The Hubble Space Telescope (HST) Advanced Camera for Surveys (ACS) Quicklook Project

Matthew Bourque[1], Sara Ogaz[1], Alex Viana[2], Meredith Durbin[3], Norman Grogin[1]

[1] Space Telescope Science Institute, Baltimore, Maryland 21218. email: bourque@stsci.edu, ogaz@stsci.edu, grogin@stsci.edu

[2] Dept. of Astronomy, The University of Washington, Box 351580, U.W. Seattle, Washington, 98195, email:mdurbin@uw.edu

[3] Terbium Labs, Baltimore, Maryland 21201. email: alexcostaviana@gmail.com

Abstract—What is an abstract, really?

1 Introduction

The Advanced Camera for Surveys (ACS) is a third-generation imaging instrument on board the Hubble Space Telescope (HST), installed in 2002 during Servicing Mission 3B. It is comprised of three detectors: (1) the Wide Field Camera (WFC), which is designed for wide-field imaging and spectroscopy in visible to near-infrared wavelengths, (2) the High Resolution Channel, which is designed for high resolution near-ultraviolet to near-infrared wavelength images and coronography, and (3) the Solar Blind Channel (SBC), desingned for far-ultraviolet imaging and spectroscopy. ACS expererienced an electronics failure in 2007 that affected the WFC and HRC detectors, until 2009 when astronauts succesfully restored the WFC detector during Servicing Mission 4; the HRC still remains unoperational.

Besides these few hiccups, the ACS instrument has been steadily acquiring astronomical images over its 15 on-orbit lifetime. Figure 1 shows an estimates of the number of observations over time for each of the three detectors. To date, there have been nearly 200,000 of observations total. Further information about the ACS instrument including its history, configuration, performance, and scientific capability can be found in the ACS Instrument Handbook (Avila et al., 2017).

ACS data, along with all other data from the other HST instruments past and present (e.g. The Wide Field Camera 3 (WFC3), The Cosmic Origins Spectrography (COS), etc.), are primarily stored and publicly-available in the Barbara A. Mikulski Archive for Space Telescopes (MAST)[1] (Barbara, 2017). Through MAST, users can request and retreive data for any publicly-available dataset via `ftp`, `sftp`, or DVD by mail[2]. The ACS data, like most all other astronomical data, are stored in the Flexible Image Transport System (FITS) filetype (FITS, 2008). This filetype has several unique characteristics, as will be discussed in section 1.1.

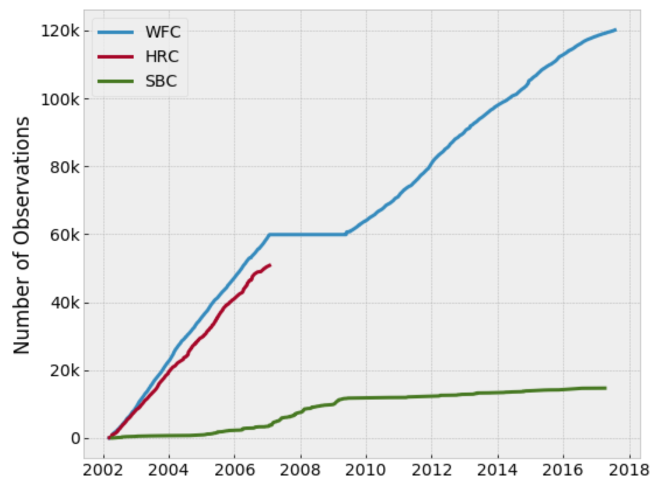

Fig. 1. The number of observations over time for each of the three detectors on ACS.

The ACS Quicklook Project is a `python`-based application for discovering, viewing, and querying all publicly-available ACS data. It consists of several subsystems: (1) A filesystem that stores ACS instrument data files and "Quicklook" JPEGs in an organized Network File System (NFS), (2) A `MySQL` database that stores image metadata of each observation, (3) A `python/Flask`-based web application for interacting with the filesystem and database, and (4) A `python` code library (named `acsql`) that contains code for connecting to the database, ingesting new data, logging production code execution, and building/maintaining the database and web application. Each of these subsystems are explained in further detail in the Methodology section of this paper.

This paper aims to outline and detail the ACS Quicklook project as part of the Towson University Computer Science Masters Program Graduate Project. The remaining subsections in this chapter discuss the motivation and use cases for this application, as well as details on the underlying data structure on top of which this project was built. Chapter 2 discusses related work to this project and how the ACS

Manuscript received Month DD, YYYY

1. named after the U.S. Senator from Maryland who has been a pivitol political driving force behind the manned servicing missions, the Hubble Space Telescope, and the forthcoming James Webb Space Telescope

2. Not all HST data are publicly available; most HST data of scientific targets are considered proprietary for up to one calendar year, after which they are publicly released.

Quicklook project differs from existing similar applications. Chapter 3 details the implementations of each of the ACS Quicklook subsystems. Chapter 4 outlines the results of the project, namely the project deliverables. Lastly, chapters 5 and 6 conclude the paper with a discussion of possible extensions and modifications to the application.

It should be noted that the work that went into this project by the authors was accomplished on behalf of the Space Telescope Science Institute (STScI) located in Baltimore, Maryland. STScI is the home institution for instrument, data, and user support of HST, the forthcoming James Webb Space Telescope (JWST), and MAST. STScI is part of the Association of Universities for Research in Astronomy (AURA).

1.1 Data Structure

The design of the ACS Quicklook application, especially the database, is heavily dependant on the underlying data structure of ACS FITS files. As such, it is important for the reader to understand this data structure and thus the next four sections are dedicated to giving an overview on the subject.

1.1.1 *Filenames*

Each ACS data file is named in a consistent fashion:

```
<rootname>_<filetype>.fits
```

where each `<rootname>` consists of nine unique alphanumeric characters, and `<filetype>` is one of several three-character filetype options (discussed in proceeding section 1.1.4). For example, one ACS observation has the rootname `j6mf16lhq_raw.fits` (Principle Investigator Gary Bernstein, observation date `2016-09-22`). Each character in the 9-character `rootname` has meaning, and is discussed in section 5.2 of the Introduction to the HST Data Handbooks (Smith et al., 2011). The `.fits` extension at the end of the filename signifies that the file is of FITS format.

Note about rootname caveat.

1.1.2 *FITS file structure*

Each ACS FITS file consists of several "Extensions", with each extension serving a purpose to describe a particular aspect of the observation. Each extension consists of two parts: (1) an extension "header", which contain key/value pairs describing image metadata (for example, `DATE-OBS = '2016-09-22'` indicates that the observation date was `2016-09-22`) (discussed in the next section), and (2) the extension data, which may be a binary table or, more commonly, a multi-dimensional array of detector pixel values.

The type of extension data can also vary. The most common extension data types are (1) 'science' (`SCI`), in which the extension data describe a scientific observation, (2) 'error' (`ERR`), in which the extension data describe the uncertainty in the pixel values of the `SCI` data, and (3) 'data quality' (`DQ`), in which the extension data describe the quality of the pixel values for the detector (for example, they may indicate that certain pixels were affected by cosmic rays durring the observation). Typically, for a given file, the 1st extension is the `SCI` extension, the 2nd extension is the `ERR` extension, and the 3rd extension is the `DQ` extension. Furthermore, the 0th extension typically has no extension data and only an extension header that contains metadata that is common to all extensions. This is referred to as the 'Primary Header'.

TABLE 1
ACS/WFC FITS file extensions

Extension	Purpose	Image Dimensions (pixels)	Data Type
0	Primary header	–	String
1	SCI, Chip 2	(4096, 2048)	Float
2	ERR, Chip 2	(4096, 2048)	Float
3	DQ, Chip 2	(4096, 2048)	Integer
4	SCI, Chip 1	(4096, 2048)	Float
5	ERR, Chip 1	(4096, 2048)	Float
6	DQ, Chip 1	(4096, 2048)	Integer

TABLE 2
ACS/HRC and ACS/SBC FITS file extensions

Extension	Purpose	Image Dimensions (pixels)	Data Type
0	Primary header	–	String
1	SCI	(1024, 1024)	Float
2	ERR	(1024, 1024)	Float
3	DQ	(1024, 1024)	Integer

Tables 1-3 describe the different extensions of ACS FITS files for each of the three ACS detectors. Note that there are two sets of `SCI`/`ERR`/`DQ` extensions for WFC since WFC is comprised of two separate CCD chips.

Over the years, there have been several tools written in various programming languages to read in FITS files and automatically convert their extension data to multi-dimensional array data types and their extension headers to dictionary data types. For this project, the `astropy.fits python` library is used extensivly to read and interact with ACS FITS files (Robitaille et al., 2013).

1.1.3 *FITS file extension headers*

As mentioned in the previous section, each FITS extension contains a "header", which contains key/value pairs of metadata associated with the extension data. Such metadata includes various data that describes the astronomical observation (e.g. target name, exposure time, principle investigator name, etc.), telemetry of ACS or HST in general at the time of observation (e.g. temperature of the ACS instrument, orientation of the telescope pointing, position of the telescope relative to Earth, etc.) or the FITS file itself (e.g. the number of extensions, file creation date, etc.). A subsection of an example header is shown in Figure 2.

Extension headers may contain a large number of keyword/value pairs. Some extension headers contain upwards of 300 keywords, while others may contain only ∼40 keywords.

```
SIMPLE  =                    T / data conform to FITS standard
BITPIX  =                   16 / bits per data value
NAXIS   =                    0 / number of data axes
EXTEND  =                    T / File may contain standard extensions
NEXTEND =                    6 / Number of standard extensions
GROUPS  =                    F / image is in group format
DATE    = '2016-09-22'         / date this file was written (yyyy-mm-dd)
FILENAME= 'j6mf16lhq_raw.fits                     ' / name of file
FILETYPE= 'SCI      '          / type of data found in data file

TELESCOP= 'HST'                / telescope used to acquire data
INSTRUME= 'ACS   '             / identifier for instrument used to acquire data
EQUINOX =               2000.0 / equinox of celestial coord. system

              / DATA DESCRIPTION KEYWORDS

ROOTNAME= 'j6mf16lhq                         ' / rootname of the observation set
IMAGETYP= 'DARK              ' / type of exposure identifier
PRIMESI = 'ACS   '             / instrument designated as prime

              / TARGET INFORMATION

TARGNAME= 'DARK                          ' / proposer's target name
RA_TARG =   0.000000000000E+00 / right ascension of the target (deg) (J2000)
DEC_TARG=   0.000000000000E+00 / declination of the target (deg) (J2000)

              / PROPOSAL INFORMATION

PROPOSID=                 9433 / PEP proposal identifier
LINENUM = '16.055         '    / proposal logsheet line number
PR_INV_L= 'Bernstein                     ' / last name of principal investigator
PR_INV_F= 'Gary                ' / first name of principal investigator
PR_INV_M= '                    ' / middle name / initial of principal investigat

              / EXPOSURE INFORMATION

SUNANGLE=            93.563698 / angle between sun and V1 axis
MOONANGL=            33.222004 / angle between moon and V1 axis
SUN_ALT =            68.062172 / altitude of the sun above Earth's limb
FGSLOCK = 'FINE              ' / commanded FGS lock (FINE,COARSE,GYROS,UNKNOWN)
GYROMODE= '3'                  / number of gyros scheduled, T=3+OBAD
REFFRAME= 'GSC1    '           / guide star catalog version
MTFLAG  = ' '                  / moving target flag; T if it is a moving target

DATE-OBS= '2003-01-27'         / UT date of start of observation (yyyy-mm-dd)
TIME-OBS= '15:20:01'           / UT time of start of observation (hh:mm:ss)
EXPSTART=   5.266663890058E+04 / exposure start time (Modified Julian Date)
EXPEND  =   5.266665048715E+04 / exposure end time (Modified Julian Date)
EXPTIME =          1000.000000 / exposure duration (seconds)--calculated
```

Fig. 2. An example header.

1.1.4 FITS filetypes for ACS

As discusses in section 1.1.1, each ACS observation may result in several FITS filetypes. Each filetype describes the observation in a different way. The set of available filetypes for a given observation is dependent on the characteristics of the observation, the details of which are beyond the scope of this paper. Also beyond the scope of this paper are the vast details that surround each filetype; each one has a different scientific application that is not important to understanding the ACS Quicklook project. However, to provide at least some context, below we give a brief description of each possible filetype that a given observation may contain:

- **raw** - The raw, uncalibrated data that comes directly from HST
- **flt** - nominally calibrated data
- **flc** - nominally calibrated data plus corrected for Charge Transfer Efficieny (CTE) deficits.
- **drz** - Geometric distortion-corrected data
- **drc** - Geometric distrotion-corrected plus CTE corrected data
- **spt** - Telescope telemetry data
- **jit** - Telescope pointing data
- **jif** - Telescope drifting data
- **crj** - Cosmic ray rejected data
- **crc** - Cosmic ray rejected plus CTE corrected data
- **asn** - Observation association table.

As noted earlier, a given observation may not result in the set of all possible filetypes. For example, the observation `j6mf16lhq` only has the filetypes `raw`, `flt`, `jit`, `jif`, and `spt`.

1.2 Key Metadata

There are several metadata key/value pairs that are particulary important for the ACS Quicklook application, specifically the web application. For some reference, and context, these metadata are briefly described below. Note that the `rootname` and `proposal_type` are not metadata from extension headers, but rather are metadata that were explicitly added to the database schema.

APERTURE - The portion of the `WFC`, `HRC`, or `SBC` detector that that was used during an observation. Can either be the entire detector (called a "full-frame image") (e.g. `WFC`), or a subsection of the detector (called a "subarray") (e.g. `WFC1-1K`).

DATE-OBS - The date of the observation in the format `YYYY-MM-DD`, measured in Universal Time (e.g. `'2017-08-05'`).

DEC_TARG - The declination of the target (i.e. the angular distance the target north or south of the celestial equator) (e.g. `41.2842`).

DETECTOR - The detector used for the observation. Can either be `WFC`, `HRC`, or `SBC`.

EXPFLAG - Indicates if an observation was interrupted (e.g. `INTERRUPTED`) or not (e.g `NORMAL`).

EXPSTART - The exposure start time of the observation, in units of Modified Julian Date (e.g. `52473.8`).

EXPTIME - The exposure duration of the observation, in units of seconds (e.g. `1000.0`).

FILTER1 - The selected element from the `ACS` filter wheel # 1 (e.g. `F606W`).

FILTER2 - The selected element from the `ACS` filter wheel # 1 (e.g. `F814W`).

IMAGETYP - The type of exposure for the observation (e.g. `BIAS`, `EXT`, etc.).

OBSTYPE - The type of observation, either `IMAGING`, `SPECTROSCOPIC`, `CORONOGRAPHIC`, or `INTERNAL`.

proposal_type - The type of proposal that the observation belongs to, such as Calibration (i.e. `CAL`) or General Observer (i.e. `GO`).

PROPOSID - The proposal number that the observation belongs to (e.g. `10695`).

RA_TARG - The right ascension of the target (i.e. the angular distance of the target east and west on the celestial sphere) (e.g. `49.5375`).

rootname - The 8-character unique rootname of the observation (e.g. `j5915401`).

SUBARRAY - A boolean flag that indicates if the observation is a full-frame `APERTURE` (i.e. `0`) or a subarray `APERTURE` (i.e. `1`).

TARGNAME - The name of the target (e.g. `M87`, `NGC-4536`, `ANDROMEDA-I`, etc.).

TIME-OBS - The time of the start of the observation in the format `HH:MM:SS`, measured in Universal Time (e.g. `14:21:56`).

1.3 Motivation

The motivation for the ACS Quicklook system is driven by several shortcomings of the FITS file structure as well as the current capabilities of MAST from a specific user perspective (inteded users and their use cases are discussed in section 1.2). Some of these shortcomings are described below along with the intended way the ACS Quicklook application will address them.

Data retreival letency: Currently, users who wish to retreive data from the MAST archive must submit a retreival request via the MAST online interface. Once the retreival request is processed (usually automatically unless it is a request of a large number of datasets), the data are either transfered to the user directly via `sftp`, transfered to a "staging area" in which the user can log into and copy the data via `ftp` at their leisure, or sent by mail via DVD, depending on which option the user selects. In the case of any one of these options, the time between a download request and the the time in which the user has fully retreived the data is a non-significant amount of time. In the fastest scenario of the `sftp` option, a typical request can take minutes to hours to be completed. The ACS Quicklook system attempts to circumnavigate this retreival process by making the full data products instantly available via read-only access of the filesystem subystem, as well as a subset of the data products (and corresponding metadata) instantly available to view through the web application.

File I/O: Users who

Data redundancy: Something.

Data discovery: Something

1.4 Use Cases

The intended user of ACS Quicklook are ACS instrument scientists, analysts, or scientific users who wish to perform one or more of the following use cases:

1. View

2 Related Work

Topics to discuss:

1. The MAST archive 2. The MAST portal 3. The WFC3/Quicklook project 4. Other Astronomy Institutions 5. How ACS/Quicklook is different

3 Methodology

In this chapter, we disucss the methods by which we implemented the various subsystems of the ACS Quicklook system. Additionally, we discuss the programming standards and standard workflows that were employed to promote code quality such as readability, maintainability, extensibility, etc; we believe that this aspect of the project is at least equaly important to the system as its individual components.

3.1 Version control

All code associated with this project (including this paper iteself) is version controlled using the `git` Version Control System (VCS) (git, 2017). The `git` repository for the project is named `acsql`. The `git` repository is also hosted on `GitHub`, a repository hosting service (GitHub, 2017), and is publicly available at http://github.com/spacetelescope/acsql/.

Several feature branches of the code were created throughout the building of this project such that the `master` branch (which is considered the "production" branch) always contained operational code (while the code in the branches may contain unfinished implementations). Such branches include `create-database` (for implementation of the database schema), `add-logging` (for implementation of system logging), `build-ingest` (for implementation of data ingestion software), and `web-application` (for implementation of the web application). For each merge of a feature branch, a `tag` and `release` was created for the `master` branch, which allows a specific version of the `master` branch to be saved in the repository. These releases are available at https://github.com/spacetelescope/acsql/releases.

Additionally, using `GitHub` allowed for issue tracking of bugs, features, and potential enhancements to the code repository. Current open issues of the repository can be found at https://github.com/spacetelescope/acsql/issues.

3.2 Programming and Documentation Standards

All code contained within this project was written to adhere to specific standards and conventions, namely (1) the `PEP8` Style Guide for `python` code (van Rossum, 2001), (2) The `PEP257` `python` guide for module and function docstring conventions (Goodger, 2001), and (3) the `numpydocs` documentation standard (NumPy Documentation, 2017). More details on each of these standards and conventions are given below.

The `PEP8` Style Guide for `python` code (abbreviated for 'Python Enhancement Proposal #8') documents `python` coding conventions including variable naming, spacing, line length, module layout, function layout, comments, and design patterns. Only in specific cases were these conventions not followed, such as using a single line of code, even if it exceeded the recoomended 80 characters, to allow for greater readability. By following these conventions, the style of the `acsql` code is constistent amongst each module and attempts to reflect the style of industry-grade `python` code.

```python
def get_proposal_type(proposid):
    """Return the ``proposal_type`` for the given ``proposid``.

    The ``proposal_type`` is the type of proposal (e.g. ``CAL``,
    ``GO``, etc.).  The ``proposal_type`` is scraped from the MAST
    proposal status webpage for the given ``proposid``.  If the
    ``proposal_type`` cannot be determined, a ``None`` value is returned.

    Parameters
    ----------
    proposid : str
        The proposal ID (e.g. ``12345``).

    Returns
    -------
    proposal_type : int or None
        The proposal type (e.g. ``CAL``).
    """
```

Fig. 3. An example of the PEP257 and numpydoc docstring conventions, using the get_proposal_type function from acsql.ingest.ingest.

ingest.ingest.get_proposal_type(*proposid*)

Return the `proposal_type` for the given `proposid`.

The `proposal_type` is the type of proposal (e.g. `CAL`, `GO`, etc.). The `proposal_type` is scraped from the MAST proposal status webpage for the given `proposid`. If the `proposal_type` cannot be determined, a `None` value is returned.

Parameters: **proposid** : str

The proposal ID (e.g. `12345`).

Returns: **proposal_type** : int or None

The proposal type (e.g. `CAL`).

Fig. 4. The readthedocs documentation for the acsql example function seen in Figure N.

The PEP257 guide for docstring conventions describes standard conventions used for function and module docstrings (i.e. the API documentation found in block comments at the beginning of modules or immediately after function declarations). Like PEP8, following these conventions ensure consistency amongst the acsql code documentation. Furthermore, the numpydocs documentation convention provides some additional details on top of the PEP257 conventions and is used in many python packages including the numpy (numerical python) and scipy (scientific python) packages (van der Walt et al., 2011). Figure N shows an example of these conventions, taken from the ascql.ingest.ingest.get_proposal_type function.

Another benefit to using PEP257 and numpydoc docstring conventions is that API documentation creation tools such as sphinx (Brandi et al., 2007) or epydoc (Loper, 2004) can automatically convert the docs into other output formats such as HTML and PDF. For this project, we use sphinx to convert API documentation to HTML, and host the webpages online using the readthedocs, which is an open-source, community supported tool for hosting and browing documentation (Read the Docs, 2017). The documentation for acsql is hosted at http://acsql.readthedocs.io/. The output documentation as seen on readthedocs for the example function in figure N is provided in Figure N.

```
filesystem/
    jcp0/
        jcp001kwq/
            jcp001kwq_flc.fits
            jcp001kwq_flt.fits
            jcp001kwq_raw.fits
            jcp001kwq_spt.fits
            jcp001kwj_jit.fits
            jcp001kwj_jif.fits
        jcp001010/
            jcp001010_asn.fits
            jcp001010_drc.fits
            jcp001010_drz.fits
            jcp001010_jif.fits
            jcp001010_jit.fits
        jcp014tyq/
            ...
        jcp001ktq/
            ...
        ...
    jcp3/
        ...
    jcp7/
        ...
    ...
```

Fig. 5. A representation of the directory structure within the acsql filesystem, using a few observations as an example.

3.3 Filesystem: Archive of ACS data

The acsql filesystem is a Network File System (NFS) that stores all on-orbit ACS data on disk in an organized set of directories and subdirectories hosted at STScI. Figure N shows an example of this directory structure: The parent directory is the first four characters of the 9-character rootname, which maps directly to an individual PROPOSID. The subdirectories of the parent directories are named after the full 9-character rootname such that each parent directory contains the rootname subdirectories that were observed for that particular PROPOSID. Each rootname subdirectory contains every available filetype (as described in Section 1.1.4) for the particular observation is stored.

Figure N shows how the total size of the filesystem has evolved over the lifetime of the mission; currently, the filesystem occupies ~40 TB of storage space. Note that the file sizes across the detectors and across the various filetypes may vary depending on the nature of the particular obseration (for example, full-frame observations result in larger file sizes than subarray observations, calibrated filetypes have larger file sizes than un-calibrated filetypes, etc.).

3.4 Filesystem: Archive of JPEGs and Thumbnails

In addition to the ACS data products described in the last section, the acsql filesystem also stores "Quicklook" JPEG and thumbnail images of each RAW, FLT, and FLC filetype (when applicable) in an organized directory structure. These images are used by the acsql web application to allow users to quickly and easily view the contents of the data without having to pysically open the corresponding .fits files.

The JPEGimages are are generated by taking the two-dimensional data from the SCI extension(s), sigma-clipping the top and bottom 1% of the values (as to avoid large outlier values and enhance the scaling of the image), and saving the data to a a JPEG format. The thumbnail images are created by simply resizing the corresponding JPEG into a 128x128 pixel image and saving to a .thumb extension; the purpose

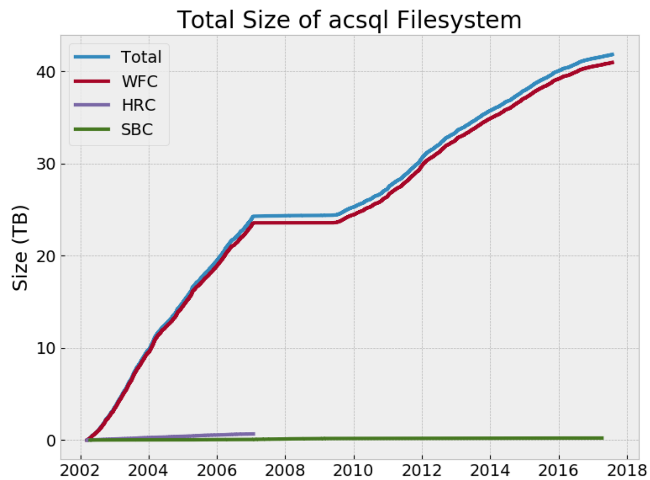

Fig. 6. The size of the `acsql` filesystem as a function of observation date.

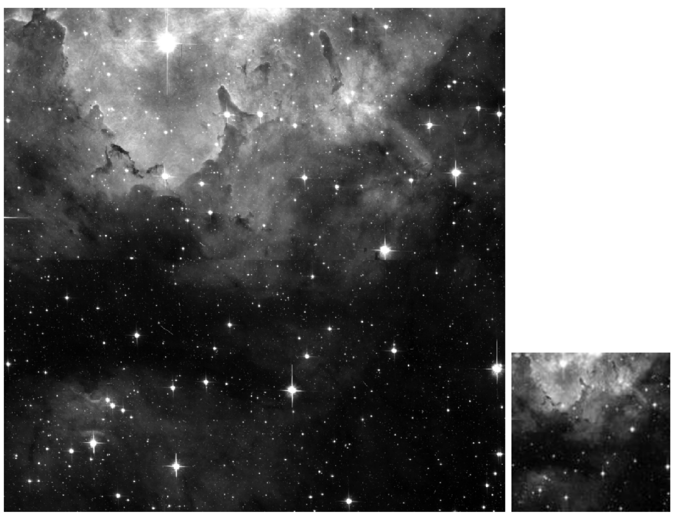

Fig. 7. An example of a `JPEG` image (left) and its corresponding `thumbnail` image using example dataset `jcs718koq`.

of these `thumbnail` images are to be able to view many of them on a single webpage in the `acsql` web application. An example of a `JPEG` image and its corresponding `thumbail` is shown in Figure N.

Unlike the ACS data products portion of the filesystem, the `JPEG` and `thumbnail` portions of the filesystem are organized based on the 5-digit `PROPOSID` of the corresponding observation instead of the first four characters of the `rootname`. This design was chosen as a means to simplify the design of the web application; users often which to view data based on the 5-digit `PROPOSID` and less often on the details of the `rootname`. An example of this sturcture is shown in Figure N. Note that the `thumbail` filesystem only contains thumbnails created from `FLT` filetypes, since thumbnails are only inteded for navigation and quick-viewing.

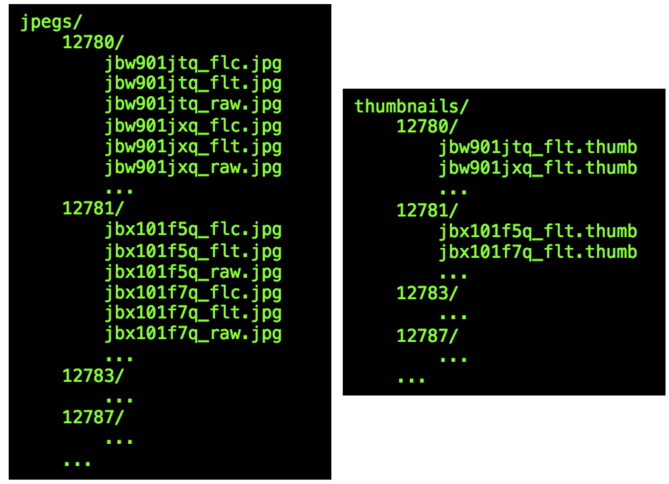

Fig. 8. A representation of the directory structure for the `JPEG` (left) and `thumbail` (right) portion of the `acsql` filesystem, using a few observations as an example.

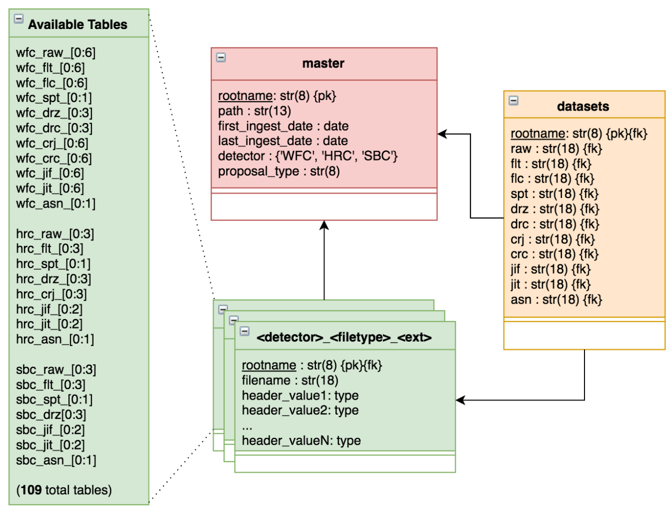

Fig. 9. The relational database schema for the `acsql` database.

3.5 Database: Relational Schema

Another major component of the `acsql` project is a relational database that stores all `FITS` header key/value pairs for each ACS filetype and `FITS` file extension across all on-orbit ACS observations. Such a database allows users to perform relational queries for any observational metadata.

To accomplish this, we implemented the relational schema shown in Figure N. The `acsql` database contains 111 tables in total: one `master` table which contains basic information about each `rootname` that is important for the `acsql` database in general, one `datasets` table which indicate which filetypes are available for a particular `rootname`, and 109 'header' tables which stores the header key/value pairs, one for each `detector/filetype/extension` combination (e.g. `wfc_raw_0`). Each of these tables are described in detail below.

The `master` table contains information that is particularly useful for maintaining and using the `acsql` database. Its primary key is the first 8 characters of the 9-character `rootname` for the partiuclar observation (recall from section 1.1.1 that only the first 8 characters of a rootname are actually unique). The path column contains the location of the observation in the acsql filesystem. The first_ingest_date and last_ingest_date contains the date in which the observation was first inserted into the database and the date in which the observation was most recenly updated in the database, respectively. The last_ingest_date allows the database maintainer to determine when data in the database may become outdated and require re-ingestion.

The datasets table lists which filetypes are available for each observation. If a particular filetype is available for the given rootname, the value for the appropriate column in the table is the full <rootname>_<filetype>.fits filename (for example, the raw column contains the value jcs718koq_raw.fits for rootname jcs718ko). If a particular filetype is not available, the value of the column is NULL. This table allows a user to determine which header tables are queryable for a given rootname. The rootname in the datasets table acts as both a primary key for the table as well as a foreign key that maps to the rootname of the master table.

The remaining 109 tables were designed to be in direct correspondance with the header metadata key/value pairs found in observations files; each column is named in the same manner as the header keys, with the value of that column reflecting the header value. There is one table for each detector, filetype, and extension combination; collectively, these are referred to as the 'header' tables. Like with the datasets table, the rootname column serves as a primary key for the header tables as well as a foreign key that maps to the rootname of the master table.

3.6 Database: MySQL + SQLAlchemy

The acsql database is stored on a MySQL server (Version 5.6) (Oracle, 2017) that is hosted at STScI. The database schema was implimented using SQLAlchemy, which is an open-source SQL toolkit and Object Relational Mapper (ORM) for python (Bayer, 2006). As an ORM, SQLAlchemy enables python classes to be easily translated to SQL-based database tables, and vice versa. Additionall, SQLAlchemy provides python methods for connecting to a SQL-based database and performing typical SQL tasks such as inserts, updates, and queries.

There are several key functions and classes that were used to construct the acsql database (all of which can be found in the acsql.database.database_interface.py module). One such function is the load_connection, as shown in Figure N. This function creates three SQLAlchemy objects that are used to establish a connection with the acsql database: engine, base, and session, each described below.

The engine object contains the Python Database API Specification (also known as DBAPI), which provides a low-level API for python-specific, commonly-used database tasks (Lemburg, 2017). It is created from the sqlalchemy.create_engine method, which requires a user-supplied connection_string. The connection_string is a string that contains information

```python
def load_connection(connection_string):
    """Return ``session``, ``base``, and ``engine`` objects for
    connecting to the ``acsql`` database.

    Create an ``engine`` using an given ``connection_string``. Create a
    ``base`` class and ``session`` class from the ``engine``. Create an
    instance of the ``session`` class. Return the ``session``,
    ``base``, and ``engine`` instances.

    Parameters
    ----------
    connection_string : str
        The connection string to connect to the ``acsql`` database. The
        connection string should take the form:
        ``dialect+driver://username:password@host:port/database``

    Returns
    -------
    session : sesson object
        Provides a holding zone for all objects loaded or associated
        with the database.
    base : base object
        Provides a base class for declarative class definitions.
    engine : engine object
        Provides a source of database connectivity and behavior.
    """

    engine = create_engine(connection_string, echo=False, pool_timeout=100000)
    base = declarative_base(engine)
    Session = sessionmaker(bind=engine)
    session = Session()

    return session, base, engine
```

Fig. 10. The load_conenction function, which is used to build a connection to the acsql database.

about the type of database, the specific database dialect being used, and the user credentials (e.g. username, password, port number, and host server name). In the case of the acsql, this connection string takes the form of 'mysql+pymysql://username:password@host:port/acsql'. The connection_string is imported from a user supplied config file within the acsql library (as will be discussed in section 3.9).

The base object serves as a base class for declarative class definitions (i.e. the classes that are used to define the database tables). It is created from the sqlalchemy.ext.declarative.declatative_base method. Perhaps most importantly, the base object contains methods for creating and dropping tables from the class definitions (e.g. base.metadata.create_all() and base.metadata.drop_all(), respectively).

The session object provides a primary usage interface for database operations, and is created via the sqlalchemy.sessionmaker method, which takes as a parameter the engine object. The methods of the session object are primarily used to query the database (i.e. session.query()) as well as commiting inserts or updates (i.e. session.commit()).

The master and datasets tables were implemented via explicit class definitions in database_interface, and are shown in Figures N and N, respectively. Each table column is defined using the sqlalchemy.Column object, which is a class that can be initialized with the datatype that will be stored in the column (e.g. String, Float, Integer,

```python
class Master(base):
    """ORM for the  master table."""
    def __init__(self, data_dict):
        self.__dict__.update(data_dict)

    __tablename__ = 'master'
    rootname = Column(String(8), primary_key=True, index=True, nullable=False)
    path = Column(String(15), unique=True, nullable=False)
    first_ingest_date = Column(Date, nullable=False)
    last_ingest_date = Column(Date, nullable=False)
    detector = Column(Enum('WFC', 'HRC', 'SBC'), nullable=False)
    proposal_type = Column(Enum('CAL/ACS', 'CAL/OTA', 'CAL/STIS', 'CAL/WFC3',
                                'ENG/ACS', 'GO', 'GO/DD', 'GO/PAR', 'GTO/ACS',
                                'GTO/COS', 'NASA', 'SM3/ACS', 'SM3/ERO',
                                'SM4/ACS', 'SM4/COS', 'SM4/ERO', 'SNAP'),
                           nullable=True)
```

Fig. 11. The class definition for constructing the `master` table via `SQLAlchemy`.

```python
class Datasets(base):
    """ORM for the datasets table."""
    def __init__(self, data_dict):
        self.__dict__.update(data_dict)

    __tablename__ = 'datasets'
    rootname = Column(String(8), ForeignKey('master.rootname'),
                      primary_key=True, index=True, nullable=False)
    raw = Column(String(18), nullable=True)
    flt = Column(String(18), nullable=True)
    flc = Column(String(18), nullable=True)
    spt = Column(String(18), nullable=True)
    drz = Column(String(18), nullable=True)
    drc = Column(String(18), nullable=True)
    crj = Column(String(18), nullable=True)
    crc = Column(String(18), nullable=True)
    jif = Column(String(18), nullable=True)
    jit = Column(String(18), nullable=True)
    asn = Column(String(18), nullable=True)
```

Fig. 12. The class definition for constructing the `datasets` table via `SQLAlchemy`.

etc.) as well as parameters that set SQL-like constraints and parameters on the column values. These include, but are not limited to, primary keys (e.g. the `primary_key=True` parameter in the `master.rootname` column), foreign key constrains (e.g. the `ForeignKey` constraint in the `datasets.rootname` column), uniqueness constraints (e.g. the `unique=True` parameters in the `master.path` column), and `NULL` constraints (e.g. the `nullable=False` parameter in the `master.first_ingest_date` column). `SQLAlchemy` determines the name of the table via the `__tablename__` attribute, and determines the name of the columns by the name of the variable used to initialize each `Column` object.

Since there are 109 header tables, some of which have hundreds of columns, it is not practical to construct a class definition for each table in a similar manner to that of the `master` and `datasets` table. Instead, these class definitions were implemented via the `database_interface.orm_factory` function, which is a factory function that creates and returns a class definition for each header table, based on the given `class_name`

```python
def orm_factory(class_name):
    """Create a SQLAlchemy ORM Classes with the given ``class_name``.

    Parameters
    ----------
    class_name : str
        The name of the class to be created

    Returns
    -------
    class : obj
        The SQLAlchemy ORM
    """

    data_dict = {}
    data_dict['__tablename__'] = class_name.lower()
    data_dict['rootname'] = Column(String(8), ForeignKey('master.rootname'),
                                   primary_key=True, index=True,
                                   nullable=False)
    data_dict['filename'] = Column(String(18), nullable=False, unique=True)
    data_dict = define_columns(data_dict, class_name)
    data_dict['__table_args__'] = {'mysql_row_format': 'DYNAMIC'}

    return type(class_name.upper(), (base,), data_dict)
```

Fig. 13. The `orm_factory` function, used to define class definitions for header tables.

that reflects the `detector/filetype/extension` combination (e.g. `wfc_raw_0`). The `orm_factory` function is shown in Figure N. Similar to the `Master` and `Datasets` classes, some of the columns in the `orm_factory` function are explicitly defined via the `SQLAlchemy Column` class. However, the columns that correspond to header key/value pairs are defined in a separate function named `define_columns`, shown in Figure N.

The purpose of the `define_columns` function is to define `SQAlchemy Column` objects for each header keyword in the headers of the particular `detector/filetype/extension` combination (provided in the given `class_name` parameter). This is accomplished by reading in a text file (named <class_name>.txt that contains the header keywords and their datatype (one per line) for the given `class_name`. An portion of an example text file is shown in Figure N.

Furthermore, the 109 text files used to define the header table columns are also generated in an automated fashion via the `acsql.database.make_tabledefs.py` module. This module uses a set of example FITS files to scrape its header contents, determine all of the header keywords and their datatypes, and write the results to a text file. Similarly, the `acsql.database.update_tabledefs.py` is used to add new header keywords by comparing the header contents of a given FITS file and the existing column definition text files[3].

With the implementation of the `orm_factory` and `define_columns` function, it is then trivial to create class definitions for each of the 109 header tables. An example of this is shown in Figure N, where the several of the `WFC` header tables are defined.

With the `master`, `datasets`, and each of the 109 header tables defined in the `database_interface` module, cre-

3. New header keywords are occaisonally introduced to ACS data proceeding updates to its calibration software

```python
def define_columns(data_dict, class_name):
    """Dynamically define the class attributes for the ORM

    Parameters
    ----------
    data_dict : dict
        A dictionary containing the ORM definitions
    class_name : str
        The name of the class/ORM.

    Returns
    -------
    data_dict : dict
        A dictionary containing the ORM definitions, now with header
        definitions added.
    """

    special_keywords = ['RULEFILE', 'FWERROR', 'FW2ERROR', 'PROPTTL1',
                        'TARDESCR', 'QUALCOM2']

    with open(os.path.join(os.path.split(__file__)[0], 'table_definitions',
                           class_name.lower() + '.txt'), 'r') as f:
        data = f.readlines()
    keywords = [item.strip().split(', ') for item in data]
    for keyword in keywords:
        if keyword[0] in special_keywords:
            data_dict[keyword[0].lower()] = get_special_column(keyword[0])
        elif keyword[1] == 'Integer':
            data_dict[keyword[0].lower()] = Column(Integer())
        elif keyword[1] == 'String':
            data_dict[keyword[0].lower()] = Column(String(50))
        elif keyword[1] == 'Float':
            data_dict[keyword[0].lower()] = Column(Float(precision=32))
        elif keyword[1] == 'Decimal':
            data_dict[keyword[0].lower()] = Column(Float(precision='13,8'))
        elif keyword[1] == 'Date':
            data_dict[keyword[0].lower()] = Column(Date())
        elif keyword[1] == 'Time':
            data_dict[keyword[0].lower()] = Column(Time())
        elif keyword[1] == 'DateTime':
            data_dict[keyword[0].lower()] = Column(DateTime)
        elif keyword[1] == 'Bool':
            data_dict[keyword[0].lower()] = Column(Boolean)
        else:
            raise ValueError('unrecognized header keyword type: {}:{}'.format(
                keyword[0], keyword[1]))

        if 'aperture' in data_dict:
            data_dict['aperture'] = Column(String(50), index=True)

    return data_dict
```

Fig. 14. The `define_columns` function, used to define columns used in the header tables.

ating the database tables on the `MySQL` server is accomplished by executing the `base.metadata.create_all()` method.

3.7 Data ingestion software

3.8 Website:

3.9 `acsql` Library

4 Results

Topics to discuss: 1. GitHub repository 2. ReadTheDocs documentation repository 3. Quantification of Database records 4. Quantification of Code repository 5. Website location

```
DETECTOR, String
NEXTEND, Integer
EXTEND, Bool
SIMPLE, Bool
NAXIS, Integer
LINENUM, String
GROUPS, Bool
DATE, String
EQUINOX, Float
INSTRUME, String
PROPOSID, Integer
ASN_ID, String
ASN_PROD, Bool
ASN_STAT, String
DEC_TARG, Float
FILETYPE, String
ASN_TAB, String
PRIMESI, String
RA_TARG, Float
TARGNAME, String
TELESCOP, String
PR_INV_F, String
BITPIX, Integer
PR_INV_M, String
PR_INV_L, String
```

Fig. 15. The contents of an example text file used to define the columns of a header table in the `define_columns` function. The example table used here is the `wfc_asn_0` table.

```python
WFC_raw_0 = orm_factory('WFC_raw_0')
WFC_raw_1 = orm_factory('WFC_raw_1')
WFC_raw_2 = orm_factory('WFC_raw_2')
WFC_raw_3 = orm_factory('WFC_raw_3')
WFC_raw_4 = orm_factory('WFC_raw_4')
WFC_raw_5 = orm_factory('WFC_raw_5')
WFC_raw_6 = orm_factory('WFC_raw_6')

WFC_flt_0 = orm_factory('WFC_flt_0')
WFC_flt_1 = orm_factory('WFC_flt_1')
WFC_flt_2 = orm_factory('WFC_flt_2')
WFC_flt_3 = orm_factory('WFC_flt_3')
WFC_flt_4 = orm_factory('WFC_flt_4')
WFC_flt_5 = orm_factory('WFC_flt_5')
WFC_flt_6 = orm_factory('WFC_flt_6')

WFC_flc_0 = orm_factory('WFC_flc_0')
WFC_flc_1 = orm_factory('WFC_flc_1')
WFC_flc_2 = orm_factory('WFC_flc_2')
WFC_flc_3 = orm_factory('WFC_flc_3')
WFC_flc_4 = orm_factory('WFC_flc_4')
WFC_flc_5 = orm_factory('WFC_flc_5')
WFC_flc_6 = orm_factory('WFC_flc_6')
```

Fig. 16. An example of how the `orm_factory` function is called to create class defintions for the header tables.

5 Conclusion

The conclusion goes here.

6 Discussion

Topics to discuss:

1. Possible simplification based on MAST archive 2. Possible extensions to other insturments

Appendix A

acsql code

Appendix one text goes here.

Acknowledgments

The authors would like to thank...

References

[1] *A Guide to NumPy/SciPy Documentation*, [Online; accessed 2017-08-05], available at https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt

[2] Avila, R., et al., 2017, *ACS Instrument Handbook*, Version 16.0 (Baltimore: STScI)

[3] Bayer, M., 2006, *SQLAlchemy: The database toolkit for Python*, [Online; accessed 2017-02-21], available at http://www.sqlalchemy.org/.

[4] Brandi, G., et al., 2007, *Sphinx: Python Documentation Generator*, available at http://www.sphinx-doc.org/en/stable/.

[5] *Definition of the Flexible Image Transport System (FITS): The FITS Standard*, 2008, International Astronomical Union FITS Working Group, available at https://fits.gsfc.nasa.gov/standard30/fits_standard30aa.pdf.

[6] *git*, [Online; accessed 2017-08-05], available at https://git-scm.com.

[7] *GitHub*, [Online; accessed 2017-08-05], available at https:/github.com.

[8] Goodger, D., 2001, *PEP 257 – Docstring Conventions*, Python Developer's Guide, available at https://www.python.org/dev/peps/pep-0257/.

[9] Lemburg, M., 2017, *PEP 249 – Python Database API Specification v2.0*, Python Developer's Guide, available at https://www.python.org/dev/peps/pep-0249/.

[10] Loper, E., 2004, *Epydoc: Generating API Documentation in Python*, Proceedings of the Second Annual Python Conference, available at http://epydoc.sourceforge.net/.

[11] *MySQL 5.6 Reference Manual*, Oracle, [Online; accessed 2017-08-13], available at https://dev.mysql.com/doc/refman/5.6/en/

[12] *Read the Docs*, [Online; accessed 2017-08-05], available at https://readthedocs.org.

[13] Robitaille, T.P., et al., 2013, *Astropy: A community Python package for astronomy*, Astronomy & Astrophysics, 558, A33.

[14] Smith, E., et al., 2011, *Introduction to the HST Data Handbooks*, Version 8.0 (Baltimore: STScI)

[15] *The Barbara A. Mikulski Archive for Space Telescopes*, [Online; accessed 2017-07-30], available at https://archive.stsci.edu/.

[16] van der Walt, S., Colbert, C., and Varoquaux, G., 2011, *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, 13, 22-30.

[17] van Rossum, G., 2001, *PEP 8 – Style Guide for Python Code*, Python Developer's Guide, available at https://www.python.org/dev/peps/pep-0008/.