The Hubble Space Telescope Advanced Camera for Surveys Quicklook Application

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

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

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

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

Abstract—The Hubble Space Telescope (HST) Advanced Camera for Surveys (ACS) has been acquiring thousands of astronomical images each year since its installation in 2002. The ACS Quicklook Application (`acsql`) provides a means for users to discover and interact with these data through a database-driven web application. The system is comprised of several components: (1) A ∼40 TB network file system, which stores all on-orbit ACS data files on disk, (2) a `MySQL` database, which stores observational metadata in a normalized relational form and allows users to build custom datasets based on observational parameters, (3) A Python/`Flask`-based web application, which allows users to view "Quicklook" JPEG images of any publicly-available ACS data along with their metadata, and (4) a Python code library, which provides a platform on which users can build automated instrument calibration and monitoring routines. The `acsql` application may be extended to support the forthcoming James Webb Space Telescope (JWST) mission, which is scheduled to launch in 2019.

1 Introduction

The Advanced Camera for Surveys (ACS) is an imaging instrument on board the Hubble Space Telescope (HST) that was installed in 2002 during Servicing Mission 3B (shuttle mission STS-109)[1]. 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), designed for far-ultraviolet imaging and spectroscopy. ACS experienced an electronics failure in 2007 that affected the WFC and HRC detectors until 2009 when astronauts successfully restored the WFC detector during Servicing Mission 4 (shuttle mission STS-125)[2]; the HRC still remains unoperational.

Despite this electronics failure, ACS has been steadily acquiring astronomical images over its 15 year on-orbit lifetime. Figure 18 shows an estimate of the number of observations over time for each of the three detectors. To date, there have been nearly 200,000 observations total. Further information about the ACS instrument including its history, configuration, performance, and scientific capability can be found in the ACS Instrument Handbook[3].

ACS data, along with the data from the other HST instruments past and present (e.g. The Wide Field Camera 3 (WFC3), The Cosmic Origins Spectrograph (COS), etc.), are publicly available and primarily stored in the Barbara A. Mikulski Archive for Space Telescopes (MAST)[1][4]. Through

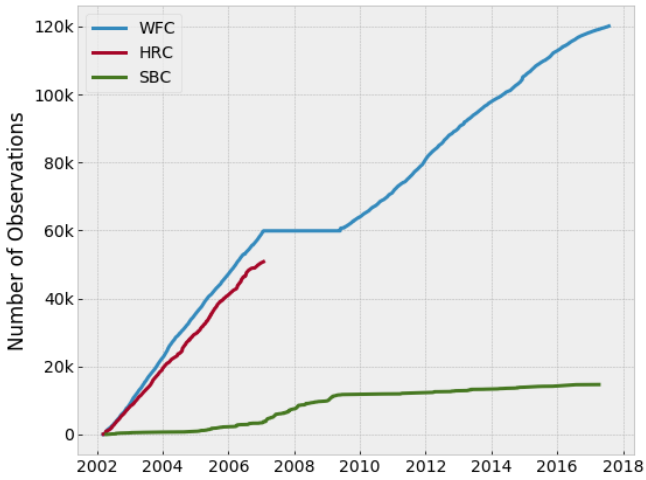


Fig. 1: *The number of observations over time for each of the three detectors on ACS.*

MAST, users can request and retrieve data for any publicly-available dataset via `ftp`, `sftp`, or DVD by mail[2]. Like is standard for other astronomical data, ACS data are stored using the Flexible Image Transport System (FITS) data format[5]. This data format has several unique characteristics, as will be discussed in section **??**.

The ACS Quicklook Application (hereinafter referred to as "`acsql`") is a Python-based application for discovering,

Manuscript submitted October 06, 2017

1. Named after the U.S. Senator from Maryland who has been a pivotal 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.

viewing, and querying all publicly-available ACS data. It consists of: (1) A filesystem that stores ACS instrument data files and "Quicklook" JPEGs in an organized Network File System (NFS) (hereinafter referred to as the "acsql filesystem"), (2) A MySQL database that stores observational metadata of each observation (hereinafter referred to as the "acsql database"), (3) A Python/Flask-based web application for interacting with the filesystem and database (hereinafter referred to as the "acsql web application"), and (4) A Python code library that contains software for connecting to the database, ingesting new data, logging production code execution, and building/maintaining the web application (hereinafter referred to as the "acsql library" or "acsql package"). Each component is explained in further detail in Chapter **??**.

This paper outlines the acsql application as part of the Towson University Computer Science Masters Program Graduate Project. The remaining sections in this chapter discuss the motivation and use cases for this application, as well as details on the underlying data structure on which this project was built. Chapter **??** discusses how the acsql application compares to related work. Chapter **??** details the implementation of each acsql component. Chapter **??** outlines the results of the projectand the project deliverables. Lastly, chapters **??** and **??** conclude the paper with a discussion of possible extensions and modifications to the application.

We note 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)[6].

1.1 Data Structure

The design of acsql, particularly the acsql database, is heavily dependent on the underlying data structure of ACS FITS files. As such, it is important for the reader to have a conceptual understanding of this data structure. The following sections are dedicated to providing an overview on the FITS data format and ACS-specific intricacies.

1.1.1 Filenames

The data from each ACS observation is contained within a FITS data file and named in a consistent fashion:

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

Each <rootname> consists of nine unique alpha-numeric characters, and <filetype> is one of several possible three-character filetype options (discussed in section **??**). For example, one ACS observation has the filename j6mf16lhq_raw.fits (Principle Investigator Gary Bernstein, observation date 2016-09-22). Each character in the 9-character rootname has a specific meaning, and is discussed in section 5.2 of the Introduction to the HST Data Handbooks[7]. The .fits extension at the end of the filename signifies that the file is of the FITS data format.

The unique 9-character rootname is associated with an individual, distinct ACS observation. However, it should be noted that only the first eight characters of the 9-character rootname are truly unique; the last character, which identifies the source of data transmission from the telescope[3], has several possible values. However, varying values of the last character do not associate with different observations. For this reason, as will be discussed in section **??**, we adopt an 8-character rootname as a primary key for the acsql database tables to denote unique ACS observations.

1.1.2 FITS file structure

Each ACS FITS file consists of several "extensions", each describing 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 date the particular observation was made was 2016-09-22), 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 data values are the measure of the brightness of the astronomical scene, (2) 'error' (ERR), in which the data values are the measure of the uncertainty in the pixel values of the SCI extension, and (3) 'data quality' (DQ), in which the data values are 16-bit flags that describe the quality of the pixel values for the detector (for example, they may indicate that certain pixels were affected by cosmic rays during 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, containing metadata that is common to all extensions. This is referred to as the "primary header'.

Tables **??** and **??** 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.

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

Over the years there have been several tools written in various programming languages to read FITS files into memory as well as automatically convert their extension data into multi-dimensional array data types and their extension headers to dictionary or string data types. For this project, the astropy.io.fits Python module is used

3. For example, q = solid-state recorder, s = retransmitted solid-state recorder

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

extensively to read and interact with the data of ACS FITS files[8].

1.1.3 FITS file extension headers

As mentioned in section **??**, each FITS extension contains a header, which contains key/value pairs of metadata associated with the extension data. Such metadata may describe the astronomical observation (e.g. target name, exposure time, principle investigator name, etc.), telemetry of ACS instrument 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 **??**. Note that 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.

```
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: *A section of an example header, taken from the 0th extension of the file* `j6mf16lhq_raw.fits`.

1.1.4 FITS filetypes for ACS

As discussed in section **??**, each ACS observation may result in several FITS filetypes. Each filetype has a specific scientific application and 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. However, these details can be ascertained from the ACS Data Handbook[9]. To provide 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 corrected for Charge Transfer Efficiency (CTE) deficits.
- **drz** - geometric distortion-corrected data
- **drc** - geometric distortion-corrected and 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 results in the filetypes `raw`, `flt`, `jit`, `jif`, and `spt`.

1.2 Key Metadata

There are several metadata key/value pairs that are particularly important for the `acsql` application, namely the web application which often presents these metadata to the user. To provide some context for the remainder of this paper, these metadata are briefly described below. Note that the `rootname` and `proposal_type` are not metadata from extension headers, but rather are metadata explicitly added to the `acsql` database schema.

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

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. `52473.84839`).

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 # 2 (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 4 or 5-digit 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. `j9q15401`).

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 `acsql` application is driven by several limitations of the FITS file structure as well as limitations in the current capabilities of MAST from specific user perspectives. Some of these limitations are described below, along with how the `acsql` application aims to address them. We also discuss the intended users of the application and their expected use cases.

1.3.1 Data retrieval latency

Currently, users who wish to retrieve data from MAST must submit a retrieval request via the MAST online interface or an internal XML request. Once the retrieval request is processed (usually automatically unless it is a request of a large number of datasets), the data are either transferred to the user directly via Secure File Transfer Protocol (`sftp`), transferred to a "staging area" in which the user can log into and copy the data via File Transfer Protocol (`ftp`) at their leisure, or sent by mail via DVD, depending on which option the user selects. In any case, the duration between a download request and the time at which the user has fully retrieved the data may be significant. In the fastest scenario of an automatically accepted request and the `sftp` option, a typical request can take minutes to hours to be completed. Furthermore, there are limited options available for programmatically obtaining new data; users who wish to retrieve the latest available data must (1) discover which datasets are new via a query to the MAST database, (2) construct a download request, (3) submit the download request, and (4) await or retrieve the data. This workflow makes it non-trivial to discover and perform analysis on new data within the same software program.

The `acsql` application attempts to circumnavigate this retrieval process by making the full data products instantly available via read-only access of the `acsql` filesystem, as well as a view of the image data (and corresponding metadata) instantly available through the `acsql` web application. By having all of the available data centrally located, users need not to go through this request process; data can be directly read from the storage areas on disk.

1.3.2 File I/O

A significant limitation of the FITS file format is the amount of time required for file input/output. With users often finding themselves analyzing hundreds to thousands of images, this file I/O can be quite a burden on the total processing time of such analysis. The `acsql` application attempts to mitigate this processing time by storing the valuable image metadata that users often seek in a relational database.

Queries to the `acsql` database prove to be much faster than opening and closing individual FITS files, and the difference in data retrieval times increases linearly with increasing amount of files. Consider an example situation in which a user wishes to retrieve the date of observation (i.e. the `DATE-OBS` header keyword in the 0th extension of a `raw` FITS file) for one hundred files. Figure **??** shows that the total time required to gather this information is $\sim$10-fold less when querying the `acsql` database than when retrieving the data directly from the FITS files, with this gap increasing linearly with increasing number of files at a rate of $\sim$2.5 seconds per 100 files.

In order to build the `acsql` database, each ACS FITS file must ultimately be opened in order to retrieve the header information. However, this operation is only performed once and needs not to be performed by the user; once the database is built, users can retrieve data by fast database queries.

1.3.3 Data redundancy

As will be discussed in section **??**, the primary user base for the `acsql` application is a group of ACS instrument analysts who use ACS data on a daily basis. As such, each user may require access to a varying amount of ACS data over time; some data may be useful for their work on a particular day, but not on another, while some data are consistently used over time. Furthermore, instrument analysts may share the need for specific data with one or more of their colleagues.

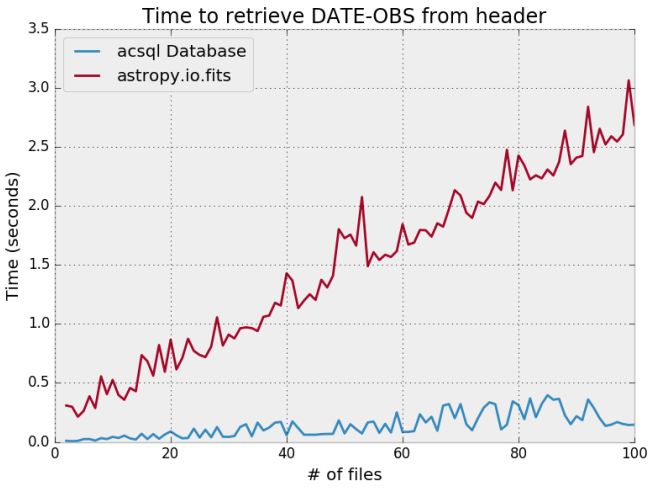


Fig. 3: *The amount of time it takes to retrieve the* `DATE-OBS` *header keyword from a varying number of ACS data files via reading the FITS file using the* `astropy.io.fits` *module (red) and querying the* `acsql` *database (blue).*

Consider a situation in which two instrument analyist require data from the proposal `11655`. Traditionally, for the two instrument analysts to obtain the data they need, they would have to submit a download request to MAST for data from `11655` and store the data in personal directories. Unless this effort is explicitly coordinated amongst the two individuals, they will each likely download and store a separate copy of this data, when in reality, only one copy is needed. This workflow leads to possible data redundancy. With the ACS instrument team employing roughly a dozen instrument analysts, the scale of this data redundancy can grow quite large.

Recently, MAST has mitigated this issue through what is known as the MAST public cache, which is a centrally-located, organized network file system that stores all publicly-available HST data and is internal to STScI. The `acsql` filesystem is built on top of the MAST public cache to provide accessibility to all ACS data in one central location. Furthermore, the `acsql` application provides the `acsql` database (for observational metadata needs) and the `acsql` web application (for quick data viewing). With the combination of these three components, instrument analysts have instant access to all ACS data at any time in one centralized location; there is no need for coordination amongst other instrument analysts or storage of separate copies.

1.3.4 Data discovery

As will be discussed in section **??**, ACS instrument analysts often wish to know which datasets (i.e. rootnames) exist for various observational parameters. For example, one may need to analyze data that were observed between a specific time period, or observed with a specific pointing on the sky. Conversely, an instrument analyst may have knowledge of specific rootnames, but wish to learn of the observational parameters associated with them.

MAST has tools which allow users to discover rootnames and/or parameters for observations. The MAST portal is a public-facing online data discovery tool for various data collections such as HST data, high level science products, and catalogs of astronomical source positions[10]. The MAST portal offers features that may be of particular interest to research astronomers at academic institutions, for example; an image of the location of the scientific target on the sky ('AstroView'), an interactive GUI for displaying results, and an option to export results to various file types. An example query using this tool is shown in Figure 4.

Though the MAST portal has many useful features, there are some limitations to the tool from the perspective of an instrument analyst at STScI. One such limitation is that the available observational metadata do not include all of the data available in the FITS headers; it is limited to a small subset of header keywords, mainly from the RAW and FLT filetypes, as well as other MAST-generated metadata not found in the FITS headers (e.g. 'Target Classification'). Another limitation is that the queries are keyed off of an 'Observation ID' as opposed to a traditional 9-character `rootname`. The Observation ID may map to one or more individual observations (and thus one or more individual `rootnames`), making it difficult for instrument analysts to discover which `rootnames` (and thus data files) that are associated with the observational metadata they seek.

The `acsql` application provides an intuitive, one-to-one correspondance between the FITS header values and the `acsql` database schema. The column names in the tables of the `acsql` database exactly match the names found in the FITS headers, and each extension of each ACS filetype is supported. Furthermore, the primary key of each table in the database is the 9-character `rootname` that instrument analysts use in their day-to-day work, as opposed to an Observation ID. With this configuration, it is straightforward to discover data via a database query using header keywords.

1.3.5 Use Cases

The intended primary users of the `acsql` application are ACS instrument analysts that work for the ACS instrument team at STScI. ACS instrument analysts use ACS data frequently in their day-to-day work, using it to help calibrate, monitor, and characterize various aspects of the ACS instrument. On any given day, an instrument analyst may need to analyze data from a number of observing modes, time periods, or filetypes. In this regard, the `acsql` application aims to limit the amount of time and effort required by the analysts to discover and access the data that they need to perform such work. It should be noted, though, that the `acsql` application is not necessarily limited to this user base; since the application contains only public data, there is potential for users external to STScI, such as scientific researchers at other academic institutions.

With the nature of ACS instrument work in mind, the `acsql` application was designed to support four main use cases, each briefly described below.

Use case 1: Visually inspect an image from the ACS archive: The application shall allow users to view a 'Quicklook' JPEG of any publicly-available image in the ACS archive, along with useful corresponding observational metadata.

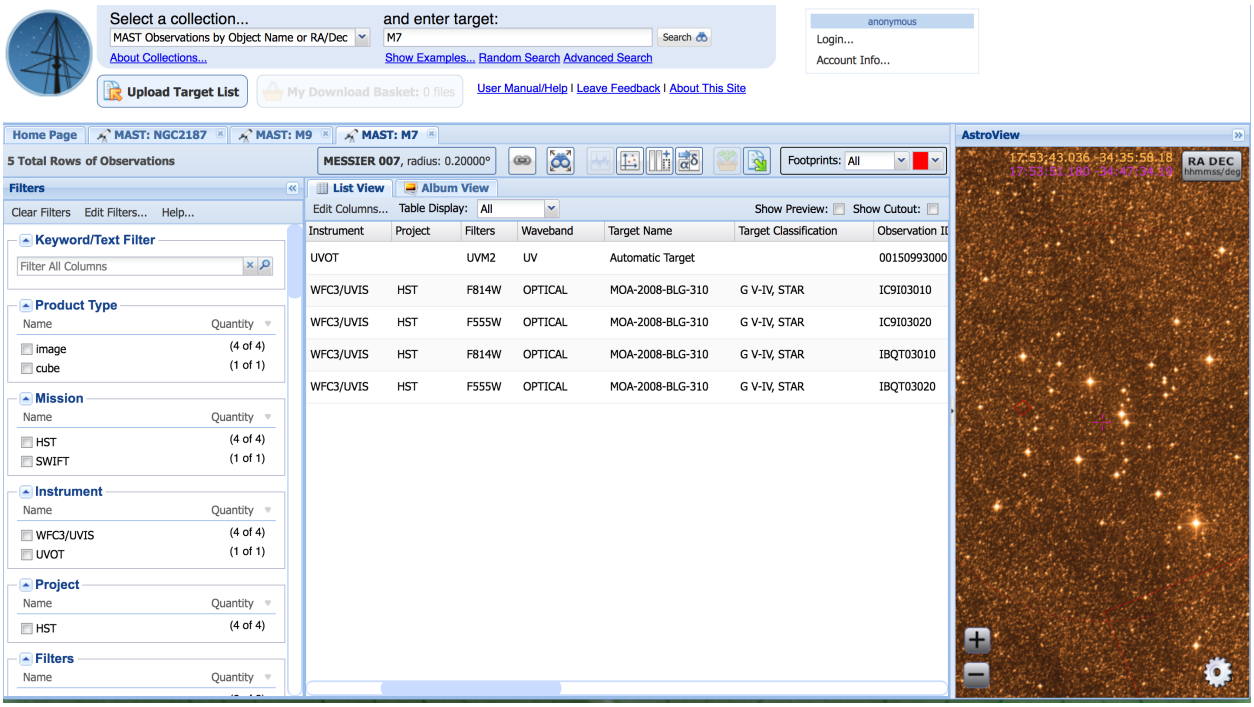


Fig. 4: *An example of a query to the MAST data discovery portal.*

Use case 2: Determine which datasets exist in the ACS archive for a given set of observational parameters: The application shall allow users to determine which 9-character rootnames exist for a defined set of observational parameters via ACS header keywords.

Use case 3: Determine the observational parameters for a given dataset: The application shall allow users to determine observational parameters (i.e. FITS header keyword values) for a given 9-character `rootname`.

Use case 4: Programmatically analyze images across custom datasets: The application shall allow users to directly read ACS image data into memory and perform analysis for data obtained in Use Case (2).

2 Related Work

The main inspiration for the construction of the `acsql` application was derived from the success of a similar application for the Wide Field Camera 3 (WFC3) instrument (also on board HST), known as the WFC3/Quicklook (`wfc3ql`) application. The `wfc3ql` application shares many of the core components that `acsql` does, including a filesystem that stores all WFC3 data on disk, a relational database that stores WFC3 FITS header information, and a web application for user interactivity with the system[11][12]. The `wfc3ql` application has been developed by the WFC3 team at STScI since the instrument was installed on HST in 2009 during Servicing Mission 4[2]. The genesis of the application was a single script that performed a download request to MAST for new data, created Quicklook JPEGs from the resulting images, and saved them to a centrally-located directory. Over time, the project has evolved into the filesystem-database-web stack that more closely resembles that of the `acsql` application.

Despite the similarities, it should be noted that the implementation of the `wfc3ql` infrastructure differs significantly from that of `acsql` (implementation details of the `acsql` components are outlined in Chapter **??**). For example, unlike the `acsql` filesystem, which contains only publicly-available data, the `wfc3ql` filesystem is essentially a duplicate of the MAST archive for WFC3, contains all data, including proprietary data which can only be used internally by the WFC3 team at STScI. Since the `wfc3ql` filesystem contains proprietary data, much support is needed in the software infrastructure to restrict data to certain file directories, user groups, and file permissions. Additionally, the `wfc3ql` database only supports a small subset of FITS filetypes and extensions, while `acsql` supports all ACS filetypes and extensions.

With `wfc3ql` being continuously developed over the past eight years by a team of several developers, many features have been implemented for the application that are not currently implemented on `acsql`. For example, the `wfc3ql` application supports the daily visual inspection of new WFC3 data, some data of which are proprietary. Such a feature is not possible for `acsql`, as the `acsql` filesystem is built using the MAST cache of only publicly-available data (and thus the `acsql` application can be considered open-source and non-proprietary). Secondly, `wfc3ql` currently contains several instrument calibration and monitoring routines built upon the `wfc3ql` automation platform and code library (known as `pyql`), while `acsql` has yet to implement any such routines. A third example is that the `wfc3ql` web application supports the tracking of WFC3 instrument image 'anomalies' (i.e. anomalous features in images that are tracked by instrument team members[13]). These fea-

tures, amongst others, clearly serve as possible extensions to the acsql system; such extensions are discussed further in Chapter **??**.

3 Methodology and Implementation

In this chapter, we discuss the methods by which we implemented the various components of the acsql application. Additionally, we discuss the programming standards and workflows (sections **??** and **??**) that were employed to foster code qualities such as readability, maintainability, extensibility, etc.; we believe that this aspect of the project is equally important to the application and its future maintenance as its individual components.

3.1 Version control

All software associated with this project (including this paper itself) is version-controlled using the git Version Control System (VCS)[14]. The git repository for the project is named "acsql" and is hosted on GitHub, a repository hosting service[15]. The repository is publicly available at http://github.com/spacetelescope/acsql/.

Several feature branches of the software were created throughout the building of the acsql application such that the master branch (which is considered the 'production' branch) always contained operational software while the feature branches may have contained unfinished implementations. Such feature branches include create-database (for implementation of the acsql database schema), add-logging (for implementation of system logging), build-ingest (for implementation of the data ingestion software), and web-application (for implementation of the acsql web application). For each merge of a feature branch, a tag and release was created for the master branch, which allowed 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, GitHub allowed for issue tracking for bugs, questions, 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[16], (2) The PEP257 Python guide for module and function docstrings [17], and (3) the numpydocs documentation standard [18]. 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 did we diverge from these conventions, such as exceeding the recommended 80 characters line length to allow for greater readability. By following these conventions, the style of the acsql software is constistent across each module and reflects the style of industry-grade Python code.

The PEP257 guide for docstring conventions describes standards 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 details in addition to the PEP257 conventions and is used in many Python packages including the numpy (numerical Python) and scipy (scientific Python) packages[19]. Figure **??** shows an example of these conventions, taken from the acsql.ingest.ingest.get_proposal_type function.

Another benefit of using PEP257 and numpydoc docstring conventions is that API documentation creation tools such as sphinx[20] or epydoc[21] can automatically convert the docstrings 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 readthedocs, an open-source, community supported tool for hosting and browsing documentation[22]. The output documentation as seen on readthedocs for the example function in Figure **??** is provided in Figure **??**. The documentation for acsql is available at http://acsql.readthedocs.io/.

3.3 Filesystem: Archive of ACS data

The acsql filesystem is a Network File System (NFS) that stores all publically available on-orbit ACS data on disk in an organized set of directories and subdirectories hosted at STScI. Figure **??** shows an example of this directory structure. The parent directory is the first four characters of the 9-character rootname, which has a one-to-one correspondence with an individual PROPOSID. The subdirectories of the parent directories are named after the full 9-character rootname such that each parent directory contains a rootname subdirectory for each rootname that were observed for the particular PROPOSID. Furthermore, each rootname subdirectory contains every available file-

```
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. 5: *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. 6: *The* `readthedocs` *documentation for the* `acsql` *example function seen in Figure* **??**.

type (as described in Section **??**) for the particular observation.

```
filesystem/
    jcp0/
        jcp001kwq/
            jcp001kwq_flc.fits
            jcp001kwq_flt.fits
            jcp001kwq_raw.fits
            jcp001kwq_spt.fits
            jcp001kwq_jif.fits
            jcp001kwq_jit.fits
        jcp001010/
            jcp001010_asn.fits
            jcp001010_drc.fits
            jcp001010_drz.fits
            jcp001010_jif.fits
            jcp001010_jit.fits
        jcp014tyq/
            ...
        jcp001kt1/
            ...
        ...
    jcp3/
        ...
    jcp7/
        ...
    ...
```

Fig. 7: *A representation of the directory structure within the* `acsql` *filesystem, using a few observations as an example.*

Figure **??** shows how the total size of the filesystem has evolved over the lifetime of the ACS 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 observation (for example, full-frame observations result in larger file sizes than subarray observations, calibrated filetypes have larger file sizes than un-calibrated filetypes, etc.).

Currently, the `acsql` filesystem utilizes the MAST public cache (described in section **??**) to store all of the ACS data. We purposely omit details of the location and exact structure of the MAST cache as to not expose sensitive data that is internal to STScI. We note, however, that it is possible for users external to STScI to reconstruct the `acsql` filesystem (or a chosen subset of the filesystem) by requesting the publicly-available data from MAST and organizing the files

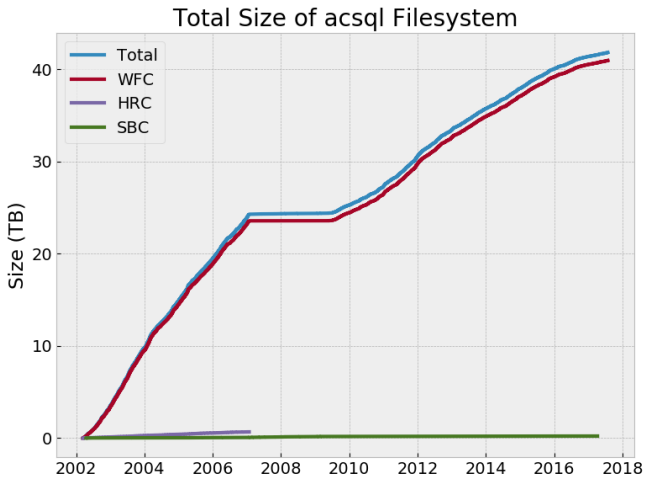


Fig. 8: *The size of the* `acsql` *filesystem as a function of observation date.*

in a similar manner to that shown in Figure **??**.

3.4 Filesystem: Archive of JPEGs and Thumbnails

In addition to the ACS data products described in 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 utilized by the `acsql` web application to allow users to quickly and easily view ACS data without having to physically open the corresponding FITS files.

The JPEG images are generated by (1) taking the two-dimensional data from the `SCI` extension(s), (2) sigma-clipping the top and bottom 1% of the values (as to avoid large outlier values and enhance the scaling of the image), and (3) saving the data as a JPEG format. The thumbnail images are created by simply resizing the corresponding JPEG into a 128x128 pixel image and saving the resulting image to a `.thumb` extension; the purpose 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 thumbnail is shown in Figure **??**.

Unlike the ACS data products portion of the filesystem (section **??**), the JPEG and thumbnail portions of the filesystem are organized based on the four-or-five 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 wish to view data based on the 4/5-digit `PROPOSID` and less often on the details of the `rootname`. An example of this structure is shown in Figure **??**. Note that the `thumbnail/` filesystem only contains thumbnails created from `FLT` filetypes, since the thumbnails are only intended for navigation and quick-viewing.

3.5 Database: Relational Schema

Another major component of the `acsql` application is a relational database that stores all `FITS` header key/value pairs for each ACS filetype and each `FITS` file extension for all on-orbit ACS observations. Such a database allows users

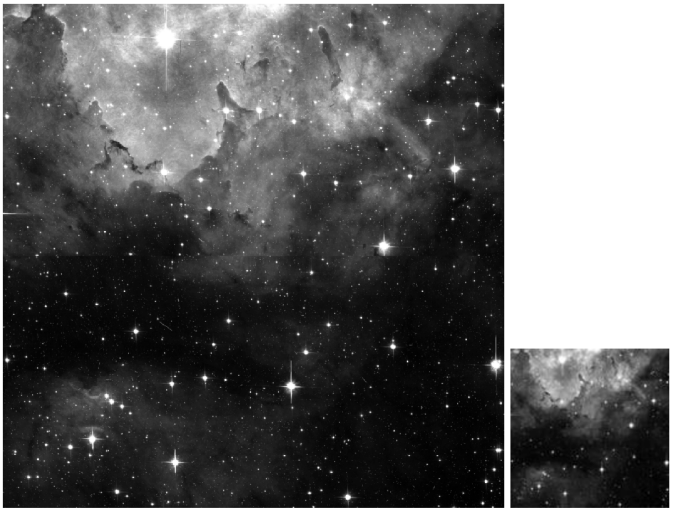


Fig. 9: *An example of a JPEG image (left) and its corresponding thumbnail image using example dataset* `jcs718koq`.

```
jpegs/
    12780/
        jbw901jtq_flc.jpg
        jbw901jtq_flt.jpg
        jbw901jtq_raw.jpg
        jbw901jxq_flc.jpg
        jbw901jxq_flt.jpg
        jbw901jxq_raw.jpg
        ...
    12781/
        jbx101f5q_flc.jpg
        jbx101f5q_flt.jpg
        jbx101f5q_raw.jpg
        jbx101f7q_flc.jpg
        jbx101f7q_flt.jpg
        jbx101f7q_raw.jpg
        ...
    12783/
        ...
    12787/
        ...
    ...

thumbnails/
    12780/
        jbw901jtq_flt.thumb
        jbw901jxq_flt.thumb
        ...
    12781/
        jbx101f5q_flt.thumb
        jbx101f7q_flt.thumb
        ...
    12783/
        ...
    12787/
        ...
    ...
```

Fig. 10: *A representation of the directory structure for the JPEG (top) and thumbnail (bottom) portion of the* `acsql` *filesystem, using a few observations as an example.*

to perform relational queries for any observational metadata based on the header keywords.

To accomplish this, we implemented the relational schema shown in Figure **??**. The `acsql` database contains 111 tables in total: one `master` table, which contains basic information about each `rootname` that is important for maintaining the database, 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 particular observation (recall from section **??** 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 recently 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 correspondence with the header metadata key/value pairs found in the ACS FITS 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 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`)[23] that is hosted at STScI. The database schema was implemented using `SQLAlchemy`, which is an open-source `SQL` toolkit and Object Relational Mapper (ORM) for Python[24]. As an ORM, `SQLAlchemy` enables Python classes to be easily translated to `SQL`-based database tables, and vice versa. Additionally, `SQLAlchemy` provides 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, further discussed in section **??**). One such function is `load_connection`, shown in Figure **??**. This function

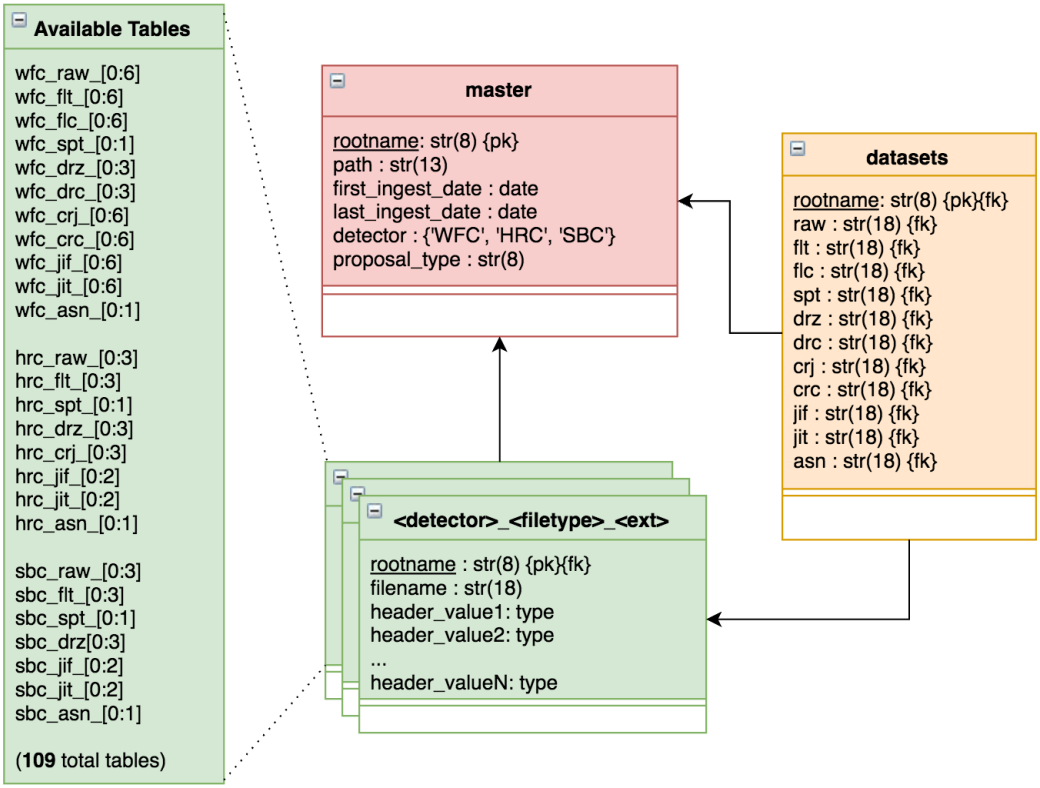


Fig. 11: *The relational database schema for the* `acsql` *database.*

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[25]. Created from the `sqlalchemy.create_engine` method, this requires a user-supplied `connection_string`. The `connection_string` contains information about the database type (e.g. `MySQL`, 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` database, 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 **??**).

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 the tables defined in 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.

```
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. 12: *The* `load_connection` *function, used to build a connection to the* `acsql` *database.*

session.query()) as well as commiting inserts or updates (i.e. session.commit()).

```
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. 13: *The class definition for constructing the* `master` *table via* `SQLAlchemy`.

```
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. 14: *The class definition for constructing the* `datasets` *table via* `SQLAlchemy`.

The `master` and `datasets` tables were implemented via explicit class definitions in `database_interface`, and are shown in Figures **??** and **??**, respectively. Each table column is defined using the `sqlalchemy.Column` object, which is a class that can be initialized with the column datatype (e.g. `String`, `Float`, `Integer`, etc.) as well as parameters that set `SQL`-like constraints 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.

```
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. 15: *The* `orm_factory` *function, used to define class definitions for header tables.*

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` that reflects the `detector / filetype / extension` combination (e.g. `wfc_raw_0`). The `orm_factory` function is shown in Figure **??**. 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 **??**.

The purpose of the `define_columns` function is to define `SQLAlchemy 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`. A portion of an example text file is shown in Figure **??**.

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 small set of example ACS FITS files to scrape their 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` module adds any new header keywords to the text files by comparing the header contents of a given FITS file to the existing column

```
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. 16: *The* `define_columns` *function, used to define columns used in the header tables.*

```
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. 17: *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.*

```
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. 18: *An example of how the* `orm_factory` *function is called to create class definitions for the header tables.*

definition text files[4].

With the implementation of the `orm_factory` and `define_columns` function, it is trivial to create class definitions for each of the 109 header tables. An example of this is shown in Figure **??**, where 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, creating the database tables on the `MySQL` server is accomplished by executing the `base.metadata.create_all()` method.

4. New header keywords are occasionally introduced to ACS data proceeding updates to its calibration software.

3.7 Data ingestion software: Algorithm

Another critical component of the `acsql` application is the modules that are used to ingest data into the `acsql` database and to create the 'Quicklook' JPEGs and thumbnails. The ingestion algorithm is encapsulated within a collection of modules in the `acsql` package, namely the `acsql.ingest.ingest` and `acsql.scripts.ingest_production.py` modules (further discussed in section **??**). Collectively, we refer to this collection of modules as the "ingestion software". We describe the overall algorithm of the ingestion software below.

1. ***Identify newly available public ACS data in the***

filesystem: This is accomplished by comparing the list of `rootnames` in the filesystem with the list of `rootnames` in the `master` table of the database. Any `rootnames` that exist in the filesystem but not in the database are considered new `rootnames` to be ingested.

2. ***Loop over each `rootname` (in a parallelized manner):*** The `ingest.py` module takes a single `rootname` as input, such that if there are multiple `rootnames` to be ingested, the calls to the `ingest.py` module can be parallelized over many CPUs. The ingestion of one `rootname` does not depend on the ingestion of another, and there is no importance to the order in which files are ingested. Note that steps 3 through 9 are written such that a single `rootname` is being ingested (i.e. inside of the loop).

3. ***Update the `master` table with information about the `rootname`:*** At this point, the `master` table can be updated with data pertaining to the `rootname`. A generic `insert_or_update` function was written (available in the `acsql.utils.utils` module) to determine if an entry should be inserted (in the case of first-time ingestion) or updated (in the case of re-ingestion). This function uses various `SQLAlchemy` methods and the class definitions described in section **??** to perform the `insert` or `update` operation. The `insert_or_update` function is shown in Figure **??**.

4. ***Loop over the available filetypes for the given `rootname`:*** The available filetypes are determined by traversing down a level in the tree structure of the filesystem and identifying which files are present. Once determined, the ingestion algorithm processes each `<rootname>_<filetype>.fits` file individually. Note that steps 5 through 9 are written such that a single filetype is being ingested (i.e. inside of the next nested loop).

5. ***Create a Python dictionary with metadata about the file*:** To reduce the amount of variables passed around to functions, a data container in the form of a Python dictionary data type is created to hold metadata needed by the remainder of the ingestion process. We refer to this data container as the `file_dict`. The `file_dict` contains metadata such as the absolute path of the file in the filesystem, the filetype, the available FITS file extensions of the file, and the absolute paths to where the Quicklook JPEGs and thumbnails will be written.

6. ***For each FITS file extension, extract the header information and update the appropriate header table in the `acsql` database:*** The header information is read into a Python dictionary via the `astropy.io.fits` module. Besides some minor fixes for a few corner cases (such as converting hyphens to underscores in header keys as to avoid Python errors), it is rather trivial to perform an `insert` or `update` operation via the `insert_or_update` function (see Figure **??**).

7. ***Update the `datasets` table for the given `filetype`:*** At this point, an entry in the `datasets` table is either inserted (if it is the first `filetype` for the `rootname` being ingested), or updated (if a `filetype` under the same `rootname` had already been ingested).

8. ***If the `filetype` is either `raw`, `flt`, or `flc`, then create a Quicklook JPEG image:*** JPEGs are produced only for `raw`, `flt`, and `flc` filetypes, since these are the filetypes that contain two-dimensional image data. The image data are read into multidimensional `numpy` array data types via the `astropy.io.fits` module. The data are then rescaled as to avoid an undesirable image stretch caused by extremely high or low-valued pixels, and saved to a `.jpg` format. The JPEGs are saved to the `jpegs/` portion of the `acsql` filesystem (described in section **??**).

9. ***If the `filetype` is `flt`, then create a Quicklook thumbnail image:*** Thumbnail images are only produced for `flt` filetypes since they are only meant to be viewed as a means to discover/identify the larger JPEG images via the `acsql` web application. Thumbnails are generated simply by reading in the corresponding `flt` JPEG and resizing it to 128x128 pixels. The thumbnails are saved to the `thumbnails/` portion of the `acsql` filesystem (described in section **??**).

```python
def insert_or_update(table, data_dict):
    """Insert or update a record in the given ``table`` with the data
    in the ``data_dict``.

    A record is inserted if the primary key of the record does not
    already exist in the ``table``.  A record is updated if it does
    already exist.

    Parameters
    ----------
    table : str
        The name of the table to insert/update into.
    data_dict : dict
        A dictionary containing the data to insert/update.
    """

    table_obj = getattr(acsql.database.database_interface, table)
    session, base, engine = acsql.database.database_interface.\
        load_connection(SETTINGS['connection_string'])

    # Check to see if a record exists for the rootname
    query = session.query(table_obj)\
        .filter(getattr(table_obj, 'rootname') == data_dict['rootname'])
    query_count = query.count()

    # If there are no results, then perform an insert
    if not query_count:
        tab = Table(table.lower(), base.metadata, autoload=True)
        insert_obj = tab.insert()
        try:
            insert_obj.execute(data_dict)
        except (DataError, IntegrityError, InternalError) as e:
            logging.warning('\tUnable to insert {} into {}: {}'.format(
                            data_dict['rootname'], table, e))

    else:
        query.update(data_dict)

    session.commit()
    session.close()
    engine.dispose()
```

Fig. 19: *The `insert_or_update` function from the `acsql.utils.utils` module, used at various times during the data ingestion process to determine if an entry should be inserted or updated in the `acsql` database.*

The modules of the ingestion software are intended to

```
08/15/2017 11:05:26 INFO: User: bourque
08/15/2017 11:05:26 INFO: System: plhstins1.stsci.edu
08/15/2017 11:05:26 INFO: Python Version: 3.5.2
08/15/2017 11:05:26 INFO: Python Path: /envs/anaconda3/envs/astroconda3/bin/python
08/15/2017 11:05:26 INFO: Numpy Version: 1.11.2
08/15/2017 11:05:26 INFO: Numpy Path: /site-packages/numpy
08/15/2017 11:05:26 INFO: Astropy Version: 1.2.1
08/15/2017 11:05:26 INFO: Astropy Path: /site-packages/astropy
08/15/2017 11:05:26 INFO: SQLAlchemy Version: 1.1.4
08/15/2017 11:05:26 INFO: SQLAlchemy Path: /site-packages/SQLAlchemy-1.1.4-py3.5-linux-x86_64.egg/sqlalchemy
08/15/2017 11:05:26 INFO: Gathering files to ingest
08/15/2017 11:06:00 INFO: j8zh21xv: Updated master table.
08/15/2017 11:06:01 INFO: j8zh21xv: Updated HRC_flt_0 table.
08/15/2017 11:06:01 INFO: j8zh21xv: Updated HRC_flt_1 table.
08/15/2017 11:06:01 INFO: j8zh21xv: Updated HRC_flt_2 table.
08/15/2017 11:06:01 INFO: j8zh21xv: Updated HRC_flt_3 table.
08/15/2017 11:06:01 INFO: j8zh21xv: Updated datasets table for flt.
08/15/2017 11:06:01 INFO: j8zh21xv: Creating JPEG
08/15/2017 11:06:02 INFO: j8zh21xv: Creating Thumbnail
08/15/2017 11:06:02 INFO: j8zh21xv: Updated HRC_raw_0 table.
08/15/2017 11:06:02 INFO: j8zh21xv: Updated HRC_raw_1 table.
08/15/2017 11:06:03 INFO: j8zh21xv: Updated HRC_raw_2 table.
08/15/2017 11:06:03 INFO: j8zh21xv: Updated HRC_raw_3 table.
08/15/2017 11:06:03 INFO: j8zh21xv: Updated datasets table for raw.
08/15/2017 11:06:03 INFO: j8zh21xv: Creating JPEG
08/15/2017 11:06:04 INFO: j8zh21xv: Updated HRC_spt_0 table.
08/15/2017 11:06:05 INFO: j8zh21xv: Updated HRC_spt_1 table.
08/15/2017 11:06:05 INFO: j8zh21xv: Updated datasets table for spt.
08/15/2017 11:06:05 INFO: j8zh21xv: End ingestion
```

Fig. 20: *An example log file for the ingestion of a single file (`j8zh21xv`).*

be executed daily (as an automatically-spawned process) to keep the `acsql` application up-to-date on any public data as it becomes available.

3.8 Data ingestion software: `logging`

Since the data ingestion software is intended to be executed by an automatic process and not by a human, we implemented a system to log the status of the ingestion to an output text file to be analyzed at a later time. Such log files can be used to assess if there were any issues with the ingestion process, such as if a new header keyword had appeared (requiring an update to the appropriate header table in the database). An example log file showing the ingestion of a single rootname (`j8zh21xv`) is provided in Figure **??**.

When the ingestion software executes, it creates an empty log file is created with the filename `<module_name>_<timestamp>.log`, where `<module_name>` is the name of the ingestion module (in production, this is `ingest_production.py`, as will be discussed in section **??**), and `<timestamp>` is the current time in the format `YYYY-MM-DD-HH-MM`. The naming convention of the log file allows system maintainers to determine which log file corresponds to a specific ingestion run.

Next, the Python `logging` module configures the format of the log statements by (1) setting the default logging `level` to `INFO` (meaning that, unless otherwise specified, each call to `logging` by the ingestion module will result in an `INFO` statement), (2) setting the timestamp format to `YYYY-MM-DD HH:MM:SS`, and (3) setting the logging message format to `<timestamp> <level>: <message>`.

With the logging settings configured, any call to the `logging` module within the ingestion software results in a log statement. For example, the command `logging.info('Gathering files to ingest')` results in a timestamped log message, e.g. `08/15/2017 11:05:26 INFO: Gathering files to ingest` (as shown in line 10 of Figure **??**.)

Calls to the `logging` module are strategically placed within the ingestion software to provide enough context of the status of the ingestion without cluttering the log file with too much detail. In most cases, logging statements only occur after a change of state to the filesystem or database (i.e. an updated database table, the creation of a JPEG or thumbnail, etc.).

3.9 Web Application

3.9.1 Overview

The front-end of the `acsql` application is the web application. The web application is built using Python and `Flask`, which is a Python based web framework[26]. Currently, the web application has two main features/modes of use: (1) viewing JPEGs and image metadata for any publicly-available ACS `raw`, `flt`, and `flc` image (when applicible), and (2) performing relational queries on the `acsql` database. To illustrate these features, we show examples of some of the different webpages that make up the web application in Figures **??** through **??**, and further describe each below.

Figure **??** shows the `acsql` homepage. The homepage, as well as all other webpages in the web application, contains a menu bar at the top containing four links: (1) to the database query page, (2) to the archive links page, (3) to the `acsql` code repository on GitHub, and (4) to the `readthedocs` documentation page. Clicking the 'ACS Quicklook' button in the menu bar allows the user to return to the homepage,

regardless of which webpage they are currently viewing. Additionally, the homepage also contains button-type links to the database query page and the archive links page.

Figure **??** shows an example of the database query page. This page allows users to fill out a form that in turn creates a query of the `acsql` database and executes on hitting the 'Submit' button at the bottom of the page. Note that the form contains only a subset of the many possible database parameters; we limited the database query page to only options that we deemed to be particularly useful by the user base. As such, we expect that this page be further expanded and/or modified in the future as more use cases become apparent.

In this example, we use the form to find all data taken with the `WFC` detector, are of Observation Type `IMAGING`, and were observed by the Principle Investigator Zolt Levay[5]. Note that there are three output options: (1) an `HTML table`, which returns a webpage showing the selected 'Output Columns' (in this case, the Rootname, PI last/first name, Proposal ID, Aperture, Date/Time of observation, and the Target Name; see Figure **??**), (2) a CSV file downloaded to the user's machine containing a comma-separated table of the selected output columns (see Figure **??**), or (3) a webpage showing the thumbnail images of all of the resulting data (see Figure **??**).

Figure **??** shows the archive links page. This page contains links to every proposal that contains publicly available ACS data via the 4-5 digit `PROPOSID`. Clicking one of the links opens a new window to a 'view proposal' webpage. Similar to the database query results shown in Figure **??**, the 'view proposal' webpage shows thumbnail images of each ACS dataset available in the given proposal. The 'view proposal' pages contain several buttons and filters near the top of the page; using these will sort and/or filter out the thumbnail images based on the chosen parameters. Figure **??** shows an example of this webpage, using proposal 14039.

In the 'view proposal' webpages, users may click on any one of the thumbnail images to bring up the 'view image' webpage for that image. An example of this webpage is shown in Figure **??**, using the dataset `jcs718kmq`. This webpage contains the Quicklook JPEG of the image along with useful metadata of the dataset near the top of the page. It also contains an additional link (`proposal`) to view the other images within the particular proposal via the appropriate 'view proposal' webpage, as well as links to view the corresponding `RAW` or `FLC` JPEG image (when applicable).

3.9.2 Implementation

The web application is implemented by several Python modules, namely:

- `acsql_webapp.py`
- `data_containers.py`
- `form_options.py`
- `query_form.py`
- `query_lib.py`

as well as several subdirectories containing useful static files, style sheets, and HTML templates, namely:

- `static/css/*.css`
- `static/js/*.js`
- `static/img/*`
- `templates/*.html`

Below we further describe these modules, directories, and files, and how they relate to each other.

The `acsql_webapp` module is the main module for running the `acsql` web application; this module must be executed in order to serve the web application (either on a dedicated web server, or locally on a user's machine and accessed through the user's `localhost`). This module uses the `Flask` web framework to receive incoming `HTML` requests. It is structured such that each `acsql` webpage relates to an individual function within `acsql_webapp` (for example, accessing the `/archive/` webpage results in the execution of the `archive` function). Each of these functions contain the appropriate Python logic and rendering of `HTML` templates to return the desired webpage.

Much of the functionality required to gather data needed for the various `acsql` webpages (e.g. lists of images, image metadata, paths to JPEG images on disk, etc.) is imported into `acsql_webapp` from the `data_containers.py` module. That is to say, many of the functions within the `acsql_webapp` module call functions from the `data_containers` module to gather the data that is eventually passed on to the `HTML` template. This design choice was made in order to separate the data from the functionality, often regarded as a best practice in software engineering.

Similarly, the `form_options.py` module is a container module for storing the possible form options (e.g. apertures, filters, output column, proposal types, etc.) for the database query form page. These data could have been defined within the `acsql_webapp` module, but we chose to separate out these data.

The `query_form.py` and `query_lib.py` modules are used extensively to render the database query webpage and convert a completed form into an executable query on the `acsql` database, respectively. The `query_form` module contains several class objects for supplying specific types of form fields (e.g. check box fields, text fields, multiple select fields) as well as functions to validate form entries. Several of the form objects are extensions of components (via subclassing) provided by the `wtforms` library, which is a form validation and rendering library for Python web development (wtforms)[27]. The query_lib module contains functions that parse, build, validate, and return `SQLAlchemy query` objects from the results of a submitted database query form. These queries are then executed on the `acsql` database and results are used to render the database query result HTML, CSV, and/or thumbnail webpages.

Like many web applications, we also store static files such as `HTML` templates, `CSS` templates, and static images in separate `static/` and `templates/` directories. The `static/css/` directory contains the `CSS` templates used in the `HTML` templates. Similarly, the `static/js/` directory contains several JavaScript libraries, also used in the `HTML` templates. Lastly, the `templates/` directory contains the

5. Zolt Levay is a Science Visuals Developer at STScI, and has been the PI of many programs that have acquired images for HST public releases.

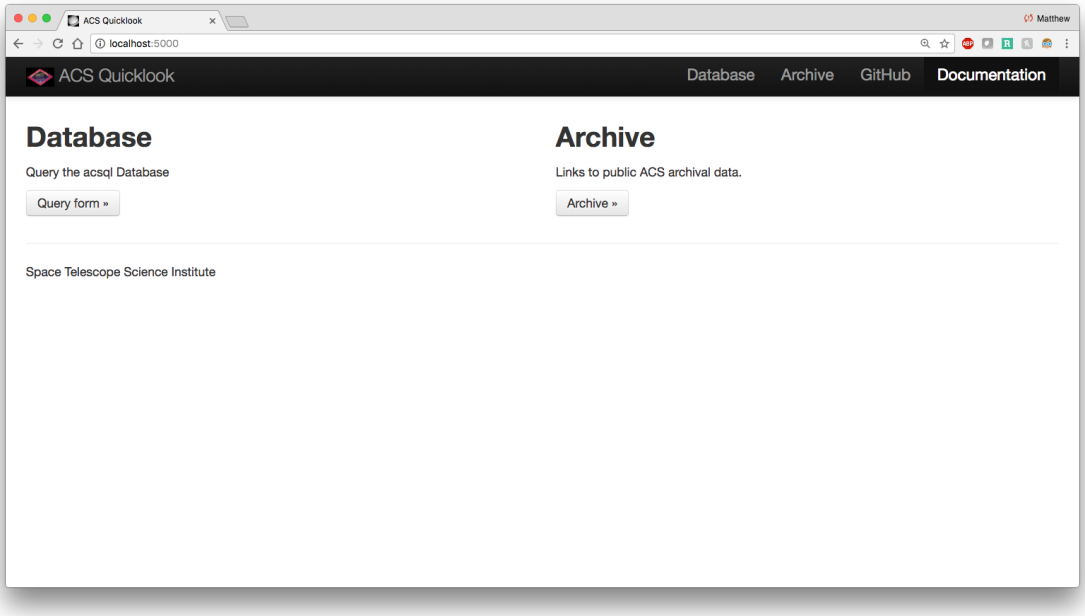


Fig. 21: *The homepage of the* `acsql` *web application.*

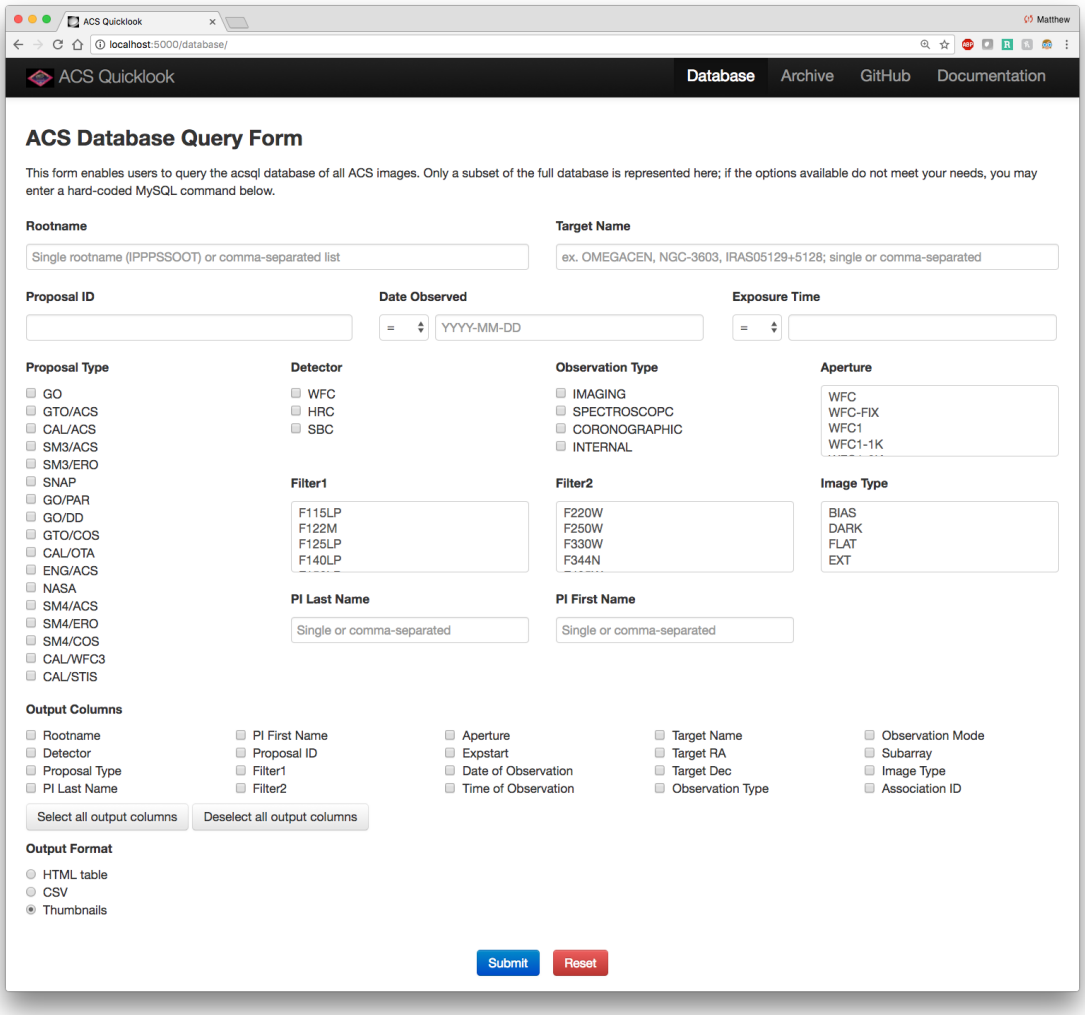
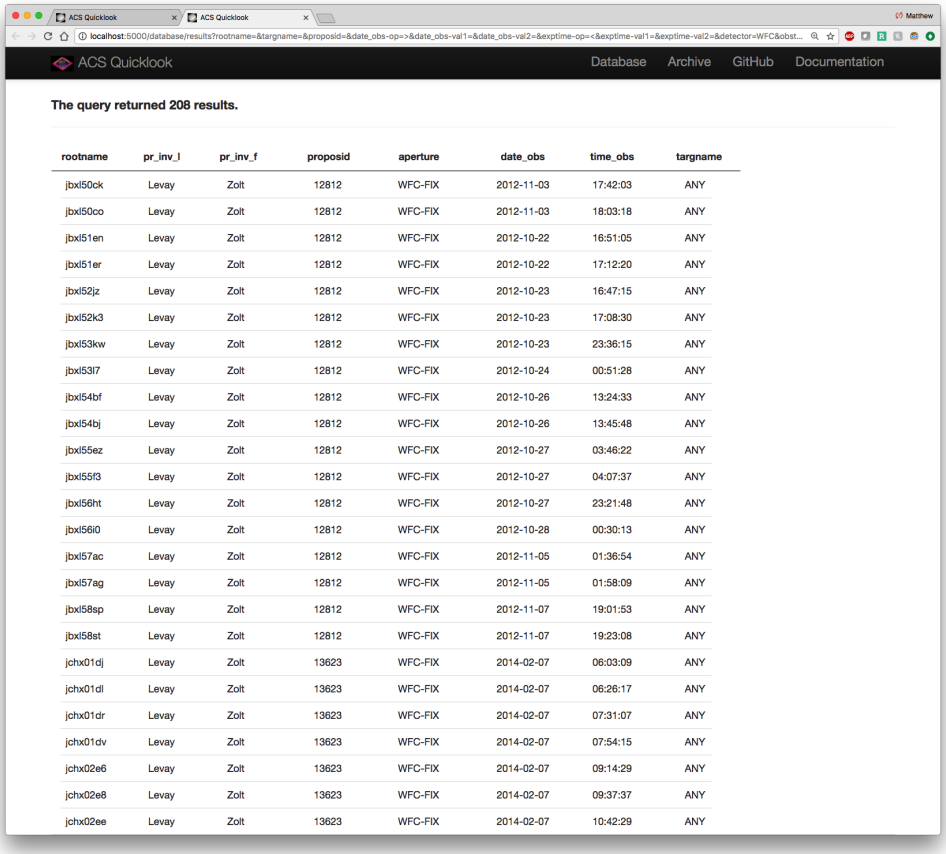


Fig. 22: *A portion of the database query page of the* `acsql` *web application.*



rootname	pr_inv_l	pr_inv_f	proposid	aperture	date_obs	time_obs	targname
jbxl50ck	Levay	Zolt	12812	WFC-FIX	2012-11-03	17:42:03	ANY
jbxl50co	Levay	Zolt	12812	WFC-FIX	2012-11-03	18:03:18	ANY
jbxl51en	Levay	Zolt	12812	WFC-FIX	2012-10-22	16:51:05	ANY
jbxl51er	Levay	Zolt	12812	WFC-FIX	2012-10-22	17:12:20	ANY
jbxl52jz	Levay	Zolt	12812	WFC-FIX	2012-10-23	16:47:15	ANY
jbxl52k3	Levay	Zolt	12812	WFC-FIX	2012-10-23	17:08:30	ANY
jbxl53kw	Levay	Zolt	12812	WFC-FIX	2012-10-23	23:36:15	ANY
jbxl53l7	Levay	Zolt	12812	WFC-FIX	2012-10-24	00:51:28	ANY
jbxl54bf	Levay	Zolt	12812	WFC-FIX	2012-10-26	13:24:33	ANY
jbxl54bj	Levay	Zolt	12812	WFC-FIX	2012-10-26	13:45:48	ANY
jbxl55ez	Levay	Zolt	12812	WFC-FIX	2012-10-27	03:46:22	ANY
jbxl55f3	Levay	Zolt	12812	WFC-FIX	2012-10-27	04:07:37	ANY
jbxl56ht	Levay	Zolt	12812	WFC-FIX	2012-10-27	23:21:48	ANY
jbxl56i0	Levay	Zolt	12812	WFC-FIX	2012-10-28	00:30:13	ANY
jbxl57ac	Levay	Zolt	12812	WFC-FIX	2012-11-05	01:36:54	ANY
jbxl57ag	Levay	Zolt	12812	WFC-FIX	2012-11-05	01:58:09	ANY
jbxl58sp	Levay	Zolt	12812	WFC-FIX	2012-11-07	19:01:53	ANY
jbxl58st	Levay	Zolt	12812	WFC-FIX	2012-11-07	19:23:08	ANY
jchx01dj	Levay	Zolt	13623	WFC-FIX	2014-02-07	06:03:09	ANY
jchx01dl	Levay	Zolt	13623	WFC-FIX	2014-02-07	06:26:17	ANY
jchx01dr	Levay	Zolt	13623	WFC-FIX	2014-02-07	07:31:07	ANY
jchx01dv	Levay	Zolt	13623	WFC-FIX	2014-02-07	07:54:15	ANY
jchx02e6	Levay	Zolt	13623	WFC-FIX	2014-02-07	09:14:29	ANY
jchx02e8	Levay	Zolt	13623	WFC-FIX	2014-02-07	09:37:37	ANY
jchx02ee	Levay	Zolt	13623	WFC-FIX	2014-02-07	10:42:29	ANY

Fig. 23: *The results of the database query shown in Figure* **??** *in the form of an HTML table.*

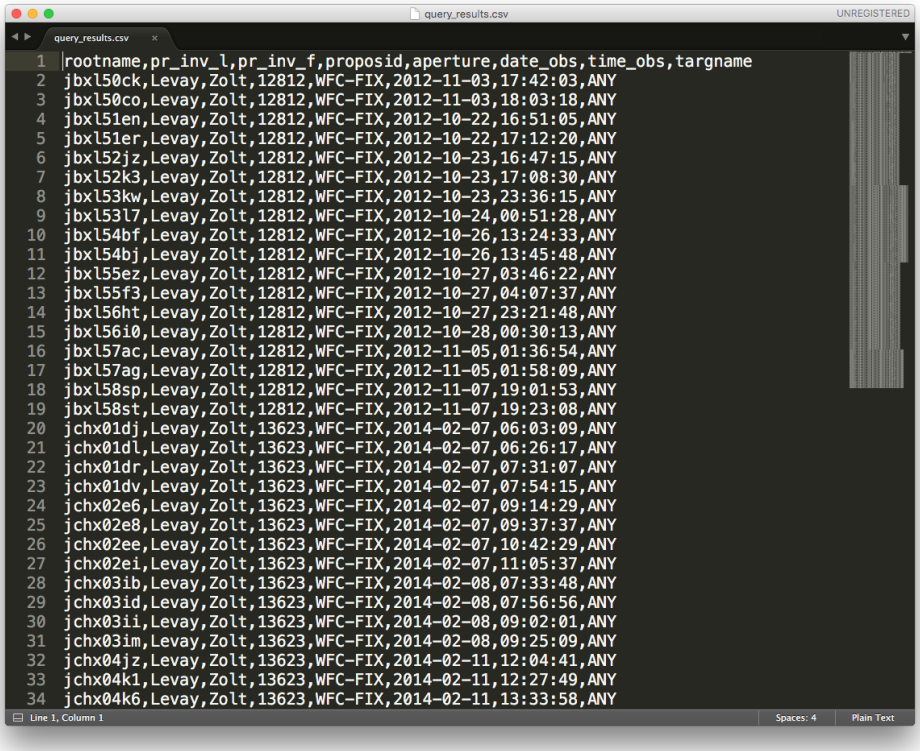


Fig. 24: *The results of the database query shown in Figure* **??** *in the form of an CSV file.*

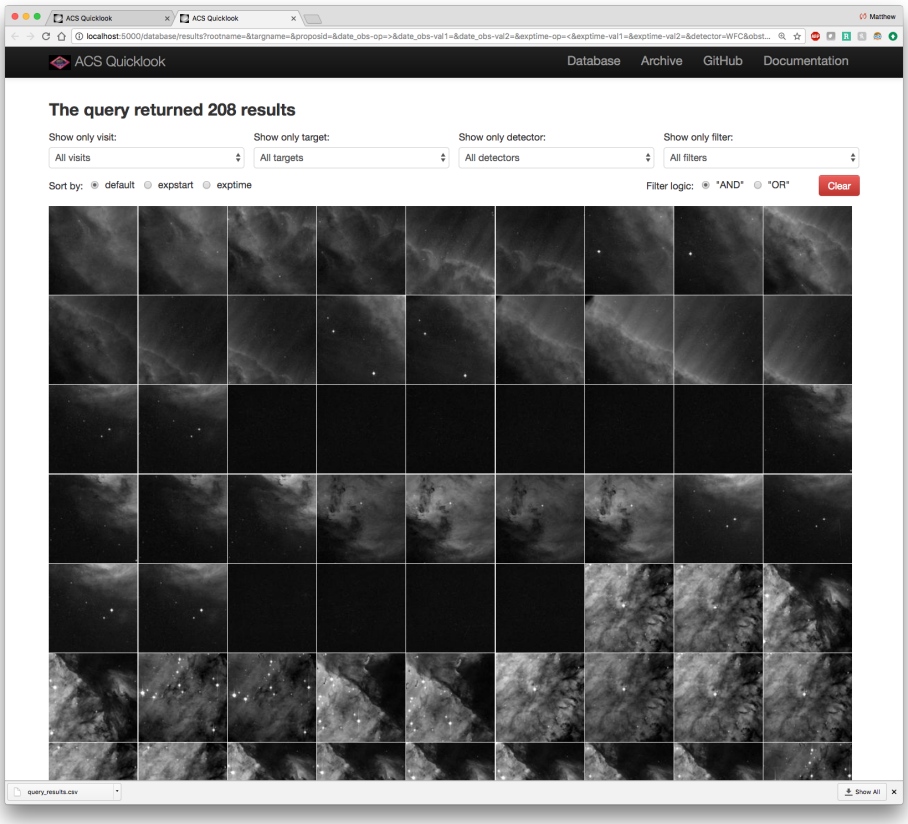


Fig. 25: *The results of the database query shown in Figure* **??** *in the form of thumbnail images.*

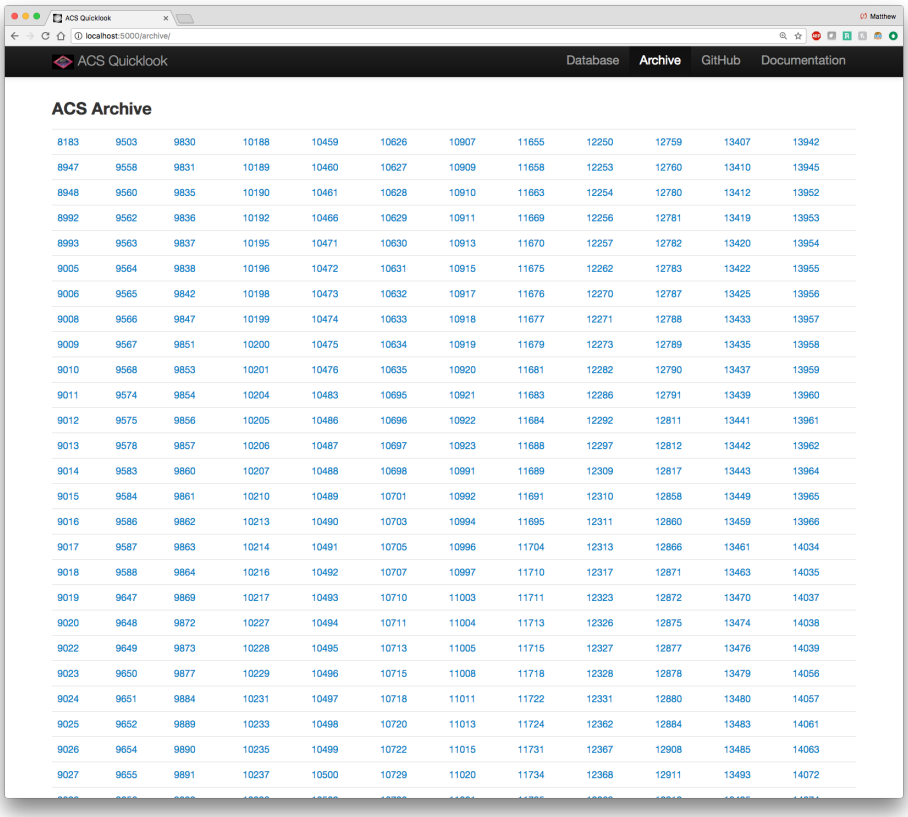


Fig. 26: *The archive links webpage of the* `acsql` *web application.*

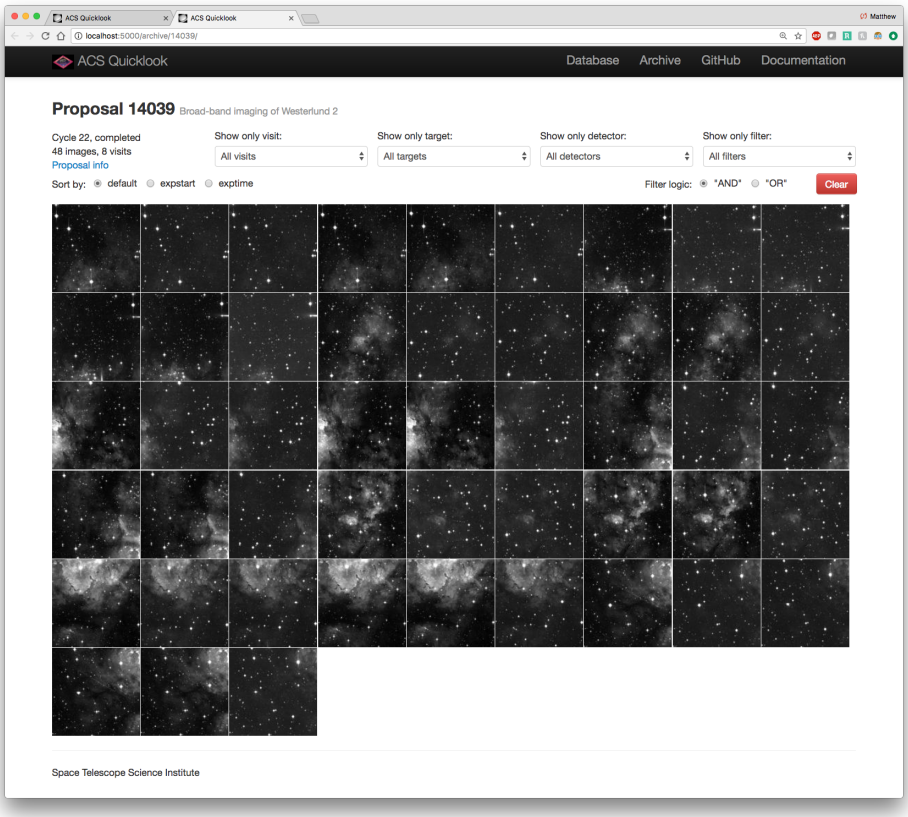


Fig. 27: *An example 'view proposal' webpage, using proposal 14039.*

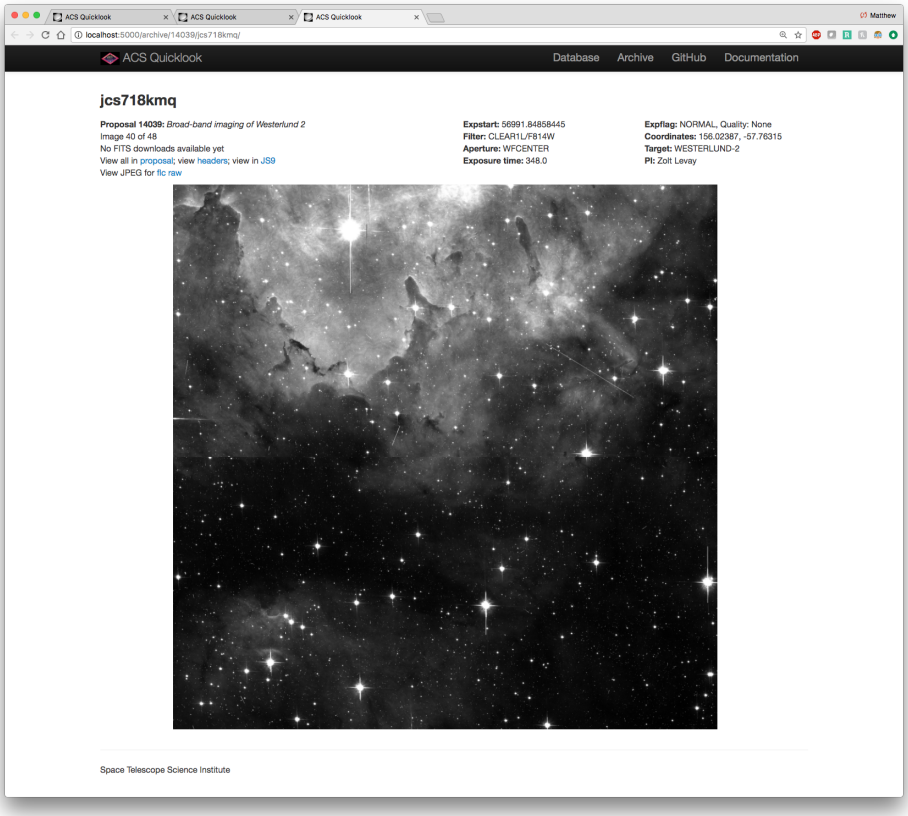


Fig. 28: *An example 'view image' webpage, using dataset* `jcs718kmq`.

HTML templates used to render the various webpages of the web application. There is one template for each type of webpage, as well as a template that houses HTML that is common to each webpage (e.g. the menu bar), named base.html. Aside from some minor tweaks, we use templates from Bootstrap, which is a front-end component library and open source toolkit for developing HTML, CSS, and JS[28].

3.10 acsql Package

All software associated with the acsql project is contained within a single git repository (also named "acsql"), which we refer to as the "acsql library", or "acsql package". The package layout is shown below:

```
acsql/
    LICENSE
    README.md
    MANIFEST.in
    setup.py
    paper/
        ...
    presentation/
        ...
    docs/
        Makefile
        requirements.rst
        source/
          conf.py
          database.rst
          index.rst
          ingest.rst
          scripts.rst
          utils.rst
          website.rst
    acsql/
        __init__.py
        database/
            __init__.py
            database_interface.py
            make_tabledefs.py
            queries.py
            reset_database.py
            table_definitions/
                *.txt
            update_tabledefs.py
        ingest/
            __init__.py
            ingest.py
            make_file_dict.py
            make_jpeg.py
            make_thumbnail.py
        scripts/
            __init__.py
            ingest_production.py
        utils/
            __init__.py
            config.yaml
            utils.py
        website/
            __init__.py
            acsql_webapp.py
            data_containers.py
            form_options.py
            query_form.py
            query_lib.py
            static/
                css/
                    *.css
                img/
                    jpegs
                    thumbnails
                js/
                    *.js
            templates/
                *.html
```

We now provide a brief description of each package component. For further details on each Python module within the acsql package, readers are encouraged to view the official API documentation hosted at http://acsql.readthedocs.io/.

LICENSE: A BSD 3-Clause license, which states that the acsql package is an open source software package and may be used and redistributed.

README.md: A README file that describes how to install and use the acsql package.

MANIFEST.in: A list of static files to be included in the tarball file when the user installs the acsql package.

setup.py: The acsql package installation script. Executing this script with python setup.py install installs the package into the software environment.

paper/: A subdirectory which contains all materials used for the creation of this document.

presentation/: A subdirectory which contains all materials used for the creation of the Towson University COSC 880 presentation.

docs/: A subdirectory which contains all materials used for the creation of the sphinx API documentation hosted on readthedocs (see section **??**).

Makefile: A make script that is used to build the sphinx API documentation from the source reStructured Text files (mentioned below) (see section **??**).

requirements.rst: A list of acsql package dependencies, used by readthedocs to build a virtual machine that constructs the resulting HTML doc pages (see section **??**).

source/: A subdirectory containing all of the reStructured Text files used for building the sphinx API documentation, one .rst file per subpackage, including a master index.rst file (see section **??**).

acsql/: A subdirectory containing all Python code that is part of the official acsql library. This is the top level of the importable acsql package.

__init__.py: A Python file indicating that the subdirectory is part of the overall acsql package.

database/: The database subpackage, containing Python modules that pertain to the acsql database (see sections **??** and **??**).

database_interface.py: The Python module for constructing and connecting to the acsql database (see

section **??**).

`make_tabledefs.py`: The Python module for creating the table definition text files (see section **??**).

`queries.py`: A Python module containing several examples of queries that can be used with the `acsql` database.

`reset_database.py`: A Python module that allows the user to 'reset' the `acsql` database (i.e. delete all tables, then create all tables).

`table_definitions/`: A subdirectory containing all of the `<detector>_<filetype>_<extension>.txt` text files, each of which contain a list of header keys along with their datatypes (see section **??**).

`update_tabledefs.py`: A Python module that allows the user to update the `table_definitions/` text files with new header keywords (see section **??**).

`ingest/`: The `ingest` subpackage, containing Python modules for ingesting new data into the `acsql` application, including database updates and the creation of JPEGs/thumbnails (see section **??**).

`ingest.py`: A Python module for performing the ingestion of a single file (see section **??**).

`make_file_dict.py`: A Python module for creating a `file_dict` for an individual file (see section **??**).

`make_jpeg.py`: A Python module for creating a JPEG image from an individual file (see section **??**).

`make_thumbnail.py`: A Python module for creating a thumbnail image from an individual JPEG (see section **??**).

`scripts/`: The `scripts` subpackage, containing Python modules for ingesting multiple files from the `acsql` filesystem. This also serves as storage place for possible future instrument calibration and monitoring routines (see section **??**).

`ingest_production`: The Python module for ingesting new ACS data as it becomes publicly available, intended to be executed regularly (see section **??**).

`utils/`: The `utils` subpackage, containing Python modules that are useful for general `acsql` operations (e.g. configuring `logging`, supplying hard-coded instrument configurations, etc.) as well as a configuration file for storing sensitive credentials and directory locations.

`config.yaml`: A text file containing hard-coded user-specific directory locations and `acsql` database credentials. Specifically, it contains values for the `acsql` database `connection_string`, as well as locations for the `filesystem`, `log_dir`, `jpeg_dir`, and `thumbnail_dir`. The contents of the `config.yaml` file can be imported via the `utils.utils.SETTINGS` dictionary.

`utils.py`: A Python module containing various functions that are generally useful for `acsql` operations, such as configuring `logging`, determining if a database entry requires an `insert` or an `update`, and hard-coded Python variables that reflect instrument/system configurations.

`website/`: The `website` subpackage, containing Python modules that are used in the construction and operations of the `acsql` web application (see section **??**).

`acsql_webapp.py`: The main Python module for running the `acsql` web application, using the Python `Flask` web framework (see section **??**).

`data_containers.py`: The Python module that contains various functions for returning data to be used by the `acsql` web application (see section **??**).

`form_options.py`: A Python module that stores form data for the database query portion of the `acsql` web application (see section **??**).

`query_form.py`: A Python module that contains class objects for building the query form for the database query portion of the `acsql` web application (see section **??**).

`query_lib.py`: A Python module that contains various functions to support the querying of the `acsql` database through the `acsql` web application (see section **??**).

`static/`: A subdirectory containing static materials used by the `acsql` web application, such as `CSS` templates (i.e. `css/`), JavaScript functions (i.e. `js/`), and symbolic links to the JPEGs and thumbnails hosted on the web application (i.e. `img/`) (see section **??**).

`templates/`: A subdirectory containing `HTML` templates used to render the various webpages of the `acsql` web application, one for each page (see section **??**).

4 RESULTS

The results of the system implementation (described in Chapter **??**) were several project deliverables:

1) Filesystem
2) Database
3) Web application
4) Software package
5) Software documentation

As mentioned in section **??**, the project deliverables will primarily be used by members of the ACS instrument team at STScI, but may also be used by users external to STScI. In the following subsections, we further describe each on of these deliverables.

4.1 Filesystem Deliverable

As described in sections **??** and **??**, the `acsql` filesystem is comprised of two parts: (1) A filesystem that stores the archive of publicly-available ACS data (i.e. FITS files), and (2) a filesystem of Quicklook JPEGs and thumbnails.

For the former, we utilized an already-existing filesystem of ACS data internal to STScI known as the "MAST public cache"; this filesystem is organized in a similar way to that shown in Figure **??**[6]. Though this service is internal to STScI, it is possible for an external user to reconstruct the filesystem, as all data within the MAST public cache is publically available to download via the MAST archive (i.e. https://archive.stsci.edu/). The location of the filesystem is determined by the `filesystem` parameter in the user-supplied `config.yaml` file (see section **??**).

Currently, the filesystem consists of ~1,030,000 total files. Figure **??** shows how this breaks down by individual filetype. We see that `spt`, `raw`, and `flt` make up the majority of the files, which is expected considering that these filetypes exist for nearly every ACS observing mode. On the other hand, we see a small amount of `crj` and `crc` files, which is also expected considering that these filetypes are only triggered for specific observing modes.

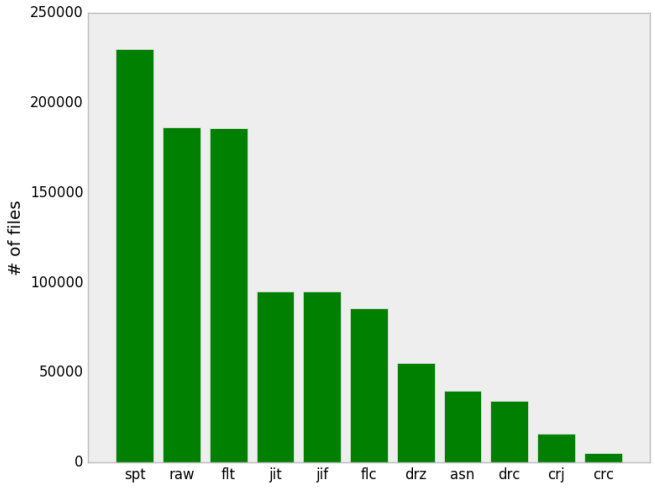


Fig. 29: *The number of files in the `acsql` filesystem by filetype.*

As for item (2), the filesystem of JPEGs and thumbnails was constructed during the ingestion of all publicly-available ACS data (via `ingest_production.py`, see section **??**). The location of the JPEG/thumbnail filesystem is determined by the `jpeg_dir` and `thumbnail_dir` parameters in the `config.yaml` file, respectively. Currently, there are ~457,000 JPEGS and ~185,000 corresponding thumbnail images in the filesystem[7].

4.2 Database Deliverable

As described in sections **??** and **??**, the `acsql` database is a `MySQL` database that stores the header information of every public ACS dataset. Currently, this database is hosted on a server that is internal to STScI. However, external users may build their own copy of the database via the `ingest_production.py` and `database_interface` modules. All 111 tables of `acsql` database (`master`, `datasets`, and 109 header tables) are up to date as of the time of this writing (September 2017). Ideally, the database will be updated via regular executions of `ingest_production.py`.

Figure **??** shows the number of records in each table of the database. Currently, there are ~4,300,000 records in total amongst the 111 tables.

4.3 Web Application Deliverable

As described in section **??**, the `acsql` web application consists of a series of Python modules and static files. Currently, the web application is not hosted on a dedicated web server (neither externally nor internally to STScI[8]), however, users can operate the application through the user's `localhost`. Detailed installation and operation instructions for the web application are provided in the official documentation (see section **??**). In short, users must (1) ensure that they have built (or have access to) the `acsql` filesystem and database, (2) have installed the Python package dependencies, and (3) have filled out the `config.yaml` file (see section **??**).

4.4 Software Package Deliverable

As described in section **??** and **??**, the `acsql` git repository serves as the software package that contains all of the software and supporting materials needed to operate the `acsql` application. The software package is open-source, and is available to download directly from the GitHub repository (http://github.com/acsql/). All future enhancements and bug fixes to the `acsql` application will occur through this version-controlled repository.

4.5 Software Documentation Deliverable

As described in section **??**, another deliverable for the project is the API documentation that is automatically generated via `sphinx`. The resulting `HTML` pages are available through the `acsql` package, but are also hosted on `readthedocs`, available at http://acsql.readthedocs.io/. This is considered the official software documentation.

5 Discussion

We consider the collection of the features implemented as described in this document to be the "official release" or "1.0 release" of the `acsql` application. This release is tagged as version `v1.0` in the `acsql` GitHub repository. It is with this release that we expect the user base to begin using the application.

Though this is the official release of the application, we expect future development by the author and/or members of the ACS instrument team at STScI. There are several

6. We omit the mention of the parent directory structure and server names in this document as well as in the `acsql` code repository as to avoid exposing possible sensitive information.

7. recall that thumbnail images are only generated for `flt` filetypes, hence the discrepency in number

8. At the time of this writing, work is being done to build a web server that is internal to STScI to allow ACS instrument team members to easily access the `acsql` web application. We expect this web server to be operational within the next few months.

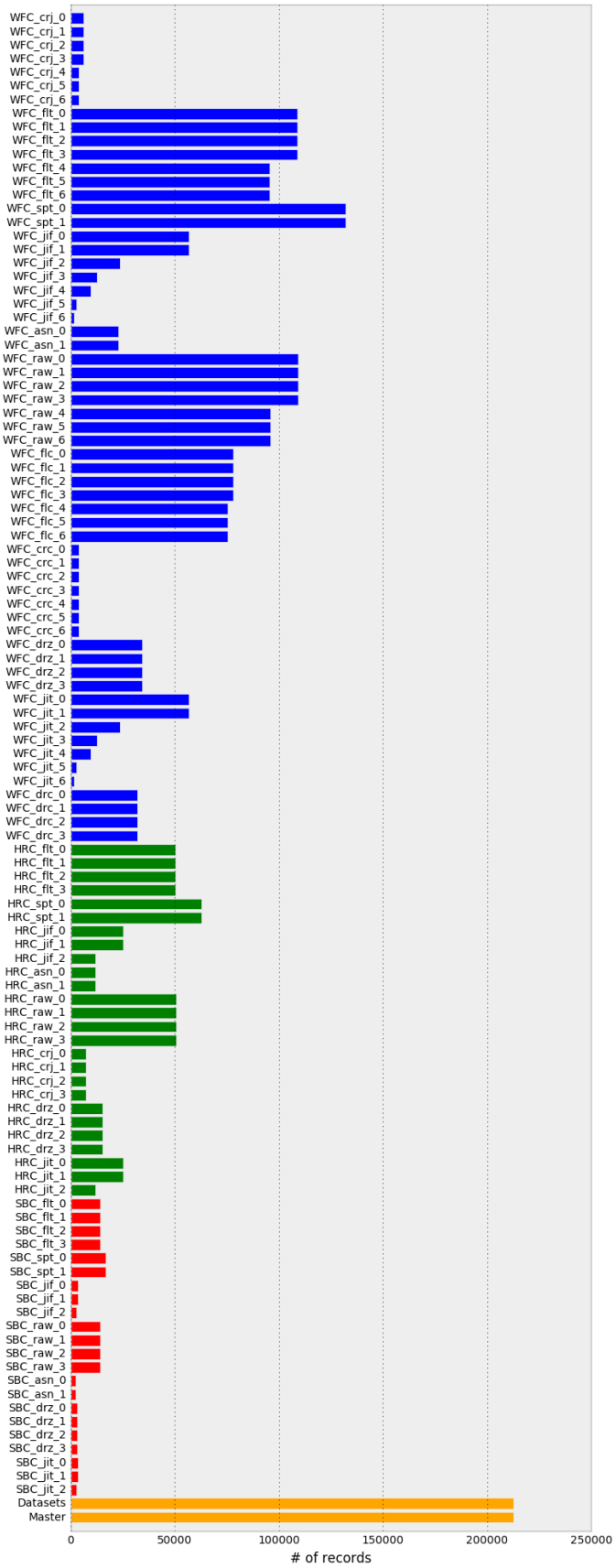


Fig. 30: *The number of records in each of the* `acsql` *database tables.*

possible features to be added to the application, namely those present in the `wfc3ql` application, as discussed in chapter **??**. Furthermore, some or all of the `acsql` application components could be extended to support other instruments and missions. Some of these possible enhancements are discussed in the proceeding sections.

5.1 Possible enhancements to the `acsql` application

As mentioned in chapter **??**, there exist several features in the `wfc3ql` application that could serve as possible enhancements to the `acsql` application. We elaborate on five of these features below:

1. ***Including proprietary data in the `acsql` filesystem:*** One of the main features of the `wfc3ql` application is that it allows users to view and diagnose new data just hours after it gets downlinked from HST. Since most HST data is proprietary up to one year after observation, this introduces a considerable security risk, and thus this feature is only made possible by taking special care of file permissions within the infrastructure of the `wfc3ql` filesystem and database. Implementing this feature in the `acsql` application would be advantageous from the perspective of an ACS instrument analyst who wishes to perform analysis on new data. However, allowing for the storage and handling of proprietary data would prohibit the application from being open-source and publicly-available to external users; the application would have to be restricted to ACS instrument team members at STScI.

2. ***Building automated instrument calibration and monitoring routines:*** The `wfc3ql` application contains several WFC3 instrument calibration and monitoring routines. These routines are in the form of individual Python modules that are executed automatically by `cron` jobs on a daily, weekly, or monthly basis. Each one of these monitors serves a specific instrument-related purpose, such as monitoring detector hysteresis effects[29], monitoring the dark current of the detector over time[30], and characterizing the behaviour of the instrument Channel Select Mechanism (CSM)[31]. Like WFC3, the ACS instrument also has various detector behaviors and characteristics that could benefit from being automatically monitored or characterized over time. Currently, such efforts are made by individual ACS instrument team members through manual execution of software. However, by using the `acsql` filesystem, database, and code library, it is possible to automate these tasks, saving employee time resources.

3. ***Adding a script execution status dashboard:*** If there were automated instrument calibration and monitoring routines implemented into the application, it would be useful to be able to visualize if the exectuions of these scripts were successful or not. The `wfc3ql` application accomplishes this with a script execution dashboard on the `wfc3ql` web application homepage. Each automated routine is labeled in the dashboard along with the date/time of last execution and a status column that indicates if the execution completed successfully or not. This helps to easily identify any disruptions in the routine `cron` jobs.

4. ***Expanding the database query form:*** The database query form webpage on the `acsql` web application allows user to construct and execute a query to the `acsql` database

using some of the most commonly used observational parameters (e.g. target name, filter, principle investigator name, etc.). However, these parameters are just a small subset of the total collection of parameters available in the `acsql` database. With the database containing all header keywords, it is possible to extend this query form use some more obscure (but useful) observational parameters. Examples of these may include telescope telemetry parameters (e.g. instrument temperature, telescope pointing, telescope latitude/longitude, etc.), engineering parameters (e.g. detector gain, detector readnoise, etc.), and photometric parameters (e.g. detector sensitivity, photometric zeropoints, etc.). We expect that the database query webpage in particular will endure the most feature development as the usefulness of certain parameters become evident from the user base.

5. ***Tracking ACS image anomalies:*** Another feature of the `wfc3ql` application is the capability to record the occurrances of image anomalies (i.e. features in anomalies that are unexpected or indicate a peculiar behavior of the instrument). The user accomplishes this in the 'view image' webpage by selecting a checkbox of the anomaly and hitting a 'submit' button. Then, a row is inserted into a separate 'anomalies' table of the `wfc3ql` database that indicates that the particular observation contains the particular anomaly. Such a feature could also be implemented in `acsql` by adding an `anomalies` table to the `acsql` database schema as well as the appropriate mechanisms to allow this functionality through the `acsql` web application. Tracking such anomalies would allows ACS instrument analysts to identify images affected by certain anomalies, as well as aid the characterization of instrument anomalies over time.

5.2 Possible extensions to other instruments or missions

This type of database-driven, web-based application for interacting with instrument data is currently only implemented for the WFC3 (`wfc3ql`) and ACS (`acsql` instruments of HST. However, the continued success of `wfc3ql` and potential success of `acsql` creates a strong case for implementing such a system for other instruments and missions. As this project has been fully developed at STScI, which is home of the operations for the Hubble Space Telescope and the forthcoming James Webb Space Telescope (JWST), it is reasonable to consider extending this application to other imaging instruments on HST (e.g. The Space Telescope Imaging Spectrography; STIS) and the future imaging instruments on JWST.

6 Conclusion

The `acsql` application, a web application built on top of a filesystem and database of ACS instrument data, has been released and is publicly available for use by ACS users. The application provides mechanisms for users to view images and their metadata for every publicly-available ACS dataset dating back to the installation of the instrument in 2002. While this document marks the version 1.0 release of this open-source application, we expect ongoing development to expand the capabilities of the `acsql` web application. With the modularization and coding standards found in the `acsql` software, we hope this application can be extended and/or used in reference to support a similar application for the forthcoming James Webb Space Telescope mission.

Acknowledgments

The authors would like to thank various members of the STScI staff that were extremely valuable for helping make this project come together: members of the WFC3/Quicklook team, for their years of ongoing development and support of the WFC3/Quicklook application, without which there would be no inspiration to create this similar application (Varun Bajaj, Ariel Bowers, Michael Dulude, Meredith Durbin, Jules Fowler, Catherine Gosmeyer, Heather Gunning, Harish Khandrika, Catherine Martlin, Abhi Rajan, Clare Shanahan, Ben Sunnquist, Alex Viana); members of the Instruments Division, who supported the creation of this project (Francesca Boffi, Norman Grogin, Elena Sabbi); members of the Operations and Engineering Division as well as the Information Technology and Services Division who provided valuable resources such as the database server, GitHub repository, web server, and disk space used to store project materials (Alex Yermolaev, Fausat Ogunsanya, Vera Gibbs); Pey-Lian Lim and Roberto Avila, who provided valuable feedback upon beta testing the web application; Sara Ogaz for assistance in developing some components of the `acsql` database as part of the Towson University COSC 657 Advance Database Management project; and finally, Jules Fowler and Sara Ogaz, who provided insightful comments and suggestions on this document.

References

[1] *SM3B*, National Aeronautics and Space Administration, Goddard Space Flight Center, [Online; accessed 2017-09-16], available at https://asd.gsfc.nasa.gov/archive/hubble/missions/sm3b.html.

[2] *SM4*, National Aeronautics and Space Administration, Goddard Space Flight Center, [Online; accessed 2017-09-16], available at https://asd.gsfc.nasa.gov/archive/hubble/missions/sm4.html.

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

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

[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] *STScI: Space Telescope Science Institute*, [Online; accessed 2017-09-16], available at www.stsci.edu.

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

[8] Lucas, R. A., et al., 2016, *ACS Data Handbook*, Version 8.0 (Baltimore: STScI)

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

[10] *The MAST Portal*, [Online; accessed 2017-09-15], available at https://mast.stsci.edu/.

[11] Bourque, M., et al., 2016, *The Hubble Space Telescope Wide Field Camera 3 Quicklook Project*, Astronomical Data Analysis Software & Systems Conference Proceedings, ADASS XXVI, ASP-CS, submitted

[12] Bourque, M., et al., 2016, *The HST/WFC3 Quicklook Project: A User Interface to Hubble Space Telescope Wide Field Camera 3 Data*, Proceedings IAU Symposium No. 325, 2016, Astroinformatics.

[13] Gosmeyer, C.M., and The Quicklook Team, 2017, *WFC3 Anomalies Flagged by the Quicklook Team*, WFC3 Instrument Science Report, 2017-22, available at http://www.stsci.edu/hst/wfc3/documents/ISRs/WFC3-2017-22.pdf

[14] *git*, [Online; accessed 2017-08-05], available at https://git-scm.com.
[15] *GitHub*, [Online; accessed 2017-08-05], available at https:/github.com.
[16] 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/.
[17] Goodger, D., 2001, *PEP 257 – Docstring Conventions*, Python Developer's Guide, available at https://www.python.org/dev/peps/pep-0257/.
[18] *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/
[19] 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.
[20] Brandi, G., et al., 2007, *Sphinx: Python Documentation Generator*, available at http://www.sphinx-doc.org/en/stable/.
[21] Loper, E., 2004, *Epydoc: Generating API Documentation in Python*, Proceedings of the Second Annual Python Conference, available at http://epydoc.sourceforge.net/.
[22] *Read the Docs*, [Online; accessed 2017-08-05], available at https://readthedocs.org.
[23] *MySQL 5.6 Reference Manual*, Oracle, [Online; accessed 2017-08-13], available at https://dev.mysql.com/doc/refman/5.6/en/.
[24] Bayer, M., 2006, *SQLAlchemy: The database toolkit for Python*, [Online; accessed 2017-02-21], available at http://www.sqlalchemy.org/.
[25] 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/.
[26] Ronacher, A., et al., 2010, [Online; accessed 2017-09-04], available at http://flask.pocoo.org/.
[27] *wtforms*, [Online; accessed 2017-09-04], available at https://github.com/wtforms/wtforms/.
[28] *Bootstrap*, [Online; accessed 2017-09-04], available at http://getbootstrap.com/.
[29] Bourque, M. and Baggett, S., 2013, *WFC3/UVIS Bowtie Monitor*, WFC3 Instrument Science Report, 2013-09, available at http://www.stsci.edu/hst/wfc3/documents/ISRs/WFC3-2013-09.pdf
[30] Bourque, M. and Baggett, S., 2016, *WFC3/UVIS Dark Calibration: Monitoring Results and Improvements to Dark Reference FIles*, WFC3 Instrument Science Report, 2016-08, available at http://www.stsci.edu/hst/wfc3/documents/ISRs/WFC3-2016-08.pdf
[31] Bushouse, H., 2005, *CSM Particulate Investigation*, WFC3 Instrument Science Report, 2005-26, available at http://www.stsci.edu/hst/wfc3/documents/ISRs/WFC3-2005-26.pdf