the input data for a job, all the input data

1. Must be on the submit host
2. The input files should be catalogued as file URL's in the Replica Catalog with the pool attribute set as local for all the locations. To make sure that local URL's are always picked up use the Local Replica Selector.

To set up Pegasus to plan workflows for Amazon you need to create an 'local-condor' entry your site catalog. The site configuration is similar to what you would create for running on a local Condor pool (See previous section).

```
 <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 3.1 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>

        <!-- required for Condor File IO and transferring stdout and stderr back -->
        <profile namespace="condor" key="should_transfer_files">YES</profile>
        <profile namespace="condor" key="transfer_output">true</profile>
        <profile namespace="condor" key="transfer_error">true</profile>
        <profile namespace="condor" key="WhenToTransferOutput">ON_EXIT</profile>

        <!-- to enable kickstart staging from local site-->
        <profile namespace="condor" key="transfer_executable">true</profile>


    </site>
```

Next you need need to update pegasus.properties to tell Pegasus to turn on Condor File IO and execute jobs on the worker nodes filesystem.

```
pegasus.data.configuration = Condor
```

By default, Pegasus creates a workflow specific execution directory that is associated with the compute site (in this case local-condor). However, in this scenario, the workflow specific execution directory needs to be created on the submit host, as native Condor File I/O only works against local directories. To ensure that the create dir job for the workflow for the local-condor site, you need to do the following

1. the paths for the scratch filesystem in the site catalog entry for local-condor site should actually refer to paths on the submit host. It is illustrated in the example site catalog entry above.
2. have an entry in the transformation catalog for pegasus::dirmanager for site local-condor that has a condor profile that makes the create dir job run in local universe instead of vanilla.

```
# This entry for pegaasus-dirmanager is to force the create dir jobs to run in the local universe
 instead of
# the vanilla universe. The CONDOR transfer settings are so that stdout and stderr are
# not redirected to initialdir instead of the submit dir

local-condor pegasus::dirmanager /path/on/submithost/pegasus/bin/dirmanager INSTALLED
 INTEL32::LINUX CONDOR:
:universe="local"CONDOR::requirements="";CONDOR::transfer_output="";CONDOR::transfer_error="";CONDOR::whentotransfe
```

Note

In Pegasus 3.1, running workflows in this setup will be made much simpler.

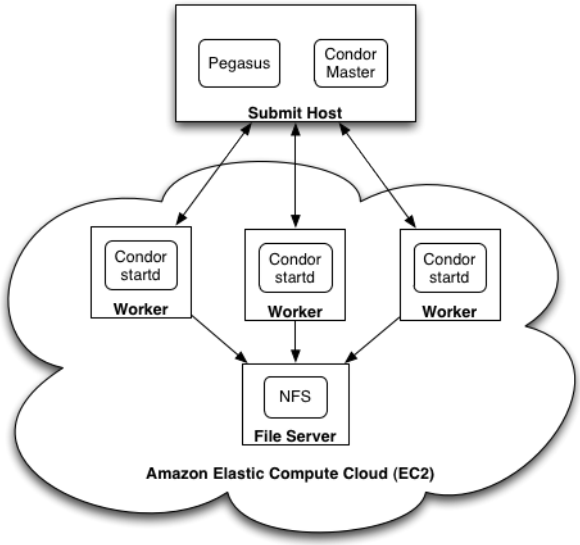
Cloud

Using Amazon

In order to use Amazon to execute workflows you need to a) set up an execution environment in EC2, and b) configure Pegasus to plan workflows for that environment.

There are many different ways to set up the execution environment in Amazon. 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 5.12. 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 workers' startd is configured to connect to the master running outside the cloud. The worker also mounts the NFS file system. More information on setting up Condor for this environment can be found at http://www.isi.edu/~gideon/condor-ec2/.

To set up Pegasus to plan workflows for Amazon you need to create an 'ec2' entry your site catalog. The site configuration is similar to what you would create for running on a local Condor pool (See previous section).

```
<site  handle="ec2" arch="x86" os="LINUX" osrelease="" osversion="" glibc="">
      <!-- These entries are required, but not used -->

      <grid  type="gt2" contact="example.com/jobmanager-pbs" scheduler="PBS" jobtype="auxillary"/>
      <grid  type="gt2" contact="example.com/jobmanager-fork" scheduler="PBS" jobtype="compute"/>
      <head-fs>
```

```
            <scratch>
                  <shared>
                        <file-server protocol="gsiftp" url="gridftp://example.com" \
                                                    mount-point="/nfs"\>
                        <internal-mount-point mount-point="/nfs"/>
                  </shared>
            </scratch>
            <storage>
                  <shared>
                        <file-server protocol="gsiftp" url="gridftp://example.com" \
                                                  mount-point=""\>
                        <internal-mount-point mount-point=""/>
                   </shared>
            </storage>
      </head-fs>
      <replica-catalog  type="LRC" url="rlsn://smarty.isi.edu" />

      <!-- following profiles reqd for ec2 -->

      <profile namespace="env" key="PEGASUS_HOME">/usr/local/pegasus/default</profile>

      <!-- The directory where a user wants to run the jobs on the
           nodes retrived from ec2 -->
      <profile namespace="env" key="wntmp">/mnt</profile>

      <!-- Use condor style vanilla universe jobs -->
      <profile namespace="pegasus" key="style">condor</profile>
      <profile namespace="condor" key="universe">vanilla</profile>

      <!-- This ensures that Condor sends stdout and stderr back to the submit host -->
      <profile namespace="condor" key="should_transfer_files">YES</profile>
      <profile namespace="condor" key="transfer_output">true</profile>
      <profile namespace="condor" key="transfer_error">true</profile>
      <profile namespace="condor" key="WhenToTransferOutput">ON_EXIT</profile>

      <!-- This ensures that the jobs generated by Pegasus will run on the remote workers -->
      <profile namespace="condor" key="requirements">(Arch==Arch)&amp;&amp;(Disk!=0)&amp;&amp;
(Memory!=0)&amp;&amp;(OpSys==OpSys)&amp;&amp;(FileSystemDomain!="")</profile>


 </site>
```

Using S3 for intermediate files

This section will show you how to use S3 to store intermediate files. In this mode, Pegasus transfers workflow inputs from the input site to S3 if they are not already in 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. This is similar to how second-level staging works (SLS), and, in fact, the current S3 implementation is an extension of SLS.

In order to use S3 for your workflows, you need to first create a config file for the S3 transfer client, s3cmd. This config file needs to be installed on both the submit host, and the worker nodes. Save the file as 's3cmd.cfg'. This file will contain your Amazon security tokens, so you will want to set the permissions so that other users cannot read it (e.g. chmod 600 s3cmd.cfg). In the listing below, replace 'YOUR_AMAZON_ACCESS_KEY_HERE' and 'YOUR_AMAZON_SECRET_KEY_HERE' with your Amazon access key and secret key.

```
[default]
access_key = YOUR_AMAZON_ACCESS_KEY_HERE
secret_key = YOUR_AMAZON_SECRET_KEY_HERE
acl_public = False
bucket_location = US
debug_syncmatch = False
default_mime_type = binary/octet-stream
delete_removed = False
dry_run = False
encrypt = False
force = False
gpg_command = /usr/bin/gpg
gpg_decrypt = %(gpg_command)s -d --verbose --no-use-agent --batch --yes --passphrase-fd
 %(passphrase_fd)s -o %(output_file)s %(input_file)s
gpg_encrypt = %(gpg_command)s -c --verbose --no-use-agent --batch --yes --passphrase-fd
 %(passphrase_fd)s -o %(output_file)s %(input_file)s
gpg_passphrase =
guess_mime_type = False
host_base = s3.amazonaws.com
```

```
host_bucket = %(bucket)s.s3.amazonaws.com
human_readable_sizes = False
preserve_attrs = True
proxy_host =
proxy_port = 0
recv_chunk = 4096
send_chunk = 4096
simpledb_host = sdb.amazonaws.com
use_https = False
verbosity = WARNING
```

Next, you need to modify your site catalog to tell s3cmd the path to your config file. You can do this by specifying an environment variable called S3CFG using an 'env' profile. This needs to be added to both the '**local**' site, and any execution sites where you want to use S3. This environment variable will be used by s3cmd in the S3 transfer jobs to locate the config file.

```
<profile namespace="env" key="S3CFG">/local/path/to/s3cmd.cfg</profile>
```

Next, you need to update your pegasus.properties file to tell Pegasus to turn on S3 mode. This will tell Pegasus to use S3 instead of rgular shared filesystem.

```
pegasus.data.configuration = S3
```

Note

The current S3 support does not work mixing storage solutions. Once S3 mode is turned on, Pegasus shared file system mode is turned off. For example, you currently can not use S3 and iRods storage at the same time.

Finally, if you want to use an existing bucket to store your data in S3 add this to your site catalog entry (replacing the existing workdirectory):

```
<scratch>
  <shared>
    <file-server protocol="s3" url="s3://user@amazon" mount-point="/existing-bucket"/>
    <internal-mount-point mount-point="/existing-bucket"/>
  </shared>
</scratch>
```

Otherwise, you can have Pegasus create a new bucket for each workflow by adding this to your site catalog entry (again, replacing any existing workdirectory):

```
<scratch>
  <shared>
    <file-server protocol="s3" url="s3://user@amazon" mount-point="/"/>
    <internal-mount-point mount-point="/"/>
  </shared>
</scratch>
```

Chapter 6. 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.

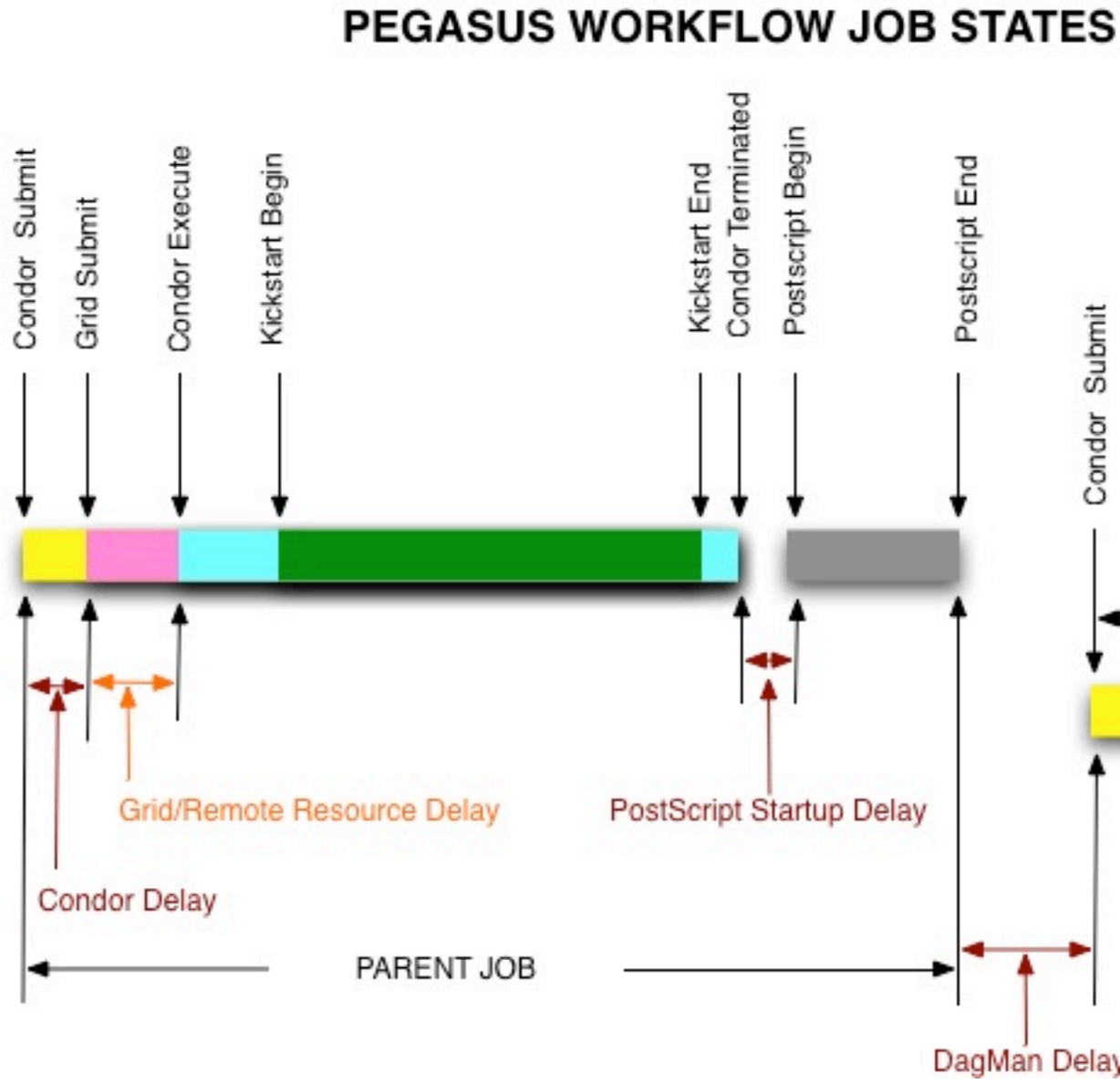
Table 6.1. Table 1: The job lifecycle when executed as part of the workflow

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 6.2. 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 7. 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>
```

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-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).

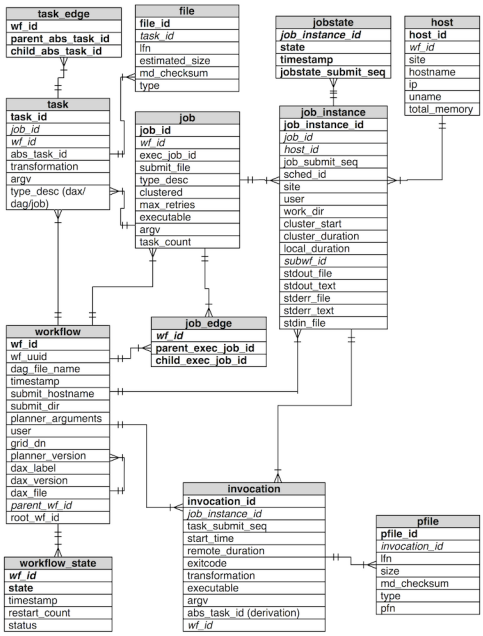
Finally, the following example:

```
$ 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.

Figure 7.1. Stampede Database Schema



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.
- **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.

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 -d /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.

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 generates workflow execution statistics. To generate statistics run the command as shown below.

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


...

****************************************** SUMMARY *****************************************
...

------------------------------------------------------------------------------------------------------
Type            Succeeded  Failed  Incomplete   Total      ||  Retries  Total Run (Retries Included)
Tasks           8          0       0            8      ||  0        8
Jobs            27         0       0            27     ||  0        27
Sub Workflows   2          0       0            2      ||  0        2
------------------------------------------------------------------------------------------------------

Workflow wall time                               : 21 mins, 9 secs,     (total 1269 seconds)

Workflow cumulative job wall time                : 8 mins, 4 secs,      (total 484 seconds)

Cumulative job walltime as seen from submit side : 8 mins, 0 secs,      (total 480 seconds)

Workflow execution statistics     : /scratch/grid-setup/run0001/statistics/workflow.txt

Job instance statistics           : /scratch/grid-setup/run0001/statistics/jobs.txt

Transformation statistics         : /scratch/grid-setup/run0001/statistics/breakdown.txt

Time statistics                   : /scratch/grid-setup/run0001/statistics/time.txt

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

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.

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.
- **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

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 cummulative 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 7.1. Workflow Statistics

#	Type	Succeeded	Failed	Incomplete	Total	Retries	Total Run	Workflow Retries
2a6df11b-9972-4ba0-b4ba-4fd39c357af4								0

#	Type	Succeeded	Failed	Incomplete	Total	Retries	Total Run	Workflow Retries
	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.
- **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 7.2. Job statistics

Job	Try	Site	Kickstart	Post	CondorQTime	Resource	Runtime	Seqexec	Seqexec-Delay
analyze_ID0000004	1	local	60.002	5.0	0.0	-	62.0	-	-
create_dir_diamond_0_local	1	local	0.027	5.0	5.0	-	0.0	-	-
findrange_ID0000002	1	local	60.001	5.0	0.0	-	60.0	-	-
findrange_ID0000003	1	local	60.002	5.0	10.0	-	61.0	-	-
preprocess_ID0000001	1	local	60.002	5.0	5.0	-	60.0	-	-
register_local_1_0	1	local	0.459	6.0	5.0	-	0.0	-	-
register_local_1_1	1	local	0.338	5.0	5.0	-	0.0	-	-
register_local_2_0	1	local	0.348	5.0	5.0	-	0.0	-	-
stage_in_local_local_0	1	local	0.39	5.0	5.0	-	0.0	-	-
stage_out_local_local_0_1	0	local	0.165	5.0	10.0	-	0.0	-	-
stage_out_local_local_1_1	0	local	0.147	7.0	5.0	-	0.0	-	-
stage_out_local_local_1_1	1	local	0.139	5.0	6.0	-	0.0	-	-
stage_out_local_local_2_1	0	local	0.145	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.
- **Max (sec.)** - the minimum runtime value of invocations with a given logical transformation name.
- **Mean (sec.)** - the mean of the invocation runtimes with a given logical transformation name.
- **Total (sec.)** - the cumulative of runtime value of invocations with a given logical transformation name

Table 7.3. 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	60.001	60.002	60.002	120.003
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 7.4. 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/


...

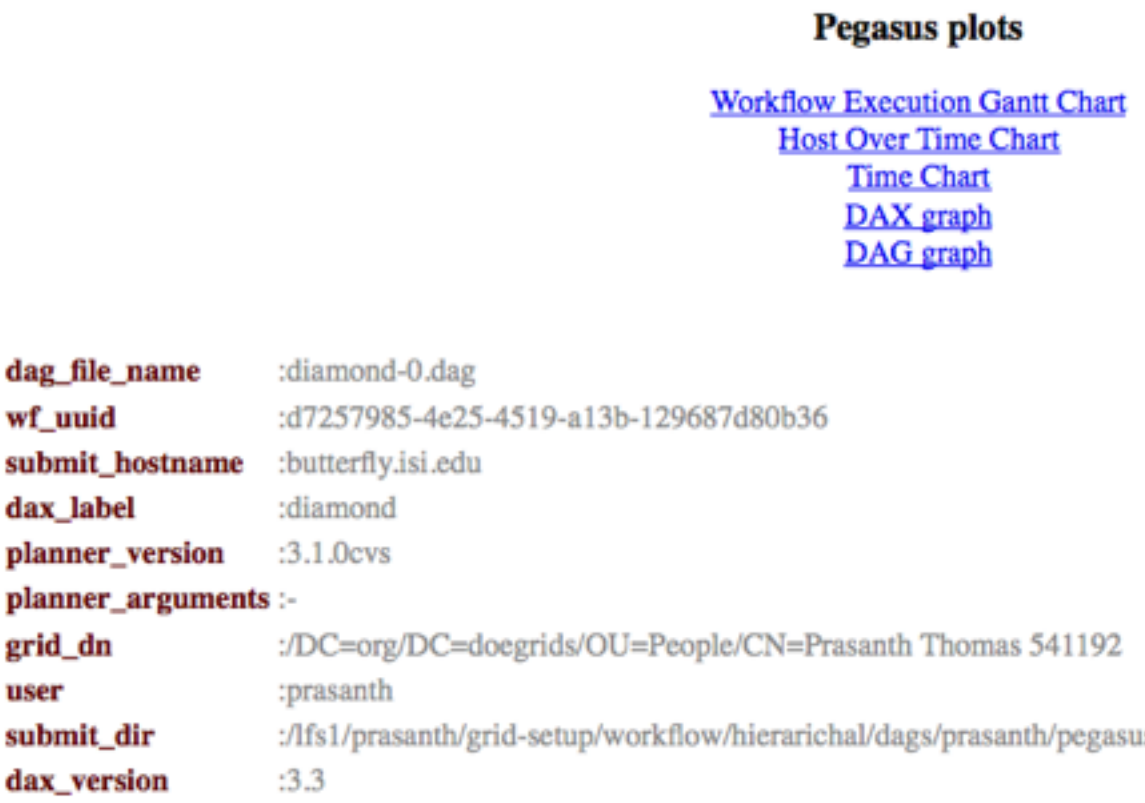
****************************************** SUMMARY *******************************************

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 7.2. pegasus-plot index page

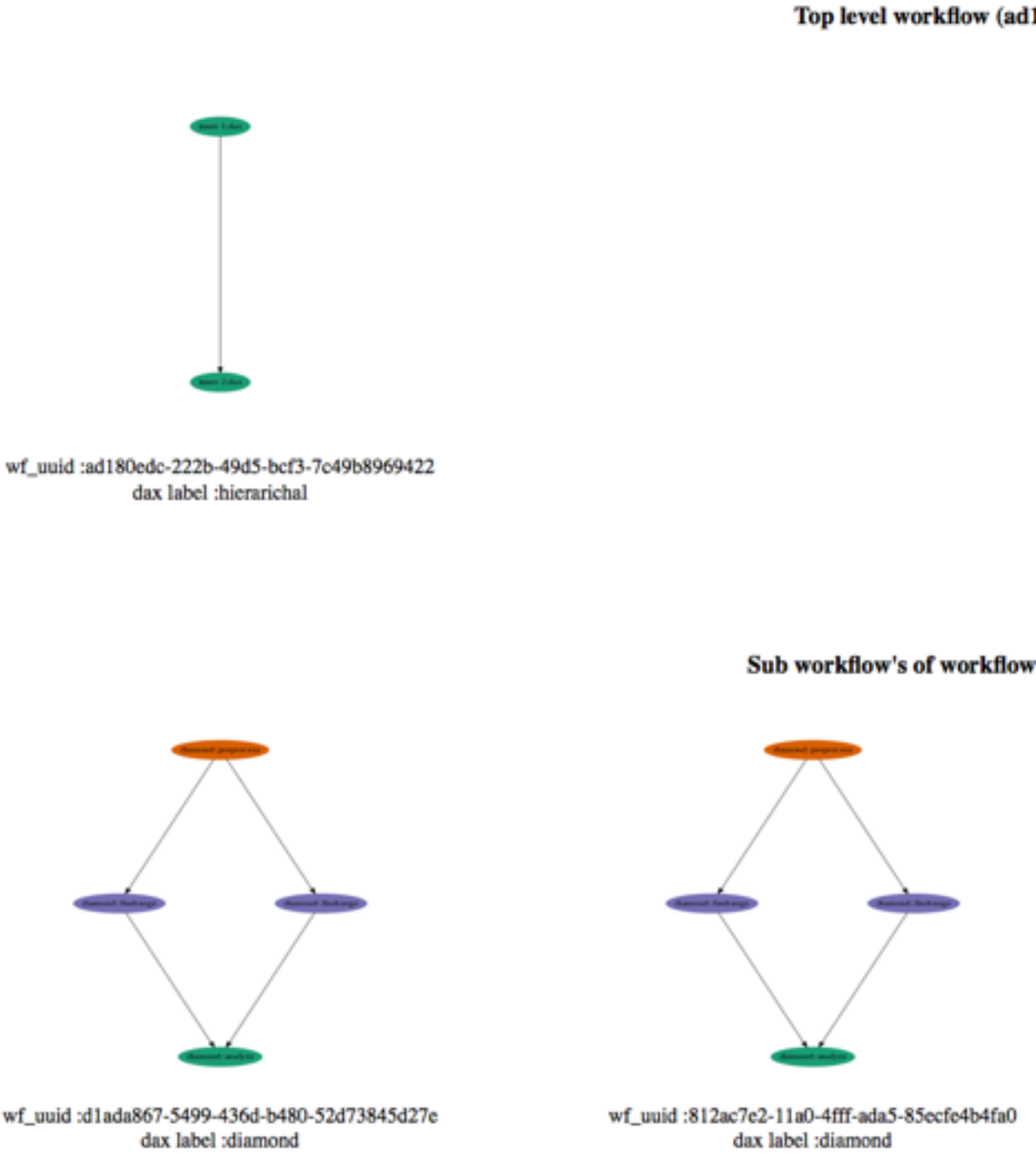


pegasus-plots generates the following plots and charts.

Dax Graph

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

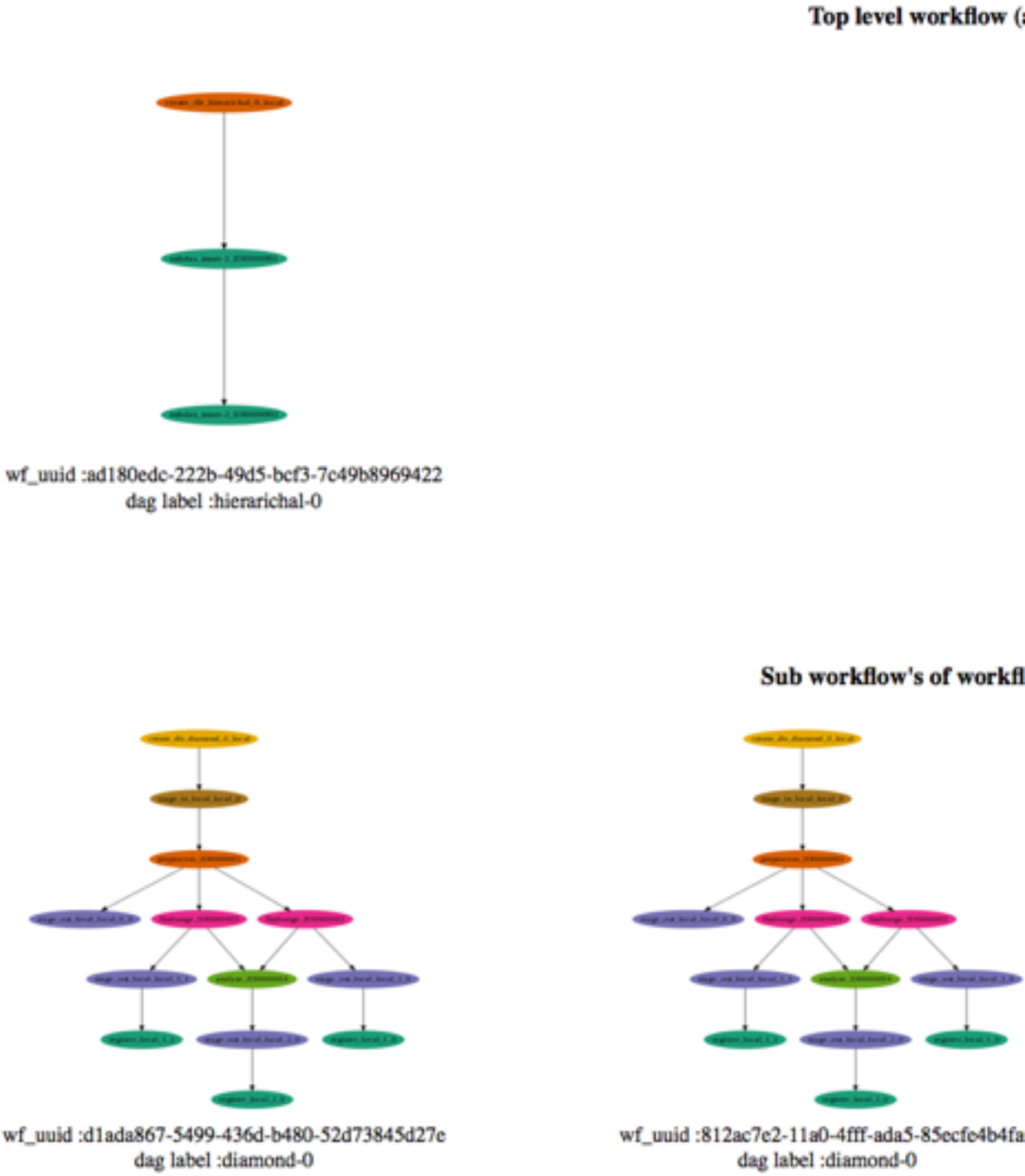
Figure 7.3. DAX Graph



Dag Graph

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

Figure 7.4. DAG Graph



Gantt workflow execution chart

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

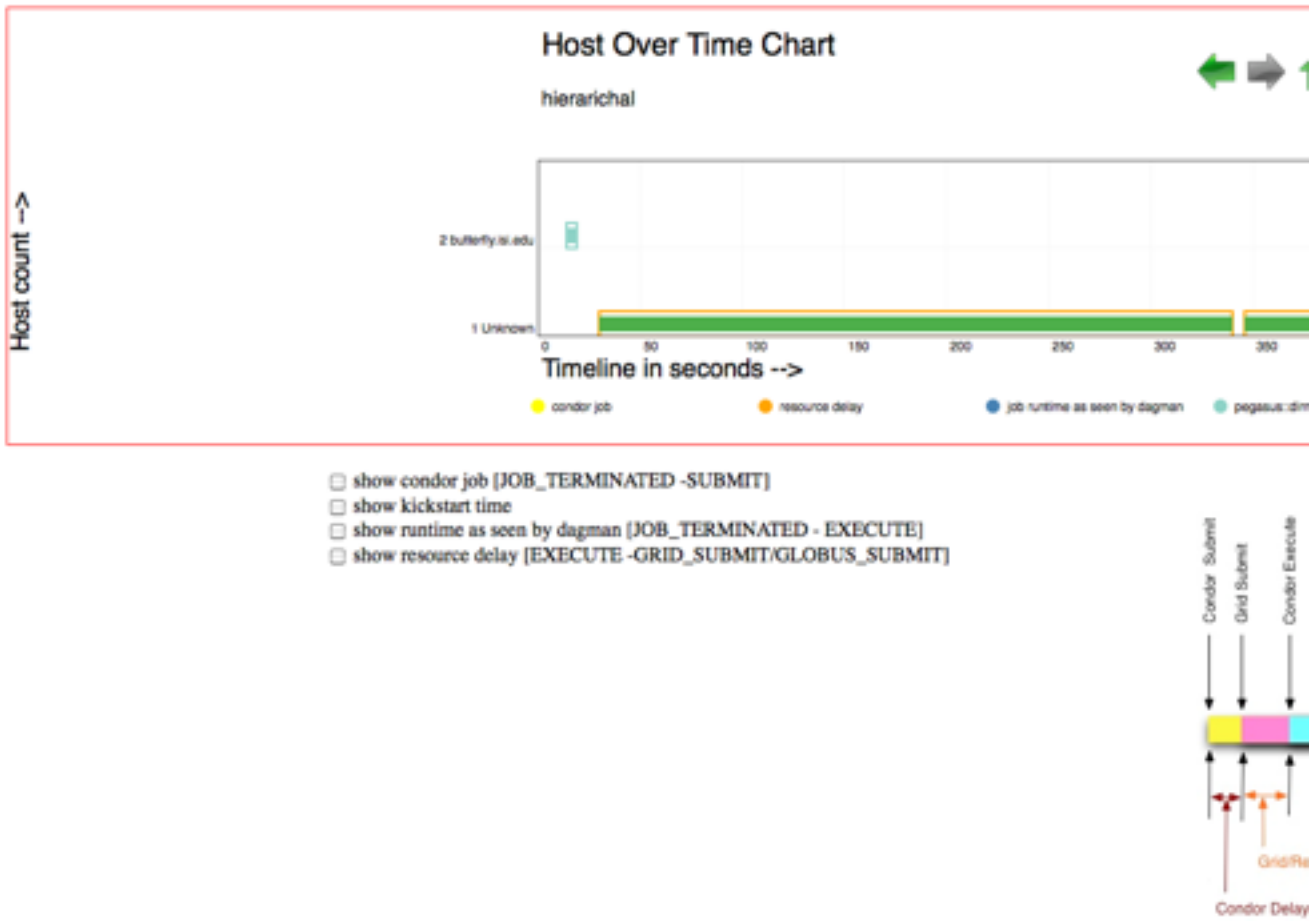
Figure 7.5. Gantt Chart

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 7.6. Host over time chart

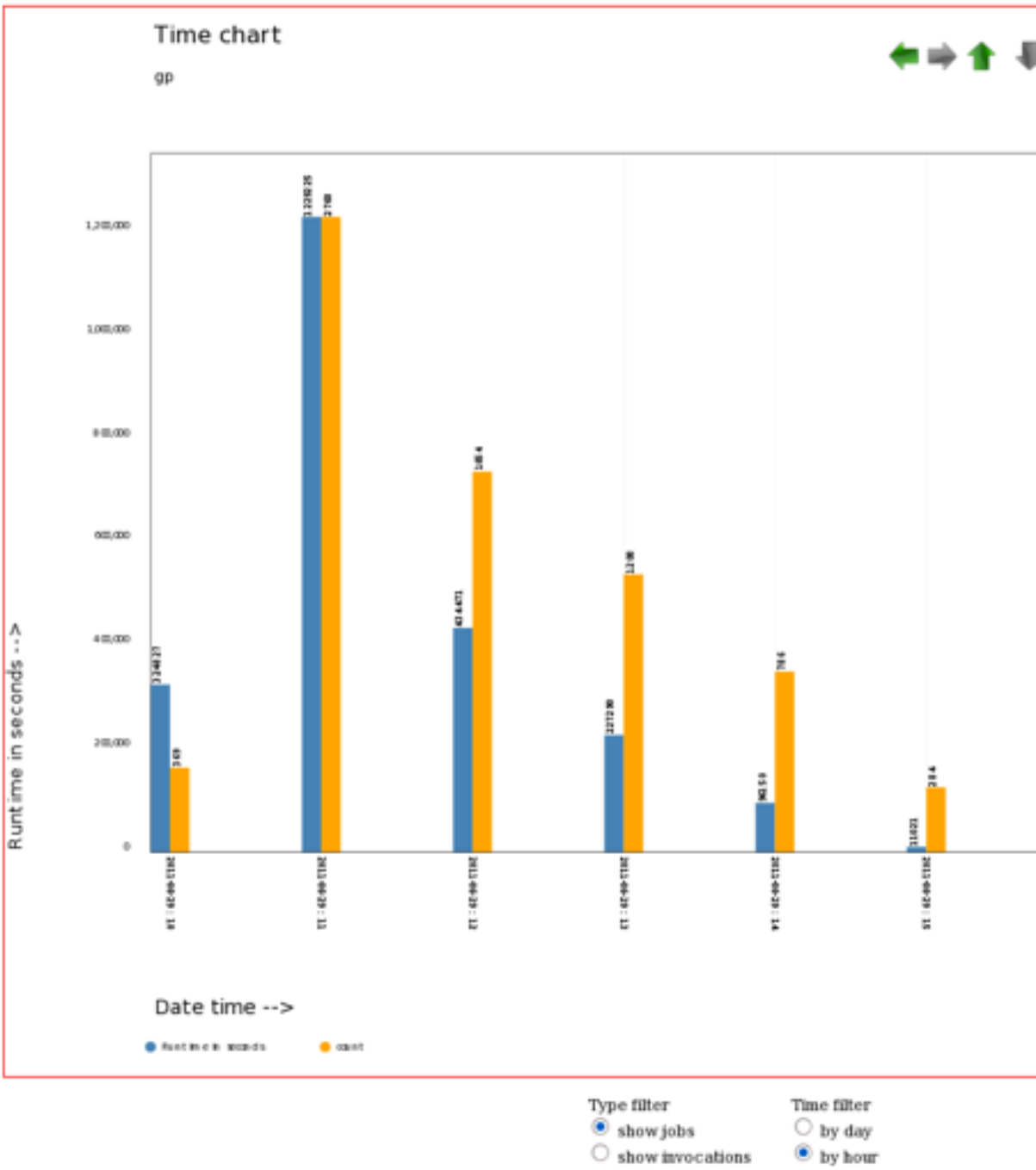


The toolbar at the top provides zoom in/out , pan left/right/top/bottom and show/hide host name functionality.The toolbar at the bottom can be used to show/hide job states. Failed job instances are shown in red border in the chart. Clicking on a sub workflow job instance will take you to the corresponding 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 7.7. Time chart

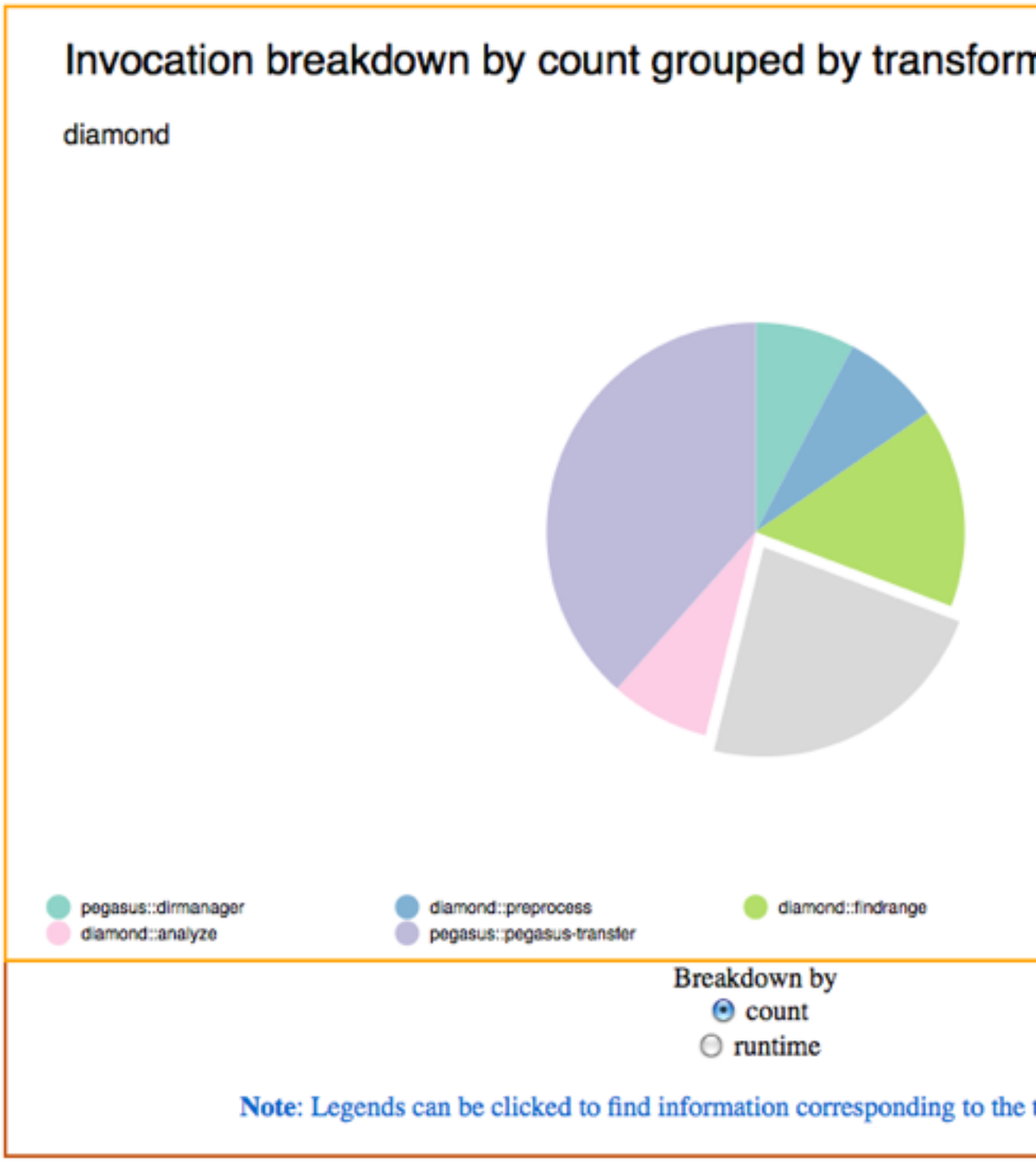


The toolbar at the top provides zoom in/out and pan left/right/top/bottom functionality.The toolbar at the bottom can be used to switch between job instances/ invocations and day/hour filtering.

Breakdown chart

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

Figure 7.8. 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.

Chapter 8. Example Workflows

These examples are included in the Pegasus distribution and can be found under `$PEGASUS_HOME/examples/`

Grid Examples

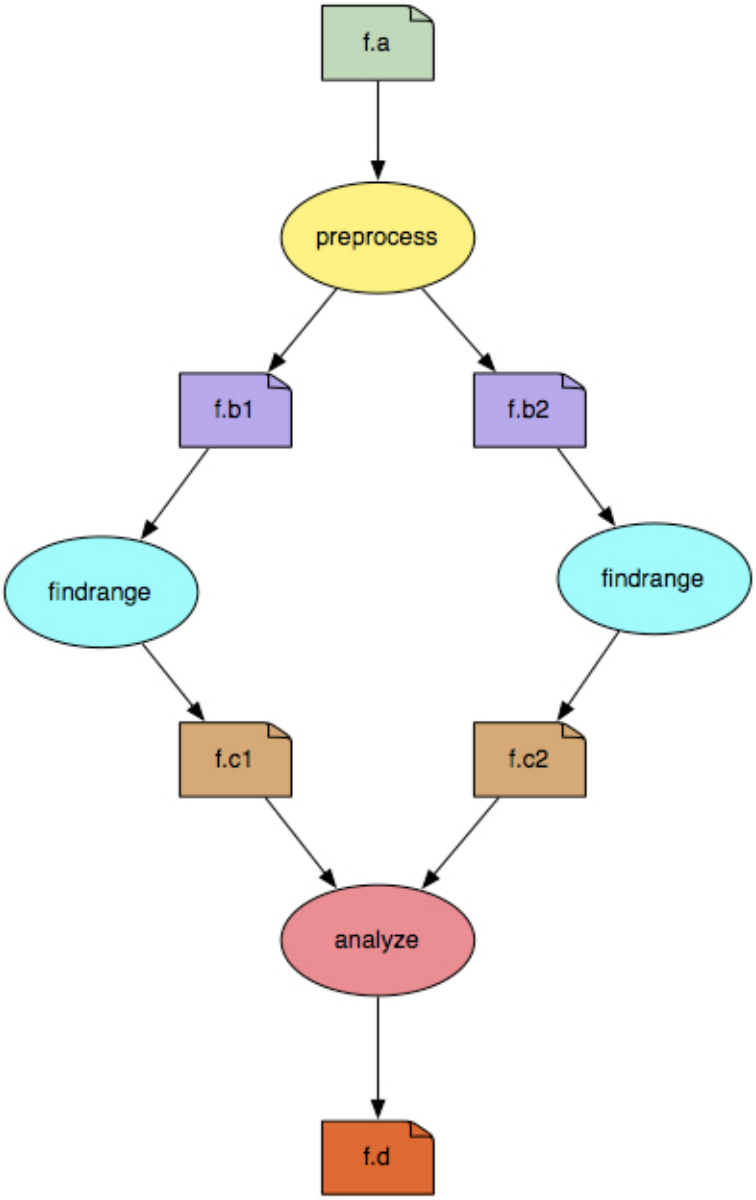
These examples assumes you have access to a cluster with Globus installed. A pre-ws gatekeeper and gridftp server is required. You also need Globus and Pegasus installed, both on the machine you are submitting from, and the cluster.

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:

```
examples/grid-blackdiamond-java/
examples/grid-blackdiamond-perl/
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

```
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

```
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

This example can be found under

```
examples/condor-blackdiamond/
```

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"). The latter requires a shared filesystem for the pool.

You can run this example with the `./submit` script, but you have to pass an argument: *condor* for universe=local or *condorpool* for universe="vanilla". Example:

```
$ ./submit condorpool
```

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.

GlideinWMS Examples

NASA/IPAC Montage

By default Pegasus generates Condor-G compute jobs, but there are a couple of other job styles available. One is to run the workflow on a local Condor pool (*pegasus.style=condor*), and an extension of that is the glideinWMS style (*pegasus.style=glideinwms*).

glideinWMS is a Condor glidein system which creates an Condor overlay over one or more grid resources. More information can be found at http://www.uscms.org/SoftwareComputing/Grid/WMS/glideinWMS/. A glidein system is usually better than Condor-G at handling short jobs. Because of the overlay effect, hides some of the issues with the low level grid protocol, and provides a more homogeneous user environment.

In this example, Pegasus is told to use the glideinwms styple by setting the style property in the site catalog:

```
<profile namespace="pegasus" key="style">glideinwms</profile>
```

Once submitted, the jobs have attributes fitting for a glideinWMS setup:

```
universe = vanilla
rank = DaemonStartTime
requirements = ((GLIDEIN_Entry_Name == "ISIViz") || (TARGET.Pegasus_Site == "ISIViz")) \
               && (IS_MONITOR_VM == False) && (Arch != "") && (OpSys != "") \
               && (Disk != -42) && (Memory > 1) && (FileSystemDomain != "")
```

This example should work against other Condor based glidein systems as well by changing the style to condor.

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 9. 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                 XML3
pegasus.catalog.site.file            ${pegasus.home}/etc/sample.sites.xml3
```

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

System:	Pegasus
Since:	2.0
Type:	remote directory location string
Default:	(no default)

This property modifies the remote storage location on various pools. If the path is relative then it is appended to the storage mount point specified in the pool.config file. If the path is absolute then it overrides the storage mount point specified in the pool config file.

pegasus.dir.storage.deep

System:	Pegasus

Since:	2.1
Type:	Boolean
Default:	false
See Also:	pegasus.dir.storage
See Also:	pegasus.dir.useTimestamp

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 and the property pegasus.dir.storage.

To this base directory, the relative submit directory structure ($user/$vogroup/$label/runxxxx) is appended.

$storage = $base + $relative_submit_directory

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

pegasus.dir.create.strategy

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

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.

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]:	MRC

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.

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]:	XML3
Value[1]:	XML

Default:	XML3

The site catalog file is available in three major flavors: The Text and and XML formats for the site catalog are deprecated. Users can use pegasus-sc-converter client to convert their site catalog to the newer XML3 format.

1. THIS FORMAT IS DEPRECATED. WILL BE REMOVED IN COMING VERSIONS. USE pegasus-sc-converter to convert XML format to XML3 Format. The "XML" format is an XML-based file. The XML format reads site catalog conforming to the old site catalog schema available at http://pegasus.isi.edu/wms/docs/schemas/sc-2.0/sc-2.0.xsd

2. The "XML3" format is an XML-based file. The XML format reads site catalog conforming to the old site catalog schema available at http://pegasus.isi.edu/wms/docs/schemas/sc-3.0/sc-3.0.xsd

pegasus.catalog.site.file

System:	Site Catalog
Since:	2.0
Type:	file location string
Default:	${pegasus.home.sysconfdir}/sites.xml3 | ${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. The default value is dependant on the value specified for the property pegasus.catalog.site . If type of SiteCatalog used is XML3, then sites.xml3 is picked up from sysconfdir else sites.xml

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 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.

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.

Transfer Configuration Properties

pegasus.transfer.*.impl

System:	Pegasus
Type:	enumeration
Value[0]:	Transfer3
Value[1]:	GUC
Default:	Transfer3
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

Transfer3 — 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]:	Bundle
Value[1]:	Chain
Value[2]:	Condor
Value[3]:	Cluster
Default:	Bundle
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.

Bundle — This is default refinement strategy in Pegasus. In this refinement strategy, the number of stage in transfer nodes that are constructed per execution site can vary. The number of transfer nodes can be specified, by associating the pegasus profile "bundle.stagein". The profile can either be associated with the execution site in the site catalog or with the "transfer" executable in the transformation catalog. The value in the transformation catalog overrides the one in the site catalog. This refinement strategy extends from the Default refiner, and thus takes care of file clobbering while staging in data.

Chain — In this refinement strategy, chains of stagein transfer nodes are constructed. A chain means that the jobs are sequentially dependant upon each other i.e. at any moment, only one stage in transfer job will run per chain. The number of chains can be specified by associating the pegasus profile "chain.stagein". The profile can either be associated with the execution site in the site catalog or with the "transfer" executable in the transformation catalog. The value in the transformation catalog overrides the one in the site catalog. This refinement strategy extends from the Default refiner, and thus takes care of file clobbering while staging in data.

Condor — In this refinement strategy, no additional staging transfer jobs are added to the workflow. Instead the compute jobs are modified to have the transfer_input_files and transfer_output_files set to pull the input

data. To stage-out the data a separate stage-out is added. The stage-out job is a /bin/true job that uses the transfer_input_file and transfer_output_files to stage the data back to the submit host. This refinement strategy is used workflows are being executed on a Condor pool, and the submit node itself is a part of the Condor pool.

Cluster In this refinement strategy, clusters of stage-in and stageout jobs are created per level of the workflow. It builds upon the Bundle refiner. The differences between the Bundle and Cluster refiner are as follows.

```
- stagein is also clustered/bundled per level. In Bundle it was
for the whole workflow.
- keys that control the clustering ( old name bundling are )
cluster.stagein and cluster.stageout
```

This refinement strategy also adds dependencies between the stagein transfer jobs on different levels of the workflow to ensure that stagein for the top level happens first and so on.

An image of the workflow with this refinement strategy can be found at

```
http://vtcpc.isi.edu/pegasus/index.php/ChangeLog#Added_a_Cluster_Transfer_Refiner
```

pegasus.transfer.sls.*.impl

System:	Pegasus
Type:	enumeration
Value[0]:	Transfer3
Value[1]:	S3
Default:	Transfer3
Since:	2.2.0
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

Transfer3 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.transfer.refiner   Condor
pegasus.selector.replica   Local
pegasus.execute.*.filesystem.local true
```

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.

S3 — This implementation refers to the s3cmd transfer client that is used for second level staging of data in the cloud. The data can be staged between the filesystem on the worker nodes and the workflow specific bucket on S3.

There should be an entry in the transformation catalog with the fully qualified name as amazon::s3cmd for the site corresponding to the cloud.

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

By default, Pegasus relies on the worker package to be installed on the shared filesystem of 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 shared scratch directory for the workflow on the remote sites.

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 RLS 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 RLS replaces it with a file based URL. This supposes 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. This ends up bypassing the GridFTP server and reduces the load on it, and is much faster.

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.proxy

System:	Pegasus
Since:	2.0
Type:	Boolean
Default:	false

By default, CondorG transfers a limited proxy to the remote site, while running jobs in the grid universe. However, certain grid ftp servers (like those in front of SRB) require a fully user proxy. In this case, the planners need to transfer the proxy along with the transfer job using Condor file transfer mechanisms. This property triggers Pegasus into creating the appropriate condor commands, that transfer the proxy from the submit host to the remote host. The source location is determined from the value of the X509_USER_KEY env profile key , that is associated with site local in the site catalog.

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]:	SeqExec
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.

SeqExec — In this mode, all the jobs are launched on the remote site via SeqExec clustering executable. This is useful when user wants to run on the local filesystem of the worker nodes. The generated input file to seqexec contains commands to create the directory on the worker node, pull data from the head node, execute the job , push data to the head node, and remove the directory on the worker node. This is still an experimental mode and will be refined more for 3.1.

Support for a new gridstart (K2) is expected to be added soon.

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.nofity

System:	DAGman wrapper
Type:	Case-insensitive enumeration
Value[0]:	Complete

Value[1]:	Error
Value[2]:	Never
Default:	Error
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-submit-dag wrapper processes properties to set DAGMan commandline arguments. The argument sets the e-mail notification for DAGMan itself. This information will be used within the Condor submit description file for DAGMan. This file is produced by the the condor_submit_dag. See notification within the section of submit description file commands in the condor_submit manual page for specification of value. Many users prefer the value NEVER.

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/ChangeLog\#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.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.

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.

Miscellaneous Properties

pegasus.code.generator

System:	Pegasus
Since:	3.0
Type:	enumeration
Value[0]:	Condor
Value[1]:	Shell
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.

pegasus.data.configuration

System:	Pegasus
Since:	3.1
Type:	enumeration
Value[0]:	Condor
Value[1]:	S3
Default:	No default

This property sets up Pegasus to run in a non shared filesystem mode using either Condor file transfers for data transfers from the submit host or using S3 block storage for running in a cloud environment.

Condor If this is set, Pegasus will run jobs in a non shared fileystem setup. The data transfers to and from the worker nodes (where the jobs run) are done using Condor File Transfers from the submit host. On loading this property, internally the following properies are set

```
pegasus.transfer.refiner   = Condor
pegasus.transfer.sls.*.impl   = Condor
pegasus.selector.replica      = Local
pegasus.execute.*.filesystem.local = true
```

S3 If this is set, Pegasus will run jobs in a non shared fileystem setup in a cloud environment. The workflows will rely on the S3 block storage for data transfers. On loading this property, internally the following properies are set

```
pegasus.execute.*.filesystem.local = true
```

```
pegasus.dir.create.impl   = S3
pegasus.file.cleanup.impl   = S3
pegasus.transfer.*.impl    = S3
pegasus.transfer.stage.sls.file   = false
pegasus.gridstart    = SeqExec
```

pegasus.job.priority.assign

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

This property can be used to turn of 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.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.gridstart

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).

It is recommended that a user when setting this property to true, also sets the following property, unless the user is running in a Condor Pool without a shared filesystem and is relying on Condor to transfer the files.

```
pegasus.gridstart    SeqExec
```

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 9.1. 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 9.2. 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 9.3. 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 9.4. 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.

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 9.5. 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 9.6. 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 9.7. 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 9.8. 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.
collapser	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.

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 a concrete workflow. Site specific or transformation specific criteria are taken into consideration while clustering smaller jobs into a larger job in the concrete 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 Concrete DAG

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 &ndash;-dax example.dax --dir ./dags &ndash;p siteX &ndash;-output local
               --cluster [ comma separated list of clustering techniques]  &ndash;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 9.1.

- **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
```

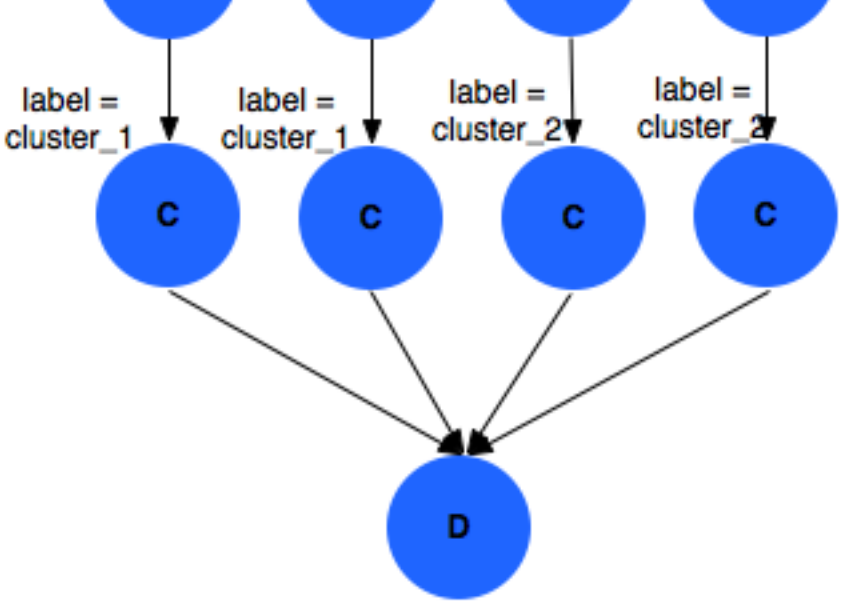
  In the above case the jobs referring to logical transformation B scheduled on siteX will be clustered on the basis of “clusters.num” value. Hence, if there are 4 jobs referring to logical transformation B scheduled to siteX, then 3 clustered jobs will be created.

Figure 9.2.

Label Clustering

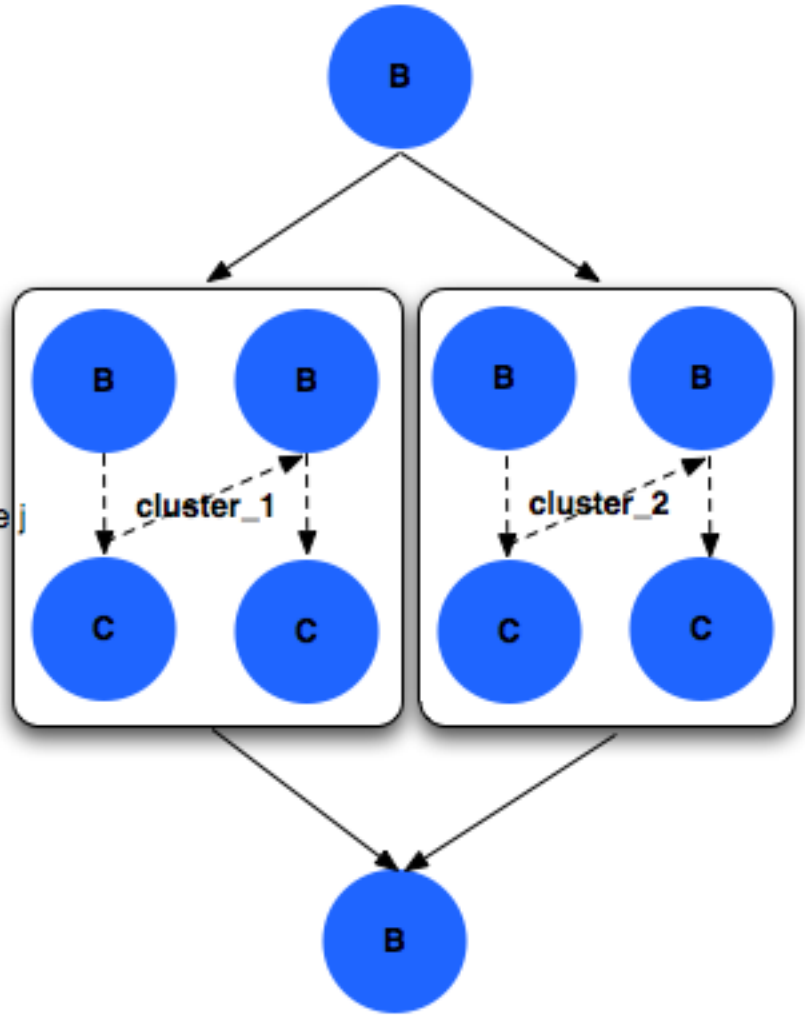
In label based clustering, the user labels the workflow. All jobs having the same label value are clustered into a single clustered job. This allows the user to create clusters or use a clustering technique that is specific to his workflows. If there is no label associated with the job, the job is not clustered and is executed as is

Fi



1. Original Workflow

The Dotted Arrows indicate the topological sort ordering in the cluster of the jobs. This is the order with which the j jobs will be executed on the single node.



2. Workflow After Label 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 >
&mldr;
  <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=&ldquo;pegasus&rdquor; key=&ldquo;user_label&rdquor;>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>
&mldr;
</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.

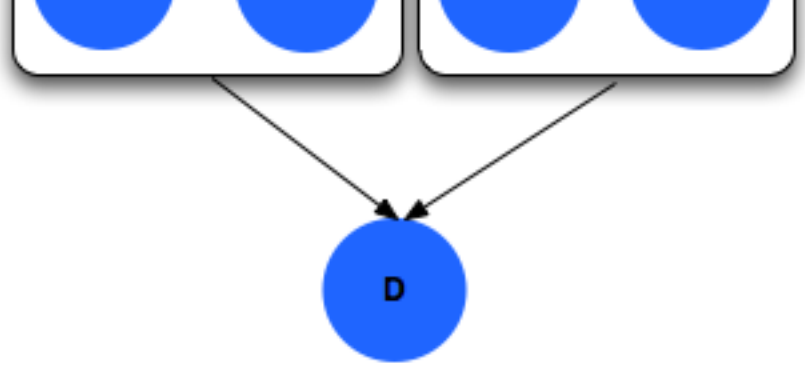
Recursive Clustering

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

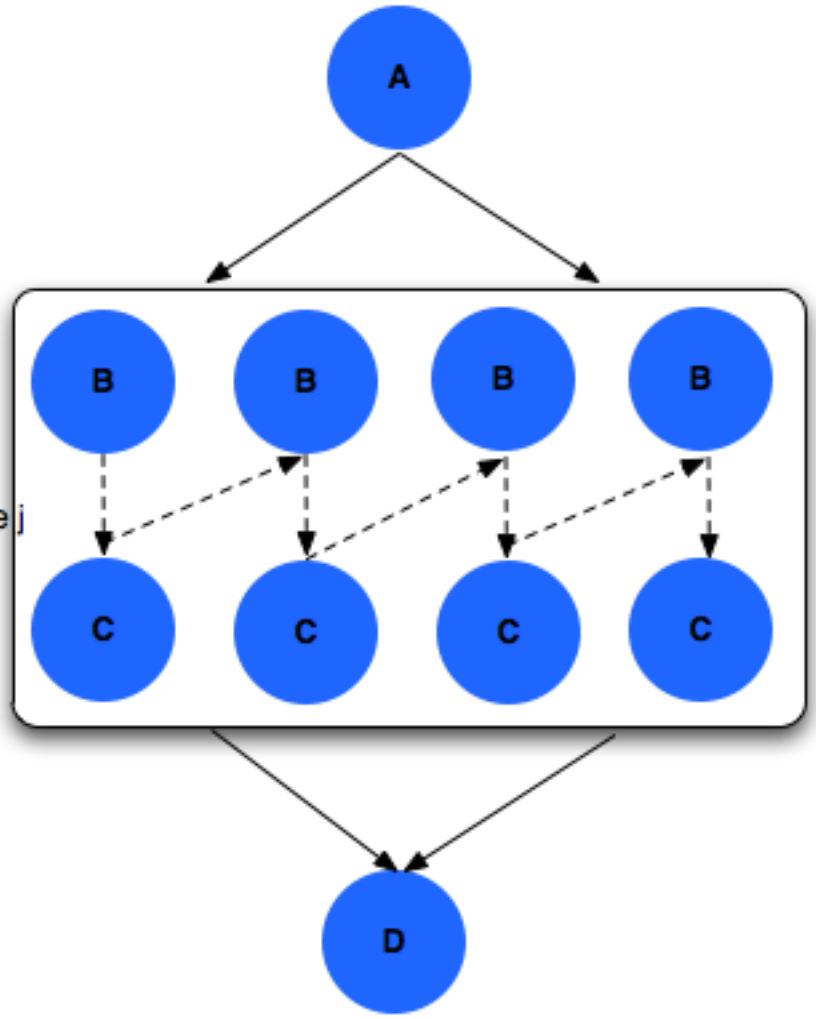
```
$ pegasus-plan &ndash;-dax example.dax --dir ./dags --cluster label,horizontal &ndash;s siteX
 &ndash;-output local --verbose
```

Fi



2. Workflow After Label Clustering

The Dotted Arrows indicate the topological sort ordering in the cluster of the jobs. This is the order with which the j jobs will be executed on the single node.



3. Workflow After Horizontal Clustering applied to Label based Clustered Workflow

Execution of the Clustered Job

The execution of the clustered job on the remote site, involves the execution of the smaller constituent jobs either

- **sequentially on a single node of the remote site**

  The clustered job is executed using **seqexec**, 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 “**seqexec**” 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     /shared/PEGASUS/bin/seqexec INSTALLED       INTEL32::LINUX NULL
```

  By default, the entry for seqexec on a site is automatically picked up if $PEGASUS_HOME or $VDS_HOME is specified in the site catalog for that site.

- **On multiple nodes of the remote site using MPI**

  The clustered job is executed using mpiexec, a wrapper mpi program written in C that is distributed as part of the PEGASUS. It is only distributed as source not as binary. The wrapper ends up being run on every mpi node, with the first one being the master and the rest of the ones as workers. The number of instances of mpiexec that are invoked is equal to the value of the globus rsl key nodecount. The master distributes the smaller constituent jobs to the workers.

  For e.g. If there were 10 jobs in the merged job and nodecount was 5, then one node acts as master, and the 10 jobs are distributed amongst the 4 slaves on demand. The master hands off a job to the slave node as and when it gets free. So initially all the 4 nodes are given a single job each, and then as and when they get done are handed more jobs till all the 10 jobs have been executed.

  To use “mpiexec” for executing the clustered job on a siteX, there needs to be an entry in the transformation catalog for an executable with the logical name mpiexec and namespace as pegasus.

```
#site  transformation   pfn            type                architecture    profiles

siteX    pegasus::seqexec     /shared/PEGASUS/bin/mpiexec INSTALLED       INTEL32::LINUX NULL
```

  Another added advantage of using mpiexec, is that regular non mpi code can be run via MPI.

  Both the clustered job and the smaller constituent jobs are invoked via kickstart, unless the clustered job is being run via mpi (mpiexec). Kickstart is unable to launch mpi jobs. If kickstart is not installed on a particular site i.e. the gridlaunch attribute for site is not specified in the site catalog, the jobs are invoked directly.

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 “collapser” 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="collapser" >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 “collapser” 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,collapser=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 compute 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. Currently, 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.

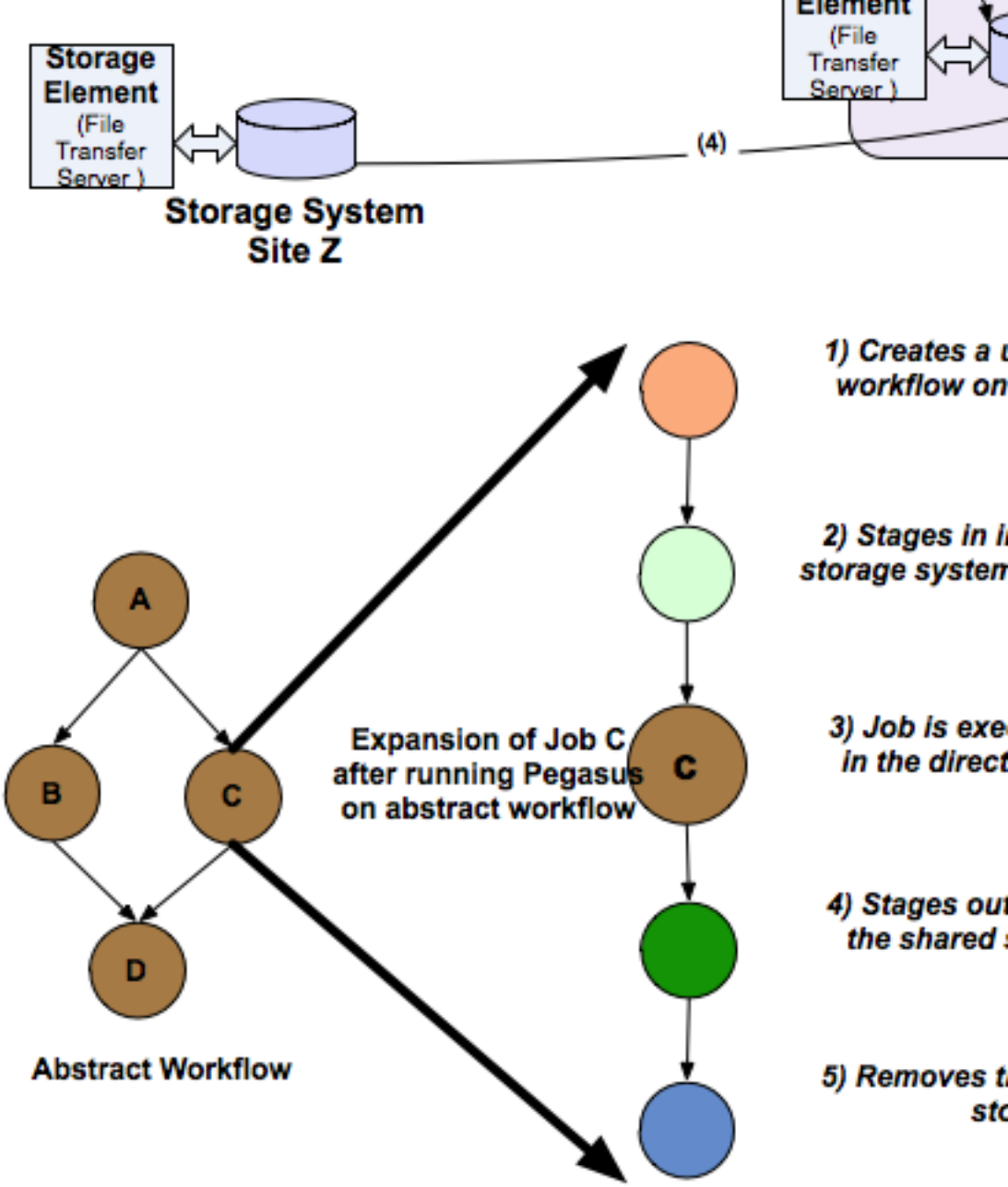


Figure 9.5. Defa
File System

Execution of jobs on storage systems shared between the worker nodes and headnode / storage element.

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.

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 compute site only if there is no way to execute it on the submit host. For e.g if the file server specified for the 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 9.9. 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

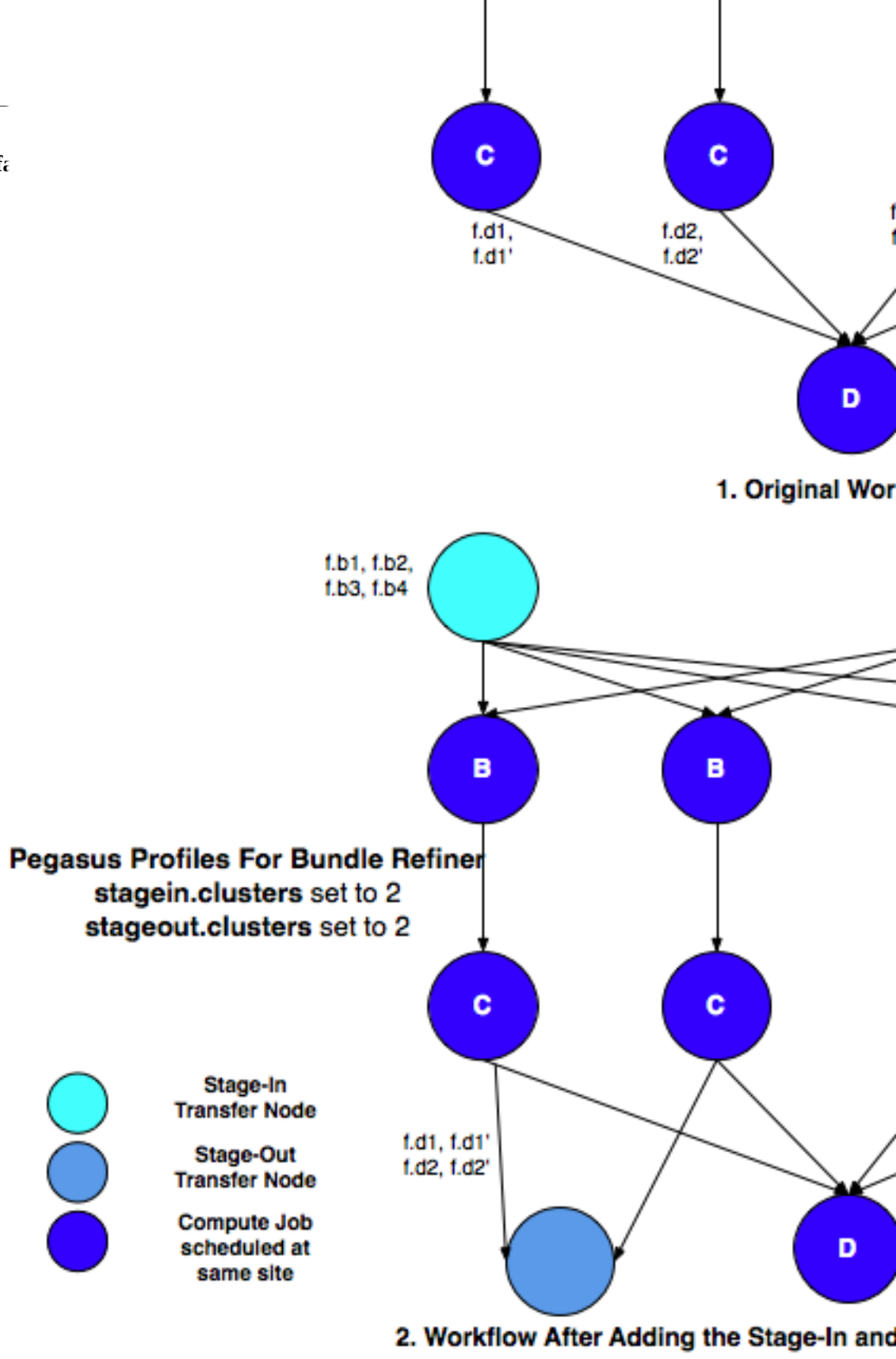
1. either with the execution sites in the site catalog

2. OR globally in the properties file

Table 9.10. Pegasus Profile Keys For the Bundle 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 9.6. Defa
File System



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 9.11. 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.

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**

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 9.12. 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.

Second Level Staging

By default, Pegasus executes the jobs in the workflow specific directory created on the shared filesystem of a compute site. However, if a user wants Pegasus can execute the jobs on the worker nodes filesystem. When the jobs are executed on the worker node, they pull the input data for the job from the workflow specific directory on the staging server (usually the shared filesystem on the compute site) to a directory on the worker node filesystem, and after the job has completed stages out the output files from the worker node to the workflow specific execution directory.

The separate data stagein and stageout jobs are still added to the workflow. They are responsible for getting the input data to the workflow specific directory on the staging server (usually the shared filesystem on the compute site) , and pushing out the output data to final storage site from that directory.

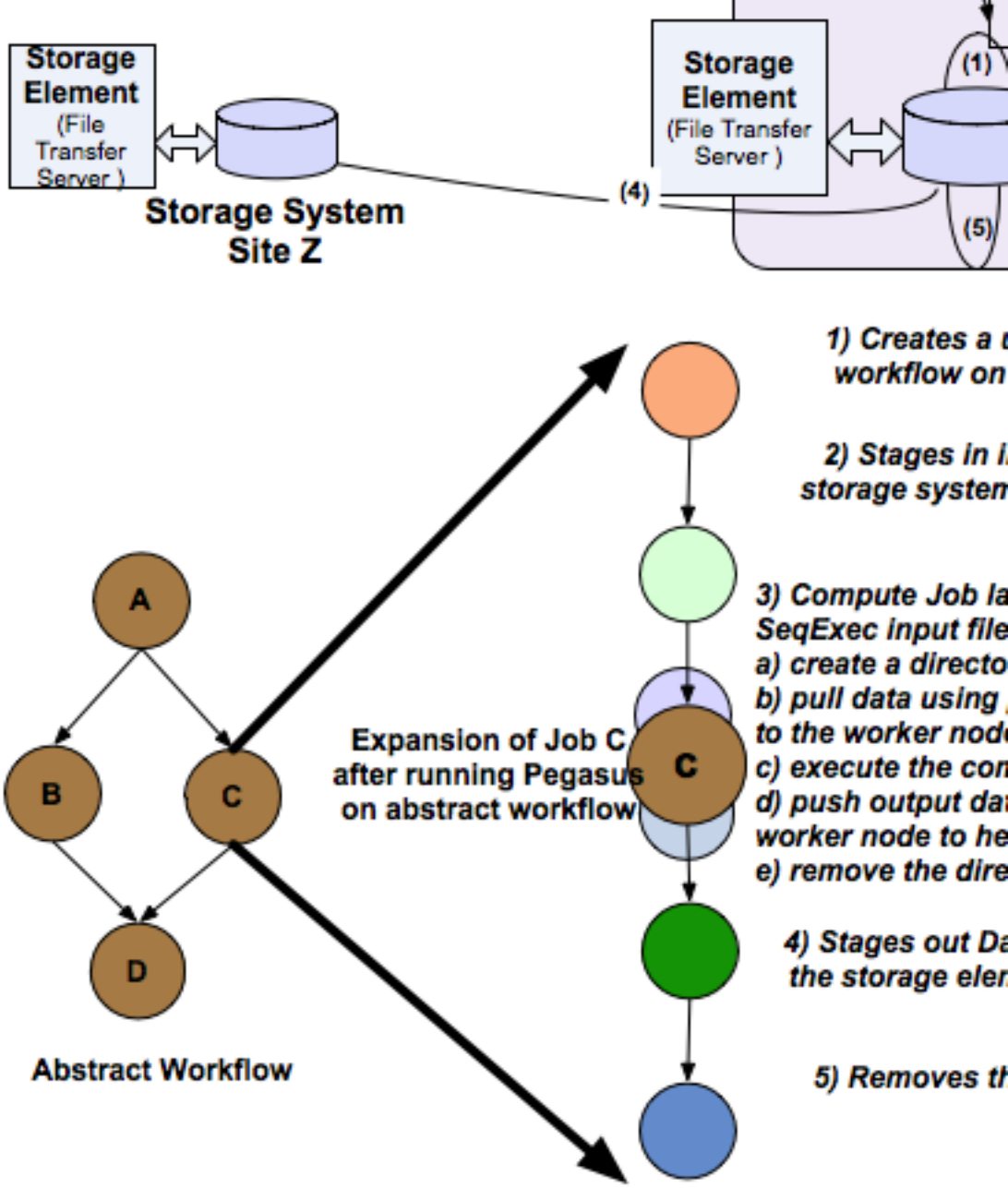


Figure 9.7. Secon

Execution of jobs on host local non shared file systems .

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.

To turn on second level staging for the workflows users should set the following properties

```
pegasus.execute.*.filesystem.local = true       # Turn on second-level staging (SLS)
pegasus.transfer.sls.s3.stage.sls.file = false  # Do not transfer .sls files via transfer jobs
pegasus.gridstart = SeqExec                     # Use SeqExec to launch the jobs
```

Hierarchical Workflows

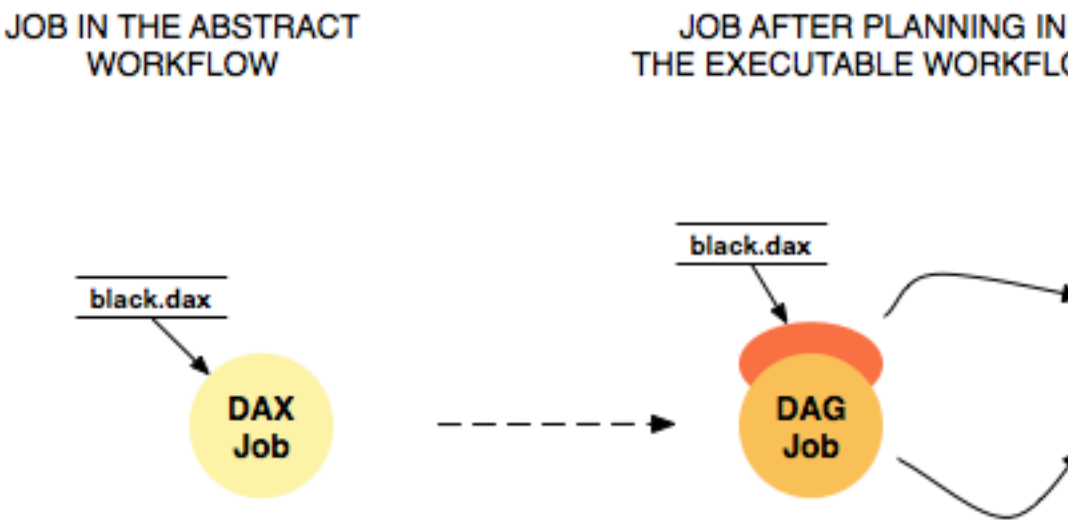
Introduction

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

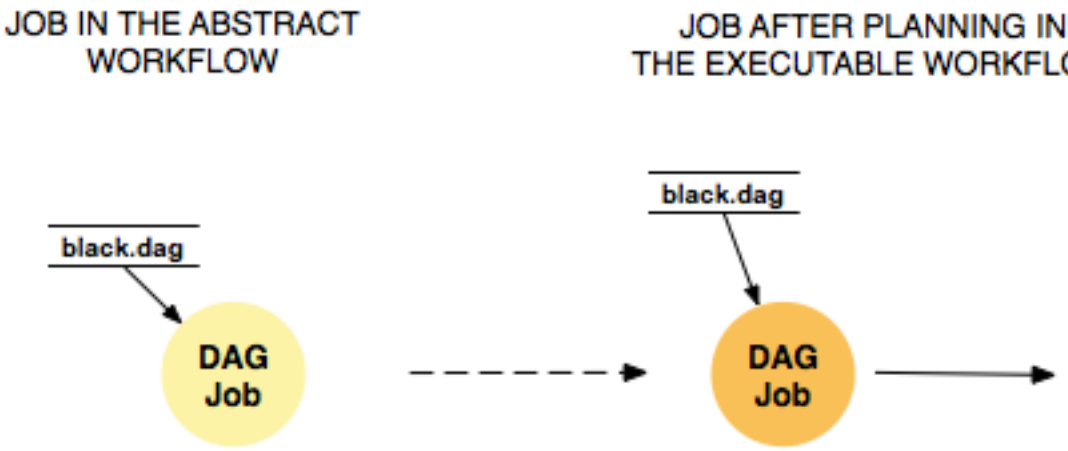
1. daxjob - refers to a sub workflow represented as a DAX. During the planning of a workflow, the DAX jobs are mapped to condor dagman jobs that have pegasus plan invocation on the dax (referred to in the DAX job) as the prescript.

Figure 9.8. 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 9.9. 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 9.13. 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
```

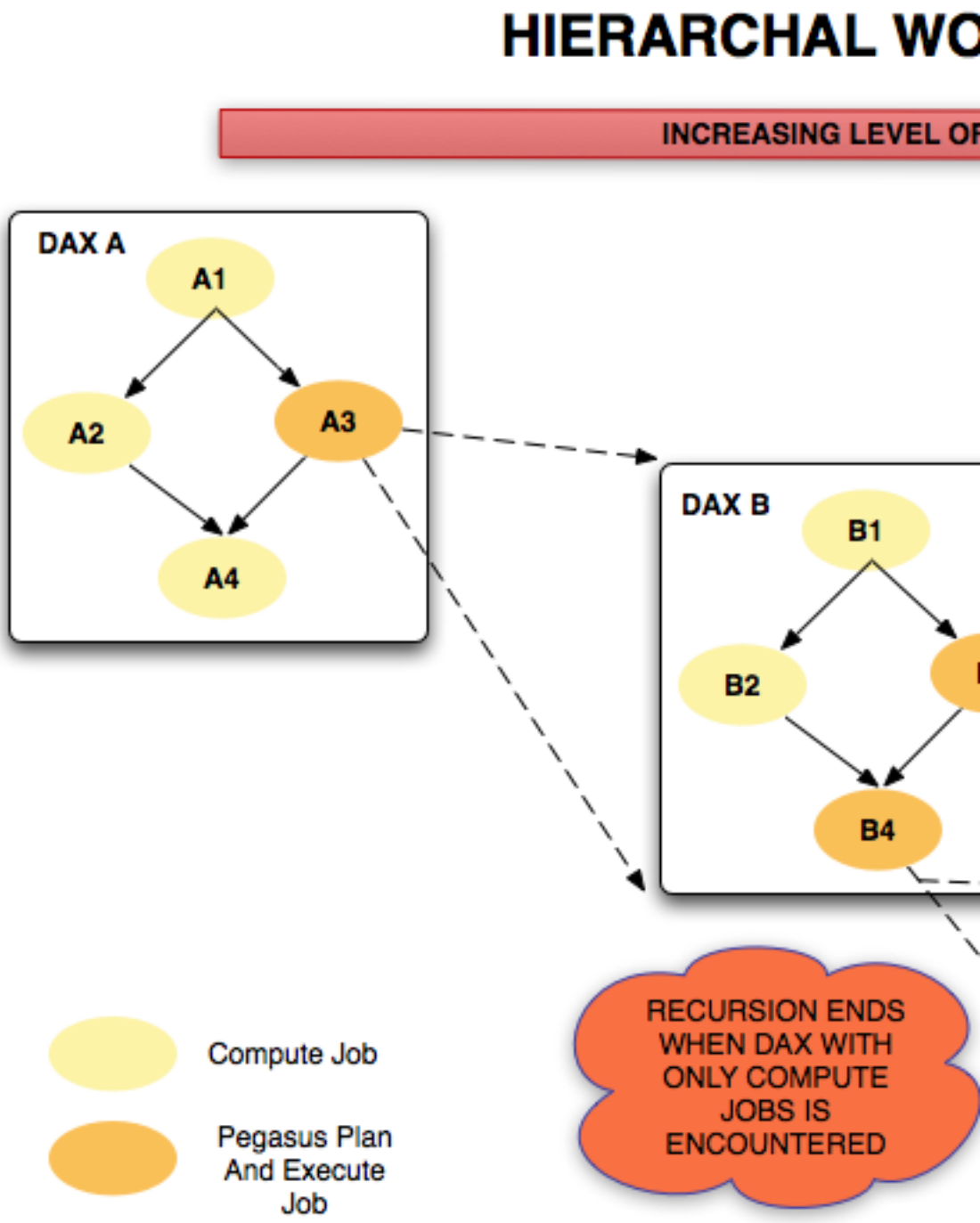
The above is useful in the case, where you are staging out the output files to a storage site, and you want the child sub workflow to stage these files from the storage output site instead of the workflow execution directory where the files were originally created.

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) .

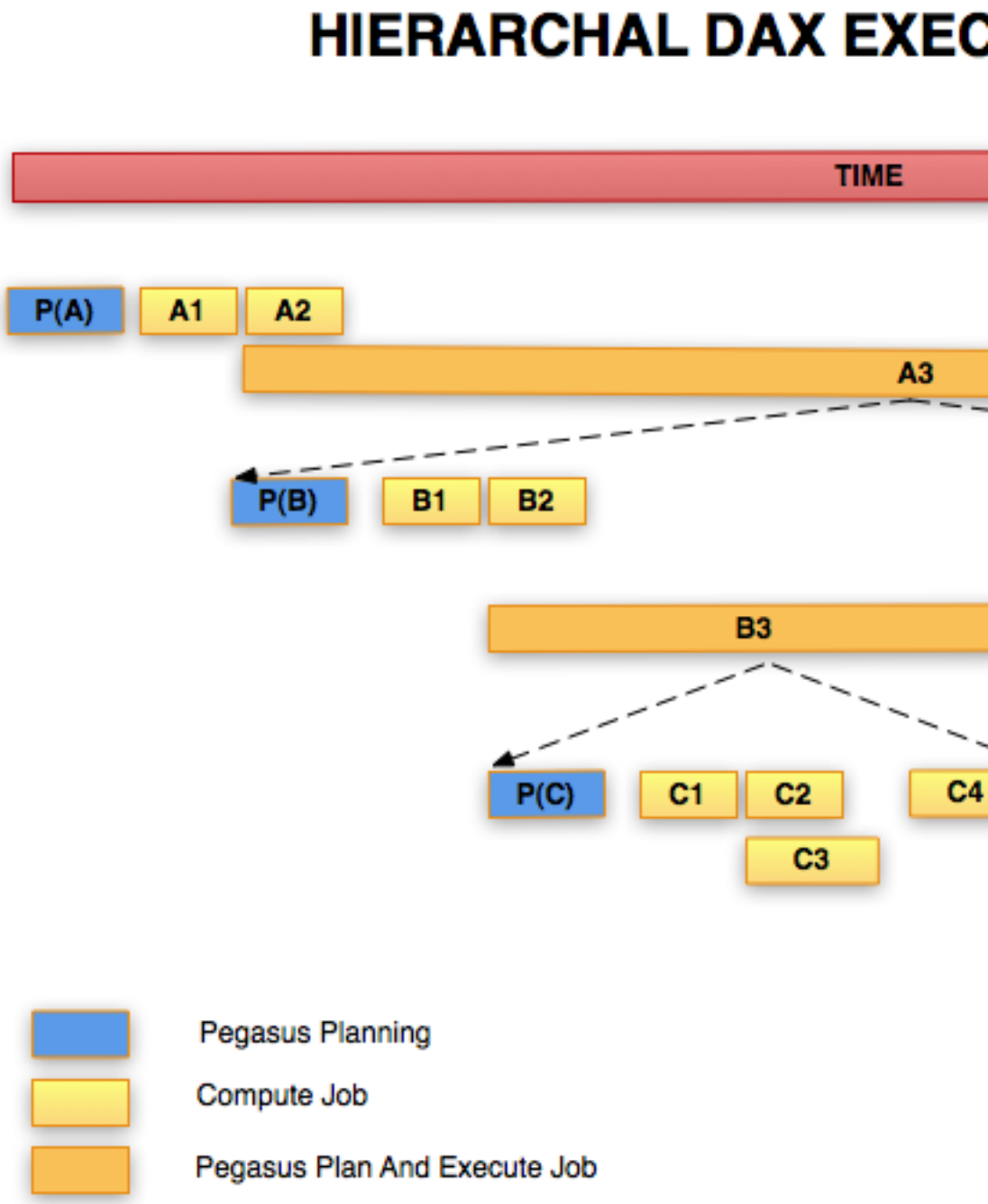
The figure below illustrates an example with recursion 2 levels deep.

Figure 9.10. R


HIERARCHAL WO
INCREASING LEVEL OF
DAX A
A1
A2
A3
A4
DAX B
B1
B2
B4
RECURSION ENDS WHEN DAX WITH ONLY COMPUTE JOBS IS ENCOUNTERED
Compute Job
Pegasus Plan And Execute Job

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

Figure 9.11. 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 9.14. 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.

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 9.15.

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

attribute	optional?	type	meaning
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 9.16.

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 9.17.

keyword	job life-cycle state	meaning
never	never	(default). Never notify of anything. This is useful to temporarily disable an existing notifications.

keyword	job life-cycle state	meaning
start	submit	create a notification when the job is submitted.
on_error	end	after a job finishes with failure (exitcode != 0).
on_success	end	after a job finishes with success (exitcode == 0).
at_end	end	after a job finishes, regardless of exitcode.
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 9.18.

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 [http://pegasus.isi.edu/wms/docs/3.0/javadoc/index.html] is well documented using Javadoc for ADAGs [http://pegasus.isi.edu/wms/docs/3.0/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.*;

public class BlackDiamondDAX {

    /**
     * Create an example DIAMOND DAX
     * @param args
     */
    public static void main(String[] args) {
        if (args.length != 3) {
            System.out.println("Usage: java ADAG <site_handle> <pegasus_location> <filename.dax>");
            System.exit(1);
        }

        try {
            Diamond(args[0], args[1]).writeToFile(args[2]);
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

    private static ADAG Diamond(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(When.start,"/pegasus/libexec/notification/email -t notify@example.com");
        dax.addNotification(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(When.start,"/pegasus/libexec/notification/email -t notify@example.com");
        j1.addNotification(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(When.start,"/pegasus/libexec/notification/email -t notify@example.com");
        j2.addNotification(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(When.start,"/pegasus/libexec/notification/email -t notify@example.com");
        j3.addNotification(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(When.start,"/pegasus/libexec/notification/email -t notify@example.com");
        j4.addNotification(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;
    }
}
```

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

This chapter contains reference material for all the command-line tools distributed with Pegasus.

pegasus-version

pegasus-version is a simple command-line tool that reports the version number of the Pegasus distribution being used.

In its most basic invocation, it will show the current version of the Pegasus software you have installed:

```
$ pegasus-version
3.1.0cvs
```

If you want to know more details about the installed version, i.e. which system it was compiled for and when, use the *long* or *full* mode:

```
$ pegasus-version -f
3.1.0cvs-x86_64_cent_5.6-20110706191019Z
```

The reported version may sometimes not be the version you would expect, if the Pegasus jar file got updated but not the remainder of the installation environment. To check this case, you use *match* mode:

```
$ pegasus-version -m
Compiled into PEGASUS: 20110706191019Z x86_64_cent_5.6
Installation provides: 20110706191019Z x86_64_cent_5.6
OK: Internal version matches installation.
Complete version info: 3.1.0cvs-x86_64_cent_5.6-20110706191019Z
```

The *match* mode implies *long* mode, and will thus show the full version as part of its output.

In *quiet* mode, which can only be applied to *match* mode, no output is written unless there is an error. You would use *quiet* mode to check the exit code of `pegasus-version` to determine a problem while being in a scripted environment.

```
$ pegasus-version -mq
$ echo $?
0
```

The manual page for `pegasus-version` will show more details, and long options availble to the user.

pegasus-plan

pegasus-plan generates an executable workflow from an abstract workflow description (DAX).

SYNOPSIS

pegasus-plan -h|--help

pegasus-plan -V|--version

pegasus-plan [-Dprop [..]] -d <dax file> [-b prefix] [--conf <path to properties file>] [-c f1[,f2[..]]] [-C <clustering technique>] [--dir <base directory for o/p files>] [-f] [--force-replan] [--inherited-rc-files] [-j job-prefix] [-n] [-o <output site>] [-r[directoryname]] [--relative-dir <relative directory to base directory>] [--relative-submit-dir <relative sub- mit directory to the base directory>] [-s site1[,site2[..]]] [-v] [-q] [-V] [-h]

DESCRIPTION

The pegasus-plan command takes in as input the DAX and generates an executable workflow usually in form of condor submit files, which can be submitted to an execution site for execution.

As part of generating the executable workflow, the planner needs to discover

- data

  The Pegasus Workflow Planner ensures that all the data required for the execution of the executable workflow is transferred to the execution site by adding transfer nodes at appropriate points in the DAG. This is done by looking up an appropriate Replica Catalog to determine the locations of the input files for the various jobs. At present the default replica mechanism used is RLS .

  The Pegasus Workflow Planner also tries to reduce the workflow, unless specified otherwise. This is done by deleting the jobs whose output files have been found in some location in the Replica Catalog . At present no cost metrics are used. However preference is given to a location corresponding to the execution site.

  The planner can also add nodes to transfer all the materialized files to an output site. The location on the output site is determined by looking up the site catalog file, the path to which is picked up from the **pegasus.catalog.site.file** property value.

- executables

  The planner looks up a Transformation Catalog to discover locations of the executables referred to in the executable workflow. Users can specify INSTALLED or STAGEABLE executables in the catalog. Stageable executables can be used by Pegasus to stage executables to resources where they are not pre-installed.

- resources

  The layout of the sites , where Pegasus can schedule jobs of a workflow are described in the Site Catalog. The planner looks up the site catalog to determine for a site what directories a job can be executed in, what servers to use for staging in and out data and what jobmanagers (if applicable) can be used for submitting jobs.

The data and executable locations can now be specified in DAX'es conforming to DAX schema version 3.2 or higher.

ARGUMENTS

Any option will be displayed with its long options synonym(s).

- **-Dprop**

  The -D options allows an experienced user to override certain properties which influence the program execution, among them the default location of the user's properties file and the PEGASUS home location. One may set several CLI properties by giving this option multiple times. The -D option(s) must be the first option on the command line. A CLI property take precedence over the properties file property of the same key.

- **-d filename**

  --dax filename

  The DAX is the XML input file that describes an abstract workflow.

  This is a mandatory option, which has to be used.

- **-b prefix**

--basename prefix

The basename prefix to be used while constructing per workflow files like the dagman file (.dag file) and other work- flow specific files that are created by Condor. Usually this prefix, is taken from the name attribute specified in the root element of the dax files.

- **-c list of cache files**

  --cache list of cache files

  A comma separated list of paths to replica cache files that override the results from the replica catalog for a particular lfn.

  Each entry in the cache file describes a LFN , the corresponding PFN and the associated attributes. The pool attribute should be specified for each entry.

```
LFN_1 PFN_1 pool=[site handle 1]
LFN_2 PFN_2 pool=[site handle 2]
...
LFN_N PFN_N [site handle N]
```

  To treat the cache files as supplemental replica catalogs set the property **pegasus.catalog.replica.cache.asrc** to true. This results in the mapping in the cache files to be merged with the mappings in the replica catalog. Thus, for a particular lfn both the entries in the cache file and replica catalog are available for replica selection.

- **-C comma separated list of clustering styles**

  --cluster comma separated list of clustering styles

  This mode of operation results in clustering of n compute jobs into a larger jobs to reduce remote scheduling overhead. You can specify a list of clustering techniques to recursively apply them to the workflow. For example, this allows you to cluster some jobs in the workflow using horizontal clustering and then use label based clustering on the intermediate workflow to do vertical clustering.

  The clustered jobs can be run at the remote site, either sequentially or by using mpi. This can be specified by setting the property pegasus.job.aggregator. The property can be overridden by associating the PEGASUS profile key **collapser** either with the transformation in the transformation catalog or the execution site in the site catalog. The value specified (to the property or the profile), is the logical name of the transformation that is to be used for clustering jobs. Note that clustering will only happen if the corresponding transformations are catalogued in the transformation catalog.

  PEGASUS ships with a clustering executable **seqexec** that can be found in $PEGASUS_HOME/bin directory. It runs the jobs in the clustered job sequentially on the same node at the remote site.

  In addition, a mpi wrapper **mpiexec** is distributed as source with the PEGASUS. It can be found in $PEGASUS_HOME/src/tools/cluster directory. The wrapper is run on every mpi node, with the first one being the master and the rest of the ones as workers. The number of instances of mpiexec that are invoked is equal to the value of the globus rsl key nodecount. The master distributes the smaller constituent jobs to the workers. For e.g. If there were 10 jobs in the clustered job and nodecount was 5, then one node acts as master, and the 10 jobs are distributed amongst the 4 slaves on demand. The master hands off a job to the slave node as and when it gets free. So initially all the 4 nodes are given a single job each, and then as and when they get done are handed more jobs till all the 10 jobs have been executed.

  By default, seqexec is used for clustering jobs unless overridden in the properties or by the **pegasus profile key collapser**.

  The following type of clustering styles are currently supported

  1. **horizontal**

     is the style of clustering in which jobs on the same level are aggregated into larger jobs. A level of the workflow is defined as the greatest distance of a node, from the root of the workflow. Clustering occurs only on jobs of the same type i.e they refer to the same logical transformation in the transformation catalog.

The granularity of clustering can be specified by associating either the PEGASUS profile key **clusters.size** or the PEGASUS profile key **clusters.num** with the transformation. The clusters.size key indicates how many jobs need to be clustered into the larger clustered job. The clusters.num key indicates how many clustered jobs are to be created for a particular level at a particular execution site. If both keys are specified for a particular transformation, then the clusters.num key value is used to determine the clustering granularity.

2. **label**

   is the style of clustering in which you can label the jobs in your workflow. The jobs with the same level are put in the same clustered job. This allows you to aggregate jobs across levels, or in a manner that is best suited to your application.

   To label the workflow, you need to associate PEGASUS profiles with the jobs in the DAX. The profile key to use for labelling the workflow can be set by the property **pegasus.clusterer.label.key** . It defaults to label, meaning if you have a PEGASUS profile key label with jobs, the jobs with the same value for the pegasus profile key label will go into the same clustered job.

- **--conf path to properties file**

  The path to properties file that contains the properties planner needs to use while planning the workflow.

- **--dir dir name**

  The base directory where you want the output of the Pegasus Workflow Planner usually condor submit files, to be generated. Pegasus creates a directory structure in this base directory on the basis of username, VO Group and the label of the workflow in the DAX.

  By default the base directory is the directory from which one runs the pegasus-plan command.

- **-f**

  --force

  This bypasses the reduction phase in which the abstract DAG is reduced, on the basis of the locations of the output files returned by the replica catalog. This is analogous to a make style generation of the executable workflow.

- **--force-replan**

  By default, for hierarchal workflows if a dax job fails, then on job retry the rescue dag of the associated workflow is submitted. This option causes Pegasus to replan the dax job in case of failure instead.

- **-g**

  --group

  The VO Group to which the user belongs to.

- **-h**

  --help displays the options to pegasus-plan command.

- **--inherited-rc-files** comma separated list of replica catalog files

  A comma separated list of paths to replica files. Locations mentioned in these have a lower priority than the locations in the DAX file. This option is usually used internally for hierarichal workflows, where the file locations mentioned in the parent (encompassing) workflow DAX, passed to the sub workflows (corresponding) to the dax jobs.

- **-j**

  --j

  The job prefix to be applied for constructing the filenames for the job submit files.

- **-n**

--no-cleanup

This results in the generation of the separate cleanup workflow that removes the directories created during the execution of the executable workflow. The cleanup workflow is to be submitted after the executable workflow has finished. If this option is not specified, then Pegasus adds cleanup nodes to the executable workflow itself that cleanup files on the remote sites when they are no longer required.

- **-o** output site

  --output output site

  The output site where all the materialized data is transferred to.

  By default the materialized data remains in the working directory on the execution site where it was created. Only those output files are transferred to an output site for which transfer attribute is set to true in the DAX.

- **-q**

  --quiet

  decreases the logging level

- **--relative-dir** dir name

  The directory relative to the base directory where the executable workflow it to be generated and executed. This over- rides the default directory structure that Pegasus creates based on username, VO Group and the DAX label.

- **--relative-submit-dir** dir name

  The directory relative to the base directory where the executable workflow it to be generated. This overrides the default directory structure that Pegasus creates based on username, VO Group and the DAX label. By specifying --relative-dir and --relative-submit-dir you can have different relative execution directory on the remote site and different relative sub- mit directory on the submit host.

- **-s** list of execution sites

  --sites list of execution sites

  A comma separated list of execution sites on which the workflow is to be executed. Each of the sites should have an entry in the site catalog, that is being used. To run on the submit host, specify the execution site as **local .**

  In case this option is not specified, all the sites in the site catalog are picked up as candidates for running the work- flow.

- **-s**

  --submit

  Submits the generated executable workflow using pegasus-run script in $PEGASUS_HOME/bin directory.

  By default, the Pegasus Workflow Planner only generates the Condor submit files and does not submit them.

- **-v**

  --verbose

  increases the verbosity of messages about what is going on.

  By default, all FATAL, ERROR, CONSOLE and WARN messages are logged.

  The logging hierarchy is as follows

  - FATAL

- ERROR
- CONSOLE
- WARN
- INFO
- CONFIG
- DEBUG
- TRACE

For example, to see the INFO, CONFIG and DEBUG messages additionally, set -vvv.

- **-V**

  --version

  Displays the current version number of the Pegasus Workflow Management System.

RETURN VALUE

If the Pegasus Workflow Planner is able to generate an executable workflow successfully, the exitcode will be 0. All runtime errors result in an exitcode of 1. This is usually in the case when you have mis-configured your catalogs etc. In the case of an error occurring while loading a specific module implementation at run time, the exitcode will be 2. This is usually due to factory methods failing while loading a module. In case of any other error occurring during the running of the command, the exit- code will be 1. In most cases, the error message logged should give a clear indication as to where things went wrong.

PEGASUS PROPERTIES

This is not an exhaustive list of properties used. For the complete description and list of properties refer to **$PEGASUS_HOME/doc/advanced-properties.pdf** or the properties chapter.

- **pegasus.catalog.replica**

  Specifies the type of replica catalog to be used. If not specified, then RLS is used as a Replica Catalog Backend.

- **pegasus.catalog.replica.url**

  Contact string to access the replica catalog. In case of RLS it is the RLI url. I

- **pegasus.dir.exec**

  A suffix to the workdir in the site catalog to determine the current working directory. If relative, the value will be appended to the working directory from the site.config file. If absolute it constitutes the working directory.

- **pegasus.catalog.transformation**

  Specifies the type of transformation catalog to be used. One can use either a file based or a database based transforma- tion catalog. At present the default is Text .

- **pegasus.catalog.transformation.file**

  The location of file to use as transformation catalog.

- **pegasus.catalog.site**

  Specifies the type of site catalog to be used. At present the default is **XML3** .

- **pegasus.catalog.site.file**

The location of file to use as a site catalog. If not specified, then default value of $PEGASUS_HOME/etc/sites.xml is used in case of the xml based site catalog and $PEGASUS_HOME/etc/sites.txt in case of the text based site catalog.

- **pegasus.code.generator**

  The code generator to use. By default, Condor submit files are generated for the executable workflow. Setting to **Shell** results in Pegasus generating a shell script that can be executed on the submit host.

FILES

- **$PEGASUS_HOME/etc/dax-3.2.xsd**

  It is the suggested location of the latest DAX schema to produce DAX output.

- **$PEGASUS_HOME/etc/tc.data.text**

  is the suggested location for the file corresponding to the Transformation Catalog.

- **$PEGASUS_HOME/etc/sc-3.0.xsd**

  is the suggested location of the latest Site Catalog schema that is used to create the XML3 version of the site catalog

- **$PEGASUS_HOME/etc/sites.xml3 | $PEGASUS_HOME/etc/sites.xml**

  is the suggested location for the file containing the site information.

- **$PEGASUS_HOME/lib/pegasus.jar**

  contains all compiled Java bytecode to run the Pegasus Workflow Planner.

pegasus-run

pegasus-run executes a workflow that has been planned using pegasus-plan.

SYNTAX

pegasus-run [options] [rundir]

DESCRIPTION

The pegasus-run command executes a workflow that has been planned using pegasus-plan. By default pegasus-run can be invoked either in the planned directory with no options and arguments or just the full path to the run directory. pegasus-run also can be used to resubmit a failed workflow by running the same command again.

ARGUMENTS

By default pegasus-run does not require any options or arguments if invoked from within the planned workflow directory. If running the command outside the workflow directory then a full path to the workflow directory needs to be specified.

pegasus-run takes the following options

- **-Dpropkey=propvalue**

  The -D options allows an advanced user to override certain properties which influence pegasus-run. One may set several CLI properties by giving this option multiple times. The -D option(s) must be the first option on the command line. CLI properties take precedence over the file-based properties of the same key.

- **-c pegasus_properties_file**

  --conf pegasus_properties_file

Provide a property file to override the default pegasus properties file from the planning directory. Ordinary users do not need to use this option unless the specifically want to override several properties

- **-d**

  --debug level

  Set the debug level for the client. Default is 0.

- **-v**

  --verbose

  Raises debug level. Each invocation increase the level by 1.

- **--grid**

  Enable grid checks to see if your submit machine is GRID enabled.

- **rundir**

  Is the full qualified path to the base directory containing the planned workflow dag and submit files. This is optional if pegasus-run command is invoked from within the run directory.

RETURN VALUE

If the workflow is submitted for execution pegasus-run returns with an exit code of 0. However, in case of error, a non-zero exit code indicates problems. An error message clearly marks the cause.

FILES AND DIRECTORIES

The following files are created, opened or written to.

- **braindump**

  This file is located in the rundir. pegasus-run uses this file to find out paths to several other files, properties configurations etc

- **pegasus.?????????.properties**

  This file is located in the rundir. pegasus-run uses this properties file by default to configure its internal settings.

- **workflowname.dag**

  pegasus-run uses the workflowname.dag or workflowname.sh file and submits it either to condor for execution or runs it locally in a shell environment

PROPERTIES

pegasus-run reads its properties from several locations. By default the properties are read from the file RUNDIR/pegasus.??????????.properties. If the --conf option is provided then the properties file provided in the conf option is used. If both the default properties file in the rundir as well as --conf file is not found or provided then the properties file located at $HOME/.pegasusrc will be used. Additionally properties can be provided individually using the -D<key>=<value> option on the command line before all other options. These properties will override properties provided using either --conf or RUNDIR/pegasus.???????.properties or the $HOME/.pegasusrc

The merge logic is CONF PROPERTIES || DEFAULT RUNDIR PROPERTIES || PEGASUSRC overriden by Command line properties

ENVIRONMENT VARIABLES

The following environment variables are used by pegasus-run

- **PATH**

  The path variable is used to locate binaries for condor-submit-dag, condor-dagman, condor-submit,pegasus-submit-dag, pegasus-dagman and pegasus-monitord

pegasus-remove

pegasus-remove removes a workflow that has been planned and submitted using pegasus-plan and pegasus-run

SYNTAX

pegasus-remove [options] [rundir]

DESCRIPTION

The pegasus-remove command removes a submitted/running workflow that has been planned and submitted using pegsus-plan and pegasus-run. The command can be invoked either in the planned directory with no options and arguments or just the full path to the run directory.

ARGUMENTS

By default pegasus-remove does not require any options or arguments if invoked from within the planned workflow directory. If running the command outside the workflow directory then a full path to the workflow directory needs to be specified or the dagid of the workflow to be removed.

pegasus-remove takes the following options

- **-d**

  --dagid dagid

  The workflow dagid to remove.

- **-v**

  --verbose

  Raises debug level. Each invocation increase the level by 1.

- **rundir**

  Is the full qualified path to the base directory containing the planned workflow dag and submit files. This is optional if pegasus-remove command is invoked from within the run directory.

RETURN VALUE

If the workflow is removed successfully pegasus-remove returns with an exit code of 0. However, in case of error, a non-zero exit code indicates problems. An error message clearly marks the cause.

ENVIRONMENT VARIABLES

The following environment variables are used by pegasus-remove

- **PATH**

  The path variable is used to locate binaries for condor-rm

pegasus-status

pegasus-status reports on the status of a workflow.

SYNTAX

`pegasus-status` -h | --help

`pegasus-status` -V | --version

`pegasus-status` [-w | --watch *`s`*] [-L | --[no]legend] [-c | --[no]color] [-U | --[no]utf8] [-Q | --[no]queue] [-v | --verbose] [-d | --debug] [-u | --user *`name`*] [-i | --[no]idle] [--[no]held] [--[no]heavy] [-j | --jobtype *`jt`*] [-s | --site *`sid`*] [-l | --long] [-S | --[no]success] [*`rundir`*]

pegasus-status shows the current state of the Condor Q and a workflow, depending on settings. If no valid run directory could be determined, including the current directory, **pegasus-status** will show all jobs of the current user and no workflows. If a run directory was specified, or the current directory is a valid run directory, status about the workflow will also be shown.

Many option will modify the behavior of this program, not withstanding a proper UTF-8 capable terminal, watch mode, the presence of jobs in the queue, progress in the workflow directory, etc.

ARGUMENTS

`-h`, `--help` Prints a concise help and exits.

`-V`, `--version` Prints the version information and exits.

`-w` *`sec`*, `--watch` *`sec`* This option enables the *watch mode.* In watch mode, the program repeated polls the status sources and shows them in an updating window. The optional argument *`sec`* to this option determines how often these sources are polled.

We *strongly* recommend to set this interval not too low, as frequent polling will degrade the scheduler performance and increase the host load. In watch mode, the terminal size is the limiting factor, and parts of the output may be truncated to fit it onto the given terminal.

Watch mode is disabled by default. The *`sec`* argument defaults to 60 seconds.

`-L`, `--legend`, `--nolegend` This option shows a legend explaining the columns in the output, or turns off legends.

By default, legends are turned off to save terminal real estate.

`-c`, `--color`, `--nocolor` This option turns on (or off) ANSI color escape sequences in the output. The single letter option can only switch on colors.

By default, colors are turned off, as they will not display well on a terminal with black background.

`-U`, `--utf8`, `--noutf8` This option turns on (or off) the output of Unicode box drawing characters as UTF-8 encoded sequences. The single option can only turn on box drawing characters.

The defaults for this setting depend on the `LANG` environment variable. If the variable contains a value ending in something indicating UTF-8 capabilities, the option is turned on by default. It is off otherwise.

`-Q`, `--queue`, `--noqueue` This option turns on (or off) the output from parsing Condor Q.

By default, Condor Q will be parsed for jobs of the current user. If a workflow run directory is specified, it will furthermore be limited to jobs only belonging to the workflow.

`-v`, `--verbose` This option increases the expert level, showing more information about the Condor Q state. Being an incremental option, two incrases are supported.

Additionally, the signals *SIGUSR1* and *SIGUSR2* will increase and decrease the expert level respectively during run-time.

By default, the simplest queue view is enabled.

`-d`, `--debug` This is an internal debugging tool and should not be used outside the development team. As incremental option, it will show Pegasus-specific ClassAd tuples for each job, more in the second level.

By default, debug mode is off.

`-u` *`name`*, `--user` *`name`* This option permits to query the queue for a different *`user`* than the current one. This may be of interest, if you are debugging the workflow of another user.

By default, the current user is assumed.

`-i`, `--idle`, `--noidle` With this option, jobs in Condor state *idle* are omitted from the queue output.

By default, *idle* jobs are shown.

`--held`, `--noheld` This option enables or disabled showing of the reason a job entered Condor's *held* state. The reason will somewhat destroy the screen layout.

By default, the reason is shown.

`--heavy`, `--noheavy` If the terminal is UTF-8 capable, and output is to a terminal, this option decides whether to use heavyweight or lightweight line drawing characters.

By default, heavy lines connect the jobs to workflows.

`-j` *`jt`*, `--jobtype` *`jt`* This option filters the Condor jobs shown only to the Pegasus jobtypes given as argument or arguments to this option. It is a multi-option, and may be specified multiple times, and may use comma-separated lists. Use this option with an argument *`help`* to see all valid and recognized jobtypes.

By default, all Pegasus jobtypes are shown.

`-s` *`site`*, `--site` *`site`* This option limits the Condor jobs shown to only those pertaining to the (remote) site *`site`* This is an multi-option, and may be specified multiple times, and may use comma-separated lists.

By default, all sites are shown.

`-l`, `--long` This option will show one line per sub-DAG, including one line for the workflow. If there is only a single DAG pertaining to the *`rundir`*, only total will be shown.

By default, only DAG totals (sums) are shown.

`-S`, `--success`, `--nosuccess` This option modifies the previous `--long` option. It will omit (or show) fully successful sub-DAGs from the output.

By default, all DAGs are shown.

`rundir` This option show statistics about the given DAG that runs in *`rundir`*. To gather proper statistics, **pegasus-status** needs to traverse the directory and all sub-directories. This can become an expensive operation on shared filesystems.

By default, the *`rundir`* is assumed to be the current directory. If the current directory is not a valid *`rundir`*, no DAG statistics will be shown.

RETURN VALUE

pegasus-status will typically return success in regular mode, and the termination signal in watch mode. Abnormal behavior will result in a non-zero exit code.

EXAMPLE

```
pegasus-status
```

This invocation will parse the Condor Q for the current user and show all her jobs. Additionally, if the current directory is a valid Pegasus workflow directory, totals about the DAG in that directory are displayed.

```
pegasus-status -l rundir
```

As above, but providing a specific Pegasus workflow directory in argument `rundir` and requesting to itemize sub-DAGs.

```
pegasus-status -j help
```

This option will show all permissible job types and exit.

```
pegasus-status -vvw 300 -Ll
```

This invocation will parse the queue, print it in high-expert mode, show legends, itemize DAG statistics of the current working directory, and redraw the terminal every five minutes with updated statistics.

RESTRICTIONS

Currently only supports a single (optional) run directory. If you want to watch multiple run directories, I suggest to open multiple terminals and watch them separately. If that is not an option, or deemed too expensive, you can ask `<pegasus-support at isi dot edu>` to extend the program.

pegasus-monitord

pegasus-monitord tracks a workflow's progress in real time and mines information.

SYNOPSIS

pegasus-monitord [--help | -help] [--verbose | -v] [--adjust | -a i] [--foreground | -N] [--no-daemon | -n] [--job | -j jobstate.log file] [--log | -l logfile] [--conf properties file] [--no-recursive] [--no-database | --no-events] [--replay | -r] [--no-notifications] [--notifications-max max_notifications] [--notifications-timeout timeout] [--sim | -s millisleep] [--db-stats] [--socket] [--output-dir | -o dir] [--dest | -d PATH or URL] [--encoding | -e bp | bson] DAGMan output file

DESCRIPTION

This program follows a workflow, parsing the output of DAGMAN's dagman.out file. In addition to generating the jobstate.log file, **pegasus-monitord** can also be used mine information from the workflow dag file and jobs' submit and output files, and either populate a database or write a NetLogger events file with that information. **pegasus-monitord** can also perform notifications when tracking a workflow's progress in real-time.

ARGUMENTS

- **-h**

  --help

  Prints a usage summary with all the available command-line options.

- **-v**

  --verbose

  Sets the log level for .**pegasus-monitord**. If omitted, the default level will be set to WARNING. When this option is given, the log level is changed to "INFO. If this option is repeated, the log level will be changed to DEBUG. The log level in **pegasus-monitord** can also be adjusted interactively, by sending the USR1 and USR2 signals to the process, respectively for incrementing and decrementing the log level.

- **-a i**

  --adjust i

  For adjusting time zone differences by i seconds, default is 0.

- **-N**

  --foreground

  Do not daemonize **pegasus-monitord**, go through the motions as if (Condor).

- **-n**

  --no-daemon

  Do not daemonize **pegasus-monitord**, keep it in the foreground (for debugging).

- **-j <jobstate.log>**

  --job <jobstate.log>

  Alternative location for the jobstate.log file. The default is to write a jobstate.log in the workflow directory. An absolute file name should only be used if the workflow does not have any sub-workflows, as each sub-workflow will generate its own jobstate.log file. If an alternative, non-absolute, filename is given with this option, **pegasus-monitord** will create one file in each workflow (and sub-workflow) directory with the filename provided by the user with this option. If an absolute filename is provided and sub-workflows are found, a warning message will be printed and **pegasus-monitord** will not track any sub-workflows.

- **-l <logfile>**

  --log-file <logfile>

  Specifies an alternative logfile to use instead of the monitord.log file in the main workflow directory. Differently from the jobstate.log file above, **pegasus-monitord** only generates one logfile per execution (and not one per workflow and sub-workflow it tracks).

- **--conf <properties file>**

  properties file is an alternative file containing properties in the **key=value** format, and allows users to override values read from the braindump.txt file. This option has precedence over the properties file specified in the braindump.txt file. Please note that these properties will apply not only to the main workflow, but also to all sub-workflows found.

- **--no-recursive**

  This options disables **pegasus-monitord** to automatically follow any sub-workflows that are found.

- **--no-database**

  --nodatabase

  --no-events

  Turns off generating events (when this option is given, **pegasus-monitord** will only generate the jobstate.log file). The default is to automatically log information to a SQLite database (see the **--dest** option below for more details). This option overrides any parameter given by the **--dest** option.

- **-r**

  --replay

  This option is used to replay the output of an already finished workflow. It should only be used after the workflow is finished (not necessarily successfully). If a jobstate.log file is found, it will be rotated. However, when using a database, all previous references to that workflow (and all its sub-workflows) will be erased from it. When outputing to a bp file, the file will be deleted. When running in replay mode, **pegasus-monitord** will always run with the **--no-daemon** option, and any errors will be output directly to the terminal. Also, **pegasus-monitord** will not process any notifications while in replay mode.

- **--no-notifications**

This options disables notifications completely, making **pegasus-monitord** ignore all the .notify files for all workflows it tracks.

- **--notifications-max <max notifications>**

  This option sets the maximum number of concurrent notifications that **pegasus-monitord** will start. When the **max notifications** limit is reached, **pegasus-monitord** will queue notifications and wait for a pending notification script to finish before starting a new one. If **max notifications** is set to 0, notifications will be disabled.

- **--notifications-timeout <timeout>**

  Normally, **pegasus-monitord** will start a notification script and wait indefinitely for it to finish. This option allows users to set up a maximum **timeout** that **pegasus-monitord** will wait for a notification script to finish before terminating it. If notification scripts do not finish in a reasonable amount of time, it can cause other notification scripts to be queued due to the maximum number of concurrent scripts allowed by **pegasus-monitord**. Additionally, until all notification scripts finish, **pegasus-monitord** will not terminate.

- **-s <millisleep>**

  --sim <millisleep>

  This option simulates delays between reads, by sleeping millisleep milliseconds. This option is mainly used by developers.

- **--db-stats**

  This option causes the database module to collect and print database statistics at the end of the execution. It has no effect if the **--no-database** option is given.

- **--socket**

  This option causes **pegasus-monitord** to start a socket interface that can be used for advanced debugging. The port number for connecting to **pegasus-monitord** can be found in the monitord.sock file in the workflow directory (the file is deleted when **pegasus-monitord** finishes). If not already started, the socket interface is also created when **pegasus-monitord** receives a USR1 signal.

- **-o <dir>**

  --output-dir <dir>

  When this option is given, .**pegasus-monitord** will create all its output files in the directory specified by **dir**. This option is useful for allowing a user to debug a workflow in a directory the user does not have write permissions. In this case, all files generated by **pegasus-monitord** will have the workflow **wf_uuid** as a prefix so that files from multiple sub-workflows can be placed in the same directory. This option is mainly used by **pegasus-analyzer**. It is important to note that the location for the output BP file or database is not changed by this option and should be set via the **--dest** option.

- **-d URL[<scheme>]<params>**

  --dest URL[<scheme>]<params>

  This option allows users to specify the destination for the log events generated by **pegasus-monitord**. If this option is omitted, *pegasus-monitord* will create a SQLite database in the workflow's run directory with the same name as the workflow, but with a .**stampede.db** prefix. For an empty **scheme**, **params** are a file path with "**-**" meaning standard output. For a x-tcp **scheme**, **params** are TCP_host[:port=14380]. For a database **scheme**, **params** are a SQLAlchemy engine URL with a database connection string that can be used to specify different database engines. Please see the examples section below for more information on how to use this option. Note that when using a database engine other than sqlite, the necessary Python database drivers will need to be installed.

- **-e <bp | bson>**

  --encoding <bp | bson>

This option specifies how to encode log events. The two available possibilities are **bp** and **bson**. If this option is not specified, events will be generated in the **bp** format.

- **DAGMan output file**

  The **DAGMan output file** is the only requires command-line argument in **pegasus-monitord** and must have the **.dag.dagman.out** extension.

RETURN VALUE

If the plan could be constructed, **pegasus-monitord** returns with an exit code of 0. However, in case of error, a non-zero exit code indicates problems. In that case, the logfile should contain additional information about the error condition.

EXAMPLES

Usually, **pegasus-monitord** is invoked automatically by **pegasus-run** and tracks the workflow progress in real-time, producing the jobstate.log file and a corresponding SQLite database. When a workflow fails, and is re-submitted with a rescue DAG, **pegasus-monitord** will automatically pick up from where it left previously and continue the jobstate.log file and the database.

If users need to create the jobstate.log file after a workflow is already finished, the **--replay | -r** option should be used when running **pegasus-monitord** manually. For example:

```
pegasus_monitord -r diamond-0.dag.dagman.out
```

will launch **pegasus-monitord** in replay mode. In this case, if a jobstate.log file already exists, it will be rotated and a new file will be created. If a diamond-0.stampede.db SQLite database already exists, **pegasus-monitord** will purge all references to the workflow id specified in the braindump.txt file, including all sub-workflows associated with that workflow id.

```
pegasus_monitord -r --no-database diamond-0.dag.dagman.out
```

will do the same thing, but without generating any log events.

```
pegasus_monitord -r --dest `pwd`/diamond-0.bp diamond-0.dag.dagman.out
```

will create the file diamond-0.bp in the current directory, containing NetLogger events with all the workflow data. This is in addition to the jobstate.log file.

For using a database, users should provide a database connection string in the format of:

```
dialect://username:password@host:port/database
```

Where **dialect** is the name of the underlying driver (**mysql**, **sqlite**, **oracle**, **postgres**) and **database** is the name of the database running on the server at the host computer.

If users want to use a different SQLite database, **pegasus-monitord** requires them to specify the absolute path of the alternate file. For example:

```
pegasus_monitord -r --dest sqlite:////home/user/diamond_database.db diamond-0.dag.dagman.out
```

Here are docs with details for all of the supported drivers: http://www.sqlalchemy.org/docs/05/reference/dialects/index.html

Additional per-database options that work into the connection strings are outlined there.

It is important to note that one will need to have the appropriate db interface library installed. Which is to say, SQLAlchemy is a wrapper around the mysql interface library (for instance), it does not provide a MySQL driver itself. The Pegasus distribution includes both SQLAlchemy and the SQLite Python driver.

As a final note, it is important to mention that unlike when using SQLite databases, using SQLAlchemy with other database servers, e.g. MySQL or Postgres, the target database needs to exist. So, if a user wanted to connect to:

```
mysql://pegasus-user:supersecret@localhost:localport/diamond
```

it would need to first connect to the server at **localhost** and issue the appropriate create database command before running **pegasus-monitord** as SQLAlchemy will take care of creating the tables and indexes if they do not already exist.

pegasus-analyzer

pegasus-analyzer is used to debug failed workflows.

SYNOPSIS

pegasus-analyzer [--help | -h] [--quiet | -q] [--strict | -s] [--monitord | -m | -t] [--debug level] [--output-dir | -o output_dir] [--file | -f dag_filename] [--dir | -d| -i input_dir] [--print | -p print_options] [--debug-job job] [--debug-dir debug_dir] [--type workflow_type]

DESCRIPTION

Pegasus-analyzer is a command-line utility for parsing the jobstate.log file and reporting succesful and failed jobs. It quickly goes through several log files, isolates jobs that did not complete sucessfully, and prints their stdout and stderr so that users can get detailed information about their workflow runs.

ARGUMENTS

Any option will be displayed with its long options synonym(s).

- **-h**

  --help

  Prints a usage summary with all the available command-line options.

- **-q**

  --quiet

  Only print the the output and error filenames instead of their contents.

- **-s**

  --strict

  Get jobs' output and error filenames from the job's submit file.

- **-m**

  -t

  --monitord

  Invoke **pegasus-monitord** before analyzing the jobstate.log file. Although **pegasus-analyzer** can be executed during the workflow execution as well as after the workflow has already completed execution, **pegasus-monitord** is always invoked with the **--replay** option. Since multiple instances of **pegasus-monitord** should not be executed simultanesouly in the same workflow directory, the user should ensure that no other instances of **pegasus-monitord** are running. If the run_directory is writable, **pegasus-analyzer** will create a jobstate.log file there, rotating an older log, if it is found. If the run_directory is not writable (e.g. when the user debugging the workflow is not the same user that ran the workflow), **pegasus-analyzer** will exit and ask the user to provide the **--output-dir** option, in order to provide an alternative location for **pegasus-monitord** log files.

- **--debug-level**

  Sets the log level for **pegasus-analyzer** . Possible values, in the more verbose to least verbose order, are:

  - INFO

- DEBUG
- WARNING
- ERROR
- CRITICAL

If omitted, the default level will be set to WARNING.

- **-o <output directory>**

  --output-dir <output directory>

  This option provides an alternative location for all monitoring log files for a particular workflow. It is mainly used when an user does not have write privileges to a workflow directory and needs to generate the log files needed by **pegasus-analyzer**. If this option is used in conjunction with the **--monitord option**, it will invoke **pegasus-monitord** using output_dir to store all output files. Because workflows can have sub-workflows, **pegasus-monitord** will create its files prepending the workflow wf_uuid to each filename. This way, multiple workflow files can be stored in the same directory. **pegasus-analyzer** has built-in logic to find the specific jobstate.log file by looking at the workflow braindump.txt file first and figuring out the corresponding wf_uuid. If output_dir does not exist, it will be created.

- **-f <dag filename>**

  --file <dag filename>

  In this option, **dag filename** specifies the path to the DAG file to use. **pegasus-analyzer** will get the directory information from the **dag filename**. This option overrides the **--dir** option below.

- **-d <input dir>**

  -i <input dir>

  --dir <input dir>

  Makes **pegasus-analyzer** look for the jobstate.log file in the **input dir** directory. If this option is omitted, **pegasus-analyzer** will look in the current directory.

- **-p <print options>**

  --print <print options>

  Tells **pegasus-analyzer** what extra information it should print for failed jobs. **print options** is a comma-delimited list of options, that include **pre**, **invocation**, and/or **all**, which activates all printing options. With the **pre** option, **pegasus-analyzer** will print the pre-script information for failed jobs. For the **invocation** option, **pegasus-analyzer** will print the invocation command, so users can manually run the failed job.

- **--debug-job <job>**

  When given this option, **pegasus-analyzer** turns on its debug_mode, which can be used to debug a particular job. In this mode, **pegasus-analyzer** will create a shell script in the **debug dir** (see below, for specifying it) and copy all necessary files to this local directory and then execute the job locally.

- **--debug-dir <debug dir>**

  When in debug mode, **pegasus-analyzer** will create a temporary debug directory. Users can give this option in order to specify a particular **debug dir** directory to be used instead.

- **--type <workflow type>**

  In this options, users specify what **workflow type** they want to debug. At this moment, the only **workflow type** available is **condor** and it is the default value if this option is not specified.

EXAMPLES

The simplest way to use **pegasus-analyzer** is to go to the run directory and invoke the analyzer:

```
pegasus-analyzer .
```

which will cause **pegasus-analyzer** to print information about the workflow in the current directory.

pegasus-analyzer output contains a summary, followed by detailed information about each job that either failed, or is in an unknown state. Here is the summary section of the output:

```
************************Summary*************************

 Total jobs         :     75 (100.00%)
 # jobs succeeded   :     41 (54.67%)
 # jobs failed      :      0 (0.00%)
 # jobs unsubmitted :     33 (44.00%)
 # jobs unknown     :      1 (1.33%)
```

jobs succeeded are jobs that have completed succesfully. **jobs failed** are jobs that have finished, but that did not complete successfully. **jobs unsubmitted** are jobs that are listed in the dag file, but no information about them was found in the jobstate.log file. Finally, **jobs unknown** are jobs that have started, but have not reached completion.

After the summary section, **pegasus-analyzer** will display information about each job in the **job failed** and **job unknown** categories.

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

=======================findrange_j3=========================

  last state: POST_SCRIPT_FAILURE
        site: local
 submit file: /home/user/diamond\-submit/findrange_j3.sub
 output file: /home/user/diamond\-submit/findrange_j3.out.000
  error file: /home/user/diamond\-submit/findrange_j3.err.000

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

 site        : local
 hostname    : server-machine.domain.com
 executable  : (null)
 arguments   : -a findrange -T 60 -i f.b2 -o f.c2
 error       : 2
 working dir :
```

In the example above, the **findrange_j3** job has failed, and the analyzer displays information about the job, showing that the job finished with a **POST_SCRIPT_FAILURE**, and lists the submit, output and error files for this job. Whenever **pegasus-analyzer** detects that the output file contains a kickstart record, it will display the breakdown containing each task in the job (in this case we only have one task). Because **pegasus-analyzer** was not invoked with the **--quiet** flag, it will also display the contents of the output and error files (or the stdout and stderr sections of the kickstart record), which in this case are both empty.

In the case of SUBDAG and subdax jobs, **pegasus-analyzer** will indicate it, and show the command needed for the user to debug that sub-workflow. For example:

```
=================subdax_black_ID000009====================

  last state: JOB_FAILURE
        site: local
 submit file: /home/user/run1/subdax_black_ID000009.sub
 output file: /home/user/run1/subdax_black_ID000009.out
  error file: /home/user/run1/subdax_black_ID000009.err
  This job contains sub workflows!
  Please run the command below for more information:
  pegasus-analyzer -d /home/user/run1/blackdiamond_ID000009.000

-----------------subdax_black_ID000009.out-----------------

Executing condor dagman ...

-----------------subdax_black_ID000009.err-----------------
```

tells the user the subdax_black_ID000009 sub-workflow failed, and that it can be debugged by using the indicated **pegasus-analyzer** command.

pegasus-statistics

pegasus-statistics reports statistics about a workflow.

SYNOPSIS

pegasus-statistics <submit directory> [-h | --help] [-o | --output <output directory>] [-c | --conf <property file path>] [-p | --statistics-level level_str] [-t | --time-filter filter_str] [-i | --ignore-db-inconsistency] [-v | --verbose] [-q | --quiet]

DESCRIPTION

pegasus-statistics generates statistics about the workflow run like total jobs/tasks/sub workflows ran , how many succeeded/failed etc. It generates job instance statistics like run time, condor queue delay etc. It generates invocation statistics information grouped by transformation name. It also generates job instance and invocation statistics information grouped by time and host.

ARGUMENTS

Any option will be displayed with its long options synonym(s).

- **-h**

  --help

  Prints a usage summary with all the available command-line options.

- **-o <output directory >**

  --output <output directory >

  Writes the output to the given directory.

- **-c <property file path >**

  --conf <property file path >

  The properties file to use. This option overrides all other property files.

- **-s level_str**

  --statistics-level level_str

  Specifies the statistics information to generate. Valid levels are: all,summary,wf_stats,jb_stats,tf_stats,ti_stats; Default is summary. The output generated by pegasus-statistics is based on the statistics_level set.

  - **all**

    generates all the statistics information.

  - **summary**

    generates the workflow statistics summary .In case of hierarchical workflow the summary is across all sub workflows.

  - **wf_stats**

    generates the workflow statistics information of each individual workflow. In case of hierarchical workflow the workflow statistics is created for each sub workflows.

  - **jb_stats**

generates the job statistics information of each individual workflow. In case of hierarchical workflow the job statistics is created for each sub workflows.

- **tf_stats**

  generates the invocation statistics information of each individual workflow grouped by transformation name .In case of hierarchical workflow the transformation statistics is created for each sub workflows.

- **ti_stats**

  generates the job instance and invocation statistics like total count and runtime grouped by time and host.

- **-t filter_str**

  --time-filter filter_str

  Specifies the time filter to group the time statistics. Valid levels are: month,week,day,hour; Default is day.

- **-i**

  --ignore-db-inconsistency

  Turns off the the check for database consistency.

- **-v**

  --verbose

  Increases the log level. If omitted, the default level will be set to WARNING. When this option is given, the log level is changed to INFO. If this option is repeated, the log level will be changed to DEBUG.

- **-q**

  --quiet

  Decreases the log level. If omitted, the default level will be set to WARNING. When this option is given, the log level is changed to ERROR.

EXAMPLE

```
pegasus-statistics   /scratch/grid-setup/run0001
```

Runs pegasus-statistics and prints the workflow summary statistics on the command line output.

```
pegasus-statistics  -o /scratch/statistics -s all /scratch/grid-setup/run0001
```

Runs pegasus-statistics and prints the workflow summary statistics on the command line output and generates all statistics information files to the given directory.

pegasus-plots

pegasus-plots generates charts and graphs that illustrate the statistics and execution of a workflow.

SYNOPSIS

pegasus-statistics <submit directory> [-h | --help] [-o | --output <output directory>] [-c | --conf <property file path>] [-m | --max-graph-nodes max_value] [-p | --plotting-level level_str] [-i | --ignore-db-inconsistency] [-v | --verbose] [-q | --quiet]

DESCRIPTION

pegasus-plots generates graphs and charts to visualize workflow run. It generates workflow execution Gantt chart, job over time chart, time chart, breakdown chart, dax and dag graph. It uses executable 'dot' to generate graphs. pegasus-plots looks for the executable in your path and generates graphs based on it's availability .

ARGUMENTS

Any option will be displayed with its long options synonym(s).

- **-h**

  --help

  Prints a usage summary with all the available command-line options.

- **-o <output directory >**

  --output <output directory >

  Writes the output to the given directory.

- **-c <property file path >**

  --conf <property file path >

  The properties file to use. This option overrides all other property files.

- **-m max_value**

  --max-graph-nodes max_value

  Maximum limit on the number of tasks/jobs in the dax/dag upto which the graph should be generated;The default value is 100

- **-p level_str**

  --plotting-level level_str

  Specifies the charts and graphs to generate. Valid levels are: all,all_charts, all_graphs,dax_graph,dag_graph,gantt_chart,host_chart, time_chart,breakdown_chart;Default is all_charts. The output generated by pegasus-plots is based on the plotting_level set.

  - **all**

    generates all charts and graphs.

  - **all_charts**

    generates all charts.

  - **all_graphs**

    generates all graphs.

  - **dax_graph**

    generates dax graph.

  - **dag_graph**

    generates dag graph.

  - **gantt_chart**

    generates the workflow execution Gantt chart.

  - **host_chart**

    generates the host over time chart.

- **time_chart**

  generates the time chart which shows the job instance/ invocation count and runtime over time.

- **breakdown_chart**

  generates the breakdown chart which shows the invocation count and runtime grouped by transformation name.

- **-i**

  --ignore-db-inconsistency

  Turns off the the check for database consistency.

- **-v**

  --verbose

  Increases the log level. If omitted, the default level will be set to WARNING. When this option is given, the log level is changed to INFO. If this option is repeated, the log level will be changed to DEBUG.

- **-q**

  --quiet

  Decreases the log level. If omitted, the default level will be set to WARNING. When this option is given, the log level is changed to ERROR.

EXAMPLE

```
pegasus-plots  -o /scratch/plots  /scratch/grid-setup/run0001
```

Runs pegasus-plots and generates all the charts and graphs to the given directory.

pegasus-transfer

pegasus-transfer is a wrapper for several file transfer clients. The tool is used whenever Pegasus plans a transfer of data.

SYNOPSIS

pegasus-transfer [-h | --help] [-l | --loglevel <level>] [-f | --file=<input file>] [--max-attempts=<attempts>]

DESCRIPTION

pegasus-transfer takes a list of url pairs, either on stdin or with an input file, determines the correct tool to use for the transfer and executes the transfer. Some of the protocols pegasus-transfer can handle are GridFTP, SRM, Amazon S3, HTTP, and local cp/symlinking. Failed transfers are retried.

ARGUMENTS

Any option will be displayed with its long options synonym(s).

- **-h**

  --help

  Prints a usage summary with all the available command-line options.

- **-l <level>**

  --loglevel <level>

The debugging output level. Valid values are debug,info.warning,error. Default value is info.

- **-f <input file>**

  --file <input file>

  File with input pairs. If not given, stdin will be used.

- **--max-attempts <attempts>**

  Maximum number of attempts for retrying failed transfers.

EXAMPLE

```
pegasus-transfer
file:///etc/hosts
file:///tmp/foo
CTRL+D
```

pegasus-sc-client

pegasus-sc-client is used to generate and modify site catalogs.

pegasus-rc-client

pegasus-rc-client - shell client for replica implementations

SYNOPSIS

- **pegasus-rc-client** [-Dprop [...]] [-c fn] [-p k=v] [[-f fn]|[-i|-d fn]|[cmd [args]]
- **pegasus-rc-client** [-c fn] -V

DESCRIPTION

The shell interface to replica catalog implementations is a prototype. It determines from various property setting which class implements the replica manager interface, and loads that driver at run-time. Some commands depend on the implementation.

ARGUMENTS

Any option will be displayed with its long options synonym(s).

- **-h, --help**

  print this help text

- **-V, --version**

  print some version identification string and exit

- **-Dprop**

  The -D options allows an experienced user to override certain properties which influence the program execution, among them the default location of the user's properties file and the PEGASUS home location. One may set several CLI properties by giving this option multiple times. The -D option(s) must be the first option on the command line. A CLI property take precedence over the properties file property of the same key.

- **-f, --file fn**

  uses non-interactive mode, reading from file fn. The special filename hyphen reads from pipes

- **-c, --conf fn**

  path to the property file

- **-v, --verbose**

  increases the verbosity level

- **-p, --pref k=v**

  enters the specified mapping into preferences (multi-use). remember quoting, e.g. -p 'format=%l %p %a'

- **-i, --insert fn**

  the path to the file containing the mappings to be inserted. Each line in the file denotes one mapping of format <LFN> <PFN> [k=v [..]]

- **-d, --delete fn**

  the path to the file containing the mappings to be deleted. Each line in the file denotes one mapping of format <LFN> <PFN> [k=v [..]].

- **-l, --lookup fn**

  the path to the file containing the LFN's to be looked up. Each line in the file denotes one LFN

- **cmd [args]**

  If not in file-driven mode, a single command can be specified with its arguments.

RETURN VALUE

Regular and planned program terminations will result in an exit code of 0. Abnormal termination will result in a non-zero exit code.

FILES

- **$PEGASUS_HOME/etc/properties**

  contains the basic properties with all configurable options.

- **$HOME/.pegasusrc**

  contains the basic properties with all configurable options.

- **pegasus.jar**

  contains all compiled Java bytecode to run the replica manager.

ENVIRONMENT VARIABLES

- **$PEGASUS_HOME**

  is the suggested base directory of your the execution environment.

- **$JAVA_HOME**

  should be set and point to a valid location to start the intended Java virtual machine as $JAVA_HOME/bin/java.

- **$CLASSPATH**

  should be set to contain all necessary files for the execution environment. Please make sure that your CLASSPATH includes pointer to the replica implementation required jar files.

PROPERTIES

The complete branch of properties pegasus.catalog.replica including itself are interpreted by the prototype. While the pegasus.catalog.replica property itself steers the backend to connect to, any meaning of branched keys is dependent on the backend. The same key may have different meanings for different backends.

- **pegasus.catalog.replica**

  determines the name of the implementing class to load at run-time. If the class resides in org.griphyn.common.catalog.replica no prefix is required. Otherwise, the fully qualified class name must be specified.

- **pegasus.catalog.replica.url**

  is used by the RLS|LRC implementations. It determines the RLI / LRC url to use.

- **pegasus.catalog.replica.file**

  is used by the SimpleFile implementation. It specifies the path to the file to use as the backend for the catalog.

- **pegasus.catalog.replica.db.driver**

  is used by a simple rDBMs implementation. The string is the fully-qualified class name of the JDBC driver used by the rDBMS implementer.

- **pegasus.catalog.replica.db.url**

  is the jdbc url to use to connect to the database.

- **pegasus.catalog.replica.db.user**

  is used by a simple rDBMS implementation. It constitutes the database user account that contains the RC_LFN and RC_ATTR tables.

- **pegasus.catalog.replica.db.password**

  is used by a simple rDBMS implementation. It constitutes the database user account that contains the RC_LFN and RC_ATTR tables.

- **pegasus.catalog.replica.chunk.size**

  is used by the pegasus-rc-client for the bulk insert and delete operations. The value determines the number of lines that are read in at a time, and worked upon at together.

COMMANDS

The commandline tool provides a simplified shell-wrappable interface to manage a replica catalog backend. The commands can either be specified in a file in bulk mode, in a pipe, or as additional arguments to the invocation.

Note that you must escape special characters from the shell.

- **help**

  displays a small resume of the commands.

- **exit, quit**

  should only be used in interactive mode to exit the interactive mode.

- **clear**

  drops all contents from the backend. Use with special care!

- **insert <lfn> <pfn> [k=v [..]]**

inserts a given lfn and pfn, and an optional site string into the backend. If the site is not specified, a null value is inserted for the site.

- **delete <lfn> <pfn> [k=v [..]]**

  removes a triple of lfn, pfn and, optionally, site from the replica backend. If the site was not specified, all matches of the lfn pfn pairs will be removed, regardless of the site.

- **lookup <lfn> [<lfn> [..]]**

  retrieves one or more mappings for a given lfn from the replica backend.

- **remove <lfn> [<lfn> [..]]**

  removes all mappings for each lfn from the replica backend.

- **list [lfn <pat>] [pfn <pat>] [<name> <pat>]**

  obtains all matches from the replica backend. If no arguments were specified, all contents of the replica backend are matched. You must use the word lfn, pfn or <name> before specifying a pattern. The pattern is meaningful only to the implementation. Thus, a SQL implementation may chose to permit SQL wild-card characters. A memory-resident service may chose to interpret the pattern as regular expression.

- **set [var [value]]**

  sets an internal variable that controls the behavior of the front-end. With no arguments, all possible behaviors are displayed. With one argument, just the matching behavior is listed. With two arguments, the matching behavior is set to the value.

DATABASE SCHEMA

The tables are set up as part of the PEGASUS database setup. The files concerned with the database have a suffix -rc.sql.

pegasus-tc-client

A full featured generic client to handle adds, delete and queries to the Transformation Catalog (TC).

SYNOPSIS

pegasus-tc-client [-Dprop [...]] OPERATION TRIGGERS [OPTIONS] [-h] [-v] [-V]

DESCRIPTION

The tc-client command is a generic client that performs the three basic operation of adding, deleting and querying of any Transformation Catalog impemented to the TC API. The client implements all the operations supported by the TC Api. It is upto the TC implementation whether they support all operations or modes.

The following 3 operations are supported by the tc-client. One of these operations have to be specified to run the client.

1. **ADD**

   This operation allows the client to add or update entries in the Transformation Catalog. Entries can be added one by one on the command line or in bulk by using the BULK Trigger and pro# viding a file with the necessary entries. Also Profiles can be added to either the logical transformation or the physical transformation.

2. **DELETE**

   This operation allows the client to delete entries from the Transformation Catalog. Entries can be deleted based on logical transformation, by resource, by transformation type as well as the transformation system information. Also Profiles associated with the logical or physical transformation can be deleted.

3. **QUERY**

   This opeartion allows the client to query for entries from the Transformation Catalog. Queries can be made for printing all the contents of the Catalog or for specific entries, for all the logical transformations or resources etc. See the TRIGGERS and VALID COMBINATIONS section for more details.

OPERATIONS

To select one of the 3 operations.

1. **-a , --add**

   Perform addition operations on the TC.

2. **-d , --delete**

   Perform delete operations on the TC.

3. *-q , --query*

   Perform query operations on the TC.

TRIGGERS

Triggers modify an OPERATIONS behaviour. E.g. if you want to perform a bulk operation you would use a BULK Trigger or if you want to perform an operation on a Logical Transformation then you would use the LFN Trigger.

The following 7 Triggers are available. See the VALID COMBINATIONS section for the correct grouping and usage.

- **-B**

  Triggers a bulk operation.

- **-L**

  Triggers an operation on a logical transformation.

- **-P**

  Triggers an operation on a physical transformation.

- **-R**

  Triggers an operation on a resource.

- **-E**

  Triggers an operation on a Profile.

- **-T**

  Triggers an operation on a Type.

- **-S**

  Triggers an operation on a System information.

OPTIONS

The following options are applicable for all the operations.

- **-l, --lfn logicalTR**

  The logical transformation to be added. The format is NAMESPACE::NAME:VERSION. The name is always required, namespace and version are optional.

- **-p, --pfn physical TR**

  The physical transfromation to be added. For INSTALLED executables its a local file path, for all others its a url.

- **-t , --type type**

  The type of physical transformation. Valid values are : INSTALLED, STATIC_BINARY, DYNAMIC_BINARY, SCRIPT, SOURCE, PACMAN_PACKAGE.

- **-r , --resource resourceID**

  The resourceID where the transformation is located.

- **-e , --profile profiles**

  The profiles for the transformation. Mulitple profiles of same namespace can be added simultaneously by seperating them with a comma ",". Each profile section is written as NAMESPACE::KEY=VALUE,KEY2=VALUE2 e.g. ENV::JAVA_HOME=/usr/bin/java2,PEGASUS_HOME=/usr/local/pegasus. To add muliple namespaces you need to repeat the -e option for each namespace.

  e.g. -e ENV::JAVA_HOME=/usr/bin/java -e GLOBUS::JobType=MPI,COUNT=10

- **-s , --system systeminfo**

  The architecture, os, osversion and glibc if any for the executable. Each system info is written in the form ARCH::OS:OSVER:GLIBC

OTHER OPTIONS

- **-Dprop**

  The -D options allows an experienced user to override certain properties which influence the program execution, among them the default location of the user's properties file and the PEGASUS home location. One may set several CLI properties by giving this option multiple times. The -D option(s) must be the first option on the command line. A CLI property take precedence over the properties file property of the same key.

- **--oldformat, -o**

  Generates the output in the old single line format

- **--conf, -c**

  path to property file

- **--verbose, -v**

  increases the verbosity level

- **--version, -V**

  Displays the version number of the Griphyn Virtual Data System software

- **--help, -h**

  Generates this help

VALID COMBINATIONS

The following are valid combinations of OPERATIONS, TRIGGERS, OPTIONS for the tc-client

- **ADD**
  - *ADD TC ENTRY*

    -a -l lfn -p pfn -t type -r resource -s system [-e profiles..]

    Adds a single entry into the transformation catalog.

  - *ADD PFN PROFILE*

    -a -P -E -p pfn -t type -r resource -e profiles ....

    Adds profiles to a specified physical transformation on a given resource and of a given type.

  - *ADD LFN PROFILE*

    -a -L -E -l lfn -e profiles ....

    Adds profiles to a specified logical transformation.

  - *Add Bulk Entries*

    -a -B -f file

    Adds entries in bulk mode by supplying a file containg the entries.

    The format of the file cotnains 6 columns. E.g.

    #RESOURCE LFN PFN TYPE SYSINFO PROFILES # isi NS::NAME:VER /bin/date INSTALLED ARCH::OS:OSVERS:GLIBC NS::KEY=VALUE,KEY=VALUE;NS2::KEY=VALUE,KEY=VALUE

- **DELETE**
  - *Delete all TC*

    -d -BPRELST

    Deletes the entire contents of the TC. WARNING : USE WITH CAUTION.

  - *Delete by LFN*

    -d -L -l lfn [-r resource] [-t type]

    Deletes entries from the TC for a particular logical transformation and additionaly a resource and or type.

  - *Delete by PFN*

    -d -P -l lfn -p pfn [-r resource] [-t type]

    Deletes entries from the TC for a given logical and physical transformation and additionaly on a particular resource and or of a particular type.

  - *Delete by Type*

    -d -T -t type [-r resource]

    Deletes entries from TC of a specific type and/or on a specific resource.

  - *Delete by Resource*

    -d -R -r resource

    Deletes the entries from the TC on a particular resource.

  - *Delete by SysInfo*

-d -S -s sysinfo

Deletes the entries from the TC for a particular system information type.

- *Delete Pfn Profile*

  -d -P -E -p pfn -r resource -t type [-e profiles ..]

  Deletes all or specific profiles associated with a physical transformation.

- *Delete Lfn Profile*

  -d -L -E -l lfn -e profiles ....

  Deletes all or specific profiles associated with a logical transformation.

- **QUERY**
  - *Query Bulk*

    -q -B Queries for all the contents of the TC.

    It produces a file format TC which can be added to another TC using the bulk option.

  - *Query LFN*

    -q -L [-r resource] [-t type]

    Queries the TC for logical transformation and/or on a particular resource and/or of a particular type.

  - *Query PFN*

    -q -P -l lfn [-r resource] [-t type]

    Queries the TC for physical transformations for a give logical transformation and/or on a particular resource and/or of a particular type.

  - *Query Resource*

    -q -R -l lfn [-t type]

    Queries the TC for resources that are registered and/or resources registered for a specific type of transformation.

  - *Query Lfn Profile*

    -q -L -E -l lfn

    Queries for profiles associated with a particular logical transformation

  - *Query Pfn Profile*

    -q -P -E -p pfn -r resource -t type

    Queries for profiles associated with a particular physical transformation

PROPERTIES

This are the properties you will need to set to use either the File or Text TC. For more details please check the $PEGASUS_HOME/etc/sample.properties file.

- **pegasus.catalog.transformation**

  Identifies what impelemntation of TC will be used. If relative name is used then the path org.griphyn.cPlanner.tc is prefixed to the name and used as the class name to load. The default value is Text. Other supported mode is File

- **pegasus.catalog.transformation.file**

  The file path where the text based TC is located. By default the path $PEGASUS_HOME/var/tc.data is used.

FILES

- **$PEGASUS_HOME/var/tc.data**

  is the suggested location for the file corresponding to the Transformation Catalog

- **$PEGASUS_HOME/etc/properties**

  is the location to specify properties to change what Tranformation Catalog Implementation to use and the implementation related PROPERTIES.

- **pegasus.jar**

  contains all compiled Java bytecode to run the PEGASUS Planner.

ENVIRONMENT VARIABLES

- **$PEGASUS_HOME**

  Path to the PEGASUS installation directory.

- **$JAVA_HOME**

  Path to the JAVA 1.4.x installation directory.

- **$CLASSPATH**

  The classpath should be set to contain all necessary PEGASUS files for the execution environment. To automatically add the CLASSPATH to you environment, in the $PEGASUS_HOME directory run the script source setup-user-env.csh or source setup-user-env.sh.

pegasus-s3

pegasus-s3 is a client for the Amazon S3 object storage service and any other storage services that conform to the Amazon S3 API, such as Eucalyptus Walrus.

URL Format

All URLs for objects stored in S3 should be specified in the following format:

```
s3[s]://USER@SITE[/BUCKET[/KEY]]
```

The protocol part can be s3:// or s3s://. If s3s:// is used, then pegasus-s3 will force the connection to use SSL and override the setting in the configuration file. If s3:// is used, then whether the connection uses SSL or not is determined by the value of the 'endpoint' variable in the configuration for the site.

The *USER@SITE* part is required, but the *BUCKET* and *KEY* parts may be optional depending on the context.

The *USER@SITE* portion is referred to as the 'identity', and the *SITE* portion is referred to as the site. Both the identity and the site are looked up in the configuration file (see pegasus-s3 Configuration) to determine the parameters to use when establishing a connection to the service. The site portion is used to find the host and port, whether to use SSL, and other things. The identity portion is used to determine which authentication tokens to use. This format is designed to enable users to easily use multiple services with multiple authentication tokens. Note that neither the *USER* nor the *SITE* portion of the URL have any meaning outside of pegasus-s3. They do not refer to real usernames or hostnames, but are rather handles used to look up configuration values in the configuration file.

The *BUCKET* portion of the URL is the part between the 3rd and 4th slashes. Buckets are part of a global namespace that is shared with other users of the storage service. As such, they should be unique.

The *KEY* portion of the URL is anything after the 4th slash. Keys can include slashes, but S3-like storage services do not have the concept of a directory like regular file systems. Instead, keys are treated like opaque identifiers for individual objects. So, for example, the keys 'a/b' and 'a/c' have a common prefix, but cannot be said to be in the same 'directory'.

Some example URLs are:

```
s3://ewa@amazon
s3://juve@skynet/gideon.isi.edu
s3://juve@magellan/pegasus-images/centos-5.5-x86_64-20101101.part.1
s3s://ewa@amazon/pegasus-images/data.tar.gz
```

Subcommands

pegasus-s3 has several subcommands for different storage service operations.

help

pegasus-s3 help

The **help** subcommand lists all available subcommands.

ls

pegasus-s3 ls [options] URL...

The **ls** subcommand lists the contents of a URL. If the URL does not contain a bucket, then all the buckets owned by the user are listed. If the URL contains a bucket, but no key, then all the keys in the bucket are listed. If the URL contains a bucket and a key, then all keys in the bucket that begin with the specified key are listed.

mkdir

pegasus-s3 mkdir [options] URL...

The **mkdir** subcommand creates one or more buckets.

rmdir

pegasus-s3 rmdir [options] URL...

The **rmdir** subcommand deletes one or more buckets from the storage service. In order to delete a bucket, the bucket must be empty.

rm

pegasus-s3 rm [options] URL...

The **rm** subcommand deletes one or more keys from the storage service.

put

pegasus-s3 put [options] FILE URL

The **put** subcommand stores the file specified by *FILE* in the storage service under the bucket and key specified by *URL*. If the URL contains a bucket, but not a key, then the file name is used as the key.

If a transient failure occurs, then the upload will be retried several times before pegasus-s3 gives up and fails.

The put subcommand can do both chunked and parallel uploads if the service supports multipart uploads (see multipart_uploads in the configuration). Currently only Amazon S3 supports multipart uploads.

This subcommand will check the size of the file to make sure it can be stored before attempting to store it.

Chunked uploads are useful to reduce the probability of an upload failing. If an upload is chunked, then pegasus-s3 issues separate PUT requests for each chunk of the file. Specifying smaller chunks (using --chunksize) will reduce the chances of an upload failing due to a transient error. Chunksizes can range from 5 MB to 1GB (chunk sizes smaller

than 5 MB produced incomplete uploads on Amazon S3). The maximum number of chunks for any single file is 10,000, so if a large file is being uploaded with a small chunksize, then the chunksize will be increased to fit within the 10,000 chunk limit. By default, the file will be split into 10 MB chunks if the storage service supports multipart uploads. Chunked uploads can be disabled by specifying a chunksize of 0. If the upload is chunked, then each chunk is retried independently under transient failures. If any chunk fails permanently, then the upload is aborted.

Parallel uploads can increase performance for services that support multipart uploads. In a parallel upload the file is split into N chunks and each chunk is uploaded concurrently by one of M threads in first-come, first-served fashion. If the chunksize is set to 0, then parallel uploads are disabled. If M > N, then the actual number of threads used will be reduced to N. The number of threads can be specified using the --parallel argument. If --parallel is 0 or 1, then only a single thread is used. The default value is 0. There is no maximum number of threads, but it is likely that the link will be saturated by ~4 threads. Very high-bandwidth, long-delay links may get better results with up to ~8 threads.

Note

Under certain circumstances, when a multipart upload fails it could leave behind data on the server. When a failure occurs the put subcommand will attempt to abort the upload. If the upload cannot be aborted, then a partial upload may remain on the server. To check for partial uploads run the **lsup** subcommand. If you see an upload that failed in the output of lsup, then run the **rmup** subcommand to remove it.

get

pegasus-s3 get [options] URL [FILE]

The **get** subcommand retrives an object from the storage service identified by *URL* and stores it in the file specified by *FILE*. If FILE is not specified, then the key is used as the file name (Note: if the key has slashes, then the file name will be a relative subdirectory, but pegasus-s3 will not create the subdirectory if it does not exist).

If a transient failure occurs, then the download will be retried several times before pegasus-s3 gives up and fails.

The get subcommand can do both chunked and parallel downloads if the service supports ranged downloads (see ranged_downloads in the configuration). Currently only Amazon S3 has good support for ranged downloads. Eucalyptus Walrus supports ranged downloads, but the current release, 1.6, is inconsistent with the Amazon interface and has a bug that causes ranged downloads to hang in some cases. It is recommended that ranged downloads not be used with Eucalyptus until these issues are resolved.

Chunked downloads can be used to reduce the probability of a download failing. When a download is chunked, pegasus-s3 issues separate GET requests for each chunk of the file. Specifying smaller chunks (uisng --chunksize) will reduce the chances that a download will fail to do a transient error. Chunk sizes can range from 1 MB to 1 GB. By default, a download will be split into 10 MB chunks if the site supports ranged downloads. Chunked downloads can be disabled by specifying a chunksize of 0. If a download is chunked, then each chunk is retried independently under transient failures. If any chunk fails permanently, then the download is aborted.

Parallel downloads can increase performance for services that support ranged downloads. In a parallel download, the file to be retrieved is split into N chunks and each chunk is downloaded concurrently by one of M threads in a first-come, first-served fashion. If the chunksize is 0, then parallel downloads are disabled. If M > N, then the actual number of threads used will be reduced to N. The number of threads can be specified using the --parallel argument. If --parallel is 0 or 1, then only a single thread is used. The default value is 0. There is no maximum number of threads, but it is likely that the link will be saturated by ~4 threads. Very high-bandwidth, long-delay links may get better results with up to ~8 threads.

lsup

pegasus-s3 lsup [options] URL

The **lsup** subcommand lists active uploads. The URL specified should point to a bucket. This command is only valid if the site supports multipart uploads. The output of this command is a list of keys and upload IDs.

This subcommand is used with **rmup** to help recover from failures of multipart uploads.

rmup

pegasus-s3 rmup [options] URL UPLOAD

The **rmup** subcommand cancels and active upload. The *URL* specified should point to a bucket, and *UPLOAD* is the long, complicated upload ID shown by the **lsup** subcommand.

This subcommand is used with **lsup** to recover from failures of multipart uploads.

pegasus-s3 Configuration

Each user should specify a configuration file that pegasus-s3 will use to look up connection parameters and authentication tokens.

Configuration file search path

This client will look in the following locations, in order, to locate the user's configuration file:

1. The -C/--conf argument
2. The S3CFG environment variable
3. ~/.s3cfg

If it does not find the configuration file in one of these locations it will fail with an error.

Configuration file format

The configuration file is in INI format and contains two types of entries.

The first type of entry is a **site entry**, which specifies the configuration for a storage service. This entry specifies the service endpoint that pegasus-s3 should connect to for the site, and some optional features that the site may support. Here is an example of a site entry for Amazon S3:

```
[amazon]
endpoint = http://s3.amazonaws.com/
```

The other type of entry is an **identity entry**, which specifies the authentication information for a user at a particular site. Here is an example of an identity entry:

```
[pegasus@amazon]
access_key = 90c4143642cb097c88fe2ec66ce4ad4e
secret_key = a0e3840e5baee6abb08be68e81674dca
```

It is important to note that user names and site names used are only logical--they do not correspond to actual hostnames or usernames, but are simply used as a convenient way to refer to the services and identities used by the client.

The configuration file should be saved with limited permissions. Only the owner of the file should be able to read from it and write to it (i.e. it should have permissions of 0600 or 0400). If the file has more liberal permissions, then pegasus-s3 will fail with an error message. The purpose of this is to prevent the authentication tokens stored in the configuration file from being accessed by other users.

Configuration variables

Table 9.19.

Variable	Scope	Description
endpoint	site	The URL of the web service endpoint. If the URL begins with 'https', then SSL will be used.
max_object_size	site	The maximum size of an object in GB (default: 5GB)
multipart_uploads	site	Does the service support multipart uploads (True/False, default: False)
ranged_downloads	site	Does the service support ranged downloads? (True/False, default: False)

Variable	Scope	Description
access_key	identity	The access key for the identity
secret_key	identity	The secret key for the identity

Example configuration

This is an example configuration that specifies a single site (amazon) and a single identity (pegasus@amazon). For this site the maximum object size is 5TB, and the site supports both multipart uploads and ranged downloads, so both uploads and downloads can be done in parallel.

```
[amazon]
endpoint = https://s3.amazonaws.com/
max_object_size = 5120
multipart_uploads = True
ranged_downloads = True

[pegasus@amazon]
access_key = 90c4143642cb097c88fe2ec66ce4ad4e
secret_key = a0e3840e5baee6abb08be68e81674dca

[magellan]
# NERSC Magellan is a Eucalyptus site. It doesn't support multipart uploads,
# or ranged downloads (the defaults), and the maximum object size is 5GB
# (also the default)
endpoint = https://128.55.69.235:8773/services/Walrus

[juve@magellan]
access_key = quwefahsdpfwlkewqjsdoijldsdf
secret_key = asdfa9wejalsdjfljasldjfasdfa

[voeckler@magellan]
# Each site can have multiple associated identities
access_key = asdkfaweasdfbaeiwhkjfbaqwhei
secret_key = asdhfuinakwjelfuhalsdflahsdl
```

pegasus-exitcode

pegasus-exitcode is a utility that examines the STDOUT of a job to determine if the job failed, and renames the STDOUT and STDERR files of a job to preserve them in case the job is retried.

Pegasus uses pegasus-exitcode as the DAGMan postscript for all jobs submitted via Globus GRAM. This tool exists as a workaround to a known problem with Globus where the exitcodes of GRAM jobs are not returned. This is a problem because Pegasus uses the exitcode of a job to determine if the job failed or not.

In order to get around the exitcode problem, Pegasus wraps all GRAM jobs with Kickstart, which records the exitcode of the job in an XML invocation record, which it writes to the job's STDOUT. The STDOUT is transferred from the execution host back to the submit host when the job terminates. After the job terminates, DAGMan runs the job's postscript, which Pegasus sets to be pegasus-exitcode. pegasus-exitcode looks at the invocation record generated by kickstart to see if the job succeeded or failed. If the invocation record indicates a failure, then pegasus-exitcode returns a non-zero result, which indicates to DAGMan that the job has failed. If the invocation record indicates that the job succeeded, then pegasus-exitcode returns 0, which tells DAGMan that the job succeeeded.

pegasus-exitcode performs several checks to determine whether a job failed or not. These checks include:

1. Is STDOUT empty? If it is empty, then the job failed.
2. Are there any <status> tags with a non-zero value? If there are, then the job failed. Note that, if this is a clustered job, there could be multiple <status> tags, one for each task. If any of them are non-zero, then the job failed.
3. Is there at least one <status> tag with a zero value? There must be at least one successful invocation or the job has failed.

In addition, pegasus-exitcode allows the caller to specify the exitcode returned by Condor using the --return argument. This can be passed to pegasus-exitcode in a DAGMan post script by using the $RETURN variable. If this value is non-zero, then pegasus-exitcode returns a non-zero result before performing any other checks. For GRAM jobs, the value of $RETURN will always be 0 regardless of whether the job failed or not.

Also, pegasus-exitcode allows the caller to specify the number of successful tasks it should see using the --tasks argument. If pegasus-exitcode does not see N successful tasks, where N is set by --tasks, then it will return a non-zero result. The default value is 1. This can be used to detect failures in clustered jobs where, for any number of reasons, invocation records do not get generated for all the tasks in the clustered job.

In addition to checking the success/failure of a job, pegasus-exitcode also renames the STDOUT and STDERR files of the job so that if the job is retried, the STDOUT and STDERR of the previous run are not lost. It does this by appending a sequence number to the end of the files. For example, if the STDOUT file is called "job.out", then the first time the job is run pegasus-exitcode will rename the file "job.out.000". If the job is run again, then pegasus-exitcode sees that "job.out.000" already exists and renames the file "job.out.001". It will continue to rename the file by incrementing the sequence number every time the job is executed.

Kickstart

Kickstart is a job wrapper that collects data about a job's execution environment, performance, and output.

SYNTAX

kickstart [-n tr] [-N dv] [-H] [-R site] [-W | -w dir] [-L lbl -T iso] [-s [l=]p | @fn] [-S [l=]p | @fn] [-i fn] [-o fn] [-e fn] [-X] [-l fn sz] (-I fn | app [appflags])

kickstart -V

The **kickstart** executable is a light-weight program which connects the *stdin*, *stdout* and *stderr* filehandles for grid jobs on the remote site.

Sitting in between the remote scheduler and the executable, it is possible for **kickstart** to gather additional information about the executable run-time behavior and resource usage, including the exit status of jobs. This information is important for the Pegasus invocation tracking as well as to Condor DAGMan's awareness of Globus job failures.

Kickstart allows the optional execution of jobs before and after the main application job that run in chained execution with the main application job. See section SUBJOBS for details about this feature.

All jobs with relative path specifications to the application are part of search relative to the current working directory (yes, this is unsafe), and by prepending each component from the PATH environment variable. The first match is used. Jobs that use absolute pathnames, starting in a slash, are exempt. Using an absolute path to your executable is the safe and recommended option.

Kickstart rewrites the commandline of any job (pre, post and main) with variable substitutions from Unix environment variables. See section VARIABLE REWRITING below for details on this feature.

Kickstart provides a temporary named pipe (fifo) for applications that are gridstart aware. Any data an application writes to the FIFO will be propagated back to the submit host, thus enabling progress meters and other application dependent monitoring. See section FEEDBACK CHANNEL below for details on this feature.

ARGUMENTS

`-n tr` In order to associate the minimal performance information of the job with the invocation records, the jobs needs to carry which *transformation* was responsible for producing it. The format is the notation for fully-qualified definition names, like namespace::name:version, with only the name portion being mandatory.

There is no default. If no value is given, "null" will be reported.

`-N dv` the job may carry which instantiation of a transformation was responsible for producing it. The format is the notation for fully-qualified definition names, like namespace::name:version, with only the name portion being mandatory.

There is no default. If no value is given, "null" will be reported.

`-H` This option avoids kickstart writing the XML preamble (entity), if you need to combine multiple kickstart records into one document.

Additionally, if specified, the environment and the resource usage segments will not be written, assuming that a in a concatinated record version, the initial run will have captured those settings.

`-R site` In order to provide the greater picture, kickstart can reflect the site handle (resource identifier) into its output.

There is no default. If no value is given, the attribute will not be generated.

`-L lbl`, `-T iso` These optional arguments denote the workflow label (from DAX) and the workflow's last modification time (from DAX). The label *lbl* can be any sensible string of up to 32 characters, but should use C identifier characters. The timestamp *iso* must be an ISO 8601 compliant time-stamp.

`-S l=p` If stat information on any file is required *before* any jobs were started, logical to physical file mappings to stat can be passed using the `-S` option. The LFN and PFN are concatenated by an equals (=) sign. The LFN is optional: If no equals sign is found, the argument is taken as sole PFN specification without LFN.

This option may be specified multiple times. To reduce and overcome commandline length limits, if the argument is prefixed with an at (@) sign, the argument is taken to be a textual file of LFN to PFN mappings. The optionality mentioned above applies. Each line inside the file argument is the name of a file to stat. Comments (#) and empty lines are permitted.

Each PFN will incur a statcall record (element) with attribute *id* set to value *initial*. The optional *lfn* attribute is set to the LFN stat'ed. The filename is part of the statinfo record inside.

There is no default.

`-s fn` If stat information on any file is required *after* all jobs have finished, logical to physical file mappings to stat can be passed using the `-s` option. The LFN and PFN are concatenated by an equals (=) sign. The LFN is optional: If no equals sign is found, the argument is taken as sole PFN specification without LFN.

This option may be specified multiple times. To reduce and overcome commandline length limits, if the argument is prefixed with an at (@) sign, the argument is taken to be a textual file of LFN to PFN mappings. The optionality mentioned above applies. Each line inside the file argument is the name of a file to stat. Comments (#) and empty lines are permitted.

Each PFN will incur a statcall record (element) with attribute *id* set to value *final*. The optional *lfn* attribute is set to the LFN stat'ed. The filename is part of the statinfo record inside.

There is no default.

`-i fn` This option allows *kickstart* to re-connect the stdin of the application that it starts. Use a single hyphen to share *stdin* with the one provided to **kickstart**.

The default is to connect *stdin* to `/dev/null`.

`-o fn` This option allows **kickstart** to re-connect the *stdout* of the application that it starts. The mode is used whenever an application produces meaningful results on its *stdout* that need to be tracked by Pegasus. The real*stdout* of Globus jobs is staged via GASS (GT2) or RFT (GT4), or whichever other means your grid middleware uses. The real *stdout* is used to propagate the invocation record back to the submit site. Use the single hyphen to share the application's *stdout* with the one that is provided to **kickstart**. In that case, the output from *kickstart* will interleave with application output. For this reason, such a mode is not recommended.

In order to provide an uncaptured *stdout* as part of the results, it is the default to connect the *stdout* of the application to a temporary file. The content of this temporary file will be transferred as payload data in the *kickstart* results. The content size is subject to payload limits, see the `-B` option. If the content grows large, only an initial portion will become part of the payload. If the temporary file grows too large, it may flood the worker node's temporary space. The temporary file will be deleted after **kickstart** finishes.

If the filename is prefixed with an exclaimation point, the file will be opened in append mode instead of overwrite mode. Note that you may need to escape the exclaimation point from the shell.

The default is to connect *stdout* to a temporary file.

`-e fn` This option allows **kickstart** to re-connect the *stderr* of the application that it starts. This option is used whenever an application produces meaningful results on *stderr* that needs tracking by Pegasus. The real *stderr* of Globus jobs is staged via GASS (GT2) or RFT (GT4). It is used to propagate abnormal behaviour from both, *kickstart* and the application that it starts, though its main use is to propagate application dependent data and heartbeats. Use a single hyphen to share *stderr* with the *stderr* that is provided to **kickstart**. This is the backward compatible behavior.

In order to provide an uncaptured *stderr* as part of the results, by default the *stderr* of the application will be connected to a temporary file. Its content is transferred as payload data in the *kickstart* results. If too large, only the an initial portion will become part of the payload. If the temporary file grows too large, it may flood the worker node's temporary space. The temporary file will be deleted after **kickstart** finishes.

If the filename is prefixed with an exclaimation point, the file will be opened in append mode instead of overwrite mode. Note that you may need to escape the exclaimation point from the shell.

The default is to connect *stderr* to a temporary file.

`-l logfn` allows to append the performance data to the specified file. Thus, multiple XML documents may end up in the same file, including their XML preamble. *stdout* is normally used to stream back the results. Usually, this is a GASS-staged stream. Use a single hyphen to generate the output on the *stdout* that was provided to **kickstart**, the default behavior.

Default is to append the invocation record onto the provided *stdout*.

`-w dir` permits the explicit setting of a new working directory once **kickstart** is started. This is useful in a remote scheduling environment, when the chosen working directory is not visible on the job submitting host. If the directory does not exist, **kickstart** will fail. This option is mutually exclusive with the `-W dir` option.

Default is to use the working directory that the application was started in. This is usually set up by a remote scheduling environment.

`-W dir` permits the explicit creation and setting of a new working directory once **kickstart** is started. This is useful in a remote scheduling environment, when the chosen working directory is not visible on the job submitting host. If the directory does not exist, **kickstart** will attempt to create it, and then change into it. Both, creation and directory change may still fail. This option is mutually exclusive with the `-w dir` option.

Default is to use the working directory that the application was started in. This is usually set up by a remote scheduling environment.

`-X` make an application executable, no matter what. It is a work-around code for a weakness of **globus-url-copy** which does not copy the permissions of the source to the destination. Thus, if an executable is staged-in using GridFTP, it will have the wrong permissions. Specifying the `-X` flag will attempt to change the mode to include the necessary x (and r) bits to make the application executable.

Default is not to change the mode of the application. Note that this feature can be misused by hackers, as it is attempted to call **chmod** on whatever path is specified.

`-B sz` varies the size of the debug output data section. If the file descriptors *stdout* and *stderr* remain untracked, **kickstart** tracks that output in temporary files. The first few pages from this output is copied into a data section in the output. In order to resize the length of the output within reasonable boundaries, this option permits a changes. Data beyond the size will not be copied, i.e. is truncated.

Warning: This is *not* a cheap way to obtain the stdio file handle data. Please use tracked files for that. Due to output buffer pre-allocation, using arbitrary large arguments may result in failures of **kickstart** itself to allocate the necessary memory.

The default maximum size of the data section is 262144 byte.

`-I fn` In this mode, the application name and any arguments to the application are specified inside of file `fn`. The file contains one argument per line. Escapeing from Globus, Condor and shell meta characters is not required. This mode permits to use the maximum possible commandline length of the underlying operationg system, e.g. 128k for Linux. Using the `-I` mode stops any further commandline processing of **kickstart** command lines.

Default is to use the *app flags* mode, where the application is specified explicitely on the command-line.

app The path to the application has to be completely specified. The application is a mandatory option.

appflags Application may or may not have additional flags.

RETURN VALUE

Kickstart will return the return value of the main job. In addition, the error code 127 signals that the call to exec failed, and 126 that reconnecting the stdio failed. A job failing with the same exit codes is indistinguishable from ***kickstart*** failures.

SUBJOBS

Subjobs are a new feature and may have a few wrinkles left.

In order to allow specific setups and assertion checks for compute nodes, **kickstart** allows the optional execution of a *prejob*. This *prejob* is anything that the remote compute node is capable of executing. For modern Unix systems, this includes #! scripts interpreter invocations, as long as the x bits on the executed file are set. The main job is run if and only if the prejob returned regularly with an exit code of zero.

With similar restrictions, the optional execution of a *postjob* is chained to the success of the main job. The postjob will be run, if the main job terminated normally with an exit code of zero.

In addition, a user may specify a *setup* and a *cleanup* job. The *setup* job sets up the remote execution environment. The *cleanup* job may tear down and clean-up after any job ran. Failure to run the setup job has no impact on subsequent jobs. The cleanup is a job that will even be attempted to run for all failed jobs. No job information is passed. If you need to invoke multiple setup or clean-up jobs, bundle them into a script, and invoke the clean-up script. Failure of the clean-up job is not meant to affect the progress of the remote workflow (DAGMan). This may change in the future.

The setup-, pre-, and post- and cleanup-job run on the same compute node as the main job to execute. However, since they run in separate processes as children of *kickstart*, they are unable to influence each others nor the main jobs environment settings.

All jobs and their arguments are subject to variable substitutions as explained in the next section.

To specify the prejob, insert the the application invocation and any optional commandline argument into the environment variable `GRIDSTART_PREJOB`. If you are invoking from a shell, you might want to use single quotes to protect against the shell. If you are invoking from Globus, you can append the RSL string feature. From Condor, you can use Condor's notion of environment settings. From Pegasus, use the *profile* command to set generic scripts that will work on multiple sites, or the transformation catalog to set environment variables in a pool-specific fashion. Please remember that the execution of the main job is chained to the success of the prejob.

To set up the postjob, use the environment variable `GRIDSTART_POSTJOB` to point to an application with potential arguments to execute. The same restrictions as for the prejob apply. Please note that the execution of the post job is chained to the main job.

To provide the independent setup job, use the environment variable `GRIDSTART_SETUP`. The exit code of the setup job has no influence on the remaining chain of jobs. To provide an independent cleanup job, use the environment variable `GRIDSTART_CLEANUP` to point to an application with possible arguments to execute. The same restrictions as for prejob and postjob apply. The cleanup is run regardless of the exit status of any other jobs.

VARIABLE REWRITING

Variable substitution is a new feature and may have a few wrinkles left.

The variable substitution employs simple rules from the Bourne shell syntax. Simple quoting rules for backslashed characters, double quotes and single quotes are obeyed. Thus, in order to pass a dollar sign to as argument to your job, it must be escaped with a backslash from the variable rewriting.

For pre- and postjobs, double quotes allow the preservation of whitespace and the insertion of special characters like \a (alarm), \b (backspace), \n (newline), \r (carriage return), \t (horizontal tab), and \v (vertical tab). Octal modes are *not* allowed. Variables are still substituted in double quotes. Single quotes inside double quotes have no special meaning.

Inside single quotes, no variables are expanded. The backslash only escapes a single quote or backslash.

Backticks are not supported.

Variables are only substituted once. You cannot have variables in variables. If you need this feature, please request it.

Outside quotes, arguments from the pre- and postjob are split on linear whitespace. The backslash makes the next character verbatim.

Variables that are rewritten must start with a dollar sign either outside quotes or inside double quotes. The dollar may be followed by a valid identifier. A valid identifier starts with a letter or the underscore. A valid identifier may contain further letters, digits or underscores. The identifier is case sensitive.

The alternative use is to enclose the identifier inside curly braces. In this case, almost any character is allowed for the identifier, including whitespace. This is the *only* curly brace expansion. No other Bourne magic involving curly braces is supported.

One of the advantages of variable substitution is, for example, the ability to specify the application as **`$HOME/bin/app1`** in the transformation catalog, and thus to gridstart. As long as your home directory on any compute node has a `bin` directory that contains the application, the transformation catalog does not need to care about the true location of the application path on each pool. Even better, an administrator may decide to move your home directory to a different place. As long as the compute node is set up correctly, you don't have to adjust any Pegasus data.

Mind that variable substitution is an expert feature, as some degree of tricky quoting is required to protect substitutable variables and quotes from Globus, Condor and Pegasus in that order. Note that Condor uses the dollar sign for its own variables.

The variable substitution assumptions for the main job differ slightly from the prejob and postjob for technical reasions. The pre- and postjob commandlines are passed as one string. However, the main jobs commandline is already split into pieces by the time it reaches *kickstart*. Thus, any whitespace on the main job's commandline must be preserved, and further argument splitting avoided.

It is highly recommended to experiment on the Unix commandline with the *echo* and *env* applications to obtain a feeling for the different quoting mechanisms needed to achieve variable substitution.

FEEDBACK CHANNEL

A long-running application may consider to stream back heart beats and other application-specific monitoring and progress data. For this reason, **kickstart** provides a feedback channel. At start-up, a transient named pipe, also known as FIFO, is created. While waiting for started jobs to finish, *kickstart* will attempt to read from the FIFO. By default, any information read will be encapsulated in XML tags, and written to *stderr* . Please note that in a Pegasus, Globus, Condor-G environment, *stderr* will be GASS streamed or staged to the submit host. At the submit host, an application specific monitor may unpack the data chunks and could for instance visually display them, or aggregate them with other data. Please note that *kickstart* only provides a feedback channel. The content and interpretation is up to, and specific for the application.

In order to make an application gridstart aware, it needs to be able to write to a FIFO. The filename can be picked up from the environment variable `GRIDSTART_CHANNEL` which is provided to all jobs. Please note that the application must be prepared to handle the PIPE signal when writing to a FIFO, and must be able to cope with failing write operations.

EXAMPLE

You can run the **kickstart** executable locallly to verify that it is functioning well. In the initial phase, the format of the performance data may be slightly adjusted.

```
$ env GRIDSTART_PREJOB='/bin/usleep 250000' \\
  GRIDSTART_POSTJOB='/bin/date -u' \\
  kickstart -l xx \\$PEGASUS_HOME/bin/keg -T1 -o-
$ cat xx
<?xml version="1.0" encoding="ISO-8859-1"?>
  ...
  </statcall>
</invocation>
```

Please take note a few things in the above example:

The output from the postjob is appended to the output of the main job on *stdout*. The output could potentially be separated into different data sections through different temporary files. If you truly need the separation, request that feature.

The log file is reported with a size of zero, because the log file did indeed barely exist at the time the data structure was (re-) initialized. With regular GASS output, it will report the status of the socket file descriptor, though.

The file descriptors reported for the temporary files are from the perspective of **kickstart**. Since the temporary files have the *close-on-exec* flag set, **kickstart**'s filedescriptors are invisible to the job processes. Still, the *stdio* of the job processes are connected to the temporary files.

Even this output already appears large. The output may already be too large to guarantee that the append operation on networked pipes (GASS, NFS) are atomically written.

The current format of the performance data is as follows:

OUTPUT FORMAT

Refer to *https://pegasus.isi.edu/wms/docs/schemas/iv-2.1/iv-2.1.html* for an up-to-date description of elements and their attributes. Check with *https://pegasus.isi.edu/wms/schema.php* for invocation schemas with a higher version number.

RESTRICTIONS

There is no version for the Condor *standard* universe. It is simply not possible within the constraints of Condor.

Due to its very nature, **kickstart** will also prove difficult to port outside the Unix environment.

Any of the pre-, main-, cleanup and postjob are unable to influence one anothers visible environment.

Do not use a Pegasus transformation with just the name *null* and no namespace nor version.

First Condor, and then Unix, place a limit on the length of the commandline. The additional space required for the gridstart invocation may silently overflow the maximum space, and cause applications to fail. If you suspect to work with many arguments, try an argument-file based approach.

A job failing with exit code 126 or 127 is indistinguishable from **kickstart** failing with the same exit codes. Sometimes, careful examination of the returned data can help.

If the logfile is collected into a shared file, due to the size of the data, simultaneous appends on a shared filesystem from different machines may still mangle data. Currently, file locking is not even attempted, although all data is written atomically from the perspective of **kickstart**.

The upper limit of characters of commandline characters is currently not checked by **kickstart**. Thus, some variable substitutions could potentially result in a commandline that is larger than permissable.

If the output or error file is opened in append mode, but the application decides to truncate its output file, as in aboves example by opening `/dev/fd/1` inside **keg**, the resulting file will still be truncated. This is correct behavior, but sometimes not obvious.

FILES

$PEGASUS_HOME/etc/iv-2.1.xsd is the suggested location of the latest XML schema describing the data on the submit host.

ENVIRONMENT VARIABLES

`GRIDSTART_TMP`	is the hightest priority to look for a temporary directory, if specified. This rather special variable was introduced to overcome some peculiarities with the FNAL cluster.
`TMP`	is the next hightest priority to look for a temporary directory, if specified.
`TEMP`	is the next priority for an environment variable denoting a temporary files directory.
`TMPDIR`	is next in the checklist. If none of these are found, either the *stdio* definition *P_tmpdir* is taken, or the fixed string`/tmp .`
`GRIDSTART_SETUP`	contains a string that starts a job to be executed unconditionally before any other jobs, see above for a detailled description.
`GRIDSTART_PREJOB`	contains a string that starts a job to be executed before the main job, see above for a detailled description.
`GRIDSTART_POSTJOB`	contains a string that starts a job to be executed conditionally after the main job, see above for a detailled description.
`GRIDSTART_CLEANUP`	contains a string that starts a job to be executed unconditionally after any of the previous jobs, see above for a detailled description.
`GRIDSTART_CHANNEL`	is the name of a FIFO for an application-specific feedback-channel, see above for a detailled description.

Chapter 10. Useful Tips

Migrating From Pegasus 2.X to Pegasus 3.X

With Pegasus 3.0 effort has been made to simplify configuration. This chapter is for existing users of Pegasus who use Pegasus 2.x to run their workflows and walks through the steps to move to using Pegasus 3.0

PEGASUS_HOME and Setup Scripts

Earlier versions of Pegasus required users to have the environment variable PEGASUS_HOME set and to source a setup file $PEGASUS_HOME/setup.sh | $PEGASUS_HOME/setup.csh before running Pegasus to setup CLASSPATH and other variables.

Starting with Pegasus 3.0 this is no longer required. The above paths are automaticallly determined by the Pegasus tools when they are invoked.

All the users need to do is to set the PATH variable to pick up the pegasus executables from the bin directory.

```
$ export PATH=/some/install/pegasus-3.0.0/bin:$PATH
```

Changes to Schemas and Catalog Formats

DAX Schema

Pegasus 3.0 by default now parses DAX documents conforming to the DAX Schema 3.2 available here [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 10.1. 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 10.2. 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 10.3. 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 10.4. 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 11. 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

Chapter 12. Pegasus Tutorial Using Self-contained Virtual Machine

These are the student notes for the Pegasus WMS tutorial on the Virtual Machine that can be downloaded from the Pegasus Website. They are designed to be used in conjunction with instructor presentation and support.

You will see two styles of machine text here:

```
Text like this is input that you should type.

Text like this is the output you should get.
```

For example:

```
$ date
Wed Jun 24 14:47:59 PST 2011
```

Downloading and Running the VM using Virtual Box

You will need to install Virtual Box to run the virtual machine on your computer. If you already have one of the tools installed, use that. Otherwise download the binary versions and install them from the Virtual Box Website [http://www.virtualbox.org/wiki/Downloads] .

The instructors have tested the image with Virtual Box 3.2.10

Download the VM for Virtual Box use

Download the corresponding disk image.

- Virtual Box Pegasus Image [http://pegasus.isi.edu/wms/download/3.1/Pegasus-3.1.0-Debian-6-x86.vbox.zip]

  It is around **1.2 GB** in size. We recommend using a command line tool like **wget** to download the image. Downloading the image using the browser sometimes corrupts the image.

  ```
  $ wget http://pegasus.isi.edu/wms/download/3.1/Pegasus-3.1.0-Debian-6-x86.vbox.zip

   http://pegasus.isi.edu/wms/download/3.1/Pegasus-3.1.0-Debian-6-x86.vbox.zip
             => `Pegasus-3.1.0-Debian-6-x86.vbox.zip'
  Resolving pegasus.isi.edu... 128.9.64.219
  Connecting to pegasus.isi.edu|128.9.64.219|:80... connected.
  HTTP request sent, awaiting response... 200 OK
  Length: 1,336,554,492 (1.2G) [application/x-zip]
  ```

  The Image is zipped. You will need to unzip it.

  If you have unzip you can do this directly

  ```
  $ unzip Pegasus-3.1.0-Debian-6-x86.vbox.zip
  ```

  After unzipping a folder named **Pegasus-3.1.0-Debian-6-x86.vbox** will be created that has the vmdk files for the VM.

Running the VM with Virtual Box

Launch Virtual Box on your machine. Follow the steps to add the vmdk file to Virtual Box and create a virtual machine inside the Virtual Box

1. In the Menu, click File and select Virtual Media Manager (File > Virtual Media Manager)
2. The Virtual Media Manager Windows opens up.
3. Click on "Add" button to add the **Pegasus-3.1.0-Debian-6-x86.vbox/Debian-6-x86.vmdk** file that you just downloaded and unzipped.

4. You will now see the Debian-6-x86.vmdk in the list of hard disks with Actual size listed as around 3.0 GB
5. Close the Window for the Virtual Media Manager

We will now create a Virtual Machine in the Virtual Box.

1. In the Menu, click Machine and select New (Machine > New)
2. It will open the New Virtual Machine Wizard. Click Continue
3. In the VM Name and OS Type Window specify the name as **PegasusVM-3.1.0**.
4. Select the **Operating System as Linux**and **Version as Debian**. Click Continue.
5. Set the base memory to **384 MB** . It defaults to 512 MB. Click Continue
6. We now select the Virtual Hard Disk to use with the machine. Select the option box for Use Existing Hard Disk. Select **Debian-6-x86.vmdk** from the list . Click Continue
7. Click Done.
8. Now in the Virtual Box , start the PegasusVM-3.1.0 machine.

Mapping and Executing Workflows using Pegasus

In this chapter you will be introduced to planning and executing a workflow through Pegasus WMS locally. You will then plan and execute a larger Montage workflow on the GRID.

When the virtual machine starts , it will automatically log you in as user **tutorial** . The password for this account is **pegasus**.

After logging on, start a terminal. There is a shortcut on the desktop for the terminal.

```
$ tutorial@pegasus-vm:$ pwd

/home/tutorial
```

In general, to run workflows on the Grid you will need to obtain Grid Credentials. The VM already has a user certificate installed for the pegasus user. To generate the proxy (grid credentials) run the **grid-proxy-init** command.

```
$ [pegasus@pegasus ~]$ grid-proxy-init

Your identity: /O=edu/OU=ISI/OU=isi.edu/CN=Tutorial User
Creating proxy ................................ Done
Your proxy is valid until: Sat Jul 30 00:31:39 2011
```

All the exercises in this Chapter will be run from the $HOME/pegasus-wms/ directory. All the files that are required reside in this directory

```
$ cd $HOME/pegasus-wms
```

Files for the exercise are stored in subdirectories:

```
$ ls

config  dax
```

You may also see some other files here.

Creating a DIAMOND DAX

We generate a 4 node diamond dax. There is a small piece of java code that uses the DAX API to generate the DAX. Open the file $HOME/pegasus-wms/dax/CreateDAX.java in a file editor:

```
$ vi dax/CreateDAX.java
```

There is a function Diamond(String site_handle, String pegasus_location) that constructs the DAX. Towards the end of the function there is some commented out code.

```
// Add analyze job
//To be uncommented for exercise 2.1

        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);

        //add job to the DAX
        dax.addJob(j4);

        //analyze job is a child to the findrange jobs
        dax.addDependency("j2", "j4");
        dax.addDependency("j3", "j4");

//End of commented out code for Exercise 2.1
```

The above snippet of code, adds a job with the ID0000004 to the DAX. It illustrates how to specify

1. the arguments for the job
2. the logical files used by the job
3. the dependencies to other jobs
4. adding the job to the dax

After uncommenting the code, compile and run the CreateDAX program.

```
$ cd dax

$ javac -classpath .:/opt/pegasus/default/lib/pegasus.jar CreateDAX.java

$ java -classpath .:/opt/pegasus/default/lib/pegasus.jar CreateDAX local /opt/pegasus/default ./
diamond.dax
```

Let us view the generated diamond.dax.

```
$ cat diamond.dax
```

Inside the DAX, you should see three sections.

1. list of input file locations
2. list of executable locations
3. definition of all jobs - each job in the workflow. 4 jobs in total.
4. list of control-flow dependencies - this section specifies a partial order in which jobs are to executed.

Replica Catalog

First lets change to the tutorial base directory.

```
$ cd $HOME/pegasus-wms
```

In this exercise you will insert entries into the Replica Catalog. The replica catalog that we will use today is a simple file based catalog. We also support and recommend the following for production runs

- Globus RLS
- JDBC implementation

A Replica Catalog maintains the LFN to PFN mapping for the input files of your workflow. Pegasus queries it to determine the locations of the raw input data files required by the workflow. Additionally, all the materialized data is registered into Replica Catalog for data reuse later on.

Pre Populated Replica Catalog

The instructors have provided a File based Replica Catalog configured for the tutorial exercises. The file is inside the config directory.

- Let us see what the file looks like.

```
$ cat config/rc.data

f.a
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/diamond/f.a
          pool="local"
2mass-atlas-990502s-j1420198.fits
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/2mass-atlas-990502s-j1420198.fits
          pool="local"
2mass-atlas-990502s-j1430092.fits
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/2mass-atlas-990502s-j1430092.fits
          pool="local"
images_20080505_143233_14944.tbl
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/images.tbl
          pool="local"
2mass-atlas-990502s-j1430080.fits
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/2mass-atlas-990502s-j1430080.fits
          pool="local"
2mass-atlas-990502s-j1440198.fits
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/2mass-atlas-990502s-j1440198.fits
          pool="local"
2mass-atlas-990502s-j1440186.fits
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/2mass-atlas-990502s-j1440186.fits
          pool="local"
2mass-atlas-990502s-j1420186.fits
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/2mass-atlas-990502s-j1420186.fits
          pool="local"
cimages_20080505_143233_14944.tbl
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/cimages.tbl
          pool="local"
dag_20080505_143233_14944.xml
     gsiftp://pegasus-vm/home/tutorial/pegasus-wms/dax/montage.dax
          pool="local"
region_20080505_143233_14944.hdr
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/region.hdr
          pool="local"
pimages_20080505_143233_14944.tbl
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/pimages.tbl
          pool="local"
statfile_20080505_143233_14944.tbl
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/statfile.tbl
          pool="local"
```

pegasus-rc-client (Optional Exercise)

You can use the **pegasus-rc-client** command to insert , query and delete from the replica catalog.

Before executing any of the pegasus-rc-client exercises lets us remove the pre populated replica catalog.

```
$ rm $HOME/pegasus-wms/config/rc.data
```

To execute the diamond dax created in **exercise 2.1**, we will need to register input file f.a in the replica catalog. The file f.a resides at /scratch/tutorial/inputdata/diamond/f.a . Let us insert a single entry into the replica catalog.

```
$  cat
```

Let us know verify if f.a has been registered successfully by querying the replica catalog using pegasus-rc-client

```
$ pegasus-rc-client  lookup f.a

 f.a gsiftp://pegasus-vm/scratch/tutorial/inputdata/diamond/f.a pool="local"
```

The **pegasus-rc-client** also allows for bulk insertion of entries. We will be inserting the entries for montage workflow using the bulk mode. The input data to be used for the montage workflow resides in the /scratch/tutorial/inputdata/0.2degree directory. We are going to insert entries into the replica catalog that point to the files in this directory.

The instructors have provided:

- A file replicas.in, the input data file for the pegasus-rc-client that contains the mappings that need to be populated in the Replica Catalog. The file is inside the config directory
- Let us see what the file looks like.

```
$ cat config/rc.in

statfile_20080505_143233_14944.tbl
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/statfile.tbl
          pool="local"
region_20080505_143233_14944.hdr
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/region.hdr
          pool="local"
pimages_20080505_143233_14944.tbl
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/pimages.tbl
          pool="local"
cimages_20080505_143233_14944.tbl
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/cimages.tbl
          pool="local"
images_20080505_143233_14944.tbl
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/images.tbl
          pool="local"
2mass-atlas-990502s-j1440198.fits
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/2mass-atlas-990502s-j1440198.fits
          pool="local"
2mass-atlas-990502s-j1430092.fits
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/2mass-atlas-990502s-j1430092.fits
          pool="local"
2mass-atlas-990502s-j1440186.fits
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/2mass-atlas-990502s-j1440186.fits
          pool="local"
2mass-atlas-990502s-j1420186.fits
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/2mass-atlas-990502s-j1420186.fits
          pool="local"
2mass-atlas-990502s-j1420198.fits
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/2mass-atlas-990502s-j1420198.fits
          pool="local"
2mass-atlas-990502s-j1430080.fits
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/2mass-atlas-990502s-j1430080.fits
          pool="local"
dag_20080505_143233_14944.xml
     gsiftp://pegasus-vm/home/tutorial/pegasus-wms/dax/montage.dax
          pool="local"
```

- Now we are ready to run rc-client and populate the data. Since each of you have an individual file replica catalog, all the 10 entries should be successfully registered.

```
$ pegasus-rc-client  --insert config/rc.in

#Successfully worked on : 12 lines
#Worked on total number of : 12 lines.
```

- Now the entries have been successfully inserted into the Replica Catalog. We should query the replica catalog for a particular lfn.

```
$ pegasus-rc-client lookup pimages_20080505_143233_14944.tbl

pimages_20080505_143233_14944.tbl
     gsiftp://pegasus-vm/scratch/tutorial/inputdata/0.2degree/pimages.tbl
           pool="local"
```

The Site Catalog

The site catalog contains information about the layout of your grid where you want to run your workflows. For each site following information is maintained

- grid gateways
- head node filesystem

- worker node filesystem
- scratch and shared file systems on the head nodes and worker nodes
- replica catalog URL for the site
- site wide information like environment variables to be set when a job is run.

Pre Populated Site Catalog

The instructors have provided a pre-populated site catalog for use in the tutorial in $HOME/pegasus-wms/config directory.

Lets see the site catalog for the Pegasus VM. It refers to two sites **local** and **cluster** .

```
$ cat $HOME/pegasus-wms/config/sites.xml3
```

```
<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="cluster" arch="x86" os="LINUX" osrelease="" osversion="" glibc="">
  <grid  type="gt2" contact="pegasus-vm/jobmanager-fork" scheduler="Fork" jobtype="auxillary"/>
  <grid  type="gt2" contact="pegasus-vm/jobmanager-condor" scheduler="SGE" jobtype="compute"/>
  <head-fs>
   <scratch>
    <shared>
     <file-server protocol="gsiftp" url="gsiftp://pegasus" mount-point="/home/tutorial/cluster-
scratch"/>
     <internal-mount-point mount-point="/home/tutorial/cluster-scratch"/>
    </shared>
   </scratch>
   <storage>
    <shared>
     <file-server protocol="gsiftp" url="gsiftp://pegasus" mount-point="/home/tutorial/cluster-
storage"/>
     <internal-mount-point mount-point="/home/tutorial/cluster-storage"/>
    </shared>
   </storage>
  </head-fs>
  <replica-catalog  type="LRC" url="rlsn://localhost"/>
  <profile namespace="env" key="GLOBUS_LOCATION" >/opt/globus/default</profile>
  <profile namespace="env" key="JAVA_HOME" >/usr</profile>
  <profile namespace="env" key="LD_LIBRARY_PATH" >/opt/globus/default/lib</profile>
  <profile namespace="env" key="PEGASUS_HOME" >/opt/pegasus/default</profile>
  <profile namespace="pegasus" key="clusters.num" >1</profile>
  <profile namespace="pegasus" key="stagein.clusters" >1</profile>
 </site>

 <site  handle="local" arch="x86" os="LINUX" osrelease="" osversion="" glibc="">
  <grid  type="gt2" contact="localhost/jobmanager-fork" scheduler="Fork" jobtype="auxillary"/>
  <grid  type="gt2" contact="localhost/jobmanager-fork" scheduler="Fork" jobtype="compute"/>
  <head-fs>
   <scratch>
    <shared>
     <file-server protocol="gsiftp" url="file://" mount-point="/home/tutorial/local-scratch"/>
     <internal-mount-point mount-point="/home/tutorial/local-scratch"/>
    </shared>
   </scratch>
   <storage>
    <shared>
     <file-server protocol="gsiftp" url="file://" mount-point="/home/tutorial/local-storage"/>
     <internal-mount-point mount-point="/home/tutorial/local-storage"/>
    </shared>
   </storage>
  </head-fs>
  <replica-catalog  type="LRC" url="rlsn://localhost"/>
  <profile namespace="env" key="GLOBUS_LOCATION" >/opt/globus/default</profile>
  <profile namespace="env" key="JAVA_HOME" >/usr</profile>
  <profile namespace="env" key="LD_LIBRARY_PATH" >/opt/globus/default/lib</profile>
  <profile namespace="env" key="PEGASUS_HOME" >/opt/pegasus/default</profile>
 </site>
</sitecatalog>
```

Generating a Site Catalog for OSG

The client pegasus-sc-client can be used to generate a site catalog and transformation catalog for the Open Science Grid.

```
$ [pegasus@pegasus pegasus-wms]$ pegasus-sc-client --vo engage --sc ./engage-osg-sc.xml \
  --source OSGMM --grid OSG -vvvv

2011.07.29 12:44:32.734 PDT: [INFO]  Adding site GridUNESP_CENTRAL
2011.07.29 12:44:32.878 PDT: [INFO]  Adding site Clemson-Palmetto
2011.07.29 12:44:32.879 PDT: [INFO]  Adding site MIT_CMS
2011.07.29 12:44:32.880 PDT: [INFO]  Adding site MIT_CMS__1
2011.07.29 12:44:32.881 PDT: [INFO]  Adding site UFlorida-HPC
2011.07.29 12:44:32.882 PDT: [INFO]  Adding site BNL_Test_2
2011.07.29 12:44:32.883 PDT: [INFO]  Adding site Purdue-Rossmann
2011.07.29 12:44:32.884 PDT: [INFO]  Adding site UConn-OSG
2011.07.29 12:44:32.885 PDT: [INFO]  Adding site USCMS-FNAL-WC1
2011.07.29 12:44:32.886 PDT: [INFO]  Adding site FNAL_FERMIGRID
2011.07.29 12:44:32.886 PDT: [INFO]  Adding site LIGO_UWM_NEMO
2011.07.29 12:44:32.887 PDT: [INFO]  Adding site SPRACE
2011.07.29 12:44:32.888 PDT: [INFO]  Adding site AGLT2
2011.07.29 12:44:32.888 PDT: [INFO]  Adding site GridUNESP_SPO
2011.07.29 12:44:32.889 PDT: [INFO]  Adding site UCR-HEP
2011.07.29 12:44:32.892 PDT: [INFO]  Adding site UMissHEP
2011.07.29 12:44:32.893 PDT: [INFO]  Adding site Purdue-Steele
2011.07.29 12:44:32.894 PDT: [INFO]  Adding site STAR-BNL
2011.07.29 12:44:32.894 PDT: [INFO]  Adding site MWT2
2011.07.29 12:44:32.895 PDT: [INFO]  Adding site Firefly
2011.07.29 12:44:32.895 PDT: [INFO]  Adding site RENCI-Blueridge
2011.07.29 12:44:32.896 PDT: [INFO]  Adding site BNL-ATLAS
2011.07.29 12:44:32.897 PDT: [INFO]  Adding site Prairiefire
2011.07.29 12:44:32.897 PDT: [INFO]  Adding site MWT2_UC
2011.07.29 12:44:32.897 PDT: [INFO]  Adding site Firefly__1
2011.07.29 12:44:32.898 PDT: [INFO]  Adding site Purdue-RCAC
2011.07.29 12:44:32.899 PDT: [INFO]  Adding site RENCI-Engagement
2011.07.29 12:44:32.923 PDT: [INFO]  Adding site SBGrid-Harvard-East
2011.07.29 12:44:32.924 PDT: [INFO]  Adding site FNAL_GPGRID_1
2011.07.29 12:44:32.924 PDT: [INFO]  Site LOCAL . Creating default entry
2011.07.29 12:44:33.034 PDT: [INFO]  Loaded 31 sites
2011.07.29 12:44:33.035 PDT:   Writing out site catalog to /home/tutorial/pegasus-wms/./engage-osg-
sc.xml
2011.07.29 12:44:33.781 PDT:   Number of SRM Properties retrieved 20
2011.07.29 12:44:33.847 PDT:   Writing out properties to /home/tutorial/pegasus-wms/./
pegasus.526913080507226851.properties
2011.07.29 12:44:33.854 PDT: [INFO]  Time taken to execute is 2.083 seconds
2011.07.29 12:44:33.854 PDT: [INFO] event.pegasus.planner planner.version 3.1.0  - FINISHED
```

Transformation Catalog

The transformation catalog maintains information about where the application code resides on the grid. 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.

Pre Populated Transformation Catalog

The instructors have provided a ready transformation catalog (tc.data.text) in the $HOME/pegasus-wms/config directory

In our case, it contains the locations where the Diamond or Montage code is installed in the Pegasus VM. Let us see the Transformation Catalog

For each transformation the following information is captured

1. 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.

$ cat $HOME/pegasus-wms/config/tc.data.text

```
tr bin/mDiff {
        site local {
                profile env "MONTAGE_HOME" "."
                pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mDiff"
                arch "x86"
                os "LINUX"
                type "STAGEABLE"
        }
}

tr bin/mFitplane {
        site local {
                pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mFitplane"
                arch "x86"
                os "LINUX"
                type "STAGEABLE"
        }
}

tr condor::dagman {
        site local {
                pfn "/usr/bin/condor_dagman"
                arch "x86"
                os "LINUX"
                type "INSTALLED"
        }
}

tr diamond::findrange:2.0 {
        site local {
                pfn "/opt/pegasus/default/bin/keg"
                arch "x86"
                os "LINUX"
                type "INSTALLED"
        }
}

tr diamond::preprocess:2.0 {
        site local {
                pfn "/opt/pegasus/default/bin/keg"
                arch "x86"
                os "LINUX"
                type "INSTALLED"
        }
}

tr mAdd:3.0 {
        site local {
                pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mAdd"
                arch "x86"
                os "LINUX"
                type "STAGEABLE"
        }
}

tr mBackground:3.0 {
        site local {
                pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mBackground"
```

```
                arch "x86"
                os "LINUX"
                type "STAGEABLE"
        }
}

tr mBgModel:3.0 {
        site local {
                pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mBgModel"
                arch "x86"
                os "LINUX"
                type "STAGEABLE"
        }
}

tr mConcatFit:3.0 {
        site local {
                pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mConcatFit"
                arch "x86"
                os "LINUX"
                type "STAGEABLE"
        }
}

tr mDiffFit:3.0 {
        site local {
                profile env "MONTAGE_HOME" "."
                pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mDiffFit"
                arch "x86"
                os "LINUX"
                type "STAGEABLE"
        }
}

tr mImgtbl:3.0 {
        site local {
                pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mImgtbl"
                arch "x86"
                os "LINUX"
                type "STAGEABLE"
        }
}

tr mJPEG:3.0 {
        site local {
                pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mJPEG"
                arch "x86"
                os "LINUX"
                type "STAGEABLE"
        }
}

tr mProjectPP:3.0 {
        site local {
                profile condor "priority" "25"
                pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mProjectPP"
                arch "x86"
                os "LINUX"
                type "STAGEABLE"
        }
}

tr mShrink:3.0 {
        site local {
                pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mShrink"
                arch "x86"
                os "LINUX"
                type "STAGEABLE"
        }
}
```

pegasus-tc-client (Optional)

We will use the **pegasus-tc-client** to add the entry for the transformation dummy into the transformation catalog.

- ```
  $ pegasus-tc-client  -a -l diamond::dummy:2.0 \
        -p /opt/pegasus/default/bin/keg -r local -t INSTALLED -s x86::LINUX

  2011.08.04 12:03:12.555 PDT:   Added tc entry sucessfully
  ```

  Let us try and query for the entry we inserted.

  ```
  $ pegasus-tc-client  -q -P -l diamond::dummy:2.0

  tr diamond::dummy:2.0 {
          site local {
                  pfn "/opt/pegasus/default/bin/keg"
                  arch "x86"
                  os "LINUX"
                  type "INSTALLED"
          }
  }
  ```

  Let us try and query the transformation catalog for all the entries in it. Let us see what our transformation catalog looks like

  ```
  $ pegasus-tc-client  -q -B

  tr bin/mDiff {
       site local {
            profile env "MONTAGE_HOME" "."
            pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mDiff"
            arch "x86"
            os "LINUX"
            type "STAGEABLE"
       }
  }

  tr bin/mFitplane {
       site local {
            pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mFitplane"
            arch "x86"
            os "LINUX"
            type "STAGEABLE"
       }
  }

  tr condor::dagman {
       site local {
            pfn "/usr/bin/condor_dagman"
            arch "x86"
            os "LINUX"
            type "INSTALLED"
       }
  }

  tr diamond::analyze:2.0 {
       site local {
            pfn "/opt/pegasus/default/bin/keg"
            arch "x86"
            os "LINUX"
            type "INSTALLED"
       }
  }

  tr diamond::findrange:2.0 {
       site local {
            pfn "/opt/pegasus/default/bin/keg"
            arch "x86"
            os "LINUX"
            type "INSTALLED"
       }
  }

  tr diamond::preprocess:2.0 {
       site local {
            pfn "/opt/pegasus/default/bin/keg"
            arch "x86"
            os "LINUX"
  ```
  ```

```
          type "INSTALLED"
     }
}

tr mAdd:3.0 {
     site local {
          pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mAdd"
          arch "x86"
          os "LINUX"
          type "STAGEABLE"
     }
}

tr mBackground:3.0 {
     site local {
          pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mBackground"
          arch "x86"
          os "LINUX"
          type "STAGEABLE"
     }
}

tr mBgModel:3.0 {
     site local {
          pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mBgModel"
          arch "x86"
          os "LINUX"
          type "STAGEABLE"
     }
}

tr mConcatFit:3.0 {
     site local {
          pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mConcatFit"
          arch "x86"
          os "LINUX"
          type "STAGEABLE"
     }
}

tr mDiffFit:3.0 {
     site local {
          profile env "MONTAGE_HOME" "."
          pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mDiffFit"
          arch "x86"
          os "LINUX"
          type "STAGEABLE"
     }
}

tr mImgtbl:3.0 {
     site local {
          pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mImgtbl"
          arch "x86"
          os "LINUX"
          type "STAGEABLE"
     }
}

tr mJPEG:3.0 {
     site local {
          pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mJPEG"
          arch "x86"
          os "LINUX"
          type "STAGEABLE"
     }
}

tr mProjectPP:3.0 {
     site local {
          profile condor "priority" "25"
          pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mProjectPP"
          arch "x86"
          os "LINUX"
          type "STAGEABLE"
     }
}
```

```
tr mShrink:3.0 {
     site local {
          pfn "gsiftp://pegasus-vm/scratch/tutorial/software/montage/3.0/x86/bin/mShrink"
          arch "x86"
          os "LINUX"
          type "STAGEABLE"
     }
}
```

Properties

Pegasus Workflow Planner is configured via the use of java properties. The instructors have provided a ready properties file at $HOME/.pegasusrc .

```
$ cat $HOME/.pegasusrc

#########################
# PEGASUS USER PROPERTIES
#########################

## SELECT THE REPLICAT CATALOG MODE AND URL
pegasus.catalog.replica = File
pegasus.catalog.replica.file = ${user.home}/pegasus-wms/config/rc.data


## SELECT THE SITE CATALOG MODE AND FILE
pegasus.catalog.site = XML3
pegasus.catalog.site.file = ${user.home}/pegasus-wms/config/sites.xml3


## SELECT THE TRANSFORMATION CATALOG MODE AND FILE
pegasus.catalog.transformation = Text
pegasus.catalog.transformation.file = ${user.home}/pegasus-wms/config/tc.data.text

## USE DAGMAN RETRY FEATURE FOR FAILURES
dagman.retry=2

## CHECK JOB EXIT CODES FOR FAILURE
dagman.post.scope=all

## STAGE ALL OUR EXECUTABLES OR USE INSTALLED ONES
pegasus.catalog.transformation.mapper = All

## WORK AND STORAGE DIR
pegasus.dir.storage = storage
pegasus.dir.exec = exec

#JOB CATEGORIES
dagman.projection.maxjobs 2
```

Planning and Running Workflows Locally

In this exercise we are going to run pegasus-plan to generate a executable workflow from the abstract workflow (diamond.dax). The Executable workflow generated, are condor submit files that are submitted locally using pegasus-run

The instructors have provided:

- A dax (diamond.dax) in the $HOME/pegasus-wms/dax directory.

You will need to write some things yourself, by following the instructions below:

- Run pegasus-plan to generate the condor submit files out of the dax.
- Run pegasus-run to submit the workflow locally.

Instructions:

- Let us run pegasus-plan on the diamond dax.

```
$ cd ~/pegasus-wms
$ pegasus-plan --dax `pwd`/dax/diamond.dax --force\
               --dir dags -s local -o local --nocleanup -v
```

The above command says that we need to plan the diamond dax locally. The condor submit files are to be generated in a directory structure whose base is dags. We also are requesting that no cleanup jobs be added as we require the intermediate data to be saved. Here is the output of pegasus-plan.

```
2011.08.04 12:06:11.390 PDT: [INFO] event.pegasus.parse.dax dax.id /home/tutorial/pegasus-wms/
dax/diamond.dax  - STARTED
2011.08.04 12:06:11.485 PDT: [INFO]  Generating Stampede Events for Abstract Workflow
2011.08.04 12:06:11.539 PDT: [INFO]  Generating Stampede Events for Abstract Workflow -DONE
2011.08.04 12:06:11.543 PDT: [INFO] event.pegasus.refinement dax.id blackdiamond_0  - STARTED
2011.08.04 12:06:11.567 PDT: [INFO] event.pegasus.siteselection dax.id blackdiamond_0  - STARTED
2011.08.04 12:06:11.596 PDT: [INFO] event.pegasus.siteselection dax.id blackdiamond_0  -
 FINISHED
2011.08.04 12:06:11.632 PDT: [INFO]  Grafting transfer nodes in the workflow
2011.08.04 12:06:11.632 PDT: [INFO] event.pegasus.generate.transfer-nodes dax.id blackdiamond_0
 - STARTED
2011.08.04 12:06:11.679 PDT: [INFO] event.pegasus.generate.transfer-nodes dax.id blackdiamond_0
 - FINISHED
2011.08.04 12:06:11.680 PDT: [INFO] event.pegasus.generate.workdir-nodes dax.id blackdiamond_0  -
 STARTED
2011.08.04 12:06:11.683 PDT: [INFO] event.pegasus.generate.workdir-nodes dax.id blackdiamond_0  -
 FINISHED
2011.08.04 12:06:11.684 PDT: [INFO] event.pegasus.generate.cleanup-wf dax.id blackdiamond_0  -
 STARTED
2011.08.04 12:06:11.685 PDT: [INFO] event.pegasus.generate.cleanup-wf dax.id blackdiamond_0  -
 FINISHED
2011.08.04 12:06:11.686 PDT: [INFO] event.pegasus.refinement dax.id blackdiamond_0  - FINISHED
2011.08.04 12:06:11.742 PDT: [INFO]  Generating codes for the concrete workflow
2011.08.04 12:06:11.831 PDT: [INFO]  Generating codes for the concrete workflow -DONE
2011.08.04 12:06:11.831 PDT: [INFO]  Generating code for the cleanup workflow
2011.08.04 12:06:11.892 PDT: [INFO]  Generating code for the cleanup workflow -DONE
2011.08.04 12:06:11.893 PDT:


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/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001


2011.08.04 12:06:11.893 PDT:   Time taken to execute is 0.902 seconds
2011.08.04 12:06:11.893 PDT: [INFO] event.pegasus.parse.dax dax.id /home/tutorial/pegasus-wms/
dax/diamond.dax  - FINISHED
```

- **Now run pegasus-run as mentioned in the output of pegasus-plan. Do not copy the command below it is just for illustration purpose.**

```
[pegasus@pegasus pegasus-wms]$ pegasus-run \
 /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001

------------------------------------------------------------------------
File for submitting this DAG to Condor           : blackdiamond-0.dag.condor.sub
Log of DAGMan debugging messages                 : blackdiamond-0.dag.dagman.out
Log of Condor library output                     : blackdiamond-0.dag.lib.out
Log of Condor library error messages             : blackdiamond-0.dag.lib.err
Log of the life of condor_dagman itself          : blackdiamond-0.dag.dagman.log

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

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

cd /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001

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

```
pegasus-status -l /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001

*** To remove your workflow run ***
pegasus-remove /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001
```

Monitoring, Debugging and Statistics

In this section, we are going to list ways to track your workflow, how to debug a failed workflow and how to generates statistics and plots for a workflow run.

Tracking the progress of the workflow and debugging the workflows.

We will change into the directory, that was mentioned by the output of pegasus-run command.

```
$ cd /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001
```

In this directory you will see a whole lot of files. That should not scare you. Unless things go wrong, you need to look at just a very few number of files to track the progress of the workflow

- **Run the command pegasus-status as mentioned by pegasus-run above to check the status of your jobs. Use the watch command to auto repeat the command every 2 seconds.**

  ```
  $ watch pegasus-status /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001

  STAT  IN_STATE  JOB
  Run      03:04  blackdiamond-0
  Run      01:00   ##findrange_j2
  Run      00:55   ##findrange_j3
  Summary: 3 Condor jobs total (R:3)

  UNREADY   READY     PRE  QUEUED    POST SUCCESS FAILURE %DONE
        3       0       0       3       0       3       0  33.3
  Summary: 1 DAG total (Running:1)
  ```

  ### Tip

  watch does not end with ESC nor (q)uit, but with Ctrl+C.

  The above output shows that a couple of jobs are running under the main dagman process. Keep a lookout to track whether a workflow is running or not. If you do not see any of your job in the output for sometime (say 30 seconds), we know the workflow has finished. We need to wait, as there might be delay in Condor DAGMan releasing the next job into the queue after a job has finished successfully.

  If output of pegasus-status is empty, then either your workflow has

  1. successfully completed
  2. stopped midway due to non recoverable error.

  We can now run pegasus-analyzer to analyze the workflow.

- Using **pegasus-analyzer** to analyze the workflow

  ```
  $ pegasus-analyzer  -i /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001

  pegasus-analyzer: initializing...

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

   Total jobs         :      8 (100.00%)
   # jobs succeeded   :      8 (100.00%)
   # jobs failed      :      0 (0.00%)
   # jobs unsubmitted :      0 (0.00%)

  ************************************Done**************************************

  pegasus-analyzer: end of status report
  ```

- Another way to monitor the workflow is to check the **jobstate.log** file. This is the output file of the monitoring daemon that is parsing all the condor log files to determine the status of the jobs. It logs the events seen by Condor into a more readable form for us.

```
$ more jobstate.log

1290676248 INTERNAL *** MONITORD_STARTED ***
1290676247 INTERNAL *** DAGMAN_STARTED 339.0 ***
...
```

  In the starting of the jobstate.log, when the workflow has just started running you will see a lot of entries with status UN_READY. That designates that DAGMan has just parsed in the .dag file and has not started working on any job as yet. Initially all the jobs in the workflow are listed as UN_READY. After sometime you will see entries in jobstate.log, that shows a job is being executed etc.

```
1290676261 create_dir_blackdiamond_0_local SUBMIT 340.0 local - 1
1290676266 create_dir_blackdiamond_0_local EXECUTE 340.0 local - 1
1290676266 create_dir_blackdiamond_0_local JOB_TERMINATED 340.0 local - 1
1290676266 create_dir_blackdiamond_0_local JOB_SUCCESS 0 local - 1
1290676266 create_dir_blackdiamond_0_local POST_SCRIPT_STARTED 340.0 local - 1
1290676271 create_dir_blackdiamond_0_local POST_SCRIPT_TERMINATED 340.0 local - 1
1290676271 create_dir_blackdiamond_0_local POST_SCRIPT_SUCCESS 0 local - 1
```

  The above shows the being submitted and then executed on the grid. In addition it lists that job is being run on the grid site local (which is your submit machine). The various states of the job while it goes through submission to execution to post processing are in UPPERCASE.

- Successfully Completed : Let us again look at the jobstate.log. This time we need to look at the last few lines of jobstate.log

```
$ tail jobstate.log

1290676542 register_local_2_0 SUBMIT 347.0 local - 8
1290676547 register_local_2_0 EXECUTE 347.0 local - 8
1290676547 register_local_2_0 JOB_TERMINATED 347.0 local - 8
1290676547 register_local_2_0 JOB_SUCCESS 0 local - 8
1290676547 register_local_2_0 POST_SCRIPT_STARTED 347.0 local - 8
1290676552 register_local_2_0 POST_SCRIPT_TERMINATED 347.0 local - 8
1290676552 register_local_2_0 POST_SCRIPT_SUCCESS 0 local - 8
1290676552 INTERNAL *** DAGMAN_FINISHED 0 ***
1290676554 INTERNAL *** MONITORD_FINISHED 0 ***
```

  Looking at the last two lines we see that DAGMan finished, and pegasus-monitord finished successfully with a status 0. This means workflow ran successfully. Congratulations you ran your workflow on the local site successfully. The workflow generates a final output file f.d that resides in the directory **/home/tutorial/local-storage/storage/f.d** .

  To view the file, you can do the following

```
$ cat /home/tutorial/local-storage/storage/f.d

--- start f.c1 ----
  --- start f.b1 ----
    --- start f.a ----
      Input File for the Diamond Workflow.--- final f.a ----
    Timestamp Today: 20101223T105659.955-08:00 (1293130619.955;60.002)
    Applicationname: preprocess @ 10.0.2.15 (VPN)
    Current Workdir: /home/tutorial/local-scratch/exec/tutorial/pegasus/blackdiamond/run0001
    Systemenvironm.: i686-Linux 2.6.32-5-686
    Processor Info.: 1 x Intel(R) Xeon(R) CPU           E5462  @ 2.80GHz @ 2797.463
    Load Averages  : 0.646 0.192 0.060
    Memory Usage MB: 502 total, 229 free, 0 shared, 39 buffered
    Swap Usage   MB: 397 total, 397 free
    Filesystem Info: /media/cdrom0            udf,iso9660    31MB total,     0B avail
    Filesystem Info: /media/floppy0           auto  7668MB total,  5436MB avail
    Output Filename: f.b1
    Input Filenames: f.a
  --- final f.b1 ----
  Timestamp Today: 20101223T105815.334-08:00 (1293130695.334;60.003)
  Applicationname: findrange @ 10.0.2.15 (VPN)
```

```
  Current Workdir: /home/tutorial/local-scratch/exec/tutorial/pegasus/blackdiamond/run0001
  Systemenvironm.: i686-Linux 2.6.32-5-686
  Processor Info.: 1 x Intel(R) Xeon(R) CPU           E5462  @ 2.80GHz @ 2797.463
  Load Averages  : 1.444 0.509 0.177
  Memory Usage MB: 502 total, 227 free, 0 shared, 39 buffered
  Swap Usage   MB: 397 total, 397 free
  Filesystem Info: /media/cdrom0            udf,iso9660    31MB total,     0B avail
  Filesystem Info: /media/floppy0           auto  7668MB total,  5436MB avail
  Output Filename: f.c1
  Input Filenames: f.b1
--- final f.c1 ----
--- start f.c2 ----
  --- start f.b2 ----
    --- start f.a ----
      Input File for the Diamond Workflow.--- final f.a ----
    Timestamp Today: 20101223T105659.955-08:00 (1293130619.955;60.003)
    Applicationname: preprocess @ 10.0.2.15 (VPN)
    Current Workdir: /home/tutorial/local-scratch/exec/tutorial/pegasus/blackdiamond/run0001
    Systemenvironm.: i686-Linux 2.6.32-5-686
    Processor Info.: 1 x Intel(R) Xeon(R) CPU           E5462  @ 2.80GHz @ 2797.463
    Load Averages  : 0.646 0.192 0.060
    Memory Usage MB: 502 total, 229 free, 0 shared, 39 buffered
    Swap Usage   MB: 397 total, 397 free
    Filesystem Info: /media/cdrom0            udf,iso9660    31MB total,     0B avail
    Filesystem Info: /media/floppy0           auto  7668MB total,  5436MB avail
    Output Filename: f.b2
    Input Filenames: f.a
  --- final f.b2 ----
  Timestamp Today: 20101223T105820.478-08:00 (1293130700.478;60.001)
  Applicationname: findrange @ 10.0.2.15 (VPN)
  Current Workdir: /home/tutorial/local-scratch/exec/tutorial/pegasus/blackdiamond/run0001
  Systemenvironm.: i686-Linux 2.6.32-5-686
  Processor Info.: 1 x Intel(R) Xeon(R) CPU           E5462  @ 2.80GHz @ 2797.463
  Load Averages  : 1.409 0.517 0.182
  Memory Usage MB: 502 total, 228 free, 0 shared, 39 buffered
  Swap Usage   MB: 397 total, 397 free
  Filesystem Info: /media/cdrom0            udf,iso9660    31MB total,     0B avail
  Filesystem Info: /media/floppy0           auto  7668MB total,  5436MB avail
  Output Filename: f.c2
  Input Filenames: f.b2
--- final f.c2 ----
Timestamp Today: 20101223T105936.718-08:00 (1293130776.718;60.000)
Applicationname: analyze @ 10.0.2.15 (VPN)
Current Workdir: /home/tutorial/local-scratch/exec/tutorial/pegasus/blackdiamond/run0001
Systemenvironm.: i686-Linux 2.6.32-5-686
Processor Info.: 1 x Intel(R) Xeon(R) CPU           E5462  @ 2.80GHz @ 2797.463
Load Averages  : 1.033 0.581 0.226
Memory Usage MB: 502 total, 228 free, 0 shared, 40 buffered
Swap Usage   MB: 397 total, 397 free
Filesystem Info: /media/cdrom0            udf,iso9660    31MB total,     0B avail
Filesystem Info: /media/floppy0           auto  7668MB total,  5436MB avail
Output Filename: f.d
Input Filenames: f.c1 f.c2
```

- Unsuccessfully Completed (Workflow execution stopped midway) : Let us again look at the jobstate.log. Again we need to look at the last few lines of jobstate.log

  $ tail jobstate.log

  ```
  1290677127 stage_in_local_local_0 EXECUTE 352.0 local - 4
  1290677127 stage_in_local_local_0 JOB_TERMINATED 352.0 local - 4
  1290677127 stage_in_local_local_0 JOB_FAILURE 1 local - 4
  1290677127 stage_in_local_local_0 POST_SCRIPT_STARTED 352.0 local - 4
  1290677132 stage_in_local_local_0 POST_SCRIPT_TERMINATED 352.0 local - 4
  1290677132 stage_in_local_local_0 POST_SCRIPT_FAILURE 1 local - 4
  1290677132 INTERNAL *** DAGMAN_FINISHED 1 ***
  1290677134 INTERNAL *** MONITORD_FINISHED 0 ***
  ```

  Looking at the last two lines we see that DAGMan finished, and pegasus-monitord finished unsuccessfully with a status 1. We can easily determine which job failed. It is stage_in_local_local_0 in this case. To determine the reason for failure we need to look at it's kickstart output file which is JOBNAME.out.NNN. where NNN is 000 - NNN

Debugging a failed workflow using pegasus-analyzer

In this section, we will run the diamond workflow but remove the input file so that the workflow fails during execution. This is to highlight how to use pegasus-analyzer to debug a failed workflow.

First of all lets rename the input file f.a

```
 $ mv /scratch/tutorial/inputdata/diamond/f.a /scratch/tutorial/inputdata/diamond/f.a.old

 $ cd $HOME/pegasus-wms
```

We will now repeat exercise **2.4 and 2.5** and submit the workflow again.

Plan and Submit the diamond workflow . Pass --submit to pegasus-plan to submit in case of successful planning

```
$  pegasus-plan --dax `pwd`/dax/diamond.dax --force \
        --dir dags -s local -o local --nocleanup --submit -v
```

Use pegasus-status to track the workflow and wait it to fail

```
$ watch pegasus-status  /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0002

STAT  IN_STATE  JOB
Run      00:18  blackdiamond-0
Summary: 1 Condor job total (R:1)

UNREADY   READY     PRE  QUEUED    POST SUCCESS FAILURE %DONE
      7       0       0       2       0       0       0   0.0
Summary: 1 DAG total (Running:1)
```

The --long option to pegasus-status of a running workflow gives more detail
[pegasus@pegasus pegasus-wms]$ pegasus-status -l /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0002

```
STAT  IN_STATE  JOB
Run      00:38  blackdiamond-0
Run      00:08   ##stage_in_local_local_0
Summary: 2 Condor jobs total (R:2)

UNRDY READY   PRE  IN_Q  POST  DONE  FAIL %DONE STATE   DAGNAME
    6     0     0     2     0     1     0  11.1 Running *blackdiamond-0.dag
Summary: 1 DAG total (Running:1)
```

We can also use --long option to pegasus-status to see the FINAL status of the workflow

```
$ pegasus-status -l /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0002

(no matching jobs found in Condor Q)
UNRDY READY   PRE  IN_Q  POST  DONE  FAIL %DONE STATE   DAGNAME
    6     0     0     0     0     1     2  11.1 Failure *blackdiamond-0.dag
Summary: 1 DAG total (Failure:1)
```

We will now run pegasus-analyzer on the failed workflow submit directory to see what job failed.

```
$ pegasus-analyzer -i $HOME/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0002
pegasus-analyzer: initializing...

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

 Total jobs         :      8 (100.00%)
 # jobs succeeded   :      1 (12.50%)
 # jobs failed      :      1 (12.50%)
 # jobs unsubmitted :      6 (75.00%)

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

============================stage_in_local_local_0============================

 last state: POST_SCRIPT_FAILURE
```

```
       site: local
submit file: /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0002/
stage_in_local_local_0.sub
output file: /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0002/
stage_in_local_local_0.out.002
 error file: /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0002/
stage_in_local_local_0.err.002

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

site        : local
hostname    : pegasus-vm.local
executable  : /opt/pegasus/3.1.0/bin/pegasus-transfer
arguments   :
exitcode    : 1
working dir : /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0002

--------------Task #1 - pegasus::pegasus-transfer - null - stdout---------------

2011-07-29 13:02:07,189    INFO:  Reading URL pairs from stdin
2011-07-29 13:02:07,190    INFO:  PATH=/opt/globus/default/bin:/opt/pegasus/default/bin:/usr/bin:/
bin
2011-07-29 13:02:07,190    INFO:  LD_LIBRARY_PATH=/opt/globus/default/lib:/usr/lib/jvm/java-6-
openjdk/jre/lib/i386/client:/usr/lib/jvm/java-6-openjdk/jre/lib/i386:/usr/lib/jvm/java-6-openjdk/
jre/../lib/i386
2011-07-29 13:02:07,225    INFO:    wget               Version: 1.12    Path: /usr/bin/wget
2011-07-29 13:02:07,253    INFO:    globus-version     Version: 5.0.2   Path: /opt/globus/default/
bin/globus-version
2011-07-29 13:02:07,301    INFO:    globus-url-copy    Version: 5.7     Path: /opt/globus/default/
bin/globus-url-copy
2011-07-29 13:02:07,316    INFO:  Command'srm-copy'not found in the current environment
2011-07-29 13:02:07,331    INFO:  Command'iget'not found in the current environment
2011-07-29 13:02:07,346    INFO:    pegasus-s3         Version: N/A     Path: /opt/pegasus/default/
bin/pegasus-s3
2011-07-29 13:02:07,348    INFO:  Sorting the tranfers based on transfer type and source/destination
2011-07-29 13:02:07,349    INFO:
 ----------------------------------------------------------------------
2011-07-29 13:02:07,349    INFO:  Starting transfers - attempt 1
2011-07-29 13:02:07,349    INFO:  /bin/cp -f -L"/scratch/tutorial/inputdata/diamond/f.a""/home/
tutorial/local-scratch/exec/tutorial/pegasus/blackdiamond/run0002/f.a"
/bin/cp: cannot stat `/scratch/tutorial/inputdata/diamond/f.a': No such file or directory
2011-07-29 13:02:07,363   ERROR:  Command'/bin/cp -f -L"/scratch/tutorial/inputdata/diamond/f.a""/
home/tutorial/local-scratch/exec/tutorial/pegasus/blackdiamond/run0002/f.a"'failed with error code 1
2011-07-29 13:02:17,417    INFO:
 ----------------------------------------------------------------------
2011-07-29 13:02:17,420    INFO:  Starting transfers - attempt 2
2011-07-29 13:02:17,422    INFO:  /bin/cp -f -L"/scratch/tutorial/inputdata/diamond/f.a""/home/
tutorial/local-scratch/exec/tutorial/pegasus/blackdiamond/run0002/f.a"
/bin/cp: cannot stat `/scratch/tutorial/inputdata/diamond/f.a': No such file or directory
2011-07-29 13:02:17,438   ERROR:  Command'/bin/cp -f -L"/scratch/tutorial/inputdata/diamond/f.a""/
home/tutorial/local-scratch/exec/tutorial/pegasus/blackdiamond/run0002/f.a"'failed with error code 1
2011-07-29 13:02:17,438    INFO:  Stats: no local files in the transfer set
2011-07-29 13:02:17,438 CRITICAL:  Some transfers failed! See above.

************************************Done*************************************

pegasus-analyzer: end of status report

[pegasus@pegasus pegasus-wms]$
```

The above tells us that the stage-in job for the workflow failed, and points us to the stdout of the job. By default, all jobs in Pegasus are launched via kickstart that captures runtime provenance of the job and helps in debugging. Hence, the stdout of the job is the kickstart stdout which is in XML.

. the duration of the job the start time for the job the node on which the job ran the stdout/stderr of the job the arguments with which it launched the job the environment that was set for the job before it was launched. 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

Kickstart and Condor DAGMan format and log files

This section explains how to read kickstart output and DAGMan Condor log files.

Kickstart

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 it's stdout. The stdout appears in the workflow submit directory as <job>.out.00n . Some useful information captured by kickstart and logged are as follows

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 directory in which the job ran
6. the stdout/stderr of the job
7. the arguments with which it launched the job
8. the environment that was set for the job before it was launched.
9. the machine information about the node that the job ran on

Reading a Kickstart Output File

Lets look at the stdout of our failed job.

```
$ cat /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0002/
stage_in_local_local_0.out.002

 <?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-2.1.xsd" version="2.1"
  start="2010-11-29T19:10:23.862-08:00" duration="0.076" transformation="pegasus::pegasus-transfer"
  derivation="pegasus::pegasus-transfer:1.0" resource="local" wf-label="blackdiamond"
  wf-stamp="2010-11-29T18:57:59-08:00" interface="eth0" hostaddr="10.0.2.15" hostname="pegasus-
vm.local"
  pid="5428" uid="501" user="pegasus" gid="501" group="pegasus" umask="0022">

 <mainjob start="2010-11-29T19:10:23.876-08:00" duration="0.063" pid="5429">
    <usage utime="0.040" stime="0.023" minflt="2758" majflt="0" nswap="0" nsignals="0" nvcsw="5"
 nivcsw="20"/>
    <status raw="256"><regular exitcode="1"/></status>
    <statcall error="0">
      <file name="/opt/pegasus/default/bin/pegasus-transfer">23212F7573722F62696E2F656E762070</file>
      <statinfo mode="0100775" size="25314" inode="2022205" nlink="1" blksize="4096" blocks="64"
              mtime="2010-11-23T13:14:52-08:00"
            atime="2010-11-29T19:10:07-08:00" ctime="2010-11-25T00:01:52-08:00" uid="501"
 user="pegasus"
              gid="501" group="pegasus"/>
    </statcall>
    <argument-vector/>
 </mainjob>
 <cwd>/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0002</cwd>
 <usage utime="0.002" stime="0.013" minflt="475" majflt="0" nswap="0" nsignals="0" nvcsw="1"
 nivcsw="5"/>


   <machine page-size="4096">
```

```
    <stamp>2010-12-23T10:56:43.817-08:00</stamp>
    <uname system="linux" nodename="pegasus-vm" release="2.6.32-5-686" machine="i686">
      #1 SMP Fri Dec 10 16:12:40 UTC 2010</uname>
   <linux>
    <ram total="527044608" free="242290688" shared="0" buffer="41041920"/>
    <swap total="417325056" free="417325056"/>
    <boot idle="1597.500">2010-12-23T10:29:16.599-08:00</boot>
    <cpu count="1" speed="2797" vendor="GenuineIntel">Intel(R) Xeon(R) CPU E5462 @ 2.80GHz</cpu>
    <load min1="0.05" min5="0.02" min15="0.00"/>
    <proc total="88" running="1" sleeping="87" vmsize="344793088" rss="123768832"/>
    <task total="101" running="1" sleeping="100"/>
   </linux>
  </machine>


  <statcall error="0" id="stdin">
    <descriptor number="0"/>
    <statinfo mode="0100664" size="142" inode="2250032" nlink="1" blksize="4096" blocks="16"
     mtime="2010-11-29T19:09:20-08:00"   atime="2010-11-29T19:10:07-08:00"
 ctime="2010-11-29T19:09:20-08:00"
     uid="501" user="pegasus" gid="501" group="pegasus"/>
  </statcall>

  <statcall error="0" id="stdout">
    <temporary name="/opt/condor/local.pegasus/spool/local_univ_execute/dir_5427/gs.out.awOX6p"
 descriptor="3"/>
    <statinfo mode="0100600" size="762" inode="2054511" nlink="1" blksize="4096" blocks="16"
          mtime="2010-11-29T19:10:23-08:00" atime="2010-11-29T19:10:23-08:00"
 ctime="2010-11-29T19:10:23-08:00"
            uid="501" user="pegasus" gid="501" group="pegasus"/>
    <data>2010-11-29 19:10:23,920    INFO:  Reading URL pairs from stdin
2010-11-29 19:10:23,921    INFO:  PATH=/usr/local/globus/default/bin:/opt/pegasus/default/bin:/usr/
bin:/bin
2010-11-29 19:10:23,921    INFO:  LD_LIBRARY_PATH=/usr/local/globus/default/lib:/usr/java/
jdk1.6.0_20/jre/lib/amd64/
2010-11-29 19:10:23,921    INFO:  Executing cp commands
/bin/cp: cannot stat `/scratch/tutorial/inputdata/diamond/f.a&apos;: No such file or directory
2010-11-29 19:10:23,932 CRITICAL:  Command &apos;/bin/cp -L &quot;/scratch/tutorial/inputdata/
diamond/f.a&quot;
    &quot;/home/tutorial/local-scratch/exec/pegasus/pegasus/blackdiamond/run0002/f.a&quot;&apos;
 failed with error code 1
</data>
  </statcall>

  <statcall error="0" id="stderr">
    <temporary name="/opt/condor/local.pegasus/spool/local_univ_execute/dir_5427/gs.err.oz9MOG"
 descriptor="4"/>
    <statinfo mode="0100600" size="0" inode="2054512" nlink="1" blksize="4096" blocks="8"
    mtime="2010-11-29T19:10:23-08:00"  atime="2010-11-29T19:10:23-08:00"
 ctime="2010-11-29T19:10:23-08:00"
    uid="501" user="pegasus" gid="501" group="pegasus"/>
  </statcall>

  <statcall error="2" id="gridstart">
    <!-- ignore above error -->
    <file name="condor_exec.exe"/>
  </statcall>
  <statcall error="0" id="logfile">
    <descriptor number="1"/>
    <statinfo mode="0100644" size="0" inode="2250072" nlink="1" blksize="4096" blocks="8"
 mtime="2010-11-29T19:10:23-08:00"
    atime="2010-11-29T19:10:23-08:00" ctime="2010-11-29T19:10:23-08:00" uid="501" user="pegasus"
 gid="501" group="pegasus"/>
  </statcall>
  <statcall error="0" id="channel">
    <fifo name="/opt/condor/local.pegasus/spool/local_univ_execute/dir_5427/gs.app.qCOCwX"
 descriptor="5" count="0"
     rsize="0" wsize="0"/>
    <statinfo mode="010640" size="0" inode="2054524" nlink="1" blksize="4096" blocks="8"
 mtime="2010-11-29T19:10:23-08:00"
     atime="2010-11-29T19:10:23-08:00" ctime="2010-11-29T19:10:23-08:00" uid="501" user="pegasus"
 gid="501"
    group="pegasus"/>
  </statcall>
  <environment>
    <env key="GLOBUS_LOCATION">/usr/local/globus/default</env>
```

```
    <env key="GRIDSTART_CHANNEL">/opt/condor/local.pegasus/spool/local_univ_execute/dir_5427/
gs.app.qCOCwX</env>
    <env key="JAVA_HOME">/usr</env>
    <env key="LD_LIBRARY_PATH">/usr/java/jdk1.6.0_20/jre/lib/amd64/server:/usr/java/jdk1.6.0_20/jre/
lib/amd64:</env>
    <env key="PEGASUS_HOME">/opt/pegasus/default</env>
    <env key="TEMP">/opt/condor/local.pegasus/spool/local_univ_execute/dir_5427</env>
    <env key="TMP">/opt/condor/local.pegasus/spool/local_univ_execute/dir_5427</env>
    <env key="TMPDIR">/opt/condor/local.pegasus/spool/local_univ_execute/dir_5427</env>
    <env key="_CONDOR_ANCESTOR_4843">4862:1291085504:2790807554</env>
    <env key="_CONDOR_ANCESTOR_4862">5427:1291086623:1798288782</env>
    <env key="_CONDOR_ANCESTOR_5427">5428:1291086623:2750667008</env>
    <env key="_CONDOR_HIGHPORT">41000</env>
    <env key="_CONDOR_JOB_AD">/opt/condor/local.pegasus/spool/local_univ_execute/dir_5427/.job.ad</
env>
    <env key="_CONDOR_LOWPORT">40000</env>
    <env key="_CONDOR_MACHINE_AD">/opt/condor/local.pegasus/spool/local_univ_execute/
dir_5427/.machine.ad</env>
    <env key="_CONDOR_SCRATCH_DIR">/opt/condor/local.pegasus/spool/local_univ_execute/dir_5427</env>
    <env key="_CONDOR_SLOT">1</env>
  </environment>
  <resource>
    <soft id="RLIMIT_CPU">unlimited</soft>
    <hard id="RLIMIT_CPU">unlimited</hard>
    <soft id="RLIMIT_FSIZE">unlimited</soft>
    <hard id="RLIMIT_FSIZE">unlimited</hard>
    <soft id="RLIMIT_DATA">unlimited</soft>
    <hard id="RLIMIT_DATA">unlimited</hard>
    <soft id="RLIMIT_STACK">unlimited</soft>
    <hard id="RLIMIT_STACK">unlimited</hard>
    <soft id="RLIMIT_CORE">0</soft>
    <hard id="RLIMIT_CORE">0</hard>
    <soft id="RESOURCE_5">unlimited</soft>
    <hard id="RESOURCE_5">unlimited</hard>
    <soft id="RLIMIT_NPROC">unlimited</soft>
    <hard id="RLIMIT_NPROC">unlimited</hard>
    <soft id="RLIMIT_NOFILE">1024</soft>
    <hard id="RLIMIT_NOFILE">1024</hard>
    <soft id="RLIMIT_MEMLOCK">32768</soft>
    <hard id="RLIMIT_MEMLOCK">32768</hard>
    <soft id="RLIMIT_AS">unlimited</soft>
    <hard id="RLIMIT_AS">unlimited</hard>
    <soft id="RLIMIT_LOCKS">unlimited</soft>
    <hard id="RLIMIT_LOCKS">unlimited</hard>
    <soft id="RLIMIT_SIGPENDING">8192</soft>
    <hard id="RLIMIT_SIGPENDING">8192</hard>
    <soft id="RLIMIT_MSGQUEUE">819200</soft>
    <hard id="RLIMIT_MSGQUEUE">819200</hard>
    <soft id="RLIMIT_NICE">0</soft>
    <hard id="RLIMIT_NICE">0</hard>
    <soft id="RLIMIT_RTPRIO">0</soft>
    <hard id="RLIMIT_RTPRIO">0</hard>
  </resource>
</invocation>
```

Condor DAGMan format and log files etc.

In this exercise we will learn about the DAG file format and some of the log files generated when the DAG runs.

- Now take a look at the DAG file...

```
$ cat $HOME/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/blackdiamond-0.dag

##########################################################################
# PEGASUS WMS GENERATED DAG FILE
# DAG blackdiamond
# Index = 0, Count = 1
##########################################################################
MAXJOBS projection 2

JOB create_dir_blackdiamond_0_local create_dir_blackdiamond_0_local.sub
SCRIPT POST create_dir_blackdiamond_0_local /opt/pegasus/default/bin/pegasus-exitcode
  /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/
create_dir_blackdiamond_0_local.out
RETRY create_dir_blackdiamond_0_local 2
```

```
JOB stage_in_local_local_0 stage_in_local_local_0.sub
SCRIPT POST stage_in_local_local_0 /opt/pegasus/default/bin/pegasus-exitcode
 /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/stage_in_local_local_0.out
RETRY stage_in_local_local_0 2

JOB preprocess_j1 preprocess_j1.sub
SCRIPT POST preprocess_j1 /opt/pegasus/default/bin/pegasus-exitcode
/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/preprocess_j1.out
RETRY preprocess_j1 2

JOB findrange_j2 findrange_j2.sub
SCRIPT POST findrange_j2 /opt/pegasus/default/bin/pegasus-exitcode
/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/findrange_j2.out
RETRY findrange_j2 2

JOB findrange_j3 findrange_j3.sub
SCRIPT POST findrange_j3 /opt/pegasus/default/bin/pegasus-exitcode
/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/findrange_j3.out
RETRY findrange_j3 2

JOB analyze_j4 analyze_j4.sub
SCRIPT POST analyze_j4 /opt/pegasus/default/bin/pegasus-exitcode
/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/analyze_j4.out
RETRY analyze_j4 2

JOB stage_out_local_local_2_0 stage_out_local_local_2_0.sub
SCRIPT POST stage_out_local_local_2_0 /opt/pegasus/default/bin/pegasus-exitcode
/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/
stage_out_local_local_2_0.out
RETRY stage_out_local_local_2_0 2

JOB register_local_2_0 register_local_2_0.sub
SCRIPT POST register_local_2_0 /opt/pegasus/default/bin/pegasus-exitcode
/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/register_local_2_0.out
RETRY register_local_2_0 2

PARENT findrange_j2 CHILD analyze_j4
PARENT preprocess_j1 CHILD findrange_j2
PARENT preprocess_j1 CHILD findrange_j3
PARENT findrange_j3 CHILD analyze_j4
PARENT analyze_j4 CHILD stage_out_local_local_2_0
PARENT stage_in_local_local_0 CHILD preprocess_j1
PARENT stage_out_local_local_2_0 CHILD register_local_2_0
PARENT create_dir_blackdiamond_0_local CHILD analyze_j4
PARENT create_dir_blackdiamond_0_local CHILD findrange_j2
PARENT create_dir_blackdiamond_0_local CHILD preprocess_j1
PARENT create_dir_blackdiamond_0_local CHILD findrange_j3
PARENT create_dir_blackdiamond_0_local CHILD stage_in_local_local_0
#####################################################################
# End of DAG
#####################################################################
```

- ... and the dagman.out file.

```
$ cat $HOME/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/blackdiamond-0.dag.dagman.out

11/25 01:10:47 ******************************************************
11/25 01:10:47 ** condor_scheduniv_exec.339.0 (CONDOR_DAGMAN) STARTING UP
11/25 01:10:47 ** /opt/condor/7.4.2/bin/condor_dagman
11/25 01:10:47 ** SubsystemInfo: name=DAGMAN type=DAGMAN(10) class=DAEMON(1)
11/25 01:10:47 ** Configuration: subsystem:DAGMAN local:<NONE> class:DAEMON
11/25 01:10:47 ** $CondorVersion: 7.4.2 Mar 29 2010 BuildID: 227044 $
11/25 01:10:47 ** $CondorPlatform: X86_64-LINUX_RHEL5 $
11/25 01:10:47 ** PID = 7844
11/25 01:10:47 ** Log last touched time unavailable (No such file or directory)
11/25 01:10:47 ******************************************************
11/25 01:10:47 Using config source: /opt/condor/config/condor_config
11/25 01:10:47 Using local config sources:
11/25 01:10:47    /opt/condor/config/condor_config.local
11/25 01:10:47 DaemonCore: Command Socket at <172.16.80.129:40035>
11/25 01:10:47 DAGMAN_DEBUG_CACHE_SIZE setting: 5242880
11/25 01:10:47 DAGMAN_DEBUG_CACHE_ENABLE setting: False
11/25 01:10:47 DAGMAN_SUBMIT_DELAY setting: 0
11/25 01:10:47 DAGMAN_MAX_SUBMIT_ATTEMPTS setting: 6
11/25 01:10:47 DAGMAN_STARTUP_CYCLE_DETECT setting: 0
11/25 01:10:47 DAGMAN_MAX_SUBMITS_PER_INTERVAL setting: 5
```

```
11/25 01:10:47 DAGMAN_USER_LOG_SCAN_INTERVAL setting: 5
11/25 01:10:47 allow_events (DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION, DAGMAN_ALLOW_EVENTS) setting:
 114
11/25 01:10:47 DAGMAN_RETRY_SUBMIT_FIRST setting: 1
11/25 01:10:47 DAGMAN_RETRY_NODE_FIRST setting: 0
11/25 01:10:47 DAGMAN_MAX_JOBS_IDLE setting: 0
11/25 01:10:47 DAGMAN_MAX_JOBS_SUBMITTED setting: 0
11/25 01:10:47 DAGMAN_MUNGE_NODE_NAMES setting: 1
11/25 01:10:47 DAGMAN_PROHIBIT_MULTI_JOBS setting: 0
11/25 01:10:47 DAGMAN_SUBMIT_DEPTH_FIRST setting: 0
11/25 01:10:47 DAGMAN_ABORT_DUPLICATES setting: 1
11/25 01:10:47 DAGMAN_ABORT_ON_SCARY_SUBMIT setting: 1
11/25 01:10:47 DAGMAN_PENDING_REPORT_INTERVAL setting: 600
11/25 01:10:47 DAGMAN_AUTO_RESCUE setting: 1
11/25 01:10:47 DAGMAN_MAX_RESCUE_NUM setting: 100
11/25 01:10:47 DAGMAN_DEFAULT_NODE_LOG setting: null
11/25 01:10:47 ALL_DEBUG setting:
11/25 01:10:47 DAGMAN_DEBUG setting:
....
11/25 01:10:47 Default node log file is:
 </home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/
blackdiamond-0.dag.nodes.log>
11/25 01:10:47 DAG Lockfile will be written to blackdiamond-0.dag.lock
11/25 01:10:47 DAG Input file is blackdiamond-0.dag
11/25 01:10:47 Parsing 1 dagfiles
11/25 01:10:47 Parsing blackdiamond-0.dag ...
11/25 01:10:47 Dag contains 8 total jobs
11/25 01:10:47 Sleeping for 12 seconds to ensure ProcessId uniqueness
11/25 01:10:59 Bootstrapping...
11/25 01:10:59 Number of pre-completed nodes: 0
11/25 01:10:59 Registering condor_event_timer...
11/25 01:11:00 Sleeping for one second for log file consistency
11/25 01:11:01 Submitting Condor Node create_dir_blackdiamond_0_local job(s)...
11/25 01:11:01 submitting: condor_submit -a dag_node_name' '=' 'create_dir_blackdiamond_0_local -
a
+DAGManJobId' '=' '339 -a DAGManJobId' '=' '339 -a submit_event_notes' '=' 'DAG' 'Node:' '
create_dir_blackdiamond_0_local -a +DAGParentNodeNames' '=' '""
 create_dir_blackdiamond_0_local.sub
11/25 01:11:01 From submit: Submitting job(s).
11/25 01:11:01 From submit: Logging submit event(s).
11/25 01:11:01 From submit: 1 job(s) submitted to cluster 340.
11/25 01:11:01  assigned Condor ID (340.0)
11/25 01:11:01 Just submitted 1 job this cycle...
11/25 01:11:01 Currently monitoring 1 Condor log file(s)
11/25 01:11:01 Event: ULOG_SUBMIT for Condor Node create_dir_blackdiamond_0_local (340.0)
11/25 01:11:01 Number of idle job procs: 1
11/25 01:11:01 Of 8 nodes total:
11/25 01:11:01  Done     Pre   Queued    Post   Ready   Un-Ready   Failed
11/25 01:11:01   ===     ===      ===     ===     ===        ===      ===
11/25 01:11:01     0       0        1       0       0          7        0
....
11/25 01:11:06 Currently monitoring 1 Condor log file(s)
11/25 01:11:06 Event: ULOG_EXECUTE for Condor Node create_dir_blackdiamond_0_local (340.0)
11/25 01:11:06 Number of idle job procs: 0
11/25 01:11:06 Event: ULOG_JOB_TERMINATED for Condor Node create_dir_blackdiamond_0_local (340.0)
11/25 01:11:06 Node create_dir_blackdiamond_0_local job proc (340.0) completed successfully.
11/25 01:11:06 Node create_dir_blackdiamond_0_local job completed
11/25 01:11:06 Running POST script of Node create_dir_blackdiamond_0_local...
11/25 01:11:06 Number of idle job procs: 0
11/25 01:11:06 Of 8 nodes total:
11/25 01:11:06  Done     Pre   Queued    Post   Ready   Un-Ready   Failed
11/25 01:11:06   ===     ===      ===     ===     ===        ===      ===
11/25 01:11:06     0       0        0       1       0          7        0
11/25 01:11:11 Currently monitoring 1 Condor log file(s)
11/25 01:11:11 Event: ULOG_POST_SCRIPT_TERMINATED for Condor Node create_dir_blackdiamond_0_local
 (340.0)
11/25 01:11:11 POST Script of Node create_dir_blackdiamond_0_local completed successfully.
11/25 01:11:11 Of 8 nodes total:
11/25 01:11:11  Done     Pre   Queued    Post   Ready   Un-Ready   Failed
11/25 01:11:11   ===     ===      ===     ===     ===        ===      ===
11/25 01:11:11     1       0        0       0       1          6        0
....
11/25 01:15:52 Event: ULOG_POST_SCRIPT_TERMINATED for Condor Node register_local_2_0 (347.0)
11/25 01:15:52 POST Script of Node register_local_2_0 completed successfully.
11/25 01:15:52 Of 8 nodes total:
11/25 01:15:52  Done     Pre   Queued    Post   Ready   Un-Ready   Failed
11/25 01:15:52   ===     ===      ===     ===     ===        ===      ===
```

```
11/25 01:15:52     8       0       0       0       0          0       0
11/25 01:15:52 All jobs Completed!
11/25 01:15:52 Note: 0 total job deferrals because of -MaxJobs limit (0)
11/25 01:15:52 Note: 0 total job deferrals because of -MaxIdle limit (0)
11/25 01:15:52 Note: 0 total job deferrals because of node category throttles
11/25 01:15:52 Note: 0 total PRE script deferrals because of -MaxPre limit (20)
11/25 01:15:52 Note: 0 total POST script deferrals because of -MaxPost limit (20)
11/25 01:15:52 **** condor_scheduniv_exec.339.0 (condor_DAGMAN) pid 7844 EXITING WITH STATUS 0
[p
```

Removing a running workflow

Sometimes you may want to halt the execution of the workflow or just permanently remove it. You can stop/halt a workflow by running the pegasus-remove command mentioned in the output of pegasus-run

```
$ pegasus-remove $HOME/pegasus-wms/dags/tutorial/pegasus/diamond/runXXXX

Job 2788.0 marked for removal
```

Generating statistics and plots of a workflow run

In this section, we will generate statistics and plots of the diamond workflow we ran using pegasus-statistics and pegasus-plots

Generating Statistics Using pegasus-statistics

pegasus-statistics generates workflow execution statistics. To generate statistics run the command as shown below

```
$ cd $HOME/pegasus-wms

$ pegasus-statistics -s all dags/tutorial/pegasus/blackdiamond/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 (Retries Included)
Tasks              4                  0                  0                  4
 ||   0                  4
Jobs               8                  0                  0                  8
 ||   0                  8
Sub Workflows      0                  0                  0                  0
 ||   0                  0
------------------------------------------------------------------------------------------------------------------------

Workflow wall time                               : 5 mins, 6 secs,      (total 306 seconds)

Workflow cumulative job wall time                : 4 mins, 0 secs,      (total 240 seconds)

Cumulative job walltime as seen from submit side : 4 mins, 2 secs,      (total 242 seconds)

Summary                          : /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/
run0001/statistics/summary.txt

Workflow execution statistics    : /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/
run0001/statistics/workflow.txt

Job instance statistics          : /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/
run0001/statistics/jobs.txt

Transformation statistics        : /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/
run0001/statistics/breakdown.txt

Time statistics                  : /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/
run0001/statistics/time.txt

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

Workflow statistics table

Workflow statistics table contains information about the workflow run like total execution time, job's failed etc.

Table 12.1. Table Workflow Statistics

Workflow runtime	5 min. 5 sec.
Cumulative workflow runtime	4 min. 0 sec.
Total jobs	8
# jobs succeeded	8
# jobs failed	0
# jobs unsubmitted	0
# jobs unknown	0

Job statistics table

Job statistics table contains the following details about the jobs in the workflow. A sample table is shown below.

- Job - the name of the job
- Site - the site where the job ran
- Kickstart(sec.) - the actual duration of the job in seconds on the remote compute node. In case of retries the value is the cumulative of all retries.
- Post(sec.) - the postscript time as reported by DAGMan .In case of retries the value is the cumulative of all retries.
- DAGMan(sec.) - the time between the last parent job of a job completes and the job gets submitted.In case of retries the value of the last retry is used for calculation.
- 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 .In case of retries the value is the cumulative of all retries.

- Resource(sec.) - the time between the remote Grid submission and start of remote execution . It is an estimate of the time job spent in the remote queue .In case of retries the value is the cumulative of all retries.
- Runtime(sec.) - the time spent on the resource as seen by Condor DAGMan . Is always >=kickstart .In case of retries the value is the cumulative of all retries.
- Seqexec(sec.) - the time taken for the completion of a clustered job .In case of retries the value is the cumulative of all retries.
- Seqexec-Delay(sec.) - the time difference between the time for the completion of a clustered job and sum of all the individual tasks kickstart time .In case of retries the value is the cumulative of all retries.

Table 12.2. Table Job Statistics

Job	Try	Site	Kickstart	Post	DAGMan	CondorQTime	Resource	Runtime	CondorQLen	Seqexec	Seqexec-Delay
analyze_j4	1	local	60.03	6.00	6.00	0.00	0.00	60.00	0	-	-
create_dir_blackdiamond_1_0_local	1	local	0.04	5.00	14.00	0.00	0.00	0.06	0	-	-
findrange_j2	1	local	60.03	5.00	6.00	0.00	0.00	65.00	0	-	-
findrange_j3	1	local	60.03	5.00	6.00	0.00	0.00	60.00	0	-	-
preprocess_j1	1	local	60.03	5.00	6.00	0.00	0.00	60.00	0	-	-
register_local_2_0	1	local	0.50	5.00	6.00	0.00	0.00	0.05	0	-	-
stage_in_local_local_1_0	1	local	0.08	6.00	6.00	0.00	0.00	0.04	0	-	-
stage_out_local_local_2_0	1	local	0.08	5.00	6.00	0.00	0.00	0.03	0	-	-

Logical transformation statistics table

Logical transformation statistics table contains information about each type of transformation in the workflow.

Table 12.3. Table: Logical Transformation Statistics

Transformation	Count	Mean	Variance	Min	Max	Total
diamond::analyze:2.0	1	60.1600	0.0000	60.1600	60.1600	60.1600
diamond::findrange:2.0	2	60.3100	0.0100	60.2500	60.3700	120.6200
diamond::preprocess:2.0	1	60.4800	0.0000	60.4800	60.4800	60.4800

Generating plots using pegasus-plots

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

```
$ cd $HOME/pegasus-wms

$ pegasus-plots -p all dags/tutorial/pegasus/blackdiamond/run0001/

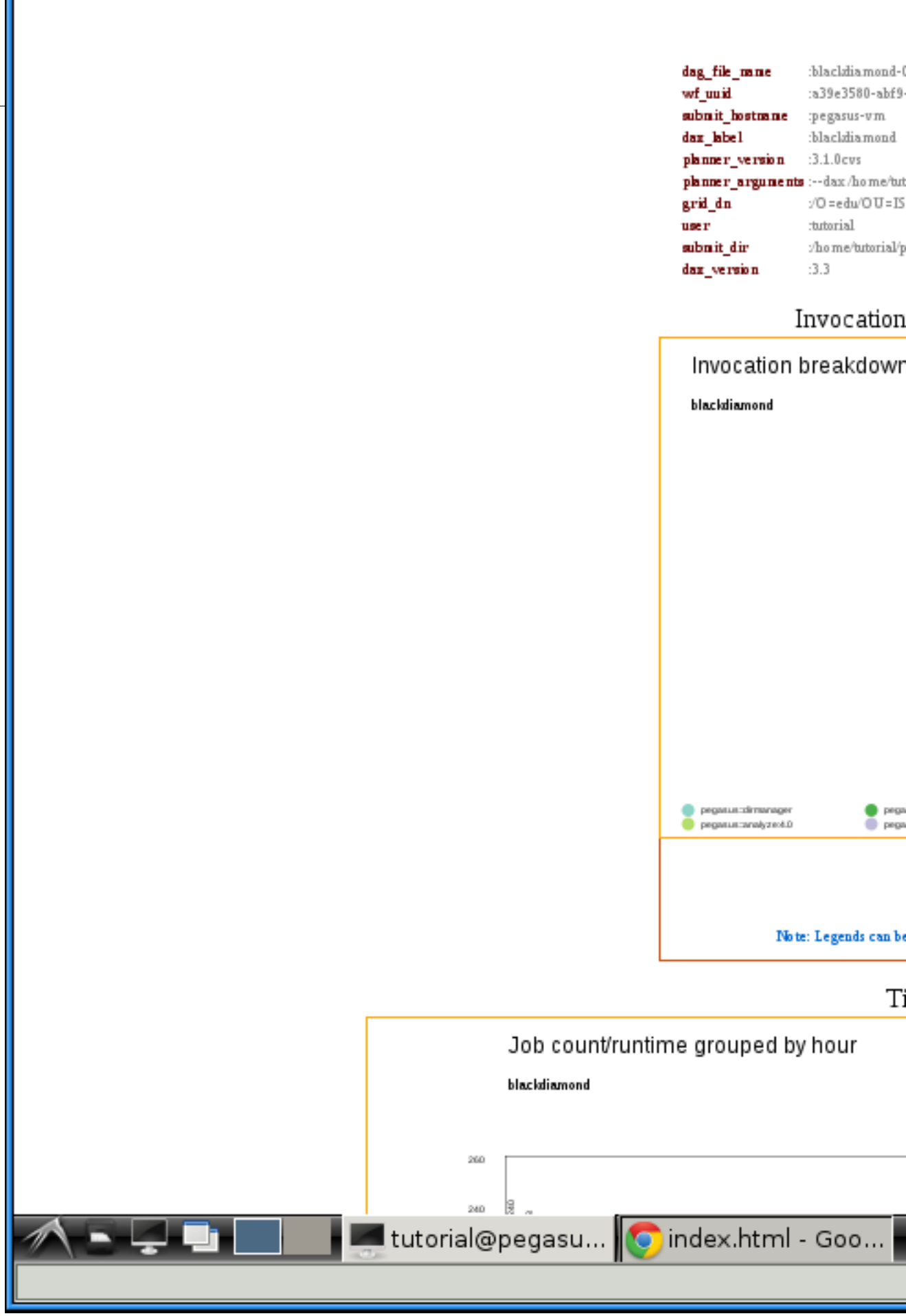
**********************************************SUMMARY*********************************************

Graphs and charts generated by pegasus-plots can be viewed by opening the generated html file in the
 web browser  :
/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/plots/index.html

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

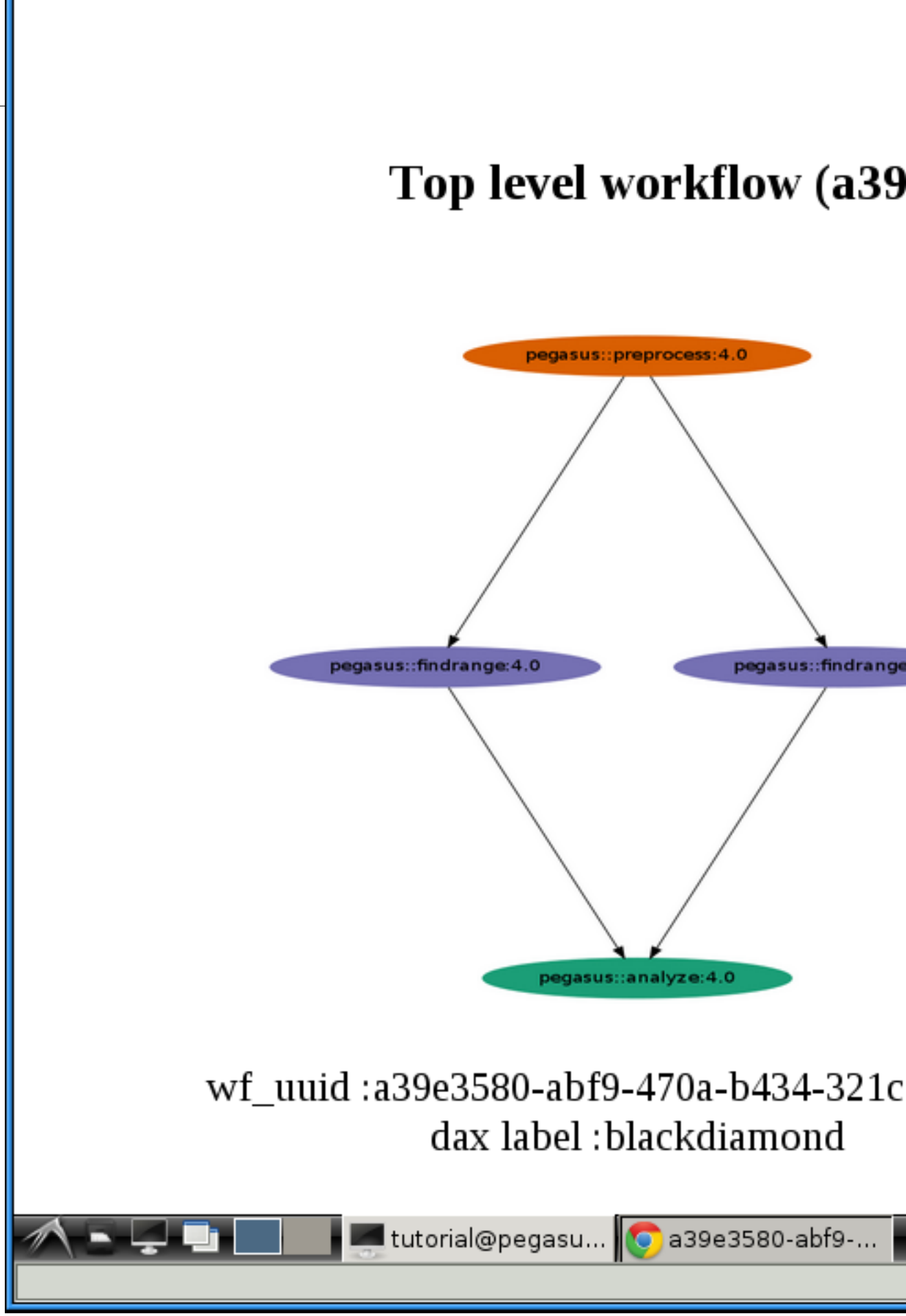
Home Page

/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/plots/index.html


dag_file_name :blackdiamond-0
wf_uuid :a39e3580-abf9-
submit_hostname :pegasus-vm
dax_label :blackdiamond
planner_version :3.1.0cvs
planner_arguments :--dax /home/tut
grid_dn :/O=edu/OU=IS
user :tutorial
submit_dir :/home/tutorial/p
dax_version :3.3
Invocation
Invocation breakdown
blackdiamond
pegasus::dirmanager
pegasus::analyzer:4.0
Note: Legends can be
Ti
Job count/runtime grouped by hour
blackdiamond
260
240
tutorial@pegasu...
index.html - Goo...

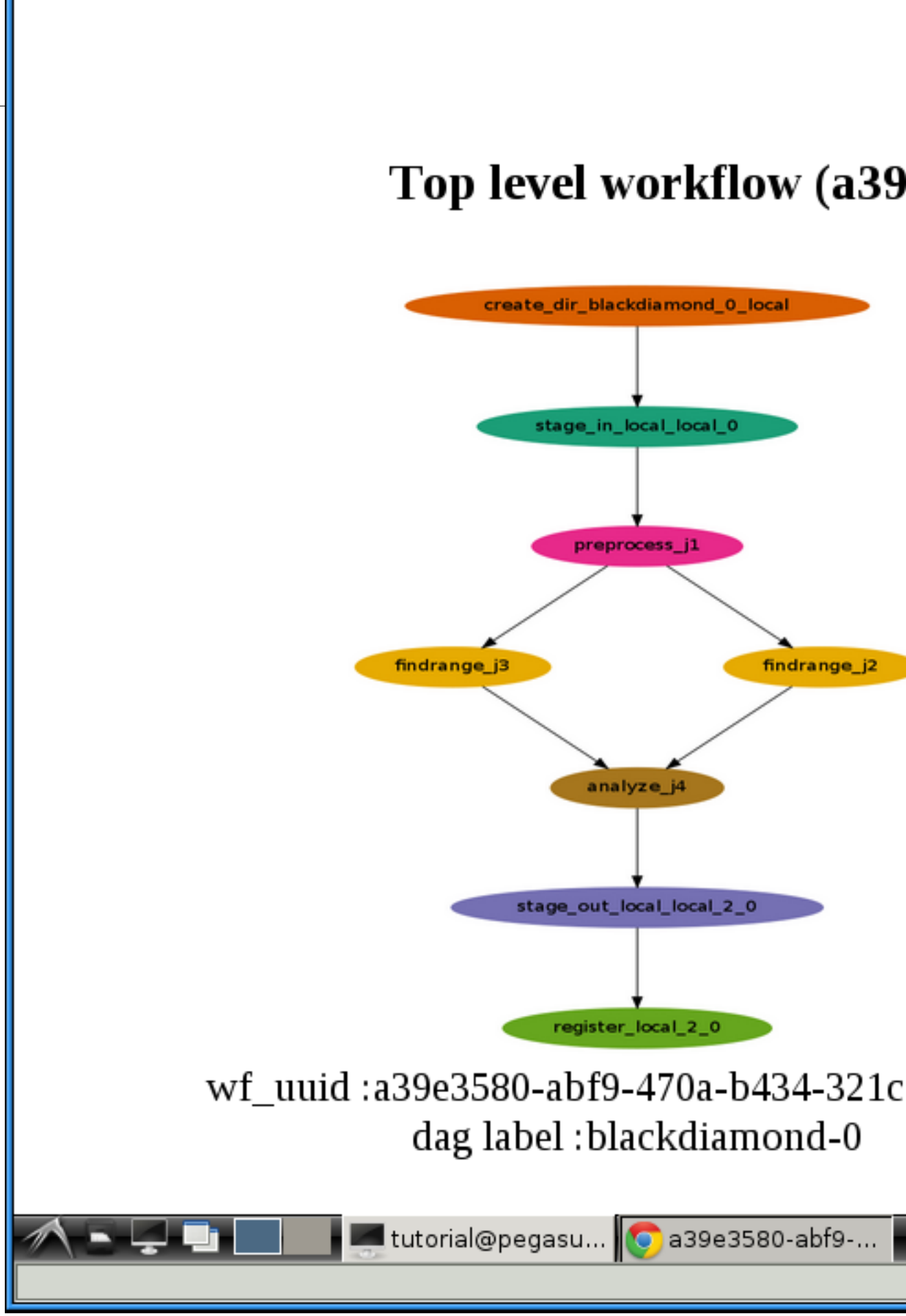
Abstract Worfklow / DAX Image

/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/plots/dax_graph/


Top level workflow (a39
pegasus::preprocess:4.0
pegasus::findrange:4.0
pegasus::findrange
pegasus::analyze:4.0
wf_uuid :a39e3580-abf9-470a-b434-321c
dax label :blackdiamond
tutorial@pegasu...
a39e3580-abf9-...

Executable Workflow / DAG Image

/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/plots/dag_graph/

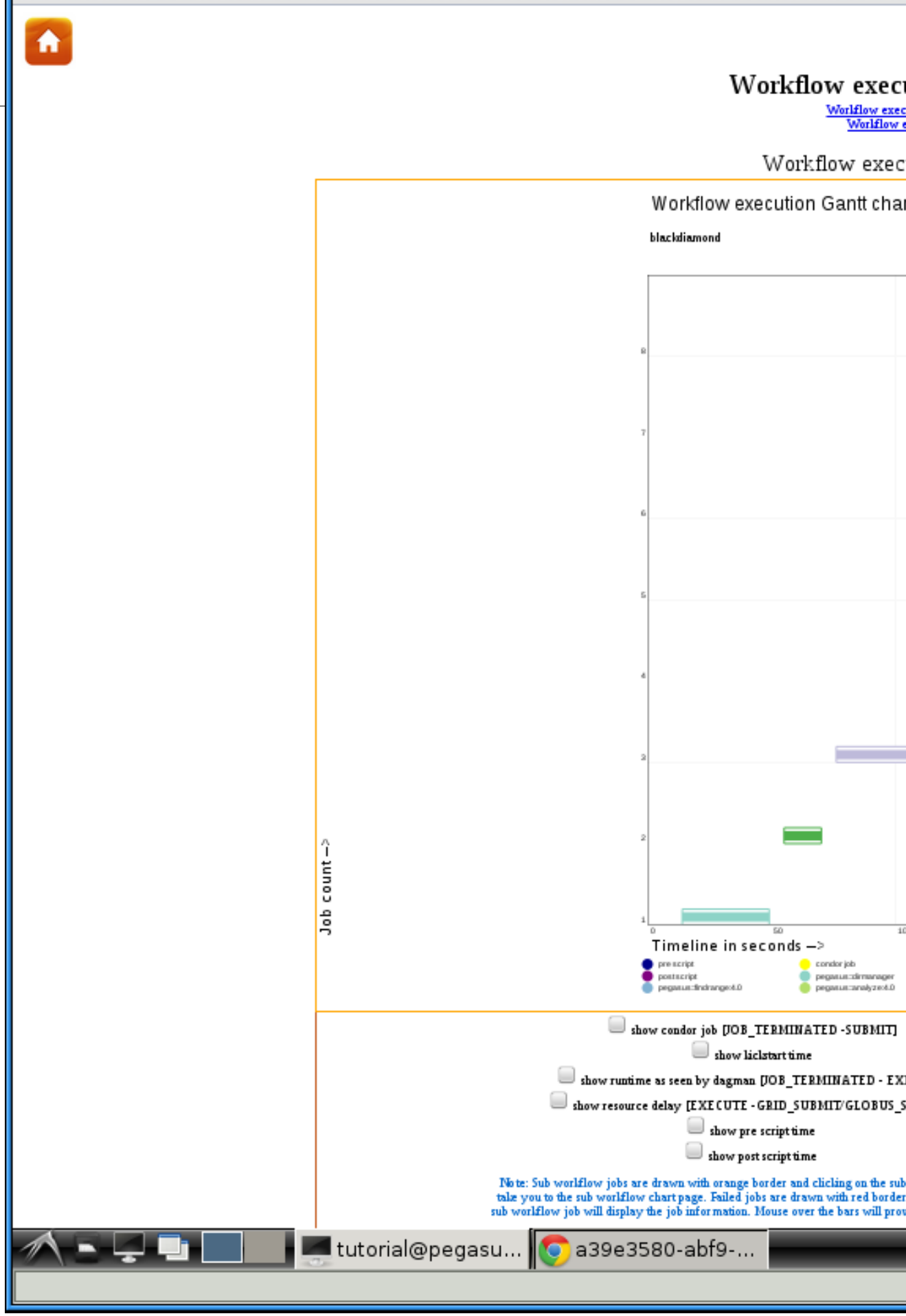

Top level workflow (a39
create_dir_blackdiamond_0_local
stage_in_local_local_0
preprocess_j1
findrange_j3
findrange_j2
analyze_j4
stage_out_local_local_2_0
register_local_2_0
wf_uuid :a39e3580-abf9-470a-b434-321c
dag label :blackdiamond-0
tutorial@pegasu...
a39e3580-abf9-...

Gantt Chart of Workflow Execution

/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/plots/gantt_chart/

X axis - time in seconds . Each tic is 60 seconds

Y axis - Job Number .

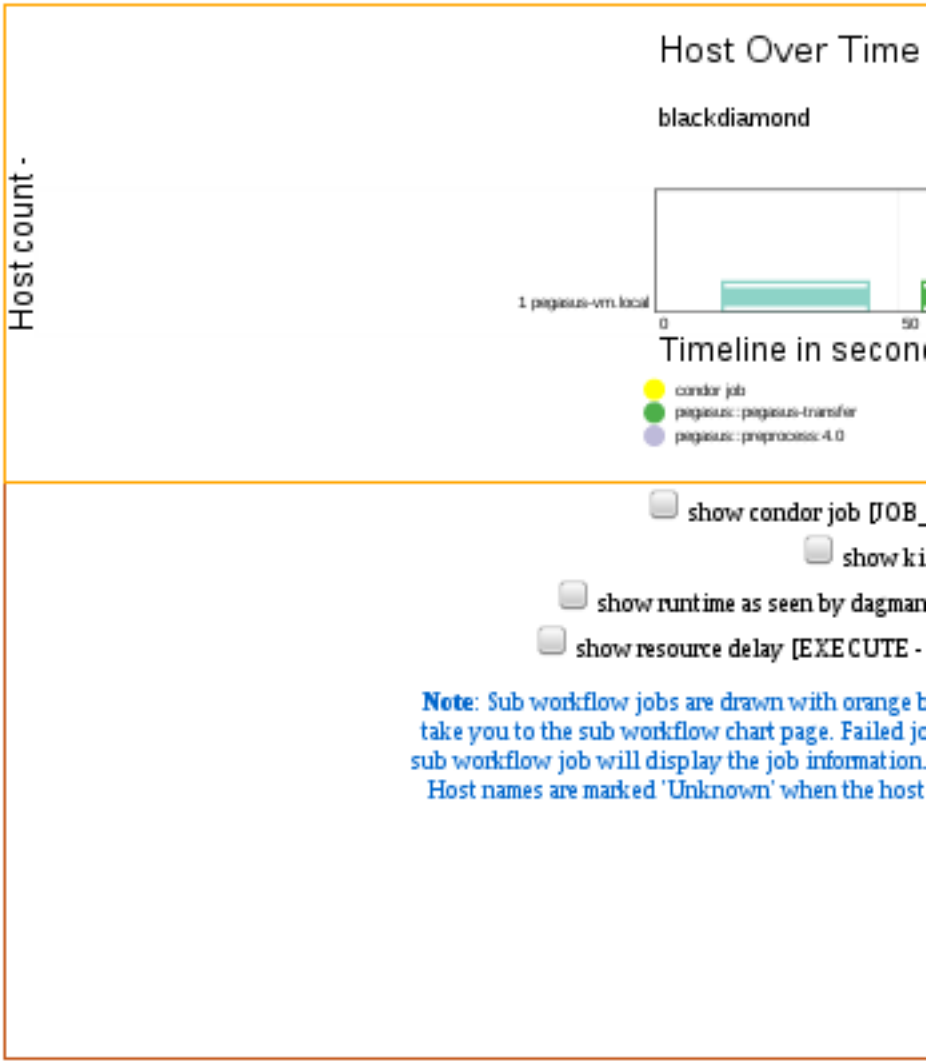

Workflow execu
Workflow exec
Workflow e
Workflow exec
Workflow execution Gantt char
blackdiamond
Job count –>
Timeline in seconds –>
pre script
post script
pegasus::findrange:4.0
condor job
pegasus::dirmanager
pegasus::analyze:4.0
show condor job [JOB_TERMINATED -SUBMIT]
show liclstart time
show runtime as seen by dagman [JOB_TERMINATED - EX
show resource delay [EXECUTE - GRID_SUBMIT/GLOBUS_S
show pre script time
show post script time
Note: Sub worlflow jobs are drawn with orange border and clicking on the sub
take you to the sub worlflow chart page. Failed jobs are drawn with red border
sub worlflow job will display the job information. Mouse over the bars will prov
tutorial@pegasu...
a39e3580-abf9-...

Host over time chart

/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/plots/host_chart/

X axis - time in seconds . Each tic is 60 seconds

Y axis - Job Number .



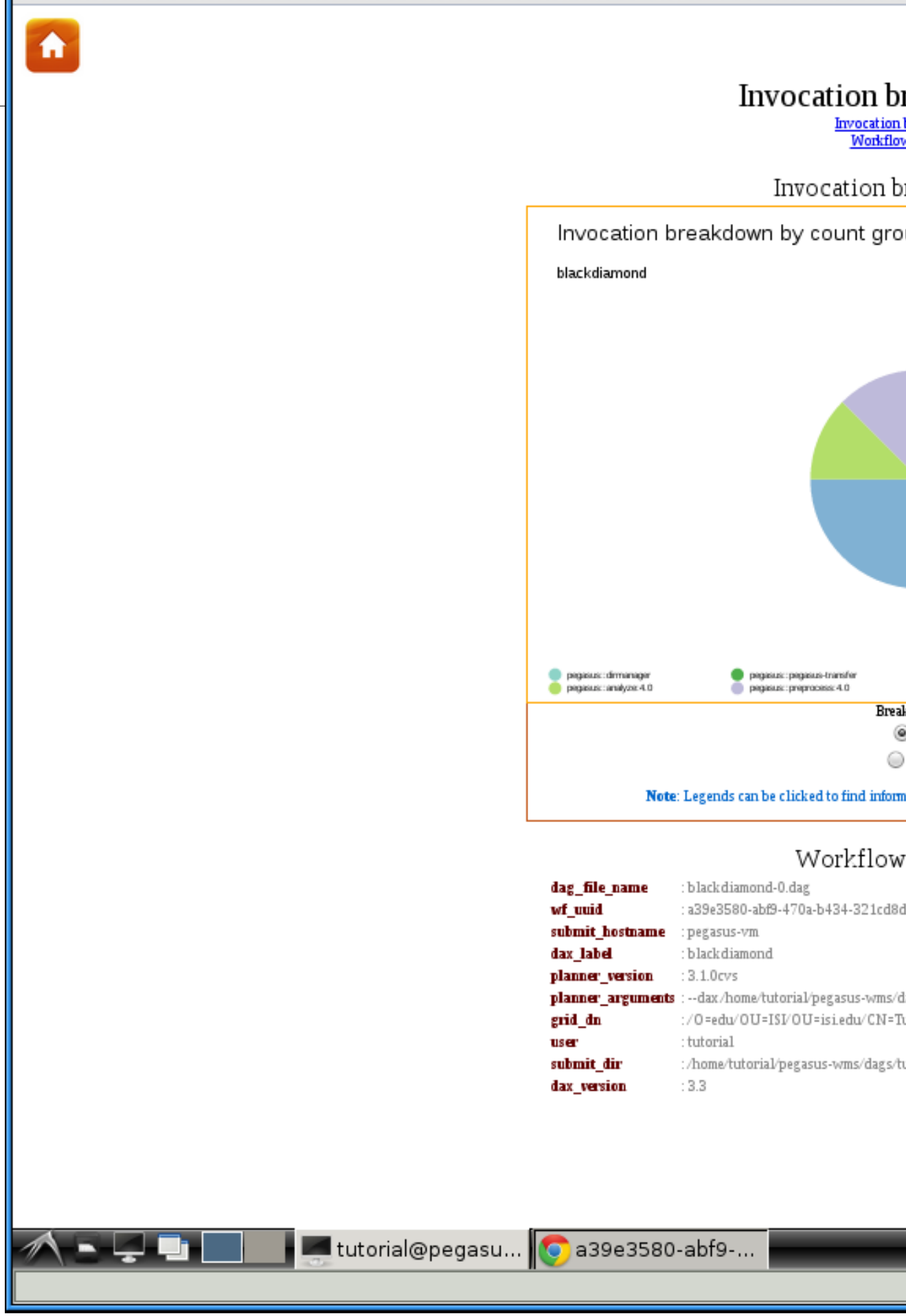
dag_file_name	: blackdiamond-
wf_uuid	: a39e3580-abf9-
submit_hostname	: pegasus-vm
dax_label	: blackdiamond
planner_version	: 3.1.0cvs
planner_arguments	: --dax /home/tut
grid_dn	: /O=edu/OU=IS
user	: tutorial
submit_dir	: /home/tutorial/
dax_version	: 3.3

Invocation Beakdown chart

/home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0001/plots/breakdown_chart/

X axis - time (seconds), count (seconds).

Y axis - Runtime (seconds).


Invocation br
Invocation b
Invocation breakdown by count grou
blackdiamond
pegasus::dirmanager
pegasus::analyze:4.0
pegasus::pegasus-transfer
pegasus::preprocess:4.0
Break
Note: Legends can be clicked to find inform
Workflow
dag_file_name : blackdiamond-0.dag
wf_uuid : a39e3580-abf9-470a-b434-321cd8d
submit_hostname : pegasus-vm
dax_label : blackdiamond
planner_version : 3.1.0cvs
planner_arguments : --dax /home/tutorial/pegasus-wms/d
grid_dn : /O=edu/OU=ISI/OU=isi.edu/CN=Tu
user : tutorial
submit_dir : /home/tutorial/pegasus-wms/dags/tu
dax_version : 3.3
tutorial@pegasu...
a39e3580-abf9-...

Planning and Executing Workflow against a Remote Resource

In this exercise we are going to run pegasus-plan to generate a executable workflow from the abstract workflow (montage.dax). The Executable workflow generated, are condor submit files that are submitted to remote grid resources using pegasus-run

The instructors have provided:

- A dax (montage.dax) in the $HOME/pegasus-wms/dax/ directory.

You will need to write some things yourself, by following the instructions below:

- Run pegasus-plan to generate the condor submit files out of the dax.

Instructions:

- Let us run pegasus-plan on the montage dax on the tg_ncsa cluster. If multiple sites are available you could provide the sites using a comma "," separated list like tg_ncsa,viz etc.

```
$ cd $HOME/pegasus-wms

$ pegasus-plan -Dpegasus.schema.dax=/opt/pegasus/default/etc/dax-2.1.xsd \
               --dir dags --sites cluster --output local --force \
               --nocleanup --dax `pwd`/dax/montage.dax --submit -v
```

The above command says that we need to plan the montage dax on the **cluster** site. The cluster site in the VM is managed by SGE that is running in the VM. The jobs for this workflow will be submitted to **jobmanager-condor** in the VM. The output data needs to be transferred back to the local host. The condor submit files are to be generated in a directory structure whose base is dags. We also are requesting that no cleanup jobs be added as we require the intermediate data on the remote host. Here is the output of pegasus-plan.

```
2011.07.29 13:21:11.377 PDT: [INFO] event.pegasus.parse.dax dax.id /home/tutorial/pegasus-wms/
dax/montage.dax  - STARTED
2011.07.29 13:21:11.386 PDT: [INFO] event.pegasus.parse.dax dax.id /home/tutorial/pegasus-wms/
dax/montage.dax  - STARTED
2011.07.29 13:21:11.493 PDT: [INFO] event.pegasus.parse.dax dax.id /home/tutorial/pegasus-wms/
dax/montage.dax  - FINISHED
2011.07.29 13:21:11.521 PDT: [INFO]  Generating Stampede Events for Abstract Workflow
2011.07.29 13:21:11.593 PDT: [INFO]  Generating Stampede Events for Abstract Workflow -DONE
2011.07.29 13:21:11.596 PDT: [INFO] event.pegasus.refinement dax.id montage_0  - STARTED
2011.07.29 13:21:11.615 PDT: [INFO] event.pegasus.siteselection dax.id montage_0  - STARTED
2011.07.29 13:21:11.644 PDT: [INFO] event.pegasus.siteselection dax.id montage_0  - FINISHED
2011.07.29 13:21:11.680 PDT: [INFO]  Grafting transfer nodes in the workflow
2011.07.29 13:21:11.681 PDT: [INFO] event.pegasus.generate.transfer-nodes dax.id montage_0  -
 STARTED
2011.07.29 13:21:11.828 PDT: [INFO] event.pegasus.generate.transfer-nodes dax.id montage_0  -
 FINISHED
2011.07.29 13:21:11.831 PDT: [INFO] event.pegasus.generate.workdir-nodes dax.id montage_0  -
 STARTED
2011.07.29 13:21:11.834 PDT: [INFO] event.pegasus.generate.workdir-nodes dax.id montage_0  -
 FINISHED
2011.07.29 13:21:11.835 PDT: [INFO] event.pegasus.generate.cleanup-wf dax.id montage_0  -
 STARTED
2011.07.29 13:21:11.836 PDT: [INFO] event.pegasus.generate.cleanup-wf dax.id montage_0  -
 FINISHED
2011.07.29 13:21:11.836 PDT: [INFO] event.pegasus.refinement dax.id montage_0  - FINISHED
2011.07.29 13:21:11.888 PDT: [INFO]  Generating codes for the concrete workflow
2011.07.29 13:21:12.354 PDT: [INFO]  Generating codes for the concrete workflow -DONE
2011.07.29 13:21:12.355 PDT: [INFO]  Generating code for the cleanup workflow
2011.07.29 13:21:12.463 PDT: [INFO]  Generating code for the cleanup workflow -DONE
2011.07.29 13:21:12.688 PDT:   Submitting job(s).
2011.07.29 13:21:12.697 PDT:   1 job(s) submitted to cluster 20.
2011.07.29 13:21:12.702 PDT:
2011.07.29 13:21:12.708 PDT:
 ------------------------------------------------------------------------
2011.07.29 13:21:12.715 PDT:   File for submitting this DAG to Condor           :
 montage-0.dag.condor.sub
```

```
2011.07.29 13:21:12.731 PDT:   Log of DAGMan debugging messages                 :
 montage-0.dag.dagman.out
2011.07.29 13:21:12.737 PDT:   Log of Condor library output                     :
 montage-0.dag.lib.out
2011.07.29 13:21:12.742 PDT:   Log of Condor library error messages             :
 montage-0.dag.lib.err
2011.07.29 13:21:12.747 PDT:   Log of the life of condor_dagman itself          :
 montage-0.dag.dagman.log
2011.07.29 13:21:12.752 PDT:
2011.07.29 13:21:12.757 PDT:
 ------------------------------------------------------------------------
2011.07.29 13:21:12.763 PDT:
2011.07.29 13:21:12.774 PDT:   Your Workflow has been started and runs in base directory given
 below
2011.07.29 13:21:12.781 PDT:
2011.07.29 13:21:12.786 PDT:   cd /home/tutorial/pegasus-wms/dags/tutorial/pegasus/montage/
run0001
2011.07.29 13:21:12.792 PDT:
2011.07.29 13:21:12.798 PDT:   *** To monitor the workflow you can run ***
2011.07.29 13:21:12.803 PDT:
2011.07.29 13:21:12.809 PDT:   pegasus-status -l /home/tutorial/pegasus-wms/dags/tutorial/
pegasus/montage/run0001
2011.07.29 13:21:12.816 PDT:
2011.07.29 13:21:12.823 PDT:   *** To remove your workflow run ***
2011.07.29 13:21:12.829 PDT:   pegasus-remove /home/tutorial/pegasus-wms/dags/tutorial/pegasus/
montage/run0001
2011.07.29 13:21:12.835 PDT:
2011.07.29 13:21:12.841 PDT:   Time taken to execute is 1.977 seconds
2011.07.29 13:21:12.841 PDT: [INFO] event.pegasus.parse.dax dax.id /home/tutorial/pegasus-wms/
dax/montage.dax  - FINISHED
```

- If you get any errors above while running pegasus-plan you can add -vvvvv to enable maximum verbosity on pegasus-run.

The above command submits the workflow to Condor DAGMan/CondorG. After submitting it starts a monitoring daemon pegasus-monitord that parses the condor log files to update the status of the jobs and push it in a work database.

Monitor the workflow using the commands provided in the output of the pegasus-run command and other commands explained earlier.

The workflow generates a single output file montage.jpg that resides in the directory **/home/tutorial/local-storage/storage/montage.jpg** if it runs successfully

The grid workflow will take time to execute on the VM. On the instructor's MAC Pro Desktop it took about **30 minutes** to run.

Advanced Exercises

Optimizing a workflow by clustering small jobs (To Be Done offline)

Sometimes a workflow may have too many jobs whose execution time is a few seconds long. In such instances the overhead of scheduling each job on a grid is too large and the runtime of the entire workflow can be optimized by using Pegasus clustering techniques. One such technique is to cluster jobs horizontally on the same level into one or more sequential jobs.

```
$ cd $HOME/pegasus-wms

$ pegasus-plan -Dpegasus.schema.dax=/opt/pegasus/default/etc/dax-2.1.xsd \
           --dir `pwd`/dags --sites cluster --output local --nocleanup --force\
           --cluster horizontal --dax `pwd`/dax/montage.dax -v
```

After clustering the executable workflow will contain 26 jobs compared to 44 in the non clustered mode.

Data Reuse

In the DAX you can specify what output data products you want to track in the replica catalog. This is done by setting the register flags with the output files for a job. For our tutorial, we only register the final output data products. So

if you were able to execute the diamond or the montage workflow successfully, we can do data reuse. Let us run **pegasus-plan** on the diamond workflow again. However, this time we will remove the **--force** option.

```
$ cd $HOME/pegasus-wms

$ pegasus-plan --dax `pwd`/dax/diamond.dax --dir `pwd`/dags -s local -o local --nocleanup -v

2011.07.29 13:22:13.022 PDT: [INFO] event.pegasus.parse.dax dax.id /home/tutorial/pegasus-wms/dax/
diamond.dax  - STARTED
2011.07.29 13:22:13.113 PDT: [INFO]  Generating Stampede Events for Abstract Workflow
2011.07.29 13:22:13.196 PDT: [INFO]  Generating Stampede Events for Abstract Workflow -DONE
2011.07.29 13:22:13.198 PDT: [INFO] event.pegasus.refinement dax.id blackdiamond_0  - STARTED
2011.07.29 13:22:13.222 PDT: [INFO] event.pegasus.reduce dax.id blackdiamond_0  - STARTED
2011.07.29 13:22:13.223 PDT: [INFO]  Nodes/Jobs Deleted from the Workflow during reduction
2011.07.29 13:22:13.223 PDT: [INFO]       analyze_j4
2011.07.29 13:22:13.223 PDT: [INFO]       findrange_j2
2011.07.29 13:22:13.223 PDT: [INFO]       findrange_j3
2011.07.29 13:22:13.224 PDT: [INFO]       preprocess_j1
2011.07.29 13:22:13.224 PDT: [INFO]  Nodes/Jobs Deleted from the Workflow during reduction  - DONE
2011.07.29 13:22:13.224 PDT: [INFO] event.pegasus.reduce dax.id blackdiamond_0  - FINISHED
2011.07.29 13:22:13.224 PDT: [INFO] event.pegasus.siteselection dax.id blackdiamond_0  - STARTED
2011.07.29 13:22:13.246 PDT: [INFO] event.pegasus.siteselection dax.id blackdiamond_0  - FINISHED
2011.07.29 13:22:13.317 PDT: [INFO]  Grafting transfer nodes in the workflow
2011.07.29 13:22:13.318 PDT: [INFO] event.pegasus.generate.transfer-nodes dax.id blackdiamond_0  -
 STARTED
2011.07.29 13:22:13.419 PDT: [INFO]  Adding stage out jobs for jobs deleted from the workflow
2011.07.29 13:22:13.421 PDT: [INFO]  The leaf file f.d is already at the output pool local
2011.07.29 13:22:13.421 PDT: [INFO] event.pegasus.generate.transfer-nodes dax.id blackdiamond_0  -
 FINISHED
2011.07.29 13:22:13.424 PDT: [INFO] event.pegasus.generate.workdir-nodes dax.id blackdiamond_0  -
 STARTED
2011.07.29 13:22:13.426 PDT: [INFO] event.pegasus.generate.workdir-nodes dax.id blackdiamond_0  -
 FINISHED
2011.07.29 13:22:13.426 PDT: [INFO] event.pegasus.generate.cleanup-wf dax.id blackdiamond_0  -
 STARTED
2011.07.29 13:22:13.428 PDT: [INFO] event.pegasus.generate.cleanup-wf dax.id blackdiamond_0  -
 FINISHED
2011.07.29 13:22:13.428 PDT: [INFO] event.pegasus.refinement dax.id blackdiamond_0  - FINISHED
2011.07.29 13:22:13.518 PDT: [INFO]  Generating codes for the concrete workflow
2011.07.29 13:22:13.927 PDT: [INFO]  Generating codes for the concrete workflow -DONE
2011.07.29 13:22:13.927 PDT:


The executable workflow generated contains only a single NOOP job.
It seems that the output files are already at the output site.
To regenerate the output data from scratch specify --force option.



pegasus-run  /home/tutorial/pegasus-wms/dags/tutorial/pegasus/blackdiamond/run0003


2011.07.29 13:22:13.927 PDT:   Time taken to execute is 1.387 seconds
2011.07.29 13:22:13.927 PDT: [INFO] event.pegasus.parse.dax dax.id /home/tutorial/pegasus-wms/dax/
diamond.dax  - FINISHED
```

You can increase the debug level to see how pegasus deletes the jobs bottom up of the workflow. Pass -vvvv to pegasus-plan command.

Hierarchal Workflows

Pegasus 3.1 allows you to create workflows of workflows i.e your workflow can contain dax jobs that refer to the sub-workflows. In this exercise, we will execute a workflow super-diamond that will execute two diamond workflows.

Let us look at superdiamond.dax in the dax directory

```
$ cat $HOME/pegasus-wms/dax/superdiamond.dax

<?xml version="1.0" encoding="UTF-8"?>
<!-- generated on: 2010-11-25T08:42:30-08:00 -->
<!-- generated by: pegasus [ ?? ] -->
<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"
 versi
on="3.2" name="superdiamond" index="0" count="1">

<!-- Section 1: Files - Acts as a Replica Catalog (can be empty) -->

   <file name="f.a">
      <pfn url="file:///scratch/tutorial/inputdata/diamond/f.a" site="local"/>
   </file>

   <file name="black-1.dax">
      <pfn url="/home/tutorial/pegasus-wms/dax/black-1.dax" site="local"/>
   </file>

   <file name="black-2.dax">
      <pfn url="/home/tutorial/pegasus-wms/dax/black-2.dax" site="local"/>
   </file>


<!-- Section 2: Executables - Acts as a Transformaton Catalog (can be empty) -->


<!-- Section 3: Transformations - Aggregates executables and Files (can be empty) -->


<!-- Section 4: Job's, DAX's or Dag's - Defines a JOB or DAX or DAG (Atleast 1 required) -->

   <dax id="d1" file="black-1.dax"  >
    <argument>-s local --force -o local</argument>
   </dax>

   <dax id="d2" file="black-2.dax"  >
    <argument>-s local --force -o local</argument>
   </dax>



<!-- Section 5: Dependencies - Parent Child relationships (can be empty) -->

   <child ref="d2">
      <parent ref="d1"/>
   </child>

</adag>
```

Now let us submit this super diamond workflow

```
$ pegasus-plan --dax `pwd`/dax/superdiamond.dax --force --submit\
               --dir dags -s local -o local --nocleanup -v

2011.07.29 13:23:35.646 PDT: [INFO] event.pegasus.parse.dax dax.id /home/tutorial/pegasus-wms/dax/
superdiamond.dax  - STARTED
2011.07.29 13:23:35.717 PDT: [INFO]  Generating Stampede Events for Abstract Workflow
2011.07.29 13:23:35.772 PDT: [INFO]  Generating Stampede Events for Abstract Workflow -DONE
2011.07.29 13:23:35.774 PDT: [INFO] event.pegasus.refinement dax.id superdiamond_0  - STARTED
2011.07.29 13:23:35.789 PDT: [INFO] event.pegasus.siteselection dax.id superdiamond_0  - STARTED
2011.07.29 13:23:35.798 PDT: [INFO] event.pegasus.siteselection dax.id superdiamond_0  - FINISHED
2011.07.29 13:23:35.842 PDT: [INFO]  Grafting transfer nodes in the workflow
2011.07.29 13:23:35.842 PDT: [INFO] event.pegasus.generate.transfer-nodes dax.id superdiamond_0  -
 STARTED
2011.07.29 13:23:35.918 PDT: [INFO] event.pegasus.generate.transfer-nodes dax.id superdiamond_0  -
 FINISHED
2011.07.29 13:23:35.922 PDT: [INFO] event.pegasus.generate.workdir-nodes dax.id superdiamond_0  -
 STARTED
2011.07.29 13:23:35.929 PDT: [INFO] event.pegasus.generate.workdir-nodes dax.id superdiamond_0  -
 FINISHED
2011.07.29 13:23:35.929 PDT: [INFO] event.pegasus.generate.cleanup-wf dax.id superdiamond_0  -
 STARTED
2011.07.29 13:23:35.931 PDT: [INFO] event.pegasus.generate.cleanup-wf dax.id superdiamond_0  -
 FINISHED
2011.07.29 13:23:35.932 PDT: [INFO] event.pegasus.refinement dax.id superdiamond_0  - FINISHED
2011.07.29 13:23:35.995 PDT: [INFO]  Generating codes for the concrete workflow
2011.07.29 13:23:36.130 PDT: [INFO] event.pegasus.parse.dax dax.id /home/tutorial/pegasus-wms/dax/
black-1.dax  - STARTED
2011.07.29 13:23:36.161 PDT: [INFO] event.pegasus.parse.dax dax.id /home/tutorial/pegasus-wms/dax/
black-2.dax  - STARTED
2011.07.29 13:23:36.488 PDT: [INFO]  Generating codes for the concrete workflow -DONE
```

```
2011.07.29 13:23:36.489 PDT: [INFO]  Generating code for the cleanup workflow
2011.07.29 13:23:36.626 PDT: [INFO]  Generating code for the cleanup workflow -DONE
2011.07.29 13:23:36.829 PDT:   Submitting job(s).
2011.07.29 13:23:36.839 PDT:   1 job(s) submitted to cluster 23.
2011.07.29 13:23:36.845 PDT:
2011.07.29 13:23:36.850 PDT:
 ------------------------------------------------------------------------
2011.07.29 13:23:36.856 PDT:   File for submitting this DAG to Condor           :
 superdiamond-0.dag.condor.sub
2011.07.29 13:23:36.862 PDT:   Log of DAGMan debugging messages                :
 superdiamond-0.dag.dagman.out
2011.07.29 13:23:36.867 PDT:   Log of Condor library output                    :
 superdiamond-0.dag.lib.out
2011.07.29 13:23:36.874 PDT:   Log of Condor library error messages            :
 superdiamond-0.dag.lib.err
2011.07.29 13:23:36.880 PDT:   Log of the life of condor_dagman itself         :
 superdiamond-0.dag.dagman.log
2011.07.29 13:23:36.886 PDT:
2011.07.29 13:23:36.892 PDT:
 ------------------------------------------------------------------------
2011.07.29 13:23:36.898 PDT:
2011.07.29 13:23:36.915 PDT:   Your Workflow has been started and runs in base directory given
 below
2011.07.29 13:23:36.920 PDT:
2011.07.29 13:23:36.926 PDT:   cd /home/tutorial/pegasus-wms/dags/tutorial/pegasus/superdiamond/
run0001
2011.07.29 13:23:36.933 PDT:
2011.07.29 13:23:36.938 PDT:   *** To monitor the workflow you can run ***
2011.07.29 13:23:36.944 PDT:
2011.07.29 13:23:36.951 PDT:   pegasus-status -l /home/tutorial/pegasus-wms/dags/tutorial/pegasus/
superdiamond/run0001
2011.07.29 13:23:36.957 PDT:
2011.07.29 13:23:36.962 PDT:   *** To remove your workflow run ***
2011.07.29 13:23:36.968 PDT:   pegasus-remove /home/tutorial/pegasus-wms/dags/tutorial/pegasus/
superdiamond/run0001
2011.07.29 13:23:36.974 PDT:
2011.07.29 13:23:36.980 PDT:   Time taken to execute is 1.821 seconds
2011.07.29 13:23:36.980 PDT: [INFO] event.pegasus.parse.dax dax.id /home/tutorial/pegasus-wms/dax/
black-1.dax  - FINISHED
```

You can track the workflow using the pegasus-status command

```
$ watch  pegasus-status -l /home/tutorial/pegasus-wms/dags/tutorial/pegasus/superdiamond/run0001
```

After the workflow has completed you will see the black-1-f.d and black-2-f.d in the storage directory

```
$ ls -lh /home/tutorial/local-storage/storage/black-*

-rw-r--r-- 1 pegasus pegasus 3.6K Nov 29 21:36 /home/tutorial/local-storage/storage/black-1-f.d
-rw-r--r-- 1 pegasus pegasus 3.6K Nov 29 21:41 /home/tutorial/local-storage/storage/black-2-f.d
```

Directory Structure For the Hierarchal Workflows

Pegasus ensures that each of the workflows have their own submit directory and execution directories.

The table below lists the submit directories for all the workflows in this exercise

Table 12.4. Table: Submit Directory Structure for Hierarchal Workflows

superdiamond (the outer level workflow)	/home/tutorial/pegasus-wms/dags/tutorial/pegasus/superdiamond/run0001
black-1 (the first sub workflow)	/home/tutorial/pegasus-wms/dags/tutorial/pegasus/superdiamond/run0001/black-1_d1
black-2 (the second sub workflow)	/home/tutorial/pegasus-wms/dags/tutorial/pegasus/superdiamond/run0001/black-2_d2

The table below lists the execution directories (one per workflow) in this exercise

Table 12.5. Table: Execution Directory Structure for Hierarchal Workflows

superdiamond (the outer level workflow)	/home/tutorial/local-scratch/exec/tutorial/pegasus/superdiamond/run0001
black-1 (the first sub workflow)	/home/tutorial/local-scratch/exec/tutorial/pegasus/superdiamond/run0001/black-1_d1
black-2 (the second sub workflow)	/home/tutorial/local-scratch/exec/tutorial/pegasus/superdiamond/run0001/black-2_d2