Gemini Fits Storage System Overview

Paul Hirst, 13 December 2011  
phirst@gemini.edu

Introduction 1

Brief Feature Overview 2

FITS Summary Page 2

Calibration Association 3

Human Oriented Calibration Checking Interface 3

Calibration Manager Interface 4

Live metadata matching 4

The calibration association XML 5

Technical overview 5

Database Backend 6

Apache Interface 6

Database Schema 6

File Ingest system and procedure. 11

Calibration Matching 13

# Introduction

The Gemini Fits Storage system was developed starting in 2009 to address a number of operational data handling issues and inefficiencies at Gemini. Essentially, this comprised a catalog of the data stored on the operations data disks, and a catalog of data stored on backup tapes. Subsequently, it was realized that the system formed a good base on which to build other data related services such as Calibration Management, PI and Gemini science staff data notification, and QA metric archiving.

The interface to the system is via the WWW. A Fits Storage installation at Gemini consists of a dedicated virtual machine which runs a PostgreSQL database backend, an apache httpd server and the FitsStorage python code. Various python scripts in the FitsStorage code are called by cron at various intervals to populate and maintain the database. The apache httpd server uses mod\_python to call python code from the FitsStorage package in order to query the database to respond to user queries.

The FitsStorage system simply references the operations data stores which are NFS mounted on the fits storage virtual machines. The system does not provide the storage itself, it simply catalogs the data on the given NFS volume. Currently it does this by polling the directory on the disk to notice new files or files that have changed; a provision exists in the code to use inotify to do this more efficiently, though the NetApp NFS mounted data volumes currently in use do not support this. One FitsStorage system is provided for each data volume – one at the Gemini North telescope (mkofits1), and one at the Gemini South telescope (cpofits1). Within Gemini, DNS aliases are present in the network, such that the hostname ‘fits’ is a CNAME to the local (ie same hemisphere) fits server.

# Brief Feature Overview

Here, we provide an overview of the main features of the system, this is not a complete list by any means. A complete list of the URLs provided by the system is available on the system help page at <http://fits>.

## FITS Summary Page

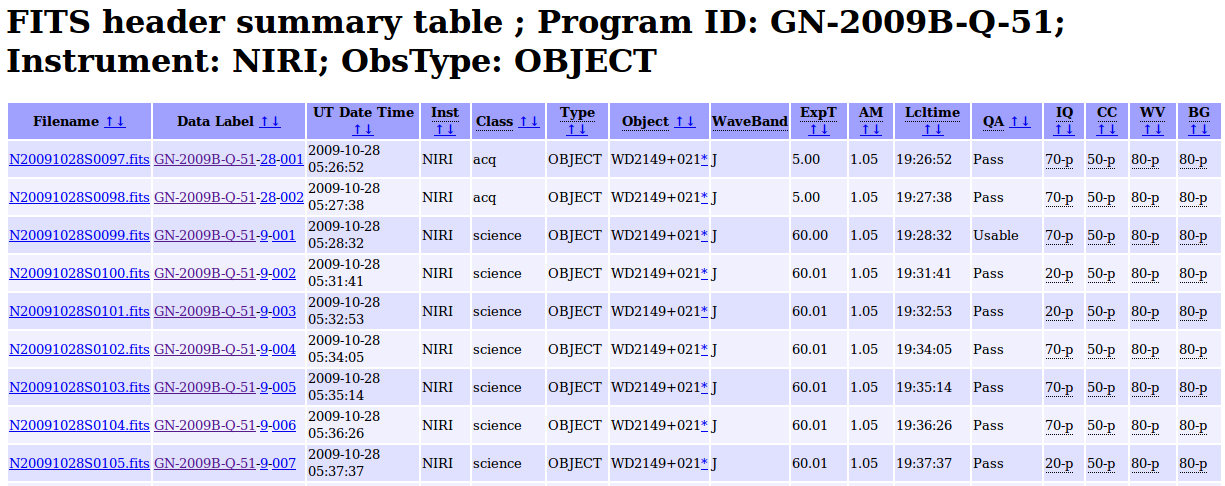
Within Gemini, the real face of the system is the fits file summary page. This allows a user to carry out a search of fits files known to the system and see a summary page of various useful details of those files. Rather than using a web form or similar, the search is specified by simply stringing together search terms as path elements of the URL. The provides a simple, clean and intuitive interface, the user can imagine that they’re simply drilling down into a directory tree.

The fits summary page is found at <http://fits/summary> (by default shows 2500 most recent files)  
Self evident (hopefully) examples of queries that can be constructed include:  
<http://fits/summary/today> (shows data from today only)  
<http://fits/summary/20101120/GMOS/Imaging/science> (shows GMOS Imaging science data from the night of 2010-11-20)  
<http://fits/summary/GN-2009B-Q-51/NIRI/OBJECT> (shows NIRI OBJECT files from that project)

Search terms in the URL can be combined arbitrarily in any order. A current complete list of the selection criteria available is given on the help page at <http://fits> and is currently as follows:

* A date in the form YYYYMMDD, eg 20100401. You can use today and yesterday as special literal words and they will act as you would expect.
* A date range in the form YYYYMMDD-YYYYMMDD, eg 20100401-20100420
* An instrument name, eg GMOS-N. Note GMOS will also work to imply either GMOS-N or GMOS-S
* AO to select only AO data, NOTAO to select only non-AO data
* A program ID, eg GN-2009B-Q-51
* An observation ID, eg GN-2009B-Q-51-9
* A data label, eg GN-2009B-Q-51-28-001
* An ObsType, eg OBJECT
* An ObsClass, eg science
* A filename, eg N20091028S0097.fits. It works if you leave off the .fits too
* A GMOS grating name, eg B600
* A GMOS mask name, eg GN2009BC009-04
* A Calibration Type, eg ARC - this is only useful with the calibration systems
* A Data Reduction state - for example RAW, PREPARED, PROCESSED\_BIAS, PROCESSED\_FLAT
* The words imaging or spectroscopy to only select imaging or spectroscopy data
* A QA state - Pass, Usable, Fail or Win - Win is taken to mean Pass or Usable (ie not Fail)
* The words Present or NotPresent to select only files which are or are not currently present on the storage disk.
* The words canonical or notcanonical. Canonical implies the most recent version of a file.
* The word photstandard to only show files that contain an observation containing a photometric standard in the Gemini list
* The string filter=xxx will only include observations where the filter is xxx

The summary html page looks like this:



## Calibration Association

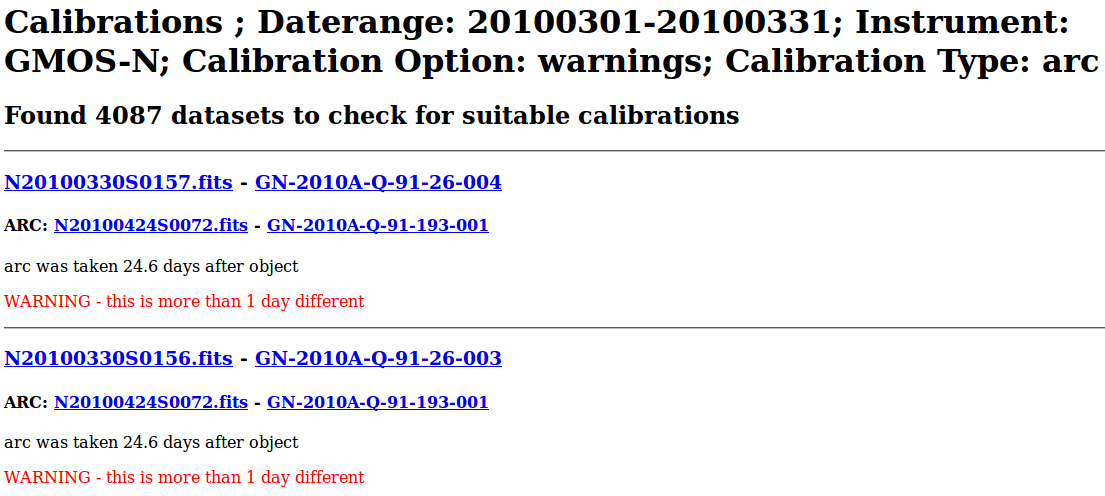
The system provides a calibration management and association feature. Essentially, given a science frame (though in fact any frame that requires calibration) and a calibration type requested, the system is able to automatically choose the best available calibration of the required type to apply to the science frame. In current Gemini operations, this is used in two ways a) with a human-oriented interface by the DA/SOSs to check that required calibrations have been taken with the correct settings, and b) with a machine-oriented calibration manager interface by the QA pipeline in order to select calibration frames to apply as part of pipeline processing. This latter interface will also we used with the Gemini data reduction package to provide automatic calibration selection within an interactive processing environment.

### Human Oriented Calibration Checking Interface

In this mode, the user uses the same selection criteria as detailed above in order to specify which files to check. The system will then go through the files in turn, determine (based on the data types) which calibrations are required for that file, then attempt to find a suitable calibration of each type. By default the system will simply list back each science frame with the calibration files it has matched with it. Extra options can be added to the URL (eg warnings, missing, takenow) which will cause the system to only list calibration matches that generate warnings, cases where the calibration is missing, or cases where it is beneficial to (re-)take the calibration.

This tool is useful for both routine daily checks that all the appropriate calibrations for the previous night of data exist, and also for spot checks for example that all data outstanding calibrations requiring a certain grating have been taken before that grating is removed from the instrument. Operationally, a cron job calls this URL periodically (daily / weekly) and emails the HTML generated to the DA/SOSs.

Example output:



### Calibration Manager Interface

While this feature uses exactly the same calibration matching code as above, the interface is oriented to be machine readable, providing calibration association results in the form of a short XML document. This interface was originally designed for use by the Gemini pipeline system. The principle here is that a client simply requests a given calibration (eg flat field) for a given science frame, and the system responds telling it which flat field to use. The calibration type requested is simply specified as part of the URL. The target data file can be specified in two ways – either as a raw data filename as part of the URL, which the database can look up internally, or by supplying all the metadata needed to carry out the association live over the http connection.

In the former case, a URL such as <http://fits/calmgr/arc/N20100330S0157.fits> (note that is is acceptable to provide a data label rather than a filename if required) will return a calibration association xml document as below.

### Live metadata matching

There is a complication in that the correct association of a processed calibration product can actually depend on the processing history of the target dataset at the point where you wish to apply the calibration. The canonical example of this is in overscan subtraction of GMOS data. Simplistically, the GMOS raw data includes an overscan strip on the edge of each data frame, resulting from ADC samples with the ADC inputs connected to a bias type source rather than actual CCD pixel registers. This can be used as part of the de-biasing procedure during data reduction – a fit is made to the overscan region, which is then subtracted from the entire data frame. The overscan region is then trimmed off the frame and discarded. If this is done for both the science dataset and also the BIAS frames, then the BIAS frames are essentially being used to subtract off the bias structure of the CCDs whereas the overscan region is being used to subtract off the DC offset of the bias, and generally this is the preferred data processing procedure. However, in some situations generally associated with large bright objects, the overscan region of the science frame can become contaminated with spurious signal and cannot be used. In that case one simply does not overscan subtract the BIAS calibration frames either and the BIAS calibration subtraction takes care of both the bias structure and the DC offset – with the latter not being as accurately corrected as if it were being measured from the overscan. The point here is that if you request a processed bias frame for an overscan subtracted science frame, you require an overscan subtracted processed bias frame, where as if you request a processed bias for a non-overscan-subtracted science frame, you require a non-overscan-subtracted processed bias frame.

This is addressed in the Gemini Recipe System, which forms the heart of the Gemini QA pipeline and will also be released as part of the user data reduction package, as follows:

Right at the point in processing where you are ready to carry out a certain type of calibration (de-bias) for example, the recipe system contacts the calibration manager, presents the actual live metadata of the part processed file you wish to calibrate, and requests a calibration match. The calibration manager then replies with a calibration product that is suitable to use on the file in its current state.

If the association is done from a raw data filename rather than from live metadata, then defaults are applied for cases where the processing route is not known.

### The calibration association XML

Calibration associations are returned as a short XML document, the root node of which is the <calibration\_associations> node, containing a <dataset> node for each dataset for which calibrations are being associated. If calibrations are associated with that dataset, the dataset node will contain a <calibration> node for each one. Both the dataset and calibration nodes contain <datalabel>, <filename>, <md5> and <ccrc> nodes. Md5 and ccrc are hash / CRC value of the data file on the fits database. The <calibration> node also contains a <url> node that specifies the URL from which the actual calibration data fits file may be retrieved. Currently these URLs point into the fits storage server itself, though it is anticipated that at some point they may be references to files at the GSA.

Here is an example calibration association XML resulting from a raw data file query using the URL:  
<http://fits/calmgr/arc/GN-2010A-Q-91-26-004> :

<calibration\_associations>  
 <dataset>  
 <datalabel>GN-2010A-Q-91-26-004</datalabel>  
 <filename>N20100330S0157.fits</filename>  
 <md5>c5f05ecac2a798c27e0105848a0657d5</md5>  
 <ccrc>36ea55f1</ccrc>  
 <calibration>  
 <caltype>arc</caltype>  
 <datalabel>GN-2010A-Q-91-193-001</datalabel>  
 <filename>N20100424S0072.fits</filename>  
 <md5>caffd39714fa6345c6a66a3eebefa969</md5>  
 <ccrc>2e2be373</ccrc>  
 <url>http://mkofits1/file/N20100424S0072.fits</url>  
 </calibration>  
 </dataset>  
</calibration\_associations>

# Technical overview

The software written for the Fits Storage project comprises a number of Python modules that interface to both an SQL database and the apache http server. All the database access is done using the SQL Alchemy Object-Relational Mapper (ORM) software. The main FitsStorage.py module basically defines database tables with rows that map to Python objects via the ORM, and that store details of the FITS files, ranging from filename, last update time, md5 hash value, and various generic and instrument specific values derived from the FITS headers.

## Database Backend

The Gemini installations of the system use a PostgreSQL database backend, though as far as I know there is no database-specific code in the python layer other than the module name used to instantiate the SQL Alchemy connection. It may well be trivial to use an alternate database backend that provides similar features to PostgreSQL. As a note of reference, we have implemented the system using SQLite, in order to provide a simple, stand-alone calibration association server within the Gemini data reduction package. We did have to make a number of small changes to the more complex query construction code to generate SQL that functioned properly on SQLite, but this was mainly to work around SQL features that are not implemented in SQLite.

Typically, we keep somewhere between 2 and 4 years worth of data from a single Gemini Telescope in the database tables on our in-house servers. This means that the main file curation tables contain of order 0.5 to 1.0 million rows. Some tables contain a row for ever version of the file – ie when the file is updated, a new row is added and the old one is not deleted, but is updated to reflect that it is no longer the current entry. Regular SQL indices are used to speed searching. The database installs have had basic performance settings adjusted (eg memory cache sizes etc) for performance, but no great effort has been put into tuning database performance. In house, the systems run on dual core virtual machines with 4GB of memory and the database storage on NetApp backed virtual disk. The performance is certainly acceptable in this situation – for example searching for a calibration for a given science data set generally takes less than 1 second.

### Apache Interface

The system uses the mod\_python apache module to invoke a custom written apache handler to handle all URLs on the server. The apache handler is provided as part of the FitsStorage code and calls other functions and modules within the Fits Storage python code to construct the HTML to send to the browser.

This does mean that apache loads the python code into memory as required and keeps it for future re-use. If the python code is updated, it is necessary to re-start the apache httpd daemon in order for the changes to take effect on the web interface.

There is no problem with apache spawning multiple httpd instances all of which can service clients simultaneously. The vast majority of www initiated database access (including all fits summaries and calibration associations) is read-only and does not lock tables or rows.

### Database Schema

The database schema consists of several main groups of tables. The primary tables of interest are the file, diskfile, header, instrument (eg gmos, niri, etc) and associated tables that contain the details of fits files known to the system. A set of tables (tape, tapewrite, tapefile) are used to maintain a tape catalog library at Gemini.

Each table has an integer ‘id’ column which serves as a primary key. Many tables contain foreign key references to other tables. These are always called table\_id (where ‘table’ is the name of the other table). The database table definitions include primary and foreign key constraints, so that the database backend can enforce this aspect of data integrity.

I do not intend to fully document the database schema in this overview document, however the basic layout of the main group of tables and their relationships to each other is shown in the diagram and described below.

File  
  
Filename  
Path

DiskFile  
  
lastmod time  
md5  
present, canonical

Header  
  
project id  
data label  
instrument  
UT datetime

Gmos  
  
filter  
FP mask  
gain  
read speed

Niri  
  
filter  
slit  
well depth  
grism

fulltextheader

fulltext

Diskfilereport

fvreport  
wmdreport

footprint

extension  
area

The database is designed with the concept of storing data for multiple versions of files as the file is edited, and also for storing data for files no longer present on the storage volume.

#### File table

The file table contains exactly one entry for each filename known to the system. It is essentially the root of the tree, and the only place that references actual filenames.

Table "public.file"

Column | Type | Modifiers

----------+---------+---------------------------------------------------

id | integer | not null default nextval('file\_id\_seq'::regclass)

filename | text | not null

path | text |

Indexes:

"file\_pkey" PRIMARY KEY, btree (id)

"ix\_file\_filename" UNIQUE, btree (filename)

#### DiskFile table

An entry in the DiskFile table refers to a specific version of a file, which may or may not be present on disk (the present column) and may or may not be the most recent (canonical column) version of that file. Various generic meta data such as last modification timestamp, file size, md5 hash value etc are stored in this table to allow robust and efficient detection of whether the file has changed. In addition, the fits verify and wmd status of the file are recorded.

Table "public.diskfile"

Column | Type | Modifiers

------------+--------------------------+-------------------------------------------------------

id | integer | not null default nextval('diskfile\_id\_seq'::regclass)

file\_id | integer | not null

present | boolean |

canonical | boolean |

ccrc | text |

md5 | text |

size | integer |

lastmod | timestamp with time zone |

entrytime | timestamp with time zone |

isfits | boolean |

fvwarnings | integer |

fverrors | integer |

wmdready | boolean |

Indexes:

"diskfile\_pkey" PRIMARY KEY, btree (id)

"ix\_diskfile\_canonical" btree (canonical)

"ix\_diskfile\_entrytime" btree (entrytime)

"ix\_diskfile\_file\_id" btree (file\_id)

"ix\_diskfile\_lastmod" btree (lastmod)

"ix\_diskfile\_present" btree (present)

Foreign-key constraints:

"diskfile\_file\_id\_fkey" FOREIGN KEY (file\_id) REFERENCES file(id)

#### DiskFileReport and FullTextHeader

These tables are essentially extensions of the diskfile table, containing long text strings in each row. Thus, they have been split off from the diskfile table for performance reasons. The long text fields are rarely searched, simply looked up by diskfile\_id and returned to for display.

DiskFileReport contains the output of the fitsverify program (indicating compliance or otherwise with the FITS standard) and the wmd / mdIngest software (indicating whether the file meets the metadata requirements for ingest into the GSA). A numeric or Boolean flag giving the number of errors or warnings, or ready-to-go state of the data from running these programs is stored in the diskfile table itself, only the long text reports from these programs are stored in this table.

Table "public.diskfilereport"

Column | Type | Modifiers

-------------+---------+-------------------------------------------------------------

id | integer | not null default nextval('diskfilereport\_id\_seq'::regclass)

diskfile\_id | integer | not null

fvreport | text |

wmdreport | text |

Indexes:

"diskfilereport\_pkey" PRIMARY KEY, btree (id)

"ix\_diskfilereport\_diskfile\_id" btree (diskfile\_id)

Foreign-key constraints:

"diskfilereport\_diskfile\_id\_fkey" FOREIGN KEY (diskfile\_id) REFERENCES diskfile(id)

FullTextHeader simply contains a plain text listing of the full FITS header of the file, including the PHU and all extensions. Again, this text is not generally searched (no user interface is provided to do so) but can be displayed by the user for any given file on request via the web interface.

Table "public.fulltextheader"

Column | Type | Modifiers

-------------+---------+-------------------------------------------------------------

id | integer | not null default nextval('fulltextheader\_id\_seq'::regclass)

diskfile\_id | integer | not null

fulltext | text |

Indexes:

"fulltextheader\_pkey" PRIMARY KEY, btree (id)

"ix\_fulltextheader\_diskfile\_id" btree (diskfile\_id)

Foreign-key constraints:

"fulltextheader\_diskfile\_id\_fkey" FOREIGN KEY (diskfile\_id) REFERENCES diskfile(id)

#### Header table

The Header table stores metadata values derived from the headers that are not instrument specific, such as project ID, data-label, instrument name, RA, Dec, etc. In actual fact, these values are not strictly header values, they are mostly AstroData descriptor values, though some of the Boolean columns are derived from the presence or absence of an AstroData type flag associated with that datafile (eg the spectroscopy column). Regardless, all this metadata is derived by the AstroData class from the FITS header of the datafile.

Table "public.header"

Column | Type | Modifiers

--------------------+-----------------------------+----------------------------------------------------

id | integer | not null default nextval('header\_id\_seq'::regclass)

diskfile\_id | integer | not null

program\_id | text |

observation\_id | text |

data\_label | text |

telescope | text |

instrument | text |

ut\_datetime | timestamp without time zone |

local\_time | time without time zone |

observation\_type | text |

observation\_class | text |

object | text |

ra | numeric(16,12) |

dec | numeric(16,12) |

azimuth | numeric(16,12) |

elevation | numeric(16,12) |

cass\_rotator\_pa | numeric(16,12) |

airmass | numeric(8,6) |

filter\_name | text |

exposure\_time | numeric(8,4) |

disperser | text |

central\_wavelength | numeric(8,6) |

focal\_plane\_mask | text |

spectroscopy | boolean |

adaptive\_optics | boolean |

raw\_iq | text |

raw\_cc | text |

raw\_wv | text |

raw\_bg | text |

qa\_state | text |

release | date |

reduction | text |

phot\_standard | boolean |

Indexes:

"header\_pkey" PRIMARY KEY, btree (id)

"ix\_header\_data\_label" btree (data\_label)

"ix\_header\_diskfile\_id" btree (diskfile\_id)

"ix\_header\_instrument" btree (instrument)

"ix\_header\_observation\_class" btree (observation\_class)

"ix\_header\_observation\_id" btree (observation\_id)

"ix\_header\_observation\_type" btree (observation\_type)

"ix\_header\_program\_id" btree (program\_id)

"ix\_header\_ut\_datetime" btree (ut\_datetime)

Foreign-key constraints:

"header\_diskfile\_id\_fkey" FOREIGN KEY (diskfile\_id) REFERENCES diskfile(id)

#### The Instrument tables

A separate table exists for each instrument known to the system, containing metadata specific to that instrument. For example, GMOS has the concept of detector binning, which is not a valid concept for most of the infrared instruments. Again, the actual values stored are either AstroData descriptor values, or the presence of AstroData type flags. It should be noted that the gmos table can be used for data from either GMOS-N or GMOS-S.

Table "public.gmos"

Column | Type | Modifiers

--------------------+---------+---------------------------------------------------

id | integer | not null default nextval('gmos\_id\_seq'::regclass)

header\_id | integer | not null

disperser | text |

filter\_name | text |

detector\_x\_bin | integer |

detector\_y\_bin | integer |

amp\_read\_area | text |

read\_speed\_setting | text |

gain\_setting | text |

focal\_plane\_mask | text |

nodandshuffle | boolean |

nod\_count | integer |

nod\_pixels | integer |

Indexes:

"gmos\_pkey" PRIMARY KEY, btree (id)

"ix\_gmos\_amp\_read\_area" btree (amp\_read\_area)

"ix\_gmos\_detector\_x\_bin" btree (detector\_x\_bin)

"ix\_gmos\_detector\_y\_bin" btree (detector\_y\_bin)

"ix\_gmos\_disperser" btree (disperser)

"ix\_gmos\_filter\_name" btree (filter\_name)

"ix\_gmos\_focal\_plane\_mask" btree (focal\_plane\_mask)

"ix\_gmos\_gain\_setting" btree (gain\_setting)

"ix\_gmos\_header\_id" btree (header\_id)

"ix\_gmos\_nod\_count" btree (nod\_count)

"ix\_gmos\_nod\_pixels" btree (nod\_pixels)

"ix\_gmos\_nodandshuffle" btree (nodandshuffle)

"ix\_gmos\_read\_speed\_setting" btree (read\_speed\_setting)

Foreign-key constraints:

"gmos\_header\_id\_fkey" FOREIGN KEY (header\_id) REFERENCES "header"(id)

Table "public.niri"

Column | Type | Modifiers

--------------------+---------+---------------------------------------------------

id | integer | not null default nextval('niri\_id\_seq'::regclass)

header\_id | integer | not null

disperser | text |

filter\_name | text |

read\_mode | text |

well\_depth\_setting | text |

data\_section | text |

coadds | integer |

camera | text |

focal\_plane\_mask | text |

Indexes:

"niri\_pkey" PRIMARY KEY, btree (id)

"ix\_niri\_camera" btree (camera)

"ix\_niri\_coadds" btree (coadds)

"ix\_niri\_data\_section" btree (data\_section)

"ix\_niri\_disperser" btree (disperser)

"ix\_niri\_filter\_name" btree (filter\_name)

"ix\_niri\_header\_id" btree (header\_id)

"ix\_niri\_read\_mode" btree (read\_mode)

"ix\_niri\_well\_depth\_setting" btree (well\_depth\_setting)

Foreign-key constraints:

"niri\_header\_id\_fkey" FOREIGN KEY (header\_id) REFERENCES "header"(id)

Table "public.gnirs"

Column | Type | Modifiers

--------------------+---------+----------------------------------------------------

id | integer | not null default nextval('gnirs\_id\_seq'::regclass)

header\_id | integer | not null

disperser | text |

filter\_name | text |

read\_mode | text |

well\_depth\_setting | text |

coadds | integer |

camera | text |

focal\_plane\_mask | text |

Indexes:

"gnirs\_pkey" PRIMARY KEY, btree (id)

"ix\_gnirs\_camera" btree (camera)

"ix\_gnirs\_coadds" btree (coadds)

"ix\_gnirs\_disperser" btree (disperser)

"ix\_gnirs\_filter\_name" btree (filter\_name)

"ix\_gnirs\_header\_id" btree (header\_id)

"ix\_gnirs\_read\_mode" btree (read\_mode)

"ix\_gnirs\_well\_depth\_setting" btree (well\_depth\_setting)

Foreign-key constraints:

"gnirs\_header\_id\_fkey" FOREIGN KEY (header\_id) REFERENCES "header"(id)

Table "public.nifs"

Column | Type | Modifiers

------------------+---------+---------------------------------------------------

id | integer | not null default nextval('nifs\_id\_seq'::regclass)

header\_id | integer | not null

disperser | text |

filter\_name | text |

read\_mode | text |

coadds | integer |

focal\_plane\_mask | text |

Indexes:

"nifs\_pkey" PRIMARY KEY, btree (id)

"ix\_nifs\_coadds" btree (coadds)

"ix\_nifs\_disperser" btree (disperser)

"ix\_nifs\_filter\_name" btree (filter\_name)

"ix\_nifs\_header\_id" btree (header\_id)

"ix\_nifs\_read\_mode" btree (read\_mode)

Foreign-key constraints:

"nifs\_header\_id\_fkey" FOREIGN KEY (header\_id) REFERENCES "header"(id)

Table "public.michelle"

Column | Type | Modifiers

------------------+---------+-------------------------------------------------------

id | integer | not null default nextval('michelle\_id\_seq'::regclass)

header\_id | integer | not null

disperser | text |

filter\_name | text |

read\_mode | text |

coadds | integer |

focal\_plane\_mask | text |

Indexes:

"michelle\_pkey" PRIMARY KEY, btree (id)

"ix\_michelle\_coadds" btree (coadds)

"ix\_michelle\_disperser" btree (disperser)

"ix\_michelle\_filter\_name" btree (filter\_name)

"ix\_michelle\_header\_id" btree (header\_id)

"ix\_michelle\_read\_mode" btree (read\_mode)

Foreign-key constraints:

"michelle\_header\_id\_fkey" FOREIGN KEY (header\_id) REFERENCES "header"(id)

### File Ingest system and procedure.

Ingesting a FITS file into the system can take up to 3 seconds or so of elapsed time on the systems deployed at Gemini[[1]](#footnote-1). For this, and other, reasons a queue based system is used for file ingest. A database table called ingestqueue contains rows for all the files waiting to be ingested. When we wish to ingest a file, we simply add its details to a new row in the ingestqueue table.

Table "public.ingestqueue"

Column | Type | Modifiers

------------+-----------------------------+----------------------------------------------------------

id | integer | not null default nextval('ingestqueue\_id\_seq'::regclass)

filename | text | not null

path | text |

inprogress | boolean |

added | timestamp without time zone |

Indexes:

"ingestqueue\_pkey" PRIMARY KEY, btree (id)

"ix\_ingestqueue\_filename" btree (filename)

"ix\_ingestqueue\_inprogress" btree (inprogress)

A number of processes can then run on the fits storage server servicing the ingest queue and ingesting files. The system is designed to allow multiple concurrent processed to service the queue simultaneously, using for example “select for update” to mark a row as in-progress, to prevent collisions and race conditions. In general, each service\_ingest\_queue process will make effective use of one CPU core - it is not sensible to run more service\_ingest\_queue processes than there are CPU cores available, though so long as this is not exceeded, the total rate of file ingest will scale well as more processes are run.

Files can be added in bulk to the ingestqueue without worry about duplication or pre-existing files. When a file is processed and removed from the ingestqueue, all equivalent entries for that file are removed from the queue. In addition, the default behavior when ingesting a file that already exists in the database is to check the last modification time of the file on disk to see if it has changed since it was last ingested. If the last modification time suggests that the file was modified, the md5 hash value of the file is next computed and compared against the database to determine if the file contents were actually changed. By default (this can be changed), the file will only be re-ingested if it has actually changed.

So far, two scripts have been developed to add files to the ingest queue. Current operation at Gemini has the files stored on a NetApp NFS export. There is no real way other than polling to detect the presence of new files or the modification of files. Thus, a cron job runs at various intervals that simply adds all the files from a recent time period onto the ingest queue. Those files that have not been modified are passed over quickly when servicing the ingest queue. Currently we re-add to the ingest queue: all files from the current UT date every 4 minutes, all files from the past 4 UT dates every 15 minutes and all files from the past 10 UT dates every 24 hours. A second script has been written which subscribes to inotify events for the filesystem and adds files to the ingestqueue as they appear or are actually updated. However the current NetApp storage cannot provide inotify events so this is not used in current operations at Gemini.

It should be noted for reference that the service\_ingest\_queue scripts work through un-modified files that are already in the database at a rate of about 20 per second on our current systems.

#### AstroData and all that

The actual metadata values derived from the FITS keywords are calculated using the AstroData python class. This is a data abstraction / access written by the Gemini Data Processing Software Group. To zero-th order it can be thought of a souped up PyFITS. In fact, the actual FITS access is done using PyFITS and the PyFITS data structures are accessible through AstroData if necessary. Two main functionalities of AstroData are used by the fits storage system: types and descriptors.

AstroData types are a number of flags that are calculated for a file. For example there is a type that says whether the file is spectroscopy. The details of how to determine this are instrument specific and messy (e.g. in GMOS if the “disperser” is set to “Mirror” you are in imaging mode, otherwise spectroscopy, in NIRI if the FILTER3 keyword value contains the string “grism”, then you are in spectroscopy mode, otherwise imaging) – but this is all hidden away by astrodata, we can simply look for the presence of the “SPECT” type without worrying about which instrument we are or how to calculate that.

AstroData descriptors are likewise a form of header abstraction. If you want to know the filter used to take a given observation, then at the fits level you typically have to look at a number of instrument dependent keywords (for example FILTER1, FILTER2, FILTER3), and combine them together in some instrument dependent way (eg cope with “Open” values, or leak blocking filters). With astrodata, you simply evaluate the filter\_name descriptor, and the details of which instrument you are using and how to determine the filter name for it are all taken care of.

AstroData is a python module. Here’s a quick example:

[fitsdata@mkofits1 ~]$ python

Python 2.6.6 (r266:84292, Apr 19 2011, 16:48:26)

[GCC 4.1.2 20080704 (Red Hat 4.1.2-50)] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>> from astrodata import AstroData

>>> ad = AstroData("/net/wikiwiki/dataflow/N20111210S0319.fits")

>>> print ad.types

['GEMINI\_NORTH', 'GEMINI', 'IMAGE', 'GMOS\_N', 'GMOS\_IMAGE', 'GMOS', 'GMOS\_RAW', 'RAW', 'UNPREPARED']

>>> ad.is\_type('SPECT')

False

>>> ad.is\_type('IMAGE')

True

>>> print ad.data\_label()

GN-2011B-SV-52-162-048

>>> print ad.filter\_name()

open1-6&z\_G0304

>>> print ad.filter\_name(pretty=True)

z

>>> print ad.read\_speed\_setting()

slow

>>> print ad.ut\_datetime()

2011-12-10 08:35:53.100000

### Calibration Matching

Having described the internals of the system, a further note on calibration matching is in order. Calibrations are located purely on the basis of the header and instrument tables. Essentially, we determine the metadata of the target (i.e. science) dataset, either by looking up the data for the corresponding raw file in the tables, or by reading the live metadata passed in over the http connection, then we construct a database query to match the target metadata against the instrument and header table, finally applying “order by” and “limit” clauses to the query to select the best available calibration. The query is formed programmatically, and the decision whether to include certain criteria in the selection can depend on the actual values of various metadata items.

In the code, this is done using the FitsStorageCalibration class. This class is subclassed for each instrument, with a factory function to instantiate and return the appropriate subclassed object as appropriate for the input data. The object is initialized with the target data set, and will determine the necessary metadata of the target data set when it is instantiated. The class then provides methods for each calibration type that construct and then execute the appropriate query to find the requested calibration. As an example, here is the code of the arc() method in the CalibrationGMOS subclass of FitsStorageCalibration:

def arc(self, sameprog=False, List=None):

query = self.session.query(Header).select\_from(join(join(Gmos, Header), DiskFile))

query = query.filter(Header.observation\_type=='ARC')

# Search only the canonical (latest) entries

query = query.filter(DiskFile.canonical==True)

# Knock out the FAILs

query = query.filter(Header.qa\_state!='Fail')

# Must Totally Match: Instrument, disperser

query = query.filter(Header.instrument==self.descriptors['instrument']) .filter(Gmos.disperser==self.descriptors['disperser'])

# Must match filter (from KR 20100423)

query = query.filter(Gmos.filter\_name==self.descriptors['filter\_name'])

# Must Match central\_wavelength

query = query.filter(Header.central\_wavelength==self.descriptors['central\_wavelength'])

# Must match focal\_plane\_mask only if it's not the 5.0arcsec slit in the target, otherwise any longslit is OK

if(self.descriptors['focal\_plane\_mask'] != '5.0arcsec'):

query = query.filter(Gmos.focal\_plane\_mask==self.descriptors['focal\_plane\_mask'])

else:

query = query.filter(Gmos.focal\_plane\_mask.like('%arcsec'))

# Must match ccd binning

query = query.filter(Gmos.detector\_x\_bin==self.descriptors['detector\_x\_bin']) .filter(Gmos.detector\_y\_bin==self.descriptors['detector\_y\_bin'])

# The science amp\_read\_area must be equal or substring of the arc amp\_read\_area

query = query.filter(Gmos.amp\_read\_area.like('%'+self.descriptors['amp\_read\_area']+'%'))

# Should we insist on the program ID matching?

if(sameprog):

query = query.filter(Header.program\_id==self.descriptors['program\_id'])

# Order by absolute time separation.

query = query.order\_by(func.abs(extract('epoch', Header.ut\_datetime - self.descriptors['ut\_datetime'])).asc())

# For now, we only want one result - the closest in time, unless otherwise indicated

if(List):

query = query.limit(List)

return query.all()

else:

query = query.limit(1)

return query.first()

An example of this being called in the Calibrations web interface looks like:

c = get\_cal\_object(session, None, header=object)

if('arc' in c.required and (caltype=='all' or caltype=='arc')):

requires=True

# Look for an arc in the same program

arc = c.arc(sameprog=True)

1. The main part of this is in running the mdIngest program on the file – it is called on each file individually as that file is ingested, so the inefficiencies of loading java and all the required jar files make this relatively slow [↑](#footnote-ref-1)