

VY Canis Majoris – load data - High level design

High Level Design Document to extract, clean and load Magnitude Data



Abhijit Majumdar

Sr IT Analyst

**VY Canis Majoris – Load Data - High Level Design Document**

**Objective of the Project:**

The objective of this project is to fetch and store the Magnitude data for VY Canis Majoris from the Association of Variable Star Observers (AAVSO) website.

The project defines an Extract Transform Load (ETL) pipeline that cleans, transforms and then loads the fetched data into a SQLite database.

The database is updated on a weekly basis (every Sunday) by a cron job that leverages GitHub Actions to trigger the job.

The project exposes the link to the database via an API, which was created using GitHub Pages. This enables the data to be used by other programs or for analysis.

The project has been created using Python and its associated libraries.

**About the Original Data:**

The database contains the Magnitude (visual band) as well as the date on which the observation was taken. The Magnitude is adjusted for any observation / aperture errors.

The data is scraped from the AAVSO website and the scraped data contains multiple data elements. These are as follows:

* Star – This column contains VY Canis Majoris only.
* JD – This is the Julian Date.
* Calendar Date – The date when the observation was made.
* Magnitude – The magnitude observed on the calendar date.
* Error – Error adjustments, if any. The adjustment can be positive or negative.
* Filter – This column will always contain “vis” (for visual band).
* Observer - The three character code for the observer, who captured the Magnitude observation.

After scraping the data, the data is transformed before being stored in a Database.

**Description of the Database and Table Schema:**

This project uses a SQLite database. SQLite was chosen to minimize the space used, since the data will be stored on the GitHub repository. The project was designed to ensure that no costs are incurred (and thus SQLite).

To minimize the use of space (since GitHub has storage limits), all of the columns of the scrapped data are not stored in the database. Instead, the table in the database stores the date and the magnitude information only.

The database contains a single table – lnd\_cma. The table schema is as follows:

* Id – Integer - This is the primary key. The Id column autoincrements.
* Date – Text – The date column is stored as text because SQLite cannot store timestamp data type. The column is mandatory and contains unique dates.
* Magnitude – Real (float) – This column is mandatory.

The database contains Magnitude data since 1980.

**Project Structure:**

The following diagram depicts the high-level project flow:

**A diagram of a software process

AI-generated content may be incorrect.**

The project is segregated into the following sections:

* Components
* Pipelines
* Unit Tests

Each of these are described in detail in the following sections below.

**Components:**

The project component section contains all the modules for the project.

**Config Entity:**

The config entity only contains one class – to store the path of the Database. The path contains the name of the folder that stores the database and the database itself.

**Fetch Data:**

This module fetches the data from the AAVSO website. The module contains the FetchData() class, which is comprised of the fetch\_data() method.

The fetch\_data() method will scrape the AAVSO website and retrieve the column names and the observation rows. This method will scrape 3 pages of data and then return the column names and observation rows.

The data is scraped using the BeautifulSoup library and returns the column names and the observation rows as lists.

**Transformation Landing Layer Data:**

This module transforms the data fetched from the AAVSO website and prepares the data to be loaded into the lnd\_cma table.

The module contains the TransformLandingLayerData() class. This class accepts the list of rows and list of columns, which are returned by the fetch\_data() method in the FetchData() class, and uses these to initiate the transformation process.

The TransformLandingLayerData() class contains the following two methods:

* Convert\_calendar\_date\_to\_datetime():
  + This method converts the “Calendar Date” feature into a datetime format in order to effectively sort the data by date.
  + The method also uses the extract\_date\_feature() from the utils module to extract the date from the original date with fractional days.
  + This method also drops any duplicate date rows, keeping the row with the date that was last reported.
* Transform\_landing\_data():
  + This method initiates the transformation of the landing data.
  + The method converts the rows and columns into a dataframe, adjusts the Magnitude feature to account of any errors (including cleaning the Magnitude feature and the Error feature).
  + The transform\_landing\_data() method returns a Pandas dataframe with the Date and Magnitude features only.
  + The Star, JD, Error, Filter and Observer features from the original dataset are dropped.

**Load Landing Layer:**

This module loads the transformed data into the database table. The module contains the LoadLandingLayer() class. The constructor of the class has the path to the database as an argument. The path to the database was defined in the config\_entity module.

The LoadLandingLayer() class contains the following method:

* Load\_landing\_layer() – This method loads the transformed data into the lnd\_cma table within the database. The method creates a SQLite connection and then loads the data based on the defined schema of the table. The method will check that the date is not present in the table, before loading the data. If the date is already present, then the row will not be loaded.

**Pipelines:**

A single data pipeline script – load\_data\_pipeline.py - combines all components mentioned above and run them to extract the data from the AAVSO website, transform the data and the load the data into the database table.

Thie load\_data\_pipeline.py pipeline script is run as a cron job using GitHub Actions. The pipeline (ETL) script runs every Sunday to load the data.

**Unit Tests:**

The Unit Test module contains all of the unit tests for the various components of the project. The project leverages GitHub Actions to create a Continuous Integration pipeline, which executes the unit tests automatically when code is pushed into the Repository.

**Project Utilities File:**

The utilities file consists of functions that are used throughout the project.

Create\_dataframe() – This function converts the data into a Pandas dataframe, using the column names and observation rows scraped from the AAVSO website.

Clean\_magnitude\_feature() – This function removes any non-numerical character from the magnitude data. This is required because some of the observations contain “less than” or “greater than” signs.

Clean\_error\_feature() – The error column will be used to adjust for differences in the Magnitude feature. However, the error column does not have many entries. Most of the column is either blank or contains double-dashes (“—“) or contains “None”. This function replaces the double-dashes and “None” with a 0 (zero).

Create\_adjusted\_magnitude() – This function adjusts the magnitude feature for any errors that was recorded in the error feature. The function converts the magnitude and error column into a numerical column, before adding the error feature to the magnitude feature. Please note that the values in the error column, if present, will contain either positive or negative values.

Extract\_date\_feature() – This function extracts the date from the “Calendar Date” feature. The “Calendar Date” column contains data in the calendar date with a fractional day format. For example, 2025 July. 29.35490.

**Custom Exception File:**

For the project, a custom exception file was created to provide a more readable error message. This file comprises of a single class – CustomException – that inherits from the overall Python Exception class. This class takes the error message, fetched from the fetch\_error\_message() function, and then feeds the error message to the Parent Class (Exception class). Once the error message is obtained, it is displayed as a readable string via the python built-in \_\_str\_\_ method.

**Creating an API:**

The database can be accessed via an API, which uses GitHub Pages to expose the API. Since GitHub Pages uses static files only, a JSON file containing the path to the database was created. GitHub Pages exposes this JSON file as an API.

The link is as follows:

https://github.com/abbeymaj/vy\_canis\_majoris/raw/refs/heads/main/db/canis\_majoris.db