# PALFA Pipeline2.0 Documentation

Patrick Lazarus, Alex Samoilov, ...
plazar@physics.mcgill.ca, alex@sequencefactory.com

February 17, 2011

**Abstract**

## 1 Installation

### 1.1 Dependencies

The PALFA pipeline has various dependencies. All of the following packages are required to run the pipeline.

- PRESTO (https://github.com/scottransom/presto)
- psrfits_utils (be sure to pull Kevin Stovall's version of psrfits_utils. It contains the merging code required to analyse PALFA Mock spectrometer data. https://github.com/kstovall/psrfits_utils.git)
- numpy (http://numpy.scipy.org/)
- M2Crypto (http://pypi.python.org/pypi/M2Crypto)
- PBSQuery (http://subtrac.sara.nl/oss/pbs_python/wiki/TorqueInstallation)
- pyfits (http://www.stsci.edu/resources/software_hardware/pyfits)
- pyodbc (http://pypi.python.org/pypi/pyodbc/)
- suds (http://pypi.python.org/pypi/suds)

*NOTE: Some of these dependencies have requirements of their own. For example, PRESTO requires cfitsio, TEMPO, numpy, scipy, etc. Be sure to follow the installation instructions for each packages.*

## 1.2 Getting Started

Here we will present basic step-by-step instructions for setting up the pipeline.

**Step 1** Create a directory where you want the pipeline to be installed.

**Step 2** Download the pipeline source files. This should be done by cloning the git repository on github.com: https://github.com/plazar/pipeline2.0.

```
$ git clone git://github.com/plazar/pipeline2.0.git
```

*NOTE: This will create a sub-directory called "pipeline2.0".*

**Step 3** Add the pipeline directory to your PYTHONPATH environment variable.

**Step 4** Create configuration files using the examples provided. Modify the settings so they are appropriate for your system. Run the `sanity_check.py` script to ensure all your configurations have the correct types, the neccessary directories exist and you have the correct privileges for files that must be read/written.

**Step 5** Set up the database. This is done using the `create_database.py` script.

**Step 6** Start the downloader using `StartDownloader.py`.

*NOTE: The downloader does not need to be run on the same computer as the job pooler. However, the directory where the downloader saves files must be accessible from the computer where the job pooler is being run.*

Alternatively, it is possible to add files to the job-tracker database without using the downloader. To do this use `add_files.py`. Be sure to set the `delete_rawdata` configuration to `False` if you do not want raw data files to be deleted when the pipeline no longer requires them.

**Step 7** Start the job pooler using `StartJobPool.py`. The job pooler will start submitting jobs when data files are finished downloading.

**Step 8** Start the uploader using `StartJobUploader.py`. The uploader will upload results to the common database when jobs are successfully processed.

# 2 The Pipeline

An overview of the pipeline.

## 2.1 Features

- Sanity check of configurations
- Dynamic zapping
- Automatic download of data files
- File size checks when downloading
- Error emails
- Automatic (configurable) retry of failed jobs
- Check of results before uploading
- Automatic (configurable) deletion of data files (upon success or terminal failure)

## 2.2 Components

### 2.2.1 Job-tracking Database

Each job run through the pipeline passes through multiple stages before it is completed. To track the state of each job, and log its history a SQLite3 database is used. The database is simply a (readable/writable) binary file formatted according to the SQLite3 format. The database file can be access using SQL statements using python's sqlite3 module, which is part of python's standard library, as of python version 2.5.

Relevant links:

- http://docs.python.org/library/sqlite3.html
- http://www.sqlite.org/

The job-tracker database governs all aspects of the pipeline. It maintains lists of requests for data, downloads, jobs, files associated with each job, processing attempts, and upload attempts.

Details for each table in the database are presented in Section 3.4.

#### 2.2.2 Downloader

#### 2.2.3 Job Pooler

#### 2.2.4 Uploader

##### Headers

##### Candidates

**Diagnostics** Diagnostics stored in the common database are split into two types: Numeric values and binary data. Some of the "binary data" diagnostics are actually plain text files.

The following per-beam diagnostics are uploaded to the common database:

- *RFI mask percentage* (Numeric value)
  Percentage of data masked due to RFI.
- *Num cands folded* (Numeric Value)
  The number of candidates folded.
- *Num cands produced* (Numeric Value)
  The total number of candidates produced, including those with sigma lower than the folding threshold.
- *Min sigma folded* (Numeric value)
  The smallest sigma value of all folded candidates from this beam.

- *Num cands above threshold* (Numeric value)
  The number of candidates produced (but not necessarily folded) that are above the desired sigma threshold.
- *RFIfind png* (Binary data)
  Output image produced by rfifind in png format.
- *Accelcands list* (Binary data)
  The combined and sifted list of candidates produced by accelsearch.
- *Zaplist used* (Binary data)
  The list of frequencies and ranges zapped from the power spectrum before searching this beam.

### 2.3 Objects Tracked

**Restores**

**Downloads**

**Jobs**

**Uploads**

## 3 Reference

### 3.1 Executables

`StartDownloader.py` A script to start the downloader background process.
usage: `StartDownloader.py`

`StartJobPool.py` A script to start the job pooler background process.
usage: `StartJobPool.py`

`StartJobUploader.py` A script to start the results uploader background process.
usage: `StartJobUploader.py`

## 3.2 Generic Queue Manager Interface

The Pipeline interacts with the system's queue manager using an abstract interface. This generic interface can be implemented with the specific queue manager installed on the system. The system-specific queue manager class must be derived from the `PipelineQueueManager` class. Each of the following methods of `PipelineQueueManager` must be implemented in the derived class:

**submit** Submits a job to queue manager with files list and output directory. Must return a unique string identifier for the submitted job. Raises ValueError if job was not submitted by queue manager.

**is_running** Given a unique identifier for a job, must return True or False whether the job is running or not, respectively.

**is_processing_file** Given a string path of a file, return True or False whether or not the job processing the input filename is running.

**delete** Given a unique identifier for a job, remove the corresponding job from the queue. Returns True if the job was delete, False if an error occured.

**status** Return the status of the queue, i.e. the number of jobs running and the number of jobs queued.

**had_errors** Given a unique identifier for a job, return True, or False whether the job terminated with an error, or not, respectively.

## 3.3 Configurations

## 3.4 Job-Tracker Database Details

The job-tracker database contains seven tables: requests, downloads, download_attempts, jobs, job_files, job_submits, and job_uploads. Each of these tables is meant to track a particular object (e.g. jobs), or events (e.g. job submissions to the queue).

A description of each database table will now be presented:

`requests` The `requests` table tracks requests for data restores. An entry in the table is inserted when a request is made. Each entry contains the following data:

**id** (UNIQUE INTEGER) A unique identifier, automatically assigned by the database upon creation of an entry.

**size** (INTEGER) Total size of the files restored, in bytes.

**guid** (TEXT) A globally unique identifier for the restore. This serves as the directory name on the FTP server where the restored files will be placed.

**status** (TEXT) The current status of the request.

**details** (TEXT) Details about the request.

**created_at** (TEXT) Date and time when the request entry was created.

**updated_at** (TEXT) Date and time when the request entry was last updated.

The possible status values for requests are:

**waiting** ...