

DATABASE DESIGN PROJECT

MMDA: A Multi-Media Data Aggregator

CMSC 424: Database Design - Fall 2017

Nick Roussopoulos

Created By:

Joseph Nyangwechi

Manan Bhalodia

1.Table of Content

2. Environment and Requirement Analysis	3
2.1 Description of the Scope of the Project	3
2.2 Conceptual and design problems	3
2.2.1 Problem 1: Defining the environment of the project	3
2.2.2 Problem 2: Requirements and system analysis:	4
2.2.3 Problem 3: Implementing web server	5
2.3 Assumptions	5
3. System Analysis and Specification	6
3.1 Information flow Diagram	6
3.2 Tasks and subtasks	7
3.2.1 Identification of Data Sources	7
3.2.2 Web interface frontend	8
3.2.3 Bulk data entry	8
3.2.4 HTML parser and automatic data entry	9
3.2.5 Building relation schema	10
3.2.6 Report and Queries	11
3.2.7 Identify Duplicate Content	12
4. Conceptual Modeling	13
4.1 Conceptual Schema	13
4.1.1 Entity Relationship Diagram	13
4.2 Functional Dependencies:	14
5. Logical Modeling	15
5.1 Logic Schema	15
5.1.1 Data Aggregate	15
5.1.2 Category	15
5.1.3 Relationships	15
5.1.4 Keyword	15
5.1.5 DAGR - Keyword	15
5.2 Relation Schema	16
6. Pseudo code of tasks	17
HTML PARSER.	17
Webs Interface, pseudo code for skeleton	17
C. Insertion of a new document	18
D. Bulk data Entry:	18

E. Support for categorization	19
F. Deletion of a DAGR:	20
G. Orphan and sterile reports.	20
F. Reach queries.	21
G. Time range reports.	22
H. Identify Duplicate Content:	22
I. Modify a DAGR.	23
8. User manual	25
8.1 Adding metadata to the Database	25
8.1.1 Adding a Manual DAGR Entry	25
8.1.2 Adding metadata of a File(s) to the Database	26
8.1.3 Adding metadata through webpages	26
8.1.4 Adding a Category to the Database	26
8.1.3 Adding a keyword to the Database	27
8.2 DAGR metadata query	27
8.2.1 Metadata Query	27
8.2.2 Results of Query	29
8.3 Orphan and Sterile Reports	29
8.4 Reach Queries	30
8.5 Modifying a DAGR	31
8.6 Time Range Queries	31
8.6.1 Select Custom Time Range	32
8.6.2 Select Preset Time Range	32
9. Testing Efforts	34
9.1 HTML Parser	34
9.2 Duplicates	34
9.3 Querying with a large data set	34
10. Improvement	35
10.1 Handling Large HTML files	35
10.2 Security Concerns	35

Phase I

2. Environment and Requirement Analysis

2.1 Description of the Scope of the Project

The project is divided into three different phases each requiring successful completion of the previous phase. The first phase deals with requirements of the system, listing any technical and conceptual problems that were encountered and their solutions, assumptions made about the system, a high level flow diagram of the system, and the tasks necessary for each part of the diagram. This section also requires us to create a web server and give a brief note saying if we were able to implement it or not.

The second phase of this project requires conceptual modeling of the system along with documenting problems encountered and solutions for those problems. This phase also requires a creation of the graphical schema using the E-R model; and a relational schema created by mapping the E-R to relations, and their third normal form with keys. You must also provide the pseudo code for each task and the embedded DML code. The final step in the phase requires you to make a connection from the database and the web server implemented in phase 1.

The third phase requires an implementation of the system and a functional demo. Along with documenting the problems and solutions required in this phase, you must also provide any revisions to the relational schema from phase two and other documentation produced in this phase. This other documentation includes a short user manual for the system, any explicit test case that the system can handle reasonably, a descriptions of the system's limitations and ways for improvement.

2.2 Conceptual and design problems

We faced several conceptual and design problems in this face of the project. Most of challenges were related to defining the environment, requirements and system analysis of the project. Some of the problem were:

2.2.1 Problem 1: Defining the environment of the project

We had to define the environment of the project by mainly defining the expected inputs and outputs of our system. Our system needed to handle inputs of different formats and support

both manual and automated input systems. Our system also needed to generate specific types of reports as described in the project description.

Solutions:

Our system will have a web interface that will enable querying different types of inputs and outputs. The web interface will make it easy for users to interact with our system.

Inputs

Our system will have a web interface that will accept the following kinds of inputs:

- HTML page url: our system will mine the specified url for metadata and store the data.
- Media files - our system will accept different formats of media files eg(images, videos, mp3, mp4)
- File directory - our system will accept bulk data entry of files within a directory.
- Any other types of files.

Outputs

The system will also allow the user to query the database for different kinds of output. The specific outputs include:

- **DAGR Metadata Query:** Typically, we would like to write queries to find DAGRs using the metadata attributes, such as date, author, type, size, or keywords in the annotations.
- **Orphan and Sterile Reports:** Generate all the DAGRs that are not referenced by a parent DAGR or have no descendant DAGRs.
- **Reach Queries:** Very often we would like to find all the DAGRs that can be reached by a given DAGR (descendants) or those that point to it (ancestors)
- **Time-Range DAGR Report:** Generate a report with several preselected data items for all the DAGRs created or entered in a given time range

2.2.2 Problem 2: Requirements and system analysis:

Our database system should be able to support the different kinds of inputs and outputs.

Solution:

In order to meet the requirements specified our system will have the following tables that will enable storage of different input and generate the specified outputs.

Table 1: Data Aggregate Table

This table will store information about every DAGR. Having every DAGR in one table will help our system be functional while also being simple.

Requirements for this table.

- Input fields for this table need to be generic enough to enable storage of metadata of different types of files.
- Every entry should be assigned an unique ID that help in identification.
- Should support categorization.

The table will also hold the following metadata about DAGR: date created, date modified, size, author, type of file, name.

Table 2: Relationships table:

Captures the relationships between different entries in the DAGR table. With this table we will be able to perform a reach query.

Table will have two columns:

Parent - Contains GUID of parent DAGR

Child - Contains GUID of child DAGR

Table 3: Categories table.

Captures the relationship between different categories.

2.2.3 Problem 3: Implementing web server

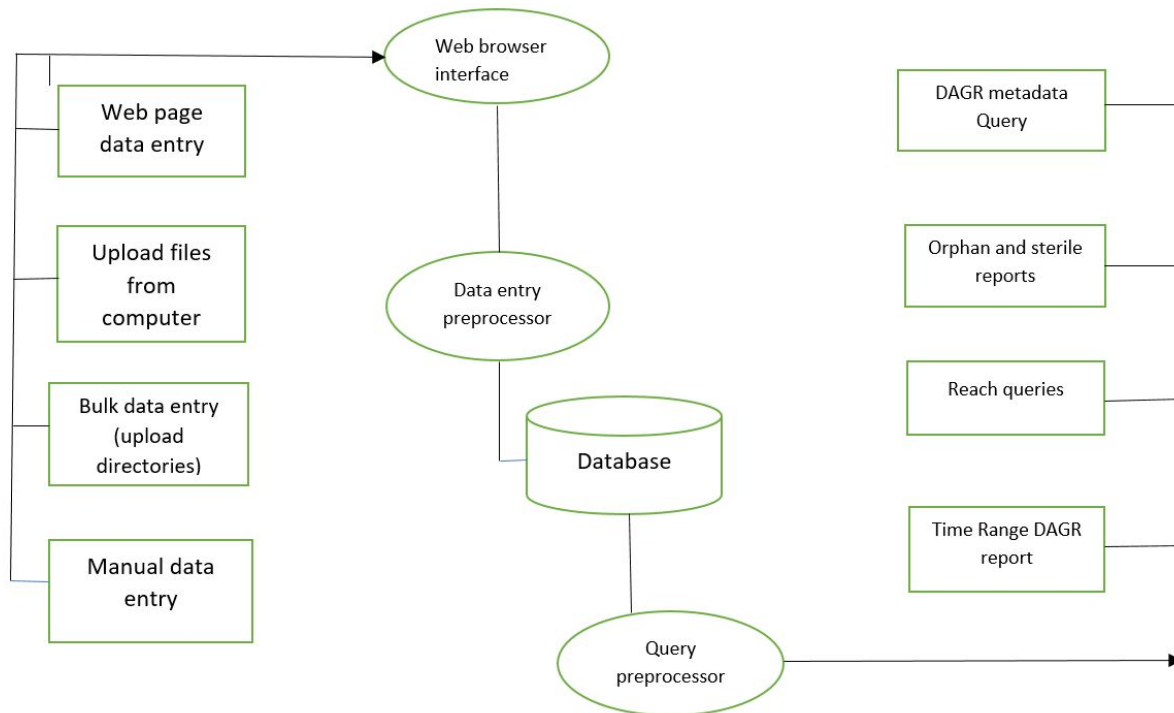
A simple web server was implemented using django and Heroku. Currently the server only displays “hello world” and is configured on quiet-hollows-36147.herokuapp.com/dagr/ .

2.3 Assumptions

- A maximum of two levels will be in the parent file
- User uses the system correctly encountering minimal errors
- There is enough space to store the data

3. System Analysis and Specification

3.1 Information flow Diagram



3.2 Tasks and subtasks

3.2.1 Identification of Data Sources

Task Number	1
Task Name	Identification of Data Sources
Performer	Joseph
Purpose	Identify of data sources that will be accepted into our system that will be used to test the functionality of the system.
Enabling Condition	System requirements have been specified
Description	Identify all the different types of data that we will need so that we can test our system.
Frequency	One time task during design and testing phase of the project
Duration	One hour
Importance	Critical to provide the correct test data for our system
Maximum Delay	N/A
Input	N/A
Output	N/A
Document Use	N/A
Operation performed	Look for both web data and files.
Subtasks	None
Error condition	N/A

3.2.2 Web interface frontend

Task Number	2
Task Name	Design Front end
Performer	Manan
Purpose	Enable users to interact with the system.
Enabling Condition	The system needs to be connected to a functioning database
Description	The front end should allow user to enter data using either manual or automatic options for both single and bulk data entry. It should also allow the user to query the database and generate reports.
Frequency	N/A
Duration	One time task done mainly at the beginning with little maintenance once system is up and running.
Importance	Critical.
Maximum Delay	Web frontend should take less than 1 second to load.
Input	N/A
Output	Web Interface frontend.
Document Use	N/A
Operation performed	Design front-end
Subtasks	Design wireframe of website.
Error condition	N/A

3.2.3 Bulk data entry

Task Number	3
Task Name	Bulk data entry
Performer	Joseph

Purpose	Allow users to upload more than one item at a time into the database.
Enabling Condition	A functioning web interface front end that is linked to the database.
Description	Allow users to upload directories of files using zip files.
Frequency	Anytime the user wants to upload more than one file
Duration	One time task at implementation phase of the database with minimum maintenance afterwards.
Importance	Critical
Maximum Delay	2 second between issuing of request and fulfillment of request.
Input	Zip files
Output	Data Aggregate items inserted into the database
Document Used	N/A
Operation performed	N/A
Subtasks	N/A
Error condition	Incorrect data selected for bulk data loading. Malicious files selected for bulk data loading.

3.2.4 HTML parser and automatic data entry

Task Number	4
Task Name	HTML parser and automatic data entry
Performer	Joseph
Purpose	Mine data from websites
Enabling Condition	Web interface frontend should be connected to the database
Description	Allows users to mine 2, levels deep, data from websites by just entering the url of the website.

Frequency	Every time a user creates a new DAGR by entering the a URL.
Duration	Initially will require significant amount of time to implement with little maintenance.
Importance	Critical system function.
Maximum Delay	2 seconds between the time the request is issued and successfully executed.
Input	URL of websites
Output	Data aggregates of URL type inserted into the database
Document Use	N/A
Operation performed	N/A
Subtasks	N/A
Error condition	Incorrect urls that do not exist.

3.2.5 Building relation schema

Task Number	5
Task Name	Building Database
Performer	Joseph
Purpose	Create relation schema that will store data aggregates and allow generation of queries and reports
Enabling Condition	A database technology and a server should set up and connected..
Description	Create relation schema that will allow storage of DAGRs, minimize data redundancy and allow different types of queries.
Frequency	One time task during system implementation.
Duration	Requires a lot of time upfront to plan for best way to store DAGRs.
Importance	Critical
Maximum Delay	N/A

Input	N/A
Output	Database schema
Document Use	N/A
Operation performed	N/A
Subtasks	N/A
Error condition	N/A

3.2.6 Report and Queries

Task Number	6
Task Name	Queries
Performer	Manan
Purpose	Allow users to query the database
Enabling Condition	Database schema stores DAGR metadata and captures the relationships between DAGRs.
Description	Queries the database to generate different reports such as DAGR metadata query, orphan and sterile reports, reach queries, Time range DAGR reports
Frequency	Every time the user asks for information from the database
Duration	Requires huge time upfront to create the queries with little to no changes in the future.
Importance	Critical
Maximum Delay	5 seconds between the time a request is issued and output is generated
Input	Query
Output	Report or Query results.
Document Use	N/A
Operation performed	N/A

Subtasks	DAGR metadata query, orphan and sterile reports, reach queries, Time range DAGR reports
Error condition	N/A

3.2.7 Identify Duplicate Content

Task Number	7
Task Name	Identify duplicate content.
Performer	Joseph
Purpose	Checks the system if similar content exists before storing DAGRs. Prevents storage of duplicate content.
Enabling Condition	Functioning frontend and backend of program.
Description	Identify duplicate content every time data is entered into the database. Items are only inserted in the database if they are not duplicate entries.
Frequency	Every time a DAGR is entered into the database
Duration	Less than 1 second
Importance	Critical to prevent storage of duplicate content.
Maximum Delay	2 second delay from the time data is entered
Input	Data entry.
Output	Outputs message indicating whether duplicate content was found.
Document Use	N/A
Operation performed	N/A
Subtasks	N/A
Error condition	N/A

PHASE II

Purpose: The purpose of this phase is to establish a proper database schema and an entity relation diagram to efficiently map out a functional database. This phase also requires a working connection of the database to the server.

Problems encountered:

1. Choosing and connecting to a database management system(DBMS) - We choose a Postgresql database due to it being ACID compliant and as a result of it being able to easily query structured data which is what we'll be storing.
2. Choosing how to create a front end - We choose html and python Django templates to construct the front end.
3. Coming up with database schema - we mapped out entity relationship diagrams and normalized them to come up with a database schema that captures the relationship between entities in 3NF.

4. Conceptual Modeling

4.1 Conceptual Schema

The conceptual schema for the database management system is a high level representation of the different entities, relationships, and constraints in the system gathered from the project specification.

4.1.1 Functional Dependencies

Data Aggregate Entity:

{GUID} -> Location, assigned_name, Author, Size, Category, Name, Type, Date Created, last_modified

{Location, name} -> GUID, assigned_name, Author, Size, Category, Type, Date Created, assigned_name

Category Entity:

{CategoryID} -> CategoryName

Relational Entity:

{ChildGUID} -> ParentGUID

DAGR - Keyword:

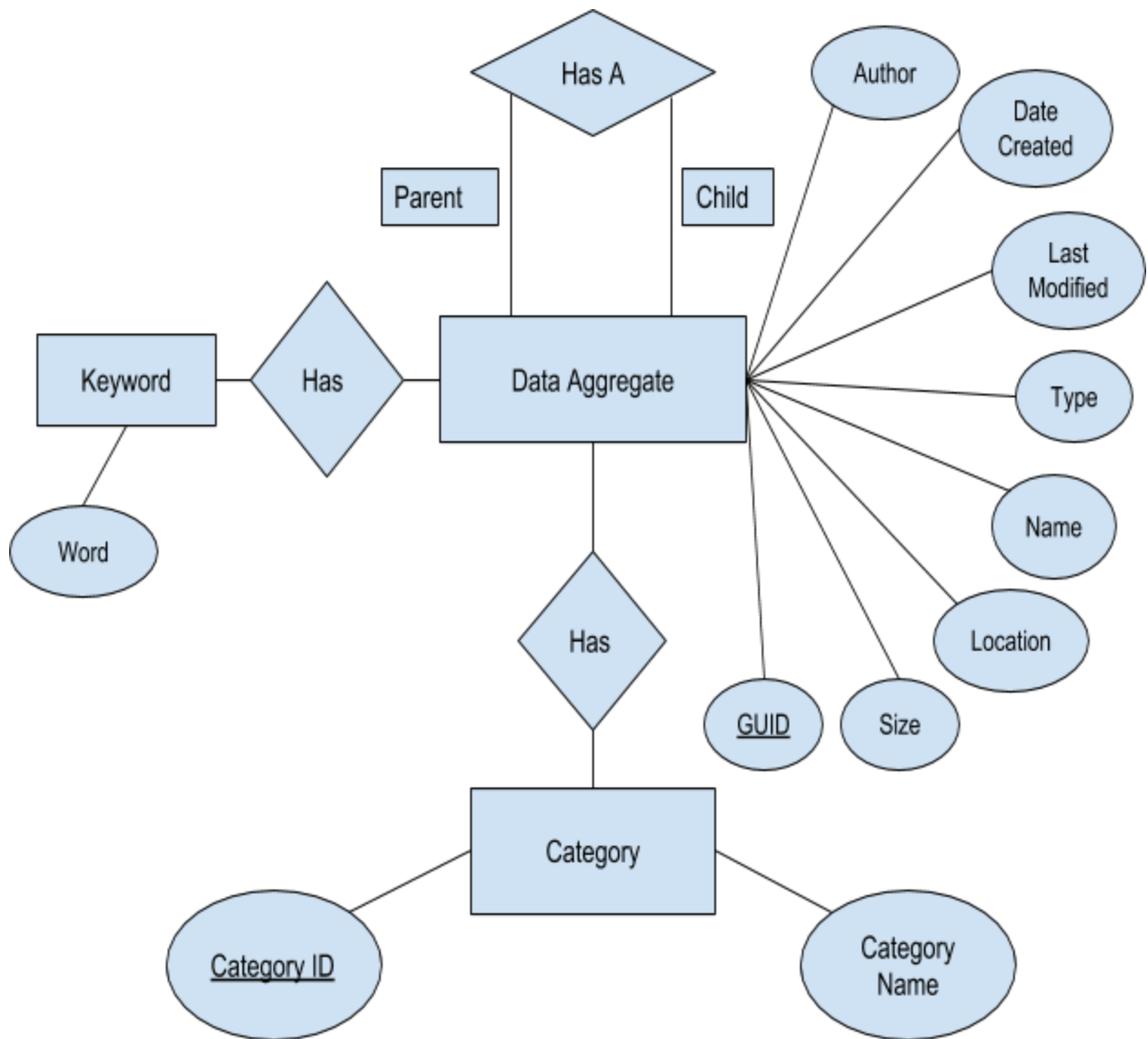
{DAGR_ID} -> Word_ID

Keyword:

{Word_ID} -> Word

{Word} -> Word_ID

4.1.2 Entity Relationship Diagram



5. Logical Modeling

5.1 Logic Schema

5.1.1 Data Aggregate

<u>GUID</u>	Location	<u>Category_ID</u>	Real Name	Assigned_Name	Size	Type	Date_created	Last_modified	Author
-------------	----------	--------------------	-----------	---------------	------	------	--------------	---------------	--------

5.1.2 Category

<u>Category ID</u>	Category_Name
--------------------	---------------

5.1.3 Relationships

<u>ParentGUID</u>	<u>ChildGUID</u>	Date_Created
-------------------	------------------	--------------

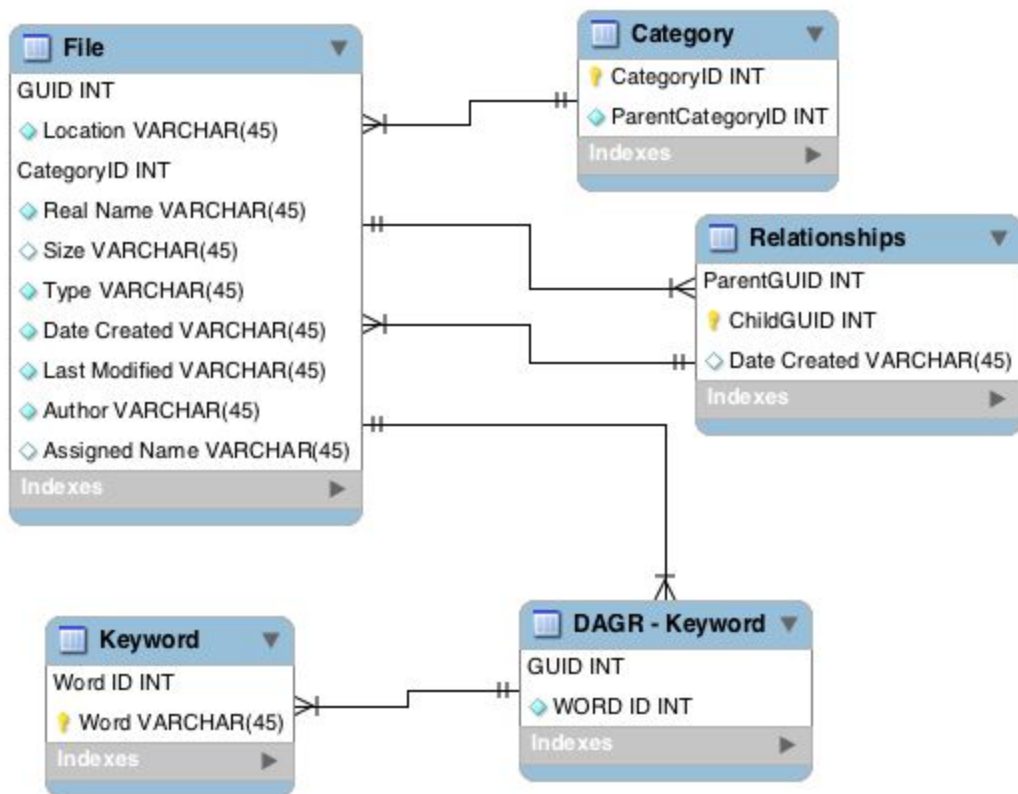
5.1.4 Keyword

<u>Word ID</u>	Word
----------------	------

5.1.5 DAGR - Keyword

<u>GUID</u>	Word_ID
-------------	---------

5.2 Relation Schema



6. Pseudo code of tasks

A. HTML PARSER.

(Actual code on GitHub:

https://github.com/mananbhalodia/MMDA/blob/master/dagr/html_parser.py)

```
a. def extractmeta (url):
    webpage = urllib.urlopen(url).read()
    soup = BeautifulSoup(webpage, "html5lib")
    meta = soup.findAll("meta")
    meta = {
        "title" : soup.findAll("meta", property="og:title")[0]['content'],
        "doctype" : soup.findAll ("meta", {"charset":True})[0]['charset'],
        "pubdate" : soup.findAll("meta", property="og:pubdate")[0]['content'],
        "lastmod" : soup.findAll("meta", {"name":"lastmod"})[0]['content'],
        "author" : soup.findAll("meta", {"name":"author"})[0]['content'],
        "keywords" : soup.findAll("meta", {"name":"keywords"})[0]['content'],
        "category" : soup.findAll("meta", property="og:type")[0]['content']
    }
    return meta
```

B. Webs Interface, pseudo code for skeleton

(Actual code on GitHub:

<https://github.com/mananbhalodia/MMDA/tree/master/dagr/templates/dagr>)

```
a. <html>
    <body>
        <form>
            Select a file to upload:
            <input type="file" id="myFile" size="50">

            <p>Click the button below do the store the file in the database.</p>
            <p>Keywords:</p>
            <input type="text" name= "First name" id="name" size="50">
            <p>Category:</p>
            <input type="text" name= "First name" id="name" size="50">

            <button type="button" onclick="myFunction()">Upload</button>
```

```

</form>
<p id="demo"></p>

<script>
function myFunction() {
    var x = document.getElementById("myFile").value;
    document.getElementById("demo").innerHTML = x;
}
</script>

</body>
</html>

```

C. Insertion of a new document

(Actual code on GitHub:

<https://github.com/mananbhalodia/MMDA/blob/master/dagr/views.py>)

a. SQL code:

INSERT INTO dagr

VALUES (guid, location, realName, LastModified, CategoryID, AssignedName, size,
type, date_created, author)

```

#add dagr by manually entering the data
def add_manually(request):
    message = None
    if request.method == 'POST':
        form_dagr = DagrForm(request.POST)
        if form_dagr.is_valid():
            try:
                form_dagr.save()
                message = form_dagr.cleaned_data['AssignedName'] + " Saved Successfully"
            except ValidationError as e:
                message = e.message
        else:
            message = "Invalid Input"
            return render(request, 'dagr/add_manually.html',{'form':form_dagr,'message':message})
    else:
        form_dagr = DagrForm()

    return render(request, 'dagr/add_manually.html',{'form':form_dagr,'message':message})

```

D. Bulk data Entry:

(Actual code on GitHub:

https://github.com/mananbhalodia/MMDA/blob/master/dagr/upload_file.py)

a. Load files

For f in files:

sql.execute("INSERT INTO dagr

Sql_files = VALUES (f.guid, f.location, f.realName, f.LastModified,f. CategoryID,
f.AssignedName, f.size, f.type, f.date_created, f.author”)

```
#useful source
#https://docs.djangoproject.com/en/1.11/ref/files/uploads/#module-django.core.files.uploadhandler

def add_file(request):
    count = None
    message = None
    if request.method == 'POST':
        form = UploadForm(request.POST, request.FILES)

        if form.is_valid():
            cd = form.cleaned_data
            #get category of uploaded files
            categ = cd['category']

            count = 0
            for f in cd['documents']:
                loc = f.name
                try:
                    dagr = handlefile(f.name, categ, f.size, f.content_type, loc)
                    count = count + 1
                except ValidationError as e:
                    message = f.name + " " + e.message
                    return render(request, 'dagr/add_file.html', {'form':form, 'count':count, 'message':message})
            else:
                message = "Invalid input"
        else:
            form = UploadForm()

    return render(request, 'dagr/add_file.html', {'form':form, 'count':count, 'message':message})
```

E. Support for categorization

(Actual code on GitHub:

<https://github.com/mananbhalodia/MMDA/blob/master/dagr/views.py>)

a. SQL code:

```
CREATE TABLE category(
    categoryID int,primary key,
    CategoryName varchar(200),
    Date_created datetime
)
```

```

class Category(models.Model):
    categoryID = models.AutoField(primary_key=True)
    categoryName = models.CharField(unique=True,max_length=100)

    class Meta:
        ordering = ('categoryName','categoryID')

    def __str__(self):
        return self.categoryName

```

F. Deletion of a DAGR:

(Actual code on GitHub:

<https://github.com/mananbhalodia/MMDA/blob/master/dagr/views.py>)

a. SQL code:

```

DELETE FROM dagr
WHERE guid = val

```

```

def delete_dagr(request, guid=None):
    guid = Dagr.objects.get(Guid=guid)
    if guid:
        guid.delete()
        m = "Successfully deleted"
    else:
        m = "Object doesn't Exist"

    return render(request, 'dagr/base.html', {'message':m})

```

G. Orphan and sterile reports.

(Actual code on GitHub:

<https://github.com/mananbhalodia/MMDA/blob/master/dagr/views.py>)

a. Orphan reports:

```

SELECT guid FROM dagr
Minus
SELECT UNIQUE childguid FROM relationships

```

B. sterile reports:

```

SELECT guid FROM dagr
Minus
SELECT UNIQUE parentGUID FROM relationships

```

```

def descendant(request, descendant=None):
    if descendant == "orphan":
        results = "Orphan"
        query = "SELECT DISTINCT Guid FROM dagr_dagr EXCEPT SELECT DISTINCT ChildGUID_id FROM dagr_Relationships;"
        #query = "SELECT DISTINCT Guid FROM dagr_Dagr MINUS  SELECT UNIQUE(childGUID) FROM dagr_Relationships"
    elif descendant == "sterile":
        results = "Sterile"

        query = "SELECT DISTINCT Guid FROM dagr_dagr EXCEPT SELECT DISTINCT ParentGUID FROM dagr_Relationships;"
    else:
        results = "No Descendants report"

    guids = [result["Guid"] for result in raw_sql(query)]
    results = Dagr.objects.filter(Guid__in=guids)
    return render(request, 'dagr/report.html', {'results':results})

```

F. Reach queries.

(Actual code on GitHub:

<https://github.com/mananbhalodia/MMDA/blob/master/dagr/views.py>)

- a. SELECT all children
FROM relationships
WHERE parentguid = val

```

def reach(request):
    results = None
    if request.method == 'POST':
        form = ReachForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            guid = Dagr.objects.get(Guid=cd['dagr_id'])
            results = list()
            if cd['ancestors']:
                results = guid.get_parents(cd['ancestors'])
            if cd['descendants']:
                results2 = guid.get_children(cd['descendants'])
                results.extend(results2)
        else:
            return render(request, 'dagr/base.html', {'message':"Invalid form input"})
    else:
        form = ReachForm()

    return render(request, 'dagr/reach.html', {'form':form,'results':results})

```

G. Time range reports.

(Actual code on GitHub:

<https://github.com/mananbhalodia/MMDA/blob/master/dagr/views.py>)

- a. `SELECT *`
`FROM dagr`
`WHERE date_created >= start AND date_created <= end`

```
def time_range(request, rang=None):
    midnight_today = datetime.datetime.combine(datetime.datetime.now().date(), datetime.time(0,0))
    now = datetime.datetime.now()
    if rang == 'yesterday':
        start_time = midnight_today + dateutil.relativedelta.relativedelta(hours=-24)
        end_time = midnight_today
    elif rang == 'last_week':
        end_time = midnight_today + dateutil.relativedelta.relativedelta(days=-1)
        start_time = end_time + dateutil.relativedelta.relativedelta(days=-7)
    elif rang == 'last_month':
        start_time = midnight_today.replace(day=1) + dateutil.relativedelta.relativedelta(months=-1)
        end_time = midnight_today.replace(day=31) + dateutil.relativedelta.relativedelta(months=-1)
    elif rang == 'last_year':
        start_time = datetime.datetime.now().replace(month=1, day=1) + dateutil.relativedelta.relativedelta(months=-12)
        end_time = datetime.datetime.now().replace(month=12, day=31) + dateutil.relativedelta.relativedelta(months=-12)
    else:
        results = "No Time Range Chosen"
        start_time = datetime.datetime.now()
        end_time = datetime.datetime.now()

    #results = Dagr.objects.all()
    #results = Dagr.objects.filter(LastModified__date__lte=end_time.date())
    results = Dagr.objects.filter(LastModified__gte=start_time, LastModified__lte=end_time)
    results = results.order_by('LastModified')

    return render(request, 'dagr/report.html', {'results': results})

def time(request):
    results = None
    if request.method == 'POST':
        form = TimeForm(request.POST)
        if form.is_valid():
            message = "Valid"
            cd = form.cleaned_data
            start_time = cd['from_date']
            end_time = cd['to']

            results = Dagr.objects.filter(LastModified__gte=start_time, LastModified__lte=end_time)

        else:
            message = "Invalid data"
    else:
        form = TimeForm()
```

H. Identify Duplicate Content:

(Actual code on GitHub:

<https://github.com/mananbhalodia/MMDA/blob/master/dagr/views.py>)

- a. `SELECT guid`
`FROM dagr as gd1, dagr`

WHERE

```
def duplicate_entry(sender, instance, created, **kwargs):
    r = Dagr.objects.filter(Location = instance.Location, RealName =instance.RealName)
    if (len(r) > 1) and created == True:
        raise ValidationError("DAGR not saved: Duplicate entry:")
    else :
        return False

#signals
models.signals.post_save.connect(duplicate_entry, sender=Dagr)
```

I.Modify a DAGR.

(Actual code on GitHub:

<https://github.com/mananbhalodia/MMDA/blob/master/dagr/views.py>)

- a. UPDATE dagr
SET column1 = value1,
WHERE dagr.guid = guid

```
def update_dagr(request, guid=None):
    message = None
    dagr = Dagr.objects.get(Guid=guid)
    if dagr:
        if request.method == 'POST':
            form_dagr = DagrForm(request.POST, instance=dagr)
            if form_dagr.is_valid():
                form_dagr.save()
                message = "Updated Successfully"
                return render(request, 'dagr/base.html', {'message': message})
            else:
                message = "Invalid Input " + str(form_dagr.errors)
            form_dagr = DagrForm(instance=dagr)
            form_dagr.fields['RealName'].widget.attrs['readonly'] = True
        else:
            return redirect('dagr:metadata')

    str_guid = str(guid)
    return render(request, 'dagr/dagr_update.html', {'message': message, 'form': form_dagr})
```


Phase III

Purpose: The purpose of this phase is to implement a working version of the MDMA Database. This includes an implementation of all the features required in the project description and handling of any edge cases. A user manual must also be created to allow the user an easy understanding of the use case of all the features.

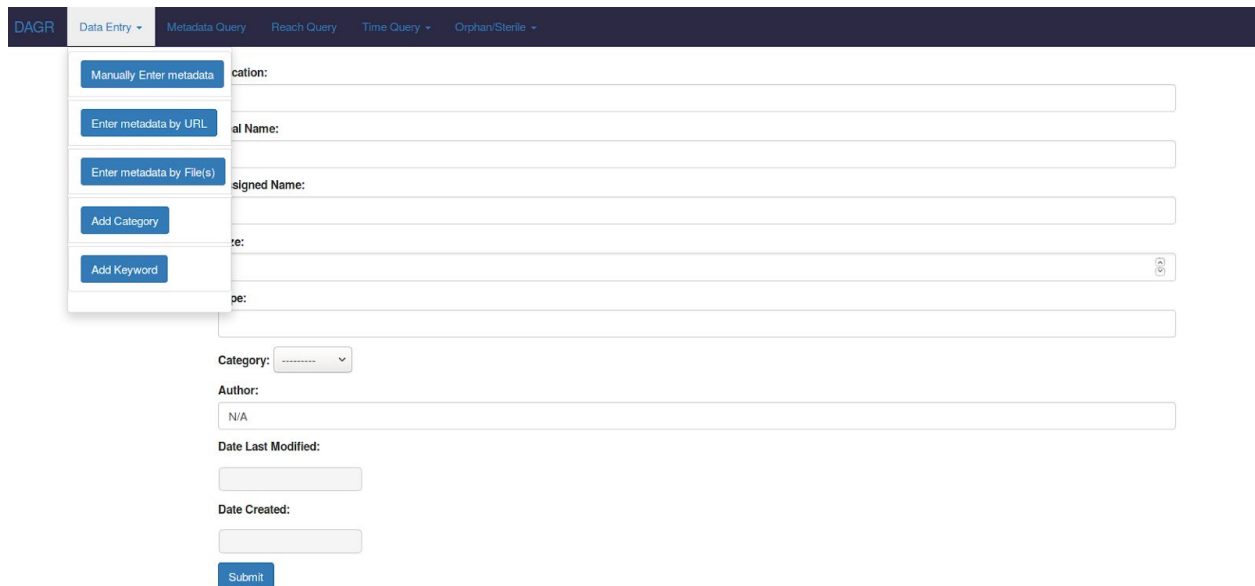
Problems Encountered:

1. **Moving the local build to a more accessible location** - We used Heroku to to deploy the local build of our project online.
2. **Build UI for application** - We used the django framework so we had to use the django frontend framework that worked well with the system. To solve this issue we used the Bootstrap and modified it to fit with the Django framework.
3. **HTML Parser** - We used python libraries like beautiful soup and Urlib to open the website url and navigate through the website. We looked at different website structures and found that the best way to extract metadata was to use the meta tag found in many website as well as tags like "<title>", "<link>", and "<a>" tags with certain properties that we can extract. We limited the links that were extracted to only those that landed on another accessible web page that contained a title (a candidate key in the DAGR table).
4. **File Extraction** - We used a Django inbuilt library that allowed us to traverse through a file and return metadata such as name, file type, size, and relative path.
5. **Database Schema** - We had to modify our original ER - model to account for keywords. Since a list of keywords would not be beneficial to store in the Keywords column we had to create a new table that stored a keyword and another table that showed the relationship between a keyword and the GUID of a DAGR.

Changes made to the relational schema from phase II: The updated ER- Model is shown on page 14. It contains a new table called "Keywords" which contains words and their unique ID; a table "DAGR - Keyword" was created to show the relationship between a DAGR and a keyword in the "Keywords" column.

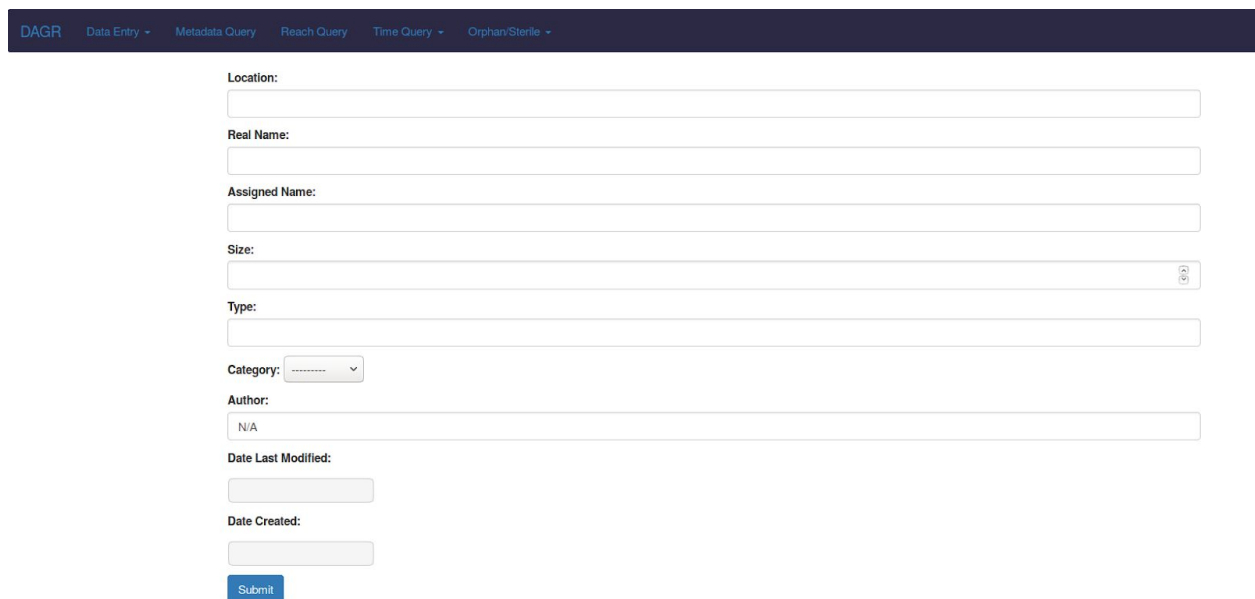
8. User manual

8.1 Adding metadata to the Database



The screenshot shows the DAGR Data Entry form. A dark blue header bar contains the text 'DAGR' and a series of navigation links: 'Data Entry', 'Metadata Query', 'Reach Query', 'Time Query', and 'Orphan/Sterile'. Below the header, a dropdown menu is open, displaying five options: 'Manually Enter metadata', 'Enter metadata by URL', 'Enter metadata by File(s)', 'Add Category', and 'Add Keyword'. The main form area contains several input fields: 'Location:', 'Real Name:', 'Assigned Name:', 'Size:', 'Type:', 'Category:' (a dropdown menu), 'Author:' (containing 'N/A'), 'Date Last Modified:', and 'Date Created:'. A blue 'Submit' button is located at the bottom of the form.

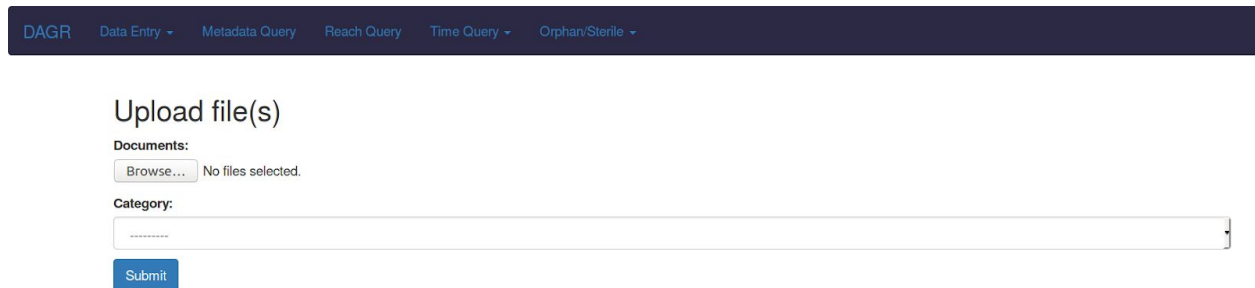
8.1.1 Adding a Manual DAGR Entry



This screenshot shows the DAGR Data Entry form with the 'Manually Enter metadata' dropdown menu open. The form layout is identical to the previous screenshot, featuring the same header, navigation links, dropdown menu, and input fields. The 'Author' field is pre-filled with 'N/A'. The 'Submit' button is at the bottom.

Enter the appropriate metadata of the DAGR in manually. A Successful submission will result in the DAGR being added to the Database. Unsuccessful submissions will result in an alert saying the DAGR was not added.

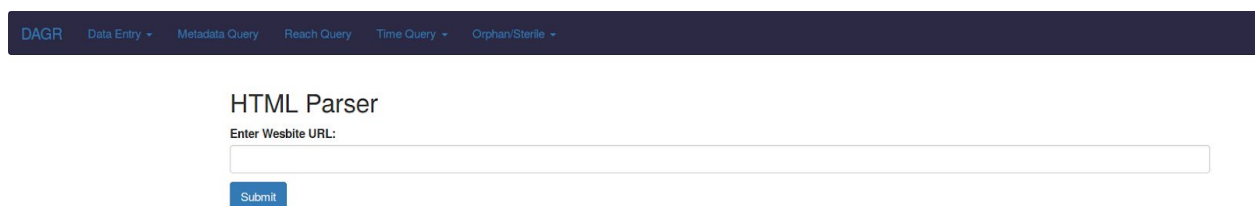
8.1.2 Adding metadata of a File(s) to the Database



The screenshot shows the 'Upload file(s)' form. At the top is a dark navigation bar with links: DAGR, Data Entry (selected), Metadata Query, Reach Query, Time Query, and Orphan/Sterile. The form has a title 'Upload file(s)'. Below it, under 'Documents:', there is a 'Browse...' button and the text 'No files selected.'. Below that, under 'Category:', there is a text input field. At the bottom is a blue 'Submit' button.

To add a file or files to the database you must select the upload files option under the “Data entry” tab. You are prompted to browse through your local files, once you select the file you can then enter a category that will be associated with the file. Successfully submitting the file will enter the metadata from the files into the database.

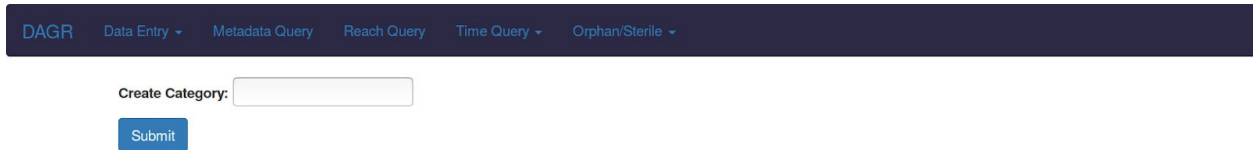
8.1.3 Adding metadata through webpages



The screenshot shows the 'HTML Parser' form. At the top is a dark navigation bar with links: DAGR, Data Entry (selected), Metadata Query, Reach Query, Time Query, and Orphan/Sterile. The form has a title 'HTML Parser'. Below it, under 'Enter Website URL:', there is a text input field. At the bottom is a blue 'Submit' button.

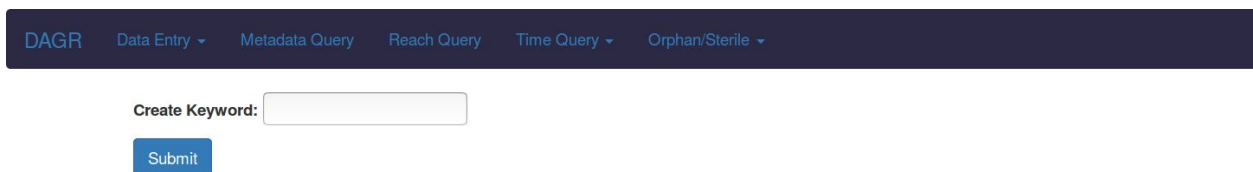
To add metadata from a URL, go to the appropriate tab under “Data Entry” and enter a URL to be parsed. Successful submission will result in the metadata to be entered into the database.

8.1.4 Adding a Category to the Database

A screenshot of a web application interface. At the top is a dark blue navigation bar with the following items: 'DAGR' (highlighted), 'Data Entry' with a dropdown arrow, 'Metadata Query', 'Reach Query', 'Time Query' with a dropdown arrow, and 'Orphan/Sterile' with a dropdown arrow. Below the navigation bar, the text 'Create Category:' is followed by a text input field. Below the input field is a blue button labeled 'Submit'.

To add a category to be available to the DAGR you must first go to the tab called “Add Category” and then enter a category to be added to the database. Once created, this category can be chosen by a DAGR.

8.1.3 Adding a keyword to the Database

A screenshot of a web application interface. At the top is a dark blue navigation bar with the following items: 'DAGR', 'Data Entry' with a dropdown arrow (highlighted), 'Metadata Query', 'Reach Query', 'Time Query' with a dropdown arrow, and 'Orphan/Sterile' with a dropdown arrow. Below the navigation bar, the text 'Create Keyword:' is followed by a text input field. Below the input field is a blue button labeled 'Submit'.

To query the database by keyword you must first add go to the “Add Keyword” tab under “Data Entry” and enter a word. The word will then be registered to the database and can then be queryable to find DAGRs with that keyword.

8.2 DAGR metadata query

8.2.1 Metadata Query

DAGR

Data Entry

Metadata Query

Reach Query

Time Query

Orphan/Sterile

Author:

Size:

Type:

Name:

Location:

Category:

Money

Sports

category 0

category 1

Keyword:

happiness

keyword0

keyword1

keyword10

Date:

Search

To search for a DAGR in the database you can search by any of these fields and a list of all matches to the query will be showed below.

8.2.2 Results of Query

DAGR

Data Entry

Metadata Query

Reach Query

Time Query

Orphan/Sterile

GUID	Real Name	Assigned Name	Type	Size	Author	Category	Date Created	Late Modified
7208	0	0	0	0.0000	Author0		Dec. 4, 2017, 3:04 p.m.	Nov. 25, 2017, 3:59 p.m.
7319	111	111	1	11.0000	Author6		Dec. 4, 2017, 3:04 p.m.	Nov. 12, 2017, 4:35 a.m.
7320	112	112	2	12.0000	Author0		Dec. 4, 2017, 3:04 p.m.	Nov. 17, 2017, 6:26 a.m.
7321	113	113	3	13.0000	Author1		Dec. 4, 2017, 3:04 p.m.	Dec. 3, 2017, 10:44 a.m.
7322	114	114	4	14.0000	Author2		Dec. 4, 2017, 3:04 p.m.	Nov. 9, 2017, 5:51 a.m.
7323	115	115	0	15.0000	Author3		Dec. 4, 2017, 3:04 p.m.	Nov. 26, 2017, 12:58 a.m.
7324	116	116	1	16.0000	Author4		Dec. 4, 2017, 3:04 p.m.	Nov. 19, 2017, 12:34 a.m.
7325	117	117	2	17.0000	Author5		Dec. 4, 2017, 3:04 p.m.	Nov. 18, 2017, 1:45 p.m.
7326	118	118	3	18.0000	Author6		Dec. 4, 2017, 3:04 p.m.	Nov. 6, 2017, 1:02 p.m.
7327	119	119	4	19.0000	Author0		Dec. 4, 2017, 3:04 p.m.	Nov. 8, 2017, 12:32 p.m.
7220	12	12	2	12.0000	Author5		Dec. 4, 2017, 3:04 p.m.	Nov. 5, 2017, 2:46 a.m.

8.3 Orphan and Sterile Reports

DAGR					Data Entry	Metadata Query	Reach Query	Time Query	Orphan/Sterile				
GUID	Real Name	Assigned Name	Type	Orphan Sterile	Size	Author	Category	Date Created	Late Modified				
7208	0	0	0		0.0000	Author0		Dec. 4, 2017,	Nov. 25, 2017,				

To find all DAGRs that have no parents, select the “Orphan” tab. To find all DAGRs that don’t have children, select the “Sterile” tab.

8.4 Reach Queries

[DAGR](#) [Data Entry](#) [Metadata Query](#) [Reach Query](#) [Time Query](#) [Orphan/Sterile](#)

Choose DAGR ID:

0
1
10
100

Choose Depth Ancestors:

Choose Depth Descendants:

Search

To find all parents or children of a DAGR you can use this feature by selecting the DAGR ID and then selecting how many levels of relationships you want to return in the appropriate box. Only either ancestors or descendants can be returned at one time.

8.5 Modifying a DAGR

DAGR

Data Entry

Metadata Query

Reach Query

Time Query

Orphan/Sterile

Update Dagr

Location:

home/

Real Name:

205

Assigned Name:

205

Size:

5

Type:

0

Category:

category 1

Author:

Author2

Date Last Modified:

2017-11-30 12:51

Date Created:

2017-12-04 03:04

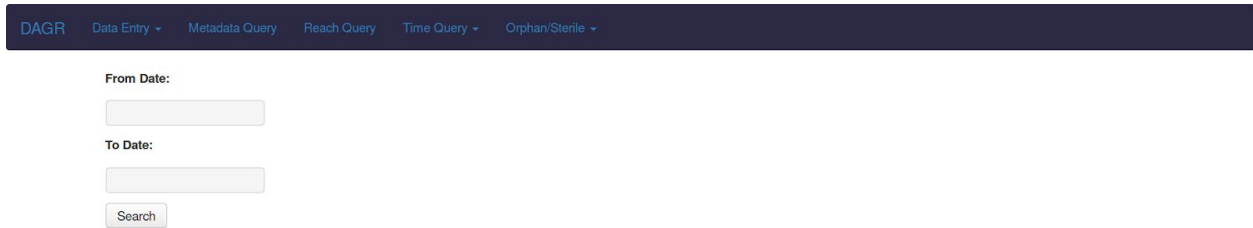
Update

Delete

To update a DAGR or to delete it from the database, select the DAGR ID from the results of a query and you will be presented with a page similar to the one above. You can edit any of the metadata or remove the DAGR by clicking on the “Delete” button. *Note: Deleted DAGRs cannot be recovered.

8.6 Time Range Queries

8.6.1 Select Custom Time Range



The screenshot shows the top navigation bar of the DAGR application with the following items: DAGR, Data Entry, Metadata Query, Reach Query, Time Query (selected), and Orphan/Sterile. Below the navigation bar, there is a form with two input fields labeled 'From Date:' and 'To Date:', and a 'Search' button. The 'Time Query' dropdown menu is open, showing a list of preset time ranges: Time Range, Yesterday, Last Week, Last Month, and Last Year.

To get all DAGRs that were either last modified or created from a certain time range, you can enter a start date and a end time. A list of results will appear below.

8.6.2 Select Preset Time Range



The screenshot shows the top navigation bar of the DAGR application with the following items: DAGR, Data Entry, Metadata Query, Reach Query, Time Query (selected), and Orphan/Sterile. Below the navigation bar, there is a form with two input fields labeled 'From Date:' and 'To Date:', and a 'Search' button. The 'Time Query' dropdown menu is open, showing a list of preset time ranges: Time Range, Yesterday, Last Week, Last Month, and Last Year.

Select one of the preset time range queries to return all DAGRs created within that time range.

9. Testing Efforts

9.1 HTML Parser

HTML Parser checks for multiple factors before entering its metadata into the database. It checks to make sure that the URL is a valid url and goes through all the meta tags to extract appropriate information. It goes through additional tags to make sure it can extract all relevant information that might still be missing. It gets the children URLs and then checks to make sure that URL is valid. The method checks to make sure that no cycles occur during the extraction process by removing the parent URL from the children. All web pages must also have a title and valid URL to be added to the database.

9.2 Duplicates

Before adding to the database, the method checks to see if a DAGR that is similar to the DAGR being added is already in the database (through location and name).

9.3 Querying with a large data set

We have randomly populated the database with a large number of DAGR (> 100) entries and tested to make sure that all queries run in an appropriate amount of time.

10. Improvement

10.1 Handling Large HTML files

Although our current code can handle any web pages, we can further improve on it by handling webpages quickly so there is not a long wait period. This would be especially useful for websites with many links as our current implementation is limited to only getting the 10 first children URLs to avoid a long delay.

10.2 Security Concerns

Our current implementation does not handle all inputs that are disguised as SQL injections. SQL injections can be masked as normal inputs and if successful they can ruin the integrity of the database. For a practical use of this database, we must implement this before use.