Scope of Project

The first task is to analyze the files that can be placed into the MMDA and figure out a database schema based on the common attributes of file metadata (date created, whether the file is local or referenced via a URL, a path to the file (URL or local), the size of the file, type of file i.e. extension, and externally linked files (if the file is HTML, for example)).

The next task is to give the user a way of adding the input files into the database. The user needs to be able to select a file or input a file path and have that file placed into the database. The file must be one of a number of pre-specified formats. If the file is HTML, the user may opt to add the files that the HTML file links to. At this stage, the MMDA needs to figure out whether the links in the HTML file are valid, i.e. whether they are broken or if they still link to a file. Only valid links should be added into the MMDA. Additionally, if a directory is selected, the user can decide which contents of the directory should be placed into the database. During each insertion of a file or directory, a GUID should be generated to identify each file and create a look-up table that maps GUIDs to DAGRs.

Finally, we need a way for the user to query the database based on the values of the metadata attributes stored. For example, the user may want to return all the files that were created past a certain date that have a common extension, e.g. .pdf.

Assumptions

1. The user is running a Unix-based machine. This means the local file system will be Unix-like, and paths to files will appear as /path/to/file rather than in the Windows format (C:\path\to\file).
2. The user must be able to access the internet to input HTML files and to follow links from HTML files, even if stored locally. Internet access also allows the user to enter queries to processed by our web server.
3. The user's inputs are not unreasonably large. The database will have limited storage capacity and the user cannot input so many files that the database will be unable to store all DAGRs to the files.

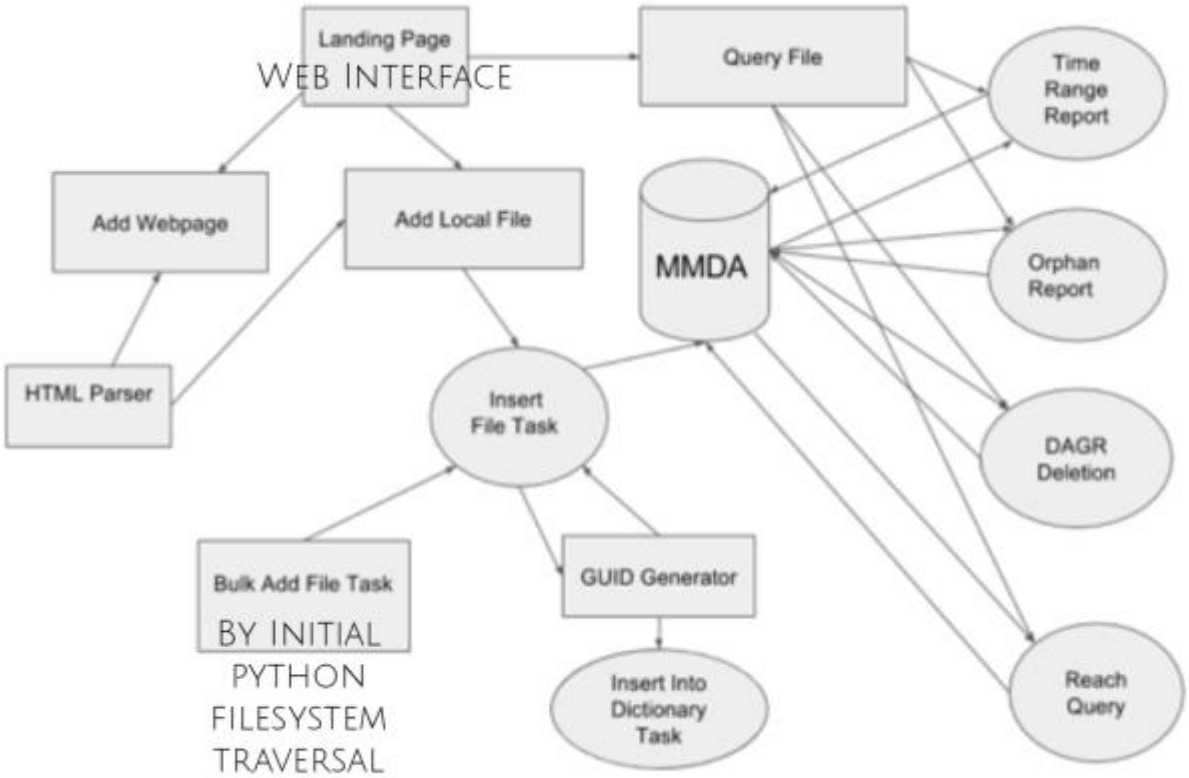
Technical Problems

Problem: Deciding how to follow links from HTML files.
Solution: When we parse an HTML file to find the files that are linked, we need to decide how many levels deep to follow links. If the user inputs an HTML file, it may contain links to other HTML files. If the user specifies a recursion depth of 1, for example, then the MMDA would only add the files directly linked to by the HTML file. However, for a recursion depth of 2, the MMDA would not only add the files directly linked by the original HTML file but also files linked to by any HTML file that were linked to by the original HTML file. This generalizes to higher levels of recursion. Our solution is to allow the user to specify the recursion depth. Our solution is to set the recursion depth to 1 by default and allow the user to specify higher levels of recursion if desired.
Problem: Generating GUIDs for files
Solution: We will use the code specified by the project description to generate GUIDs for each file on the server-side. This will be down automatically when the program decides to add a DAGR to the database. However, we need to decide how to compute the GUIDs, i.e., whether it should be based on file content or file metadata.

Problem: Choosing database software to use: SQLite vs. PostgreSQL
Solution: We are choosing to use SQLite because of the easier learning curve. After a comparison between the two, we realized that PostgreSQL is far more advanced.

Problem: Integrating our database with the back-end
Solution: Since we are using Flask for the back-end, we decided to use SQLAlchemy, a Python toolkit designed to convert relations into objects. This makes it much easier to query the database from our server.

Problem: Handling duplicate entries
Solution: Since each entry is mapped to a unique GUID, all we have to do to prevent the insertion of duplicates is compute the GUID of an entry before inserting it and performing a look-up in a hash set of GUIDs to determine if the file is already present. We only insert the file if the GUID is not present in the hash set.



TASK NAME:	Insert File Task
PERFOMER:	Python + SQLite
PURPOSE:	Insert file metadata into the MMDA database
ENABLING CONDITION:	File is one of the accepted file types and has not already been inserted into the database. In addition, if the file is linked via a URL, the link is valid i.e. not broken. User must input the file path or URL.
DESCRIPTION:	Inserts the designated file into the MMDA database.
FREQUENCY:	Whenever specified by the user.
DURATION:	Time needed to check if the file is already present plus the time needed to insert metadata into the database.
IMPORTANCE:	Critical
INPUT:	File name or URL.
OUTPUT:	Whether the insertion succeeded.
SUBTASKS:	GUID generation
ERROR CONDITION:	None

TASK NAME:	Bulk Add Task
PERFOMER:	Python + SQLite
PURPOSE:	Insert multiple file metadata into the MMDA database
ENABLING CONDITION:	Files are one of the accepted file types and have not already been inserted into the database. In addition, if the files are linked via a URL, the links is valid i.e. not broken. User must input the file paths or URLs.
DESCRIPTION:	Inserts the designated files into the MMDA database one-by-one.
FREQUENCY:	Whenever specified by the user.
DURATION:	Time needed to check if each file is already present plus the time needed to insert each file’s metadata into the database.
IMPORTANCE:	Critical
INPUT:	List of file names and/or URLs.

OUTPUT:	List of booleans specifying whether each input succeeded.
SUBTASKS:	GUID generation
ERROR CONDITION:	None

TASK NAME:	Orphan and Sterile Report
PERFOMER:	Python + SQLite
PURPOSE:	Find which files have no DAGR descendants and which files have no parents.
ENABLING CONDITION:	User request
DESCRIPTION:	Database query is made that looks for entries in the relation that do not have references to other files or have no references to themselves.
FREQUENCY:	Whenever specified by the user.
DURATION:	Time needed to perform query.
IMPORTANCE:	Critical
INPUT:	None
OUTPUT:	List of orphan and sterile files.
SUBTASKS:	SQL queries
ERROR CONDITION:	None

TASK NAME:	DAGR Deletion
PERFOMER:	Python + SQLite
PURPOSE:	Delete a file from the MMDA database
ENABLING CONDITION:	File exists within the database and user specifies the GUID to delete.
DESCRIPTION:	Performs an SQL delete query on the file.

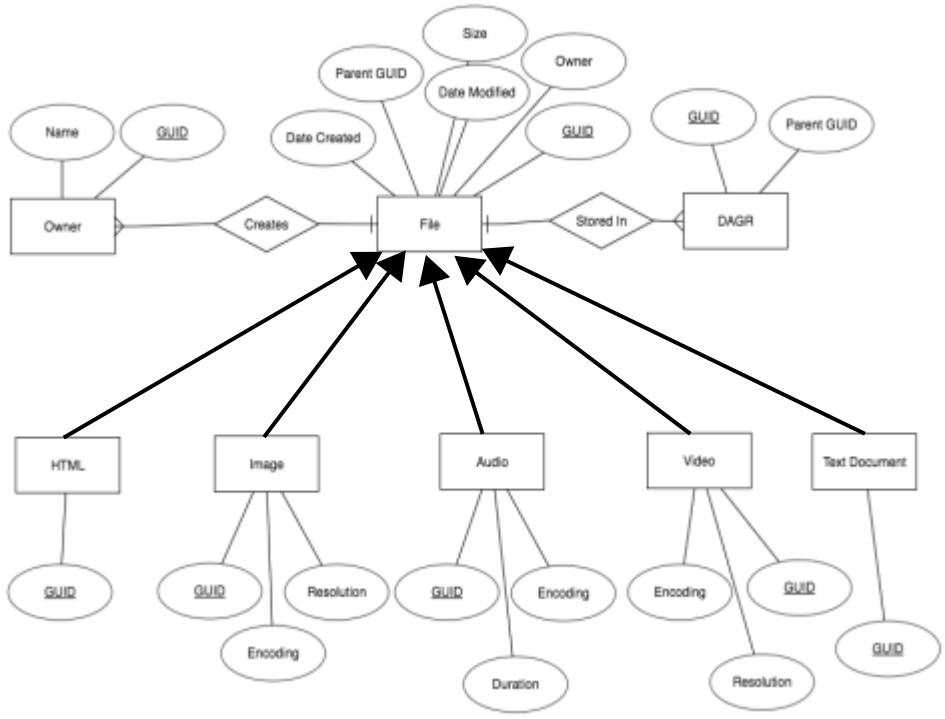
FREQUENCY:	Whenever specified by the user.
DURATION:	Time needed to perform query.
IMPORTANCE:	Critical
INPUT:	File name, URL, or GUID
OUTPUT:	Whether the deletion succeeded.
SUBTASKS:	GUID generation
ERROR CONDITION:	None

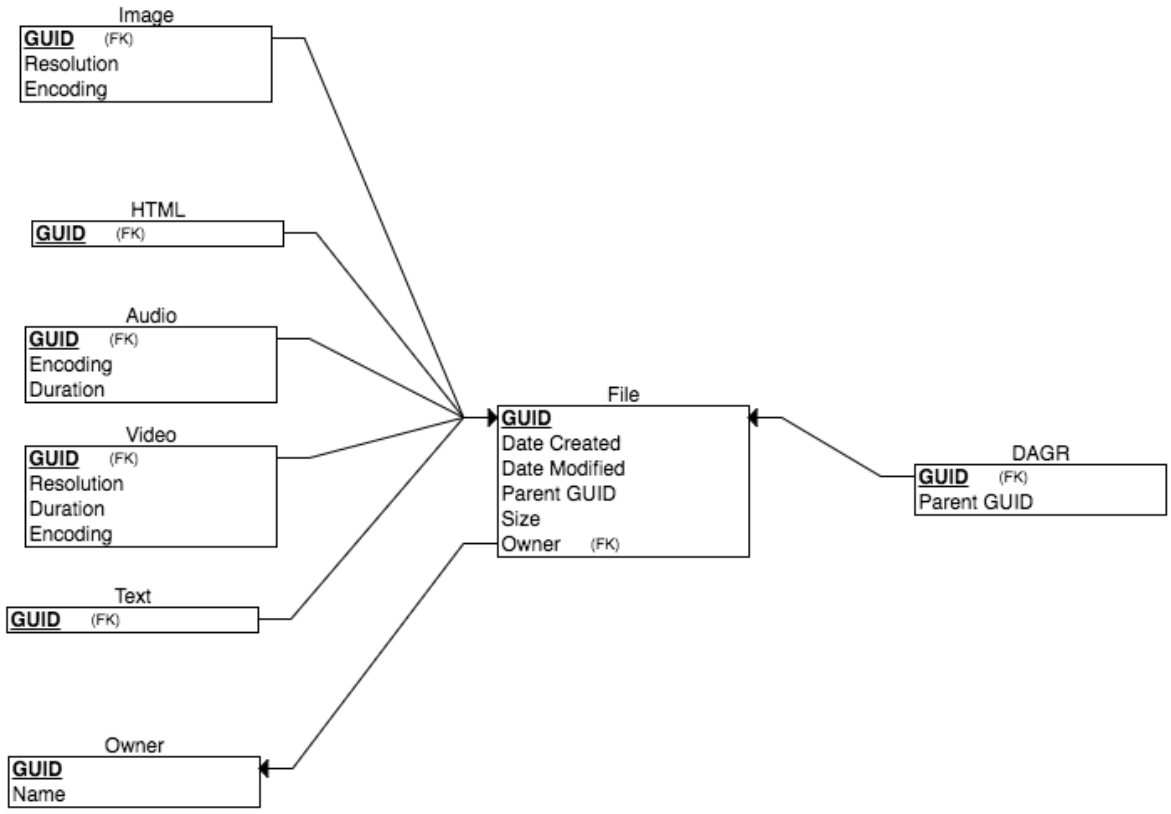
PHASE 2:

Purpose

The purpose for this part of the project is to model out an optimal schema for the database through an ER Model and its Boyce-Codd form. This helps reduce the amount of unnecessary information in each table as well as highlighting the right fields to use as primary and foreign keys. The pseudocode for each task helps to demonstrate this concept as well as show a standardized way to query certain information.

Relational Schema





Task Pseudocode and DML

TASK: INSERT FILE

If query for file based on file path returns nothing:

Generate GUID

INSERT into FILE (List of Values)

INSERT into OWNER(Name, FileGUID)

Determine file type and insert into correct table

Return True

Else:

Return False

TASK: BULK ADD

Walk through FileSystem Tree:

Call Insert File on each File

TASK: ORPHAN AND STERILE REPORT

INNER JOIN DAGR and FILE on GUID and then do a MINUS on that and FILE

TASK: DAGR DELETION

Generate array of all files with DAGR as an ancestor

For file in array:

DELETE FROM FILE where GUID=FileGUID

DELETE FROM OWNER where GUID=FileGUID
DELETE FROM DAGR where GUID=FileGUID
Determine file type and delete from the corresponding table

Database and Web Server

I was able to get my SQLite database up and running and successfully attached it to my Flask web server.