Reprocessing Task Structure

Reprocessing task structure

1/11/2012 **Under Construction**

TASK FUNCTION

Task Overview

Each Fermi data run corresponds to a single top-level stream of this task.

The generic task consists of seven job steps. One top-level stream reprocesses one Fermi run. Because each Fermi run typically contains millions of events, the reprocessing is spread across multiple CPUs by breaking the run into fixed-length 'clumps'. Each clump is reprocessed independently; the primary output (typically ROOT files) is finally merged. Secondary data products (e.g., FITS files) are created after the merging of ROOT files.

The task structure is fairly simple including various short bookkeeping steps, and two 'heavy lifters': processClump.py runs Gleam; 'mergeClumps.py' merges run fragments and produces any final data products. Depending on the task configuration, one or both of these two job steps may be substantial (xlong queue) or trivial (short queue).

To date, reprocessing instances have been accomplished using two independent consecutive pipeline tasks. The first task runs Gleam and produces all desired ROOT files. The second task reads in MERIT files and produces all desired FITS files. This task separation is done to better match the needs of a reprocessing life-cycle most efficiently: task code development, data validation, IRF development, and resource management while the tasks are running.

top-level stream	sub-stream	type	primary function	
setupRun		py	• discover run number, input files; calculate # parallel substreams	
createClumps		jy	• create substreams	
	processClump	py	• Gleam reprocessing of run fragment (clump)	
	clumpDone	jy	• no-op	
setupMerge		jy	• collect data from Pipeline II DB and write to file	
mergeClumps		py	• merge output from processClump; create post-merge data products	
runDone		jy	• register datasets; update processing history DB	

Task Details

setupRun.py

- read & parse 'runList' file
- identify run# and input files for this stream
- calculate #clumps (substreams in subtask)
- create env-vars for subtask

createClumps.jy

- read env-vars
- create task-level pipeline-vars for subTask
- create subStreams (with pipeline-var list)

[subtask] processClump.py (all parallel analysis processing)

- unpack pipeline-vars (as env-vars)
- define skipEvents/lastEvent for Gleam
- [limit # events processed]
- infile staging [disabled]
- infile slicing (skimmer) [disabled]
- infile env-vars for Gleam
- construct output filenames

- output file staging
- set output file env-vars for Gleam
- prepare .rootrc
- select and stage-in FT2 file
    - (Gleam setup)
- select jobOptions file and customize
- Run Gleam
    - (SVAC setup)
- Run svac [disabled]
- Run makeFT1 [disabled]
- Finalize staging
- create new subTask-level pipeline-vars with clump output info

[subtask] clumpDone.jy

- (nothing!)

setupMerge.jy

- unpack task-level pipeline variables
- create new task-level pipeline vars from subTask vars
- create two pipeline files: pipelineVarList.txt and clumpFileList.txt

mergeClumps.py

- open and read the two pipeline files, store in dicts
- stage in FT2 file
    - (merge files)
- create tool-specific lists of files to be merged
- loop over all file types to be merged
- construct new output file name (with proper version)
- if # files to be merged == 1, just use 'cp'
    - skimmer merge
- stage-out output file
- run skimmer
    - FT1 merge
- stage-out output file
- run fmerge
    - HADD merge (histograms)
- stage-out output file
- run hadd
    - (post-merge data product generation)
- stage-in input MERIT file, if necessary
    - FT1
- generate new output file name (wth proper version)
- stage-out output file
- runMakeFT1()
    - ELECTRONFT1
- generate new output file name (wth proper version)
- stage-out output file
- runMakeFT1()
    - LS1
- generate new output file name (wth proper version)
- stage-out output file
- runMakeLS1()
    - FILTEREDMERIT
- generate new output file name (wth proper version)
- stage-out output file
- run skimmer
    - ELECTRONMERIT
- generate new output file name (wth proper version)
- stage-out output file
- run skimmer
- Finalize staging
- Produce list of files for dataCat registration

runDone.jy

- Unpack task-level pipeline vars
- Register (merged) output file in dataCat
- Make entry in HISTORYRUNS DB table

CODE

Directories

Primary script and configuration directories

/nfs/farm/g/glast/u38/Reprocessing-tasks/P201-ROOT /config	task-specific scripts and config files for ROOT file reprocessing
/nfs/farm/g/glast/u38/Reprocessing-tasks/P201-FITS/config	task-specific scripts and config files for FITS file reprocessing
/nfs/farm/g/glast/u38/Reprocessing-tasks/commonTools	task-independent scripts and config files available to all tasks

Other code dependencies

/afs/slac/g/glast/ground/PipelineConfig/GPLtools/GPLtools-02-00-00/	common pipeline tools
/afs/slac/g/glast/ground/PipelineConfig/python/@sys/bin/python	Fermi installation of python
/nfs/farm/g/glast/u52	Location of SCons GlastRelease builds
/nfs/farm/g/glast/u35	Location of SCons ScienceTools builds
/afs/slac/g/ki/software	KIPAC Ftools installation
/afs/slac/g/glast/applications	General Fermi tools (incl. xroot
/afs/slac/g/glast/ground/GLAST_EXT	Fermi (GLAST) externals

commonTools

/nfs/farm/g/glast/u38/Reprocess-tasks/commonTools/00-01-00

findRunsRepro.py*		search dataCatalog (used to generate run list)
checkRunList.py*		check/summarize run list(s)
envCtl.py*		class for env-var management
	envCtl	constructor
	genEnv()	generate target environment
	diffEnv()	determine diffs between original and traget envs
	setEnv()	set target environment
	restoreInitEnv()	restore initial environment
	storeEnv()	store environment to disk (pickle)
	loadEnv()	load environment from disk (pickle)
	dumpEnv()	print all environment variables
svs.py*		SCons variant string generator
pickleEnv.py*		wrapper for python setup scripts (pickles resultant env)
pickleEnv.sh*		wrapper for bash setup scripts (pickles resultant env)
repTools.py		miscellaneous functions available to all repro tasks
	makeXrootFilename()	generate an repro file Xroot filename
	makeXrootScratchFilename()	generate a scratch filename (for clump files)
	getKey()	get info from FITS file
	findFt2()	query dataCat for appropriate FT2 file
	getFile()	
	getCurrentVersion()	determine current version of specified file
	fileOpen()	
	runMakeFT1()	run makeFT1 and, optionally, gtdiffrsp and gtmktime
	rcCheck()	return code management (check if any rc in list != 0)
	rcGood()	check if rc in list and == 0

	rcDump()	print all rc's in list
	modeDump()	dump 'mode' dictionary
ft1skim.py*		wrapper to run ft1skimmer (TonyJ) (OBSOLETE?)
runFT1skim.sh		wrapper to ft1skim.py (OBSOLETE?)
runMakeFT1.sh*		wrapper to run makeft1/gtdiffrsp/gtmktime
runSvac.sh*		wrapper to run SVAC tuple generation code (OBSOLETE)
runWrapper.sh*		(old) wrapper to run Gleam (CMT version)
setupFtools.sh*		create environment to run FTools
setupGR.sh*		create environment to run Gleam (GR)
setupOracle.sh*		create environment for Oracle
setupSkimmer.py		create environment to run TSkim
setupST.sh*		create environment to run ScienceTools
setupXroot.sh*		create environment to run xroot tools
trickle.py		class to control rate of task stream creation
.xrayrc		FTools setup config file

task-specific code

Task preparation	
taskConfig.xml	task definition
genRunFile.csh*	generate list of input files for reprocessing
Pipeline code	
envSetup.sh*	set up environment to run GR/ST/FT/etc (called by pipeline)
config.py	task configuration (imported by all .py)
setupRun.py*	setup for reprocessing a single run
createClumps.jy	create subprocess for processing a "clump" (part of a run)
processClump.py*	process a clump of data
clumpDone.jy	cleanup after clump processing
setupMerge.jy	setup for merging clumps
mergeClumps.py*	merge all clumps for single run
runFT1skim.sh*	skim FT1 events
runDone.jy	final bookkeeping after run reprocessed (dataCat and runHistory)
commonTools@	link to commonTools
Input data to pipeline code	
doRecon.txt	Gleam job options
fullList.txt	List of reprocessing input data files
removeMeritColumns.txt	List of columns to remove from MERIT files
runFile.txt@	Sym link to fullList.txt

Pipeline control code	
trickleStream.py*	task-specific config for trickle.py

Running environment

Setting up a proper running environment for the many varied applications and utilities needed to perform data reprocessing is an issue to be treated with care. Therefore a short discussion of this topic will be given.

A given release of Fermi code is built for a finite number of operating systems, compiler versions, hardware address size, compiler options (e.g., optimized or debug), build system (CMT or SCons). Over time, the standard location for these builds at SLAC can move about. Closely related but independent of this, SLAC and SCCS supports slowly evolving set of hardware and software architectures (RHEL5-32, RHEL5-64, RHEL6-64, etc.) and compiler versions (gcc). A system was developed to automate the matching of the best combination of hardware with Fermi software, e.g., there is no available RHEL6-64 build, but a RHEL4-32 build of GlastRelease will run on a RHEL5- or RHEL6- machine.

The first step is setting up an environment to run a particular application or utility, e.g., GlastRelease, ScienceTools, Ftools, Xroot, Oracle, etc. This typically involves defining one or more environment variables and then running a shell script prior to running the compiled executable (in '/exe'). In the case of an SCons build, one can optionally skip the explicit running of the setup shell script by invoking the '/bin' version of the application - which is really a shell script wrapper which then calls the compiled executable. One problem with this implicit shell script wrapper is that one cannot then override selected variables which are set by it. At this point in time, all setup scripts do nothing more than define environment variables.

The approach taken here is to run a setup script in a sub-process, capture all environment variables and store them, along with the original set. When a particular application is invoked, the stored environment variables are defined, the application executed, then the environment restored to its previous condition. In this way, one can easily run, say, Gleam, ScienceTools, Ftools and anything requiring a specific (and, possibly conflicting) setup within the same python script

The current system works only for SCons release builds.

Example:

One must first create the appropriate scripts that setup the desired environment. By convention, all used for reprocessing are named: setup<package>.{sh, py}, e.g., setupGR.sh.

```
import svs
import envCtl

myGR = envCtl.envCtl(envScript='setupGR.sh',name="GR set")   ## Create env-var management object
myGR.genEnv()                                                ## Generate environment for Gleam


...


myGR.setEnv()                          ## Establish Gleam environment
gleamrc = os.system(GleamCmd)          ## Run Gleam
myGR.restoreInitEnv()                  ## Restore initial environment
```