1 User Guide
   1.1 Install and Basic Usage .................................................. 3
   1.2 Configuration .............................................................. 7
   1.3 Job Manipulation ............................................................ 8
   1.4 Running Executables ........................................................ 10
   1.5 Using Different Backends .................................................. 10
   1.6 Input And Output Data ...................................................... 13
   1.7 Splitters ................................................................. 15
   1.8 PostProcessors ............................................................. 16
   1.9 Miscellaneous Functionality .............................................. 20
   1.10 Queues ............................................................... 21
   1.11 Tasks ............................................................... 22

2 GPI

3 Guide for System Administrators
   3.1 Installation ............................................................... 29
   3.2 Site config .............................................................. 29

4 Guide for developers
   4.1 GangaObject ............................................................. 31
   4.2 Schema ............................................................... 31
   4.3 Proxy objects .......................................................... 31
   4.4 Repository ............................................................ 32
   4.5 Registry .............................................................. 32
   4.6 Job monitoring .......................................................... 32
   4.7 IGangaFile ............................................................ 32

5 API Documentation .......................................................... 33
Ganga is a tool to make it easy to run data analysis jobs along with managing associated data files.
This tutorial is in several stages that cover all of the basic and more of the advanced functionality that Ganga provides. To follow it properly, you should go through each stage in turn. Please post an issue if you encounter any problems!

## 1.1 Install and Basic Usage

### 1.1.1 Installation

There are several ways to install and run Ganga:

**CVMFS**

If you have access to CVMFS, Ganga can be found at /cvmfs/ganga.cern.ch/. This will be kept up-to-date with the latest release that you can run directly with:

```
/cvmfs/ganga.cern.ch/runGanga.sh
```

This isn’t just a link to the start-up script because it needs to be run from the correct directory. However, it will take all the command line options that the normal ganga script takes.

**PyPI Install**

You can also install directly from PyPI using:

```
pip install ganga
```

If you don’t have System Administrator rights or just want to do a private install then it can be beneficial to run using a virtualenv:

```
virtualenv ~/gangaenv
source ~/gangaenv/activate
pip install ganga
```

To install pip locally if it’s not on your system and you don’t have admin access please consult: https://pip.pypa.io/en/stable/installing/

Alternatively if you want you can install it using pip to your local user environment (not recommended if you have many python projects/libraries around)

```
pip install ganga --user
```
Ganga Install Script

There is an install script provided here that will not only download and install Ganga but also all the externals required as well. To run it, download it and do the following:

```
wget http://ganga.web.cern.ch/ganga/download/ganga-install
python ganga-install --extern=GangaDirac <RELEASE>
```

(Here `<RELEASE>` is in the format x.y.z)

To update an install to the latest code from github develop (FOR DEVELOPERS ONLY, THIS IS UNSUPPORTED) try something similar to the following:

```
wget http://ganga.web.cern.ch/ganga/download/ganga-install
chmod +x ganga-install
./ganga-install 6.1.20
cd ~/Ganga/install
rm -fr 6.1.20
git clone https://github.com/ganga-devs/ganga
mv ganga 6.1.20
```

From Github

You can always download a .zip/.tar.gz of a release from github here Simply unzip the file where you want Ganga to be available and then you can run it using:

```
<install-dir>/ganga-<RELEASE>/bin/ganga
```

Note that this will not install any of the additional packages Ganga requires so you may have import errors when running. If so, please install the appropriate python packages on your system.

For LHCb

(This has been tested on lxplus, feel free to open an issue if this fails on cvmfs/CernVM)

First grab the LHCb installer script:

```
chmod +x $TMPDIR/lhcb-prepare
```

To install locally run one of the following:

- For the official Ganga releases:
  
  ./lhcb-prepare v601r14

- For an ‘in development’ release:
  
  ./lhcb-prepare -p v601r15

- To keep the git history of the project so that you can develop patches and branches etc:
  
  ./lhcb-prepare -k -t v601r15

You now have Ganga installed in $HOME/cmtuser/GANGA/GANGA_v601r15

- To update your install of Ganga to use the latest git code:
cd $HOME/cmtuser/GANGA/GANGA_v601r15

- Finally use:

  SetupProject ganga v601r15

to get a Ganga environment setup.

### 1.1.2 Starting Ganga

As described above, to run Ganga simply execute `ganga` (for PyPI install), `<installdir>/bin/ganga` (for other installs) or the helper script in CVMFS. This will start Ganga and its associated threads as well as provide you with a Ganga IPython prompt that gives you access to the Ganga Public Interface (GPI) on top of the usual IPython functionality:

Note that the first time you run Ganga it will ask you to create a default `.gangarc` file which you should probably do. In the future, if you want to recreate this default config file, add the option `-g` to the command line.

### 1.1.3 Getting Help

The documentation for all objects and functions in Ganga can be accessed using the help system:

```
[13:25:29]
Ganga In [1]: help()
************************************
*** Welcome to Ganga ***
Version: 6.1.16
Documentation and support: http://cern.ch/ganga
Type help() or help('index') for online help.

This is free software (GPL), and you are welcome to redistribute it under certain conditions; type license() for details.

This is an interactive help based on standard pydoc help.

Type 'index' to see GPI help index.
Type 'python' to see standard python help screen.
Type 'interactive' to get online interactive help from an expert.
Type 'quit' to return to Ganga.
************************************
help>
```

Now typing `index` at the prompt will list all the objects, etc. available. You can also directly access the documentation for an object using `help` directly:

```
help(Job)
```

You also have IPython’s tab-complete service available to help identify members of an object.
1.1.4 Hello World with Ganga

We are now in a position to submit our first job. This will take the defaults of the Ganga Job object which is to run `echo 'Hello World'` on the machine you’re currently running on:

```python
j = Job()
j.submit()
```

If all goes well, you should see the job submit:

```text
Ganga In [6]: j.submit()
```

If you wait a few seconds and then press Enter you should then see that the job has already transitioned through running and to completed:

```text
Ganga In [7]:
Ganga.GPIDev.Lib.Job : INFO job 0 status changed to "running"
Ganga.GPIDev.Lib.Job : INFO Job 0 Running PostProcessor hook
Ganga.GPIDev.Lib.Job : INFO job 0 status changed to "completed"
```

You can view the job in your repository using the `jobs` command which lists all job objects that Ganga knows about:

```text
Ganga In [7]: jobs
Ganga Out [7]:
Registry Slice: jobs {1 objects}
-----------------------
  fqid | status | name | subjobs | application | backend |
-----------------------
  0 | completed | | | Executable | Local |
```

You can get more info about your job by selecting it from the repository:

```text
jobs(0)
```

You can also select specific info about the job object, e.g. the application that was run:

```text
jobs(0).application
```

To check the `stdout/stderr` of a job, you can use a couple of methods:
1.1.5 Job Monitoring

While Ganga is running in interactive mode, a background thread goes through all your active jobs and checks to see what state they are in. Generally, jobs will transition from new -> submitted -> running -> completed/failed. As described above, the `jobs` command will show you the state of your jobs in the Ganga repository.

In addition to this monitoring, there is also a web GUI provided that can be started using the `--webgui` option which gives a graphical representation of your repository. [NOTE THIS NEEDS TESTING AND MAY NOT WORK AT PRESENT!]

1.1.6 Scripting and Batch Mode

You can put your ganga commands into a python script and then execute it from the Ganga prompt like this:

```python
open('submit.py', 'w').write(""
}
execfile('submit.py')
```

In addition, Ganga can be run in batch mode by just providing a script as the last argument:

```
ganga submit.py
/cvmfs/ganga.cern.ch/runGanga.sh submit.py
```

Note that by default, the monitoring is turned off while in batch mode.

1.2 Configuration

There are several ways that you can configure and control how Ganga behaves. There are 3 different ways to do this:

1. Edit the options in your `~/.gangarc` file
3. At runtime using the `config` variable:

```python
# print full config
config

# print config section
config.Logging

# edit a config option
config.Logging['Ganga.Lib'] = 'DEBUG'
```

The config system also provides a set of default options for each Ganga object which override what values the object starts with on creation. e.g.

```python
config.defaults_Executable.exe = 'ls'
```
In addition to this, you can also supply a 

```
~/.ganga.py
```

file that will be executed just as if you’d typed the commands when Ganga starts up e.g. this will show all running jobs when you start Ganga if put into the 

```
~/.ganga.py
```

file:

```python
slice = jobs.select(status='running')
print(slice)
```

### 1.3 Job Manipulation

There are several ways to control and manipulate your jobs within Ganga.

#### 1.3.1 Copying Jobs

You can copy jobs using the `copy` method or using the cop-constructor in the Job creation. The job status is always set to `new`:

```python
j = Job(name = 'original')
j2 = j.copy()
j2.name = 'copy'
j.submit()
j3 = Job(j, name = 'copy2')
```

```
Ganga Out [3]:
Registry Slice: jobs (4 objects)
------------------
<table>
<thead>
<tr>
<th>fqid</th>
<th>status</th>
<th>name</th>
<th>subjobs</th>
<th>application</th>
<th>backend</th>
<th>backend.actualCE</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>completed</td>
<td></td>
<td></td>
<td>Executable</td>
<td>Local</td>
<td>epldt017.ph.bham.ac.uk</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>completed</td>
<td>original</td>
<td></td>
<td>Executable</td>
<td>Local</td>
<td>epldt017.ph.bham.ac.uk</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>new</td>
<td>copy</td>
<td></td>
<td>Executable</td>
<td>Local</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>new</td>
<td>copy2</td>
<td></td>
<td>Executable</td>
<td>Local</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### 1.3.2 Accessing Jobs in the Repository

As shown before, you can view all the jobs that Ganga is aware of using the `jobs` command. To access a specific job from the repo with the parentheses, use it’s id number or:

```
jobs(2)
```

You can also use the square bracket (`[]`) notation to specify single jobs, lists of jobs or a job by a (unique) name:

```
jobs[2]
jobs[2:]  
jobs['copy2']
```

#### 1.3.3 Resubmitting Jobs

Jobs can fail for any number of reasons and often it’s a transient problem that resubmitting the job will solve. To do this in Ganga, simply call the `resubmit` method:

```
jobs(0).resubmit()
```

Note that, as above, this can also be used on `completed` jobs, though it’s backend and application dependent.
1.3.4 Forcing to Failed

Sometimes you may encounter a problem where the job has been marked completed by the backend but you notice in the logs that there was a problem which renders the output useless. To mark this job as failed, you can do:

```python
jobs(1).force_status('failed')
```

Note that there are PostProcessors in Ganga that can help with a lot of these kind of problems.

1.3.5 Removing Jobs

As you submit more jobs, your Ganga repository will grow and could become quite large. If you have finished with jobs it is good practise to remove them from the repository:

```python
jobs(2).remove()
```

This will remove all associated files and directories from disk.

1.3.6 Performing Bulk Job Operations

There are several job operations you can perform in bulk from a set of jobs. To obtain a list of jobs, you can either use the array syntax described above or the `select` method:

```python
# can select on ids, name, status, backend, application
jobs.select(status='new')
jobs.select(backend='Local')
jobs.select(ids=[1,3])

# can restrict on min/max id
jobs.select(1,3, application='Executable')
```

Given this selection, you can then perform a number of operations on all of the jobs at once:

```python
jobs.select(status='new').submit()
```

Available operations are: `submit`, `copy`, `kill`, `resubmit`, `remove`. These also take the `keep_going` argument which, if set to `True` will mean that it will keep looping through the jobs even if an error occurs performing the operation on one of them. These operations can also be performed on subjobs as well - see `SplittersAndPostprocessors` for more info.

1.3.7 Export and Import of Ganga Objects

Ganga is able to export a Job object (or a `select`ion of Job objects) or any other Ganga object using the `export` method which will create a human readable text file that you can edit manually and then load in using `load`:

```python
export(jobs(0), 'my_job.txt')
jlist = load('my_job.txt')
jlist[0].submit()
```

As in the above example, any jobs loaded will be put into the `new` state.
1.4 Running Executables

You can run any executable or script through Ganga using the Executable application. This accepts either a full path to an already installed exe (e.g. on CVMFS) or a Ganga File object that will be sent with your job. You can also supply arguments and environment settings with the options in the Executable object.

As an example:

```python
# Already existing Exe
j = Job()
j.application = Executable()
j.application.exe = '/bin/ls'
j.application.args = ['-l', '-h']
j.submit()

# Wait for completion
j.peek("stdout")

# Send a script
open('my_script.sh', 'w').write("#!/bin/bash
    echo 'Current dir: ' `pwd`
    echo 'Contents:'
    ls -ltr
    echo 'Args: ' $@
"")

import os
os.system('chmod +x my_script.sh')

j = Job()
j.application = Executable()
j.application.exe = File('my_script.sh')
j.submit()

# Wait for completion
j.peek("stdout")
```

If your executable requires more than one file to run, you can use the inputfiles field of the Job object to send this across as well (see Input And Output Data).

1.5 Using Different Backends

One of the main benefits of Ganga is that you can submit to different clusters/systems (in Ganga, these are termed backends) by only changing one or two lines in your scripts. Though there are often very different ways of submission for each backend, Ganga tries to hide this as much as possible. We’ll cover the main options in Ganga Core below but to get a full list, use:

```python
plugins("backends")
```

1.5.1 Local Backend

This is the default and refers to the machine that Ganga is running on. The job will be spawned as a separate process, independent of Ganga. Typical usage is:
There are no editable options for the object itself but there are two config options that you can view with
config.Local.

### 1.5.2 Batch Backends

Ganga supplies backend objects for most of the major batch systems around - LSF, PBS and Condor. You should obviously use the one that is relevant to the system you are running on. Typical usage is detailed below though as with all these, you can get more help using help(<backend>) and config.<backend>:

**LSF**

The default setup will work for LSF set up at CERN but you can change the patterns searched for in the output of the LSF commands if these are different in your particular setup.

```python
j = Job()
j.backend = LSF()
j.backend.queue = '1nh'
```

**PBS**

Very similar to the LSF backend, this is setup by default to submit to a typical Torque/Maui installation but again, can easily be changed to reflect your specific setup:

```python
j = Job()
j.backend = PBS()
j.submit()
```

**Condor**

Condor is a little different than the other backends but should still submit to most typical installations. There is also a requirements object that can be used to specify memory, architecture, etc.

```python
j = Job()
j.backend = Condor()
j.backend.getenv = "True"  # send the environment to the host
j.backend.requirements.memory = 1200
j.submit()
```

Also note that the `.getenv` option is defined as a string so in your `.gangarc`, you would need to set it to:

```ini
[Condor]
.getenv = 'True'
```

To avoid Ganga attempting to assign a boolean instead.

### 1.5.3 Dirac Backend

To submit to a Dirac instance, you will need to have the Dirac client installed and a Dirac proxy available.

1.5. Using Different Backends
Using GridPP Dirac on CVMFS

There is an installed version of Dirac configured to use the GridPP Dirac instance available on the Ganga CVMFS area. To run with this, simply do the following:

- If you are not running with on the GridPP VO, change the following in your ~/.gangarc file:

  ```
  [defaults_DiracProxy]
  group = <dirac user group>
  ```

- After this, you can run Ganga with the Dirac plugins setup and pointing to the GridPP Dirac instance by simply running:

  ```
  /cvmfs/ganga.cern.ch/runGanga-dirac.sh
  ```

Installing and Configuring the Dirac Client

If don’t have access to CVMFS then you will need to install and configure the Dirac client. How this is done will depend somewhat on your Dirac instance but for the one hosted by GridPP, first follow the installation instructions here

After successfully doing this, do the following steps to configure Ganga:

- Edit your .gangarc file and set the following options:

  ```
  [Configuration]
  RUNTIME_PATH = GangaDirac

  [LCG]
  GLITE_SETUP = /home/<username>/dirac_ui/bashrc

  [DIRAC]
  DiracEnvSource = /home/<username>/dirac_ui/bashrc

  [defaults_DiracProxy]
  group = <dirac user group>
  ```

Testing Dirac Submission Through Ganga

To test everything is working, run Ganga (either using the CMVFS helper script runGanga-dirac.sh or as normal). It should ask you to generate a proxy if you haven’t already and then leave you at the IPython prompt.

To test that all is working, try to submit a basic job to the local machine you’re running and then to DIRAC:

```python
j = Job()
j.backend = Dirac()
j.submit()
```

If you go to the Dirac portal (for GridPP, go here), you should now see your job! On completion, you can view the output of the job using:

```python
j.peek('Ganga_Executable.log')
```

1.5.4 LCG Backends

There are several ways to submit ‘directly’ to the LCG grid through Ganga that are detailed below. However, all these require that you have a Grid UI installed and a Grid Proxy available. Ganga will check if you have a proxy when it
needs it and prompt to create one if it’s not available. To set your config correctly, use the following:

### LCG backend

### ARC backend

### CREAM backend

## Remote Backend

This backend allows you to submit a job ‘through’ a gateway host. IT IS CURRENTLY BEING FIXED!!

### 1.6 Input And Output Data

Ganga tries to simplify sending input files and getting output files back as much as possible. You can specify not only what files you want but where they should be retrieved/put. There are three fields that are relevant for your job:

1. **Input Files** Files that are sent with the job and are available in the same directory on the worker node that runs it

2. **Input Data** A dataset or list of files that the job will run over but which are NOT transferred to the worker.

3. **Output Files** The name, type and location of the job output

#### 1.6.1 Basic Input/Output File usage

To start with, we’ll show a job that sends an input text file with a job and then sends an output text file back:

```python
# create a script to send
open('my_script2.sh', 'w').write("#!/bin/bash
ls -ltr
more "my_input.txt"
echo "TESTING" > my_output.txt
"")

import os
os.system('chmod +x my_script2.sh')

# create a script to send
open('my_input.txt', 'w').write('Input Testing works!')

j = Job()
j.application.exe = File('my_script2.sh')
j.inputfiles = [ LocalFile('my_input.txt') ]
j.outputfiles = [ LocalFile('my_output.txt') ]
j.submit()
```

After the job completes, you can then view the output directory and see the output file:

```python
j.peek()  # list output dir contents
j.peek('my_output.txt')
```

If the job doesn’t produce the output Ganga was expecting, it will mark the job as failed:

```python
# This job will fail
j = Job()
j.application.exe = File('my_script2.sh')
j.inputfiles = [ LocalFile('my_input.txt') ]
j.outputfiles = [ LocalFile('my_output_FAIL.txt') ]
j.submit()
```
You can also use wildcards in the files as well:

```python
# This job will pick up both 'my_input.txt' and 'my_output.txt'
j = Job()
j.application.exe = File('my_script2.sh')
j.inputfiles = [LocalFile('my_input.txt')]
j.outputfiles = [LocalFile('*.txt')]
j.submit()
```

After completion, the output files found are copied as above but they are also recorded in the job appropriately:

```python
j.outputfiles
```

This will also work for all backends as well - Ganga handles the changes in protocol behind the scenes, e.g.:

```python
j = Job()
j.application.exe = File('my_script2.sh')
j.inputfiles = [LocalFile('my_input.txt')]
j.outputfiles = [LocalFile('my_output.txt')]
j.backend = Dirac()
j.submit()
```

### 1.6.2 Input Data Usage

Generally, input data for a job is quite experiment specific. However, Ganga does provide some basic input data functionality in Core that can be used to process a set of remotely stored files without copying them to the worker. This is done with the `GangaDataset` object that takes a list of `GangaFiles` (as you would supply to the `inputfiles` field) and instead of copying them, a flat text file is created on the worker (`__GangaInputData.txt__`) that lists the paths of the given input data. This is useful to access files from Mass or Shared Storage using the mechanisms within the running program, e.g. Opening them with directly with Root.

As an example:

```python
# Create a test script
open('my_script3.sh', 'w').write("#!/bin/bash
  echo $PATH
  ls -ltr
  more __GangaInputData.txt__
  echo "MY TEST FILE" > output_file.txt
 "")
import os
os.system('chmod +x my_script3.sh')

# Submit a job
j = Job()
j.application.exe = File('my_script3.sh')
j.inputdata = GangaDataset(files=[LocalFile('*.sh')])
j.backend = Local()
j.submit()
```

### 1.6.3 File Types Available

Ganga provides several File types for accessing data from various sources. To find out what’s available, do:

```python
plugins('gangafiles')
```
LocalFile

This is a basic file type that refers to a file on the submission host that Ganga runs on. As in input file, it will pick up the file and send it with your job, as an output file it will be returned with your job and put in the \texttt{j.outputdir} directory.

DiracFile

This will store/retrieve files from Dirac data storage. This will require a bit of configuration in \texttt{~/.gangarc} to set the correct LFN paths and also where you want the data to go:

\begin{verbatim}
config.DIRAC.DiracLFNBase
config.DIRAC.DiracOutputDataSE
\end{verbatim}

To use a DiracFile, do something similar to:

\begin{verbatim}
j = Job()
j.application.exe = File('my_script2.sh')
j.inputfiles = [ LocalFile('my_input.txt') ]
j.outputfiles = [ DiracFile('my_output.txt') ]
j.backend = Dirac()
j.submit()
\end{verbatim}

Ganga won’t retrieve the output to the submission node so you need to do this manually:

\begin{verbatim}
j.outputfiles.get()
\end{verbatim}

LCGSEFile

MassStorageFile

GoogleFile

WebDAVFile

1.7 Splitters

One of the main benefits of Ganga is it’s ability to split a job description across many subjobs, changing the input data or arguments appropriately for each. Ganga then keeps these subjobs organised with the parent master job but keeps track of all their status, etc. individually. There are two main splitters that are provided in Ganga Core which are detailed below.

1.7.1 GenericSplitter

The \texttt{GenericSplitter} is a useful tool to split a job based on arguments or parameters in an application or backend. You can specify whatever attribute you want to split over within the job as a string using the \texttt{attribute} option. A typical example using the basic \texttt{Executable} application is to produce subjobs with different arguments:

\begin{verbatim}
j = Job()
j.splitter = GenericSplitter()
j.splitter.attribute = 'application.args'
j.splitter.values = [['hello', 1], ['world', 2], ['again', 3]]
j.submit()
\end{verbatim}
This produces 3 subjobs with the arguments:

```
echo hello 1  # subjob 1
echo world 2  # subjob 2
echo again 3  # subjob 3
```

Each subjob is essentially another Job object with all the parameters set appropriately for the subjob. You can check each one by using:

```
j.subjobs
j.subjobs(0).peek("stdout")
```

There may be times where you want to split over multiple sets of attributes though, for example the args and the env options in the Executable application. This can be done with the multi_attrs option that takes a dictionary with each key being the attribute values to change and the lists being the values to change. Give the following a try:

```
j = Job()
j.splitter = GenericSplitter()
j.splitter.multi_attrs = {'application.args': ['hello1', 'hello2'],
                        'application.env': [{'MYENV':'test1'}, {'MYENV':'test2'}]}
j.submit()
```

This will produce subjobs with the exe and environment:

```
echo hello1 ; MYENV = test1  # subjob 1
echo hello2 ; MYENV = test2  # subjob 2
```

### 1.7.2 GangaDatasetSplitter

The GangaDatasetSplitter is provided as an easy way of splitting over a number input data files given in the inputdata field of a job. The splitter will create a subjob with the maximum number of file specified (default is 5). A typical example is:

```
j = Job()
j.application.exe = 'more'
j.application.args = ['__GangaInputData.txt__']
j.inputdata = GangaDataset( files=[ LocalFile('*.txt') ] )
j.splitter = GangaDatasetSplitter()
j.splitter.files_per_subjob = 2
j.submit()
```

If you check the output you will see the list of files that each subjob was given using j.subjobs() as above.

### 1.8 PostProcessors

Ganga can be instructed to do many things after a job completes. Each object can be added to the postprocessors field of the Job object and they will be carried out in order. The available Post-Processing options are detailed below:

#### 1.8.1 Mergers

A merger is an object which will merge files from each subjobs and place it the master job output folder. The method to merge depends on the type of merger object (or file type). For example, if each subjob produces a root file ‘thesis_data.root’ and you would like this to be merged you can attach a RootMerger object to your job:
When the job is finished this merger object will then merge the root files and place them in \texttt{j.outputdir}. The \texttt{ignorefailed} flag toggles whether the merge can proceed if a subjob has failed. The \texttt{overwrite} flag toggles whether to overwrite the output if it already exists. If a merger fails to merge, then the merger will fail the job and subsequent postprocessors will not run. Also, be aware that the merger will only run if the files are available locally, Ganga won’t automatically download them for you (unless you use Tasks) to avoid running out of local space. You can always run the mergers separately though:

\begin{verbatim}
j.postprocessors[0].merge()
\end{verbatim}

There are several mergers available:

\textbf{TextMerger}

\begin{verbatim}
TextMerger(compress = True)
\end{verbatim}

Used for merging .\texttt{txt}, ‘.\texttt{log}’, etc. In addition to the normal attributes, you can also choose to compress the output with

\textbf{RootMerger}

\begin{verbatim}
TextMerger(compress = True)
\end{verbatim}

Used for root files. In addition to the normal attributes, you can also pass additional arguments to hadd.

\textbf{CustomMerger}

A custom merger where you can define your own merge function. For this merger to work you must supply a path to a python module which carries out the merge with

\begin{verbatim}
CustomMerger().module = '~/mymerger.py'
\end{verbatim}

In \texttt{mymerger.py} you must define a function called merge(file_list,output_file), e.g:

\begin{verbatim}
import os
def mergefiles(file_list,output_file):
    f_out = file(output_file,'w')
    for f in file_list:
        f_in = file(f)
        f_out.write(f_in.read())
        f_in.close()
    f_out.flush()
    f_out.close()
\end{verbatim}

This function would mimic the TextMerger, but with more control to the user. Note that the \texttt{overwrite} and \texttt{ignorefailed} flags will still work here as a normal merger object.

\textbf{SmartMerger}

The final merger object which can be used is the \texttt{SmartMerger()}, which will choose a merger object based on the output file extension. It supports different file types. For example the following SmartMerger would use a RootMerger for ‘thesis_data.root’ and a TextMerger for ‘stdout’.
SmartMerger(files = ['thesis_data.root','stdout'],overwrite = True)

Note that:

j.postprocessors.append(SmartMerger(files = ['thesis_data.root','stdout'],overwrite = True))

is equivalent to doing:

j.postprocessors.append(TextMerger(files = ['stdout'],overwrite = True))
j.postprocessors.append(RootMerger(files = ['thesis_data.root'],overwrite = False))

However in the second instance you gain more control as you have access to the Root/TextMerger specific attributes, but at the expense of more code. Choose which objects work best for you.

### 1.8.2 Checkers

A checker is an object which will fail otherwise completed jobs based on certain conditions. However, if a checker is misconfigured the default is to do nothing (pass the job), this is different to the merger. Currently there are three Checkers:

**FileChecker**

Checks the list of output files and fails job if a particular string is found (or not found). For example, you could do:

```python
fc = FileChecker(files = ['stdout'], searchStrings = ['Segmentation'])
```

You can also enforce that your file must exist, by setting `filesMustExists` to `True`:

```python
fc.filesMustExist = True
```

If a job does not produce a stdout file, the checker will fail the job. This FileChecker will look in your stdout file and grep the file for the string ‘Segmentation’. If it finds it, the job will fail. If you want to fail the job a string doesn’t exist, then you can do:

```python
fc.searchStrings = ['SUCCESS']
f.$$c.failIfFound = False
```

In this case the FileChecker will fail the job if the string ‘SUCCESS’ is not found.

**RootFileChecker**

This checks that all your ROOT files are closed properly and have nonzero size. Also checks the merging procedure worked properly. Adding a RootFileChecker to your job will add some protection against hadd failures, and ensure that your ROOT files are mergable. If you do:

This checker will check that each ROOT file has non-zero file size and is not a zombie. If you also have a merger, it will check the output from hadd, ensure that the sum of the subjob entries is the same as the master job entries, and check that each ROOT file has the same file structure. RootFileChecker inherits from FileChecker so you can also ensure that the ROOT files must exist.
CustomChecker

This is probably the most useful checker and allows the user to use private python code to decide if a job should fail or not. The CustomChecker will execute your script and fail the job based on the output. For example, you can make a checker in your home directory called 'mychecker.py'. In this file you must define a function called check(j), which takes in your job as input and returns True (pass) or False (fail)

```python
import os

def check(j):
    outputfile = os.path.join(j.outputdir, 'thesis_data.root')
    return os.path.exists(outputfile)
```

Then in ganga do:

```python
cc = CustomChecker(module = '~/mychecker.py')
```

This checker will then fail jobs which don’t produce a ‘thesis_data.root’ root file.

1.8.3 Notifier

The notifier is an object which will email you about your jobs upon completion. The default behaviour is to email when master jobs have finished and when subjobs have failed. Emails are not sent upon failure if the auto-resubmit feature is used. Important note: Emails will only be sent when ganga is running, and so the Notifier is only useful if you have ganga running in the background (e.g. screen session, GangaService). To make a notifier, just do something like:

```python
n = Notifier(address = 'myaddress.cern.ch')
```

If you want emails about every subjob, do

```python
n = Notifier(address = 'myaddress.cern.ch')
```

1.8.4 Management of post processors with your job

You can add multiple post processors to a Job and Ganga will order them to some degree. Mergers appear first, then checkers, then finally the notifier. It will preserve the order within each class though (e.g. The ordering of the #checkers is defined by the user). To add some postprocessors to your job, you can do something like

```python
tm = TextMerger(files=['stdout'], compress=True)
rm = RootMerger(files=['thesis_data.root'], args='-f6')
fc = FileChecker(files=['stdout'], searchStrings=['Segmentation'])
cc = CustomChecker(module='~/mychecker.py')
n = Notifier(address='myadress.cern.ch')

j.postprocessors = [tm, rm, fc, cc, n]
```

or:

```python
j.postprocessors.append(fc)
j.postprocessors.append(tm)
j.postprocessors.append(rm)
j.postprocessors.append(cc)
j.postprocessors.append(n)
```

You can also remove postprocessors:
In [21]: j.postprocessors
Out[21]: [SmartMerger {
    files = [],
    ignorefailed = False,
    overwrite = False
}, FileChecker {
    files = [],
    checkSubjobs = False,
    searchStrings = [],
    failIfFound = True
}, Notifier {
    verbose = False,
    address = ''
}]

In [22]: j.postprocessors.remove(FileChecker())

In [23]: j.postprocessors
Out[23]: [SmartMerger {
    files = [],
    ignorefailed = False,
    overwrite = False
}, Notifier {
    verbose = False,
    address = ''
}]

1.9 Miscellaneous Functionality

Ganga provides quite a lot of additional functionality to help with job management. Below are the main ones:

1.9.1 Job Templates

If there is a version of a job that you use a lot, it can be beneficial to store this as a job template and then you can easily retrieve and then only alter a few parameters of. To create a template you do exactly what you would do for a normal job except you create a `JobTemplate` object instead of a `Job` object:

```python
j = JobTemplate()
j.name = 'LsExeLocal'
j.application.exe = 'ls'
j.backend = Local()
```

To view the templates available, just do:

```python
templates
```

You can then create a job from this template by doing:

```python
j = Job(templates[0], name = 'JobFromTemplate')
j.submit()
```
1.9.2 Job Trees

As you submit more jobs of different types, it can become quite difficult to keep track of them. Ganga supports a ‘directory tree’ like structure for jobs so you can easily keep track of which jobs are associated with different calibrations, analyses, etc. Jobs are stored by id and can be thought of as soft links to the main Ganga Job Repository.

```python
# show the current job tree (empty to start with)
jobtree

# make some dirs and subdirs
jobtree.mkdir('test_old')
jobtree.mkdir('test')
jobtree.mkdir('prod')
jobtree.mkdir('/test/jan')
jobtree.mkdir('/prod/full')

# have a look at the tree
jobtree.printtree()

# remove a dir
jobtree.rm('test_old')

# create some jobs and add them
jobtree.cd('/test/jan')
jobtree.add(Job())
jobtree.cd('/prod/full')
jobtree.add(Job())
jobtree.add(Job())

# look at the tree again
jobtree.printtree()

# submit the some jobs
jobtree.getjobs().submit()
```

1.9.3 GangaBox

1.10 Queues

Many tasks in Ganga can take a lot of time from job submission to output download. Several things are already handled in the background by the Monitoring System, but you may have user tasks that you want to also push into the background that can run in parallel. This is where queues can be used.

To start with, you can view the state of the background threads by just typing `queues`:

```
Ganga In [38]: queues
Ganga Out [38]:
Ganga user threads:                      Ganga monitoring threads:            
 Name  Command  Timeout     |  Name  Command  Timeout
 ----  ------   -------     |    ----  ------   -------
User_Worker_0  idle      N/A        |  GangaWorker_0  idle
User_Worker_1  idle      N/A        |  GangaWorker_1  idle
User_Worker_2  idle      N/A        |  GangaWorker_2  idle
```

Ganga user queue:
To add a function call to the queue such as a submit call, do the following:

```python
for i in range(1, 10):
    j = Job()
    queues.add(j.submit)
```

You can also supply your own functions as well as provide arguments to these functions:

```python
def f(x):
    print x
queues.add(f, args=(123,))
```

Queues can also be used to submit subjobs in parallel:

```python
j = Job()
j.splitter = GenericSplitter()
j.splitter.attribute = 'application.args'
j.splitter.values = [i for i in range(0, 10)]
j.parallel_submit = True
j.submit()
```

## 1.11 Tasks

### 1.11.1 Introduction to Tasks

Even with Ganga, you can find that you may find managing a large set of jobs and steps in an analysis to be a bit cumbersome. The GangaTasks package was developed to help with these larger scale analyses and remove as much of the ‘busy work’ as possible. It can automatically submit jobs to keep a set number running, it can create new jobs when others complete and chain their data together, it can automatically transfer data around as required and a number of other things as well. As with all of Ganga it is based on the plugin system and so you can easily extend some elements of it to better suit your requirements.

GangaTasks essentially adds 3 new objects that control all aspects of the overall task:

- **Task** This is overall ‘container’ for the steps in your analysis. It is fairly light weight but is used to aggregate the overall status of the task and control overall settings, numbers of jobs, etc.

- **Transform** This is where most things occur. It is in some ways analogous to a Job Template in that it mostly contains the objects that will be assigned to the created jobs. This is where new Units are created and data is transferred between steps. You will generally have a Transform per ‘step’ or ‘type’ of job that you want to run.

- **Unit** This is the ‘control class’ for any created jobs and contains all the job-specific information (e.g. input data, application settings, etc.) that each actual Job will be setup with. After all the units in a Transform are created, each unit then creates a new Job and attempts to submit it. It will monitor the status of the job and will do any necessary actions (e.g. download output data) upon completion. If the job fails and it seems sensible to do so, it will also resubmit or recreate the job.
A typical example of how this structure works would be in a two stage analysis where you generate some MC in the first step and then run some analysis code on the output of this data. You would create an overall Task to manage both steps. Each step would have an associated Transform with the first being setup as MC generation and the second doing the analysis. You would set the input data of the second transform to be the output data of the first. Then, while running your Task, Units will be created to cover the number of events you wanted to create and jobs will be submitted for each of these. As these complete new units and jobs will be created by the analysis Transform to cover that step.

### 1.11.2 Basic Core Usage

It’s quite likely you will want to develop your own plugins to maximise your use of GangaTasks, however there is a set of generalised classes that can get you started. Typical use of these is shown below:

```python
# First create the overall Task
t = CoreTask()

# Now create the Transform ( -> Job template)
trf = CoreTransform()
trf.application = Executable()
trf.backend = Local()

# Set the unit splitter (unique to CoreTransform - you may have better ways of creating units in your plugins). This will create a unit based on the splitting of any given splitter
# If you put in your own splitter here, use the trf.fields_to_copy string list to tell Tasks which fields of a Job to preserve from the split. Here, Tasks already knows about GenericSplitter and knows that we want to change the 'application' object for each Unit/Master Job
trf.unit_splitter = GenericSplitter()
trf.unit_splitter.attribute = "application.args"
trf.unit_splitter.values = ['arg 1', 'arg 2', 'arg 3']

# Append the transform
t.appendTransform(trf)

# set the maximum number of active jobs to have running (allows for throttling)
t.float = 100

# run the Task
t.run()
```

After running the above commands you won’t see much happen initially as Tasks runs on a separate monitoring loop that triggers every 30s (configurable in `~/.gangarc`). Eventually though you will see the units created and then jobs for each of these units will be submitted. To see the progress of your tasks use:

```bash
tasks
tasks(0).overview()
```

Tasks can also take advantage of using queues for submission as well. Simply add:

```python
# note - done at the transform level rather than task level as different backends may not need it
trf.max_active_threads = 10  # optional - specifies the max number of submissions to queue up
trf.submit_with_threads = True
```

### 1.11.3 Job Chaining

The Tasks package also allows you to chain jobs together, i.e. have the output of one job be the input of another. This is done by setting the input data of the dependant Transform to be `TaskChainInput` type and giving the
ID of the Transform is depends on. You can have multiple transforms feed into one Transform by specifying more TaskChainInput datasets.

A typical example is shown below:

```bash
# Create a test script
open('my_script3.sh', 'w').write("#!/bin/bash
echo $PATH
ls -l
more __GangaInputData.txt__
echo "MY TEST FILE" > output_file.txt
sleep 120
"

# Create the parent task
t = CoreTask()

# Create the first transform
trf1 = CoreTransform()
trf1.application = Executable()
trf1.application.exe = File('my_script3.sh')
trf1.outputfiles = [LocalFile("*.txt")]
d = GangaDataset()
d.files = [LocalFile("*.txt")]
d.treat_as_inputfiles = True
trf1.addInputData(d)
trf1.files_per_unit = 1
trf1.submit_with_threads = True

trf1.splitter = GangaDatasetSplitter()
trf1.splitter.files_per_subjob = 2

trf1.backend = Local()
t.appendTransform(trf1)

# Create the second transform
trf2 = CoreTransform()
trf2.application = Executable()
trf2.application.exe = File('my_script3.sh')
trf2.submit_with_threads = True

d = TaskChainInput()
d.input_trf_id = trf1.getID()
trf2.addInputData(d)

trf2.splitter = GangaDatasetSplitter()
trf2.splitter.files_per_subjob = 2

trf2.backend = Local()
t.appendTransform(trf2)

# Set the Task running
t.float = 1
t.run()
```

Sections still to be added:

- Pure Python Usage
- Ganga as a Service
• Developing a New Application
The Ganga Public Interface provides a wrapper around the internals of the Python implementation to provide safety and to increase ease-of-use.

Sections still to be added:

- Pure Python Usage
- Ganga as a Service
- Developing a New Application
Guide for System Administrators

This section of the manual is intended for system administrators or interested individuals to describe how to install and manage Ganga.

3.1 Installation

Historically Ganga was installed via a custom `ganga-install` script which would fetch the latest version and its dependencies. We have since migrated away from that and there are two primary ways to get access to Ganga, one of which is mostly of interest only to particle physicists.

3.1.1 pip

At its simplest it is possible to install ganga using the standard Python `pip` tool with a simple

```
pip install ganga
```

3.1.2 CVMFS

CVMFS is a read-only file system intended for distributing software originally developed for the CERN virtual machine infrastructure.

```
/cvmfs/ganga.cern.ch/
```

3.2 Site config

It’s often the case that you want to specify default configuration settings for your users, perhaps on a group-by-group basis. You can do this by placing `.gangarc`-style INI files in a common directory on your system and pointing Ganga at it. The order of precedence for a particular setting goes `default → site config → user config → runtime setting` with those later in the chain overriding those earlier. The location that Ganga looks for the site config is controlled with an environment variable, `GANGA_SITE_CONFIG_AREA`, which you could set in your users’ default shell setup.

```
GANGA_SITE_CONFIG_AREA=/some/physics/subgroup
```

Files in this directory should be named after the Ganga version that you want to affect. They should start with the version number with the . replaced with - and can have any extension. So if you have three config files:
$ ls $GANGA_SITE_CONFIG_AREA
6-0-44.ini  6-1-6.ini  6-1-10.ini

and the user is running Ganga 6.1.6 then 6-0-44 and 6-1-6 will be loaded and 6-1-10 will be ignored.
This document is intended to detail some of the inner workings of Ganga to both document what we have done as well as make it easier for new developers to get on-board quicker.

### 4.1 GangaObject

At the core of a lot of Ganga is GangaObject. This is a class which provides most of the core functionality of Ganga including persistency, typed attribute checking and simplified construction.

**Note:** There is currently some work being done to replace the existing implementation of GangaObject with a simpler version. The user-facing interface should not change at all but more modern Python features will be used to simplify the code. This will also affect how schemas are defined but not how they are presented or persisted.

### 4.2 Schema

The schema of a GangaObject defines the set of attributes belonging to that class along with their allowed types, access control, persistency etc. Each GangaObject must define a schema which consists of a schema version number and a dictionary of Items. Schema items must define their name and a default value and can optionally define a lot more such as a list of possible types and documentation string.

### 4.3 Proxy objects

In order to provide a nice interface to users, Ganga provides a Ganga Public Interface which fulfils two main purposes. Firstly it is a reduced set of objects so that the user is not bombarded with implementation details such as Node. Secondly, all GangaObjects available through the GPI are wrapped in a runtime-generated class called a proxy.

These proxy classes exist for a number of reasons but primarily they are there for access control. While a GangaObject can has as many functions and attributes as it likes, only those attributes in the schema and those methods which are explicitly exported will be available to users of the proxy class.

When working on internal Ganga code, you should never have to deal with any proxy objects at all. Proxies should be added to objects as they are passed to the GPI and should be removed as they are passed back.
4.3.1 Attributes on proxy objects

Proxy classes and the object that they are proxying have a set number of attributes which should be present. If an object inherits from GangaObject the class can have the property _proxyClass set which will point to the relevant GPIProxyObject subclass. This is created on demand in the addProxy and GPIProxyObjectFactory methods. The proxy class (which is a subclass of GPIProxyObject and created using GPIProxyClassFactory()) will have the attribute_impl set to be the relevant GangaObject subclass.

When an instance of a proxy class is created, the_impl attribute of the instance will point to the instance of the GangaObject that is being proxied.

4.4 Repository

A repository is the physical storage of data on disk (usually persisted GangaObjects) as well as library interface to it.

4.5 Registry

A registry is an in-memory data-store which is backed by a repository.

4.6 Job monitoring

4.7 IGangaFile

All file types as of Ganga 6.1 inherit from IGangaFile. This main exception to this is the File object which as of 05/05/2016 is used as it still has more features than the IGangaFile inherited classes do.

<table>
<thead>
<tr>
<th>Script Generator</th>
<th>When is it used?</th>
</tr>
</thead>
<tbody>
<tr>
<td>getWNScriptDownload-Command</td>
<td>This generates a script which will make the file accessible from the WN when the job starts running</td>
</tr>
<tr>
<td>getWNInjectedScript</td>
<td>This generates a script which will send the file to the remote directory from the WN with no client intervention</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Special attr</th>
<th>Use/Doc</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>lfn</td>
<td>Unique to the DiracFile. This is the LFN of the file in the DFC</td>
<td>str</td>
</tr>
<tr>
<td>getReplicas</td>
<td>Unique to DiracFile returns a list of SE where the file is replicated</td>
<td>list of str</td>
</tr>
<tr>
<td>'<em>list_get__match</em>_()'</td>
<td>IGangaFile, performs a type match on file objects. can we remove this?</td>
<td>bool</td>
</tr>
</tbody>
</table>
The documentation on these pages is automatically generated from the source code. It provides an overview of all the Python objects in the code base and as such is intended for Ganga core and extension developers. If you are a user of Ganga then you probably want the GPI documentation.

All Ganga modules