CIS 450/550

Data Lake Management System

Aayushi Dwivedi | Ankit Mishra | Anwesha Das | Deepti Panuganti

1. Introduction

1.1. Project goals

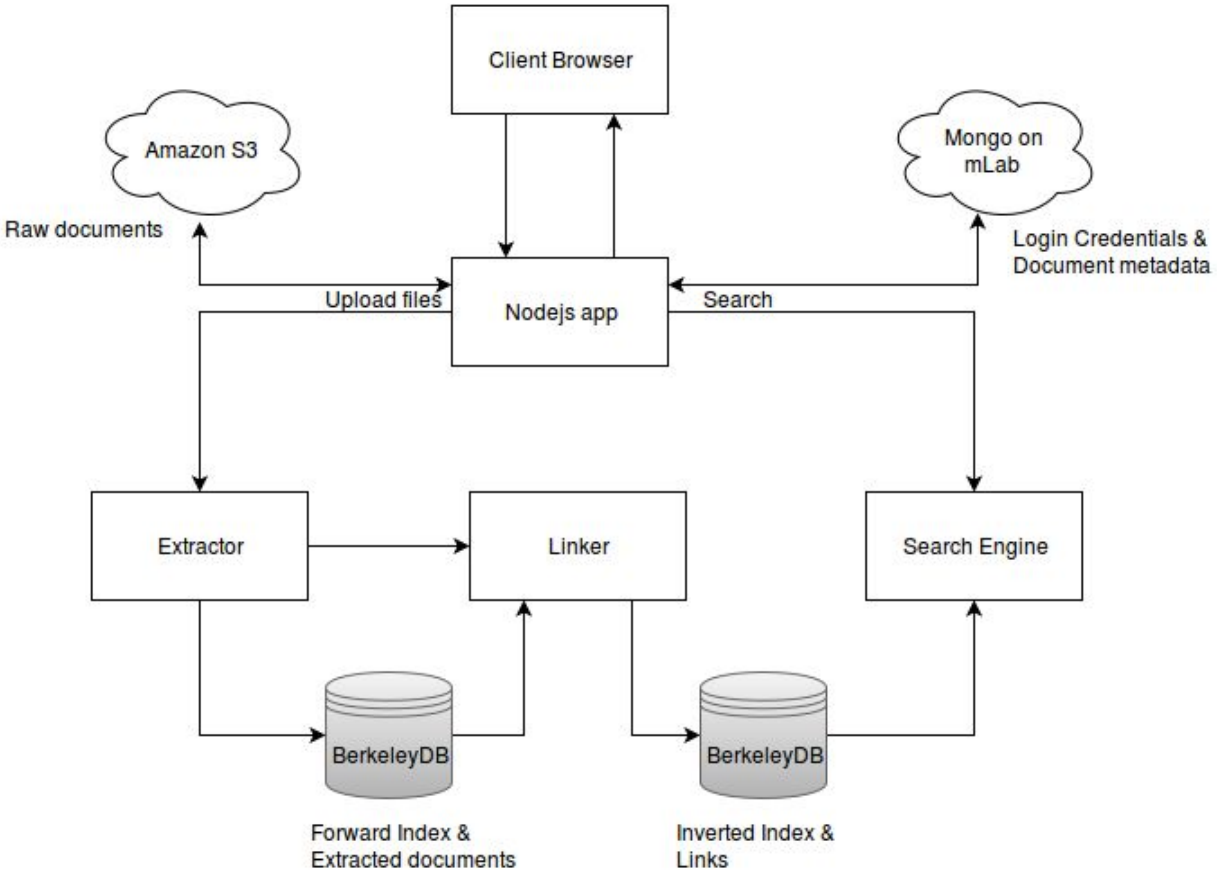
To design, implement and test a data lake management system, which allows for uploading and storing files and their extracted content, creating links between the file structure and content and searching across the links based.

1.2. Division of labor

Component	Assigned To
Web Application : Front End	**Aayushi, Deepti**
Web Application : Node Backend	**Aayushi**
Storage and Storage API	**Aayushi, Ankit**
Extractor, Forward Index	**Anwesha**
Linker, Inverted Index	**Ankit**
Search Engine	**Deepti**
Search Result UI - Graph	**Anwesha**

1.3. Project Architecture and High Level Approach

The project architecture is divided into 4 different components i.e. Extractor, Linker, Search Engine and Web application. All of these components run in parallel and independent of each other and communicate through databases.



1.4. Challenges

1. Deciding the right database for communication between the different components.
2. Dealing with heavy computation jobs in the linker. For example, a 10 mb file generated ~0.5M extracted key value pairs. Cross linking this data with, say, 10 mb of pre-existing linked data, is O(n^2), resulting in ~250 Billion combinations.
3. Integration of NodeJS web application with the java backend.
4. Speed of the search engine.

2. Components

2.1. Storage

2.1.1. MongoDB

User login credentials are stored on mongoDB. The application also maintains a replica of the uploaded files on MongoDB.

The web application uses these tables for login authentication and permission verification.

2.1.2. BerkeleyDB

The extractor, linker and search engine use Berkeley DB for data storage and communication. Berkeley DB is used to store the forward indices, extracted document to forward indices mapping, inverted index and links.

2.1.3. AWS S3

The application stores files uploaded by all the users on an S3 bucket. These files are used by the graph generation API to display the raw data for a given search result.

2.2. Extractor

2.2.1. Architecture

The extractor uses Apache Tika to detect the content type of a file. Based on the content type, it invokes the appropriate content extractor.

We also use Apache Tika to extract the metadata for each file.

The following are the set of content extractors we use -

1. **JSON**

   For JSON file formats, we use the **Jackson parser**. We create a map of key value pairs, which are extracted while iterating through the tree structure (created by the Object Mapper). If the json node is a nested json object or an array, it is assigned a dummy key value that is not used for linking. After traversing through the whole tree and putting all the key value pairs in the map, we return it as a JSON string. This is then parsed and the key value pairs are extracted.

2. **XML**

   For XML file formats, we use the **SAX parser**, which is an event based parser. It extracts the contents of the xml file and outputs the appropriate key value pairs.

   If a XML tag has a text value, then the key is the tag name and the value is the text value. In case the tag has an attribute, then the tag name combined with the attribute name is the key, and the value is the attribute value. In case a XML tag does not have any text value and has nested elements inside, it is still extracted, with the key being the XML tag name and the value as a dummy value that is not linked or considered while searching.

3. **CSV**

   We use the **CSVParser from the Apache Commons library**. We treat the first row (with the column names), combined with the row number as the key and the content of that cell as the

value. We store the extracted key value pairs in a map which is then converted into a JSON string and returned. This JSON string is then parsed and the key value pairs are extracted.

4. Others

   For all other file types, like word documents, pdf etc., we use **Apache Tika**. As discussed above, it extracts the metadata as key/value pairs, along with a key "content" and the value being the actual content of the file as one whole chunk of text.

2.2.2. Working

The extractor takes the path of the file as its input. Based on its type, it invokes the appropriate extractor and gets the extracted content, along with the metadata for every file.

Each extracted key is actually the path to the occurrence of that key in the file, starting from a root node which is the file name. For example, consider following simple JSON file called example.json -

```
{key1 : value1, key2 : [{key3 : value3, key4 : value4}, {key5 : value5}]}
```

The extracted key value pairs will look like -

example.json/content/key1 : value1

example.json/content/key2 : List1

example.json/content/key2/List1_Element1/key3 : value3

example.json/content/key2/List1_Element1/key4 : value4

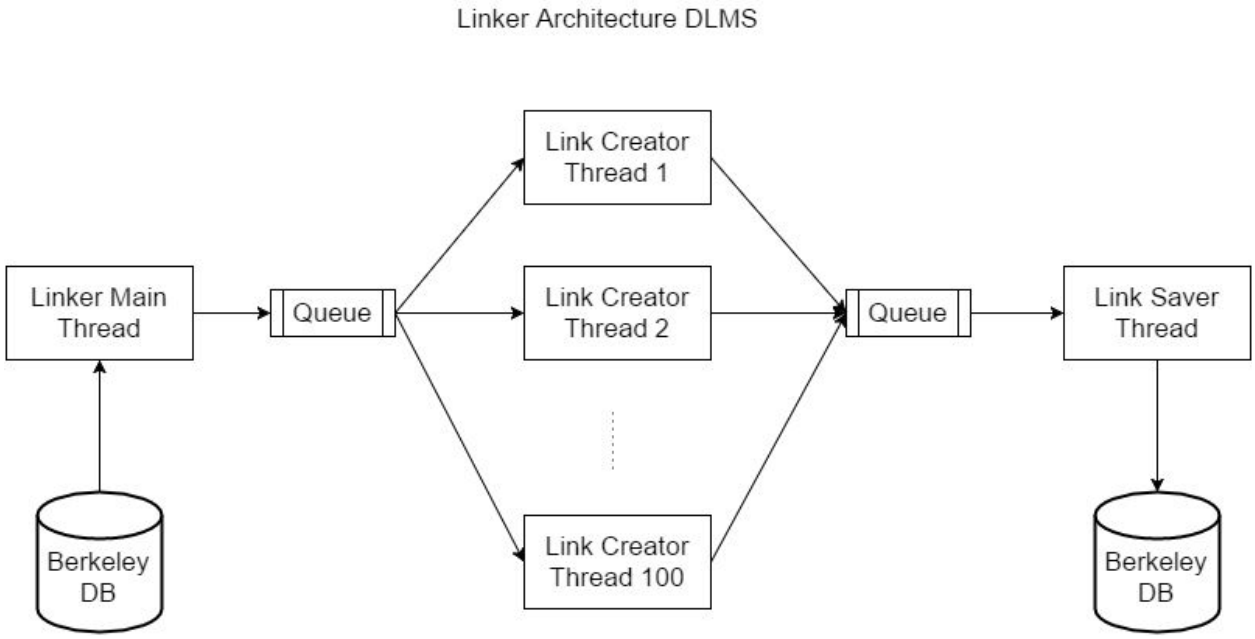
example.json/content/key2/List1_Element2/key5 : value5

These extracted pairs, along with the metadata are stored in the the **forward index** storage table.

Also, the extractor stores a document to all its extracted keys mapping in the **Flat Document** store.

2.3. Linker

2.3.1. Architecture



2.3.2. Working

The Linker architecture is shown above. The Linker consists of a main thread which is executed by the Extractor on successful extraction of a file. The Linker then performs the following steps -

1. Fetch forward indices for 1 new document : doc1
2. Create parent-child links for doc1
3. Create Inverted-index and corresponding links for doc1
4. Fetch forward indices for 1 old document : doc2
5. For all pairs of forward Indices : (f1, f2) - enqueue (f1, f2) for link creators
6. If more old documents exists, then go to step 4
7. If more new documents exists, then go to step 1

The Linker starts a thread pool of Link Creator threads that wait on the queue that stores pairs of forward indices. Once a thread extracts a pair of forward indices (f1, f2), it performs the following steps -

1. Create f1.value -> f2.path links
2. Create f1.value -> f2.filename links

3. Create f1.value -> f2.attribute links

The same steps are also performed for f2.value -> f1.path/filename/attribute. The generated links are stored in the linksQueue as Sets of Links for the Link Saver Thread.

The Linker also starts a Link Saver Thread, which waits on the Link Creator threads to write sets of links to the linksQueue. The link SAver thread extracts a set of links from the queue, merges the links based on the source and writes the links into Berkeley DB.

2.4. Search Engine

2.4.1. Architecture

1. **Input:** The input to the search engine is a query, represented as a string, and a username (the user performing the search).
2. **Output:** The final output of the search engine is a graph depicting the shortest path between the 2 keywords, or the shortest path from node to root in case of a one word query.
3. **Databases:** The search engine communicates with the linker and the inverted index. It also accesses the permissions table to ensure that private documents can only be viewed by the owner.
4. **Search:** For the search there are 2 threads one for forward search and one for backward search and each has its own thread pool for picking up nodes from the frontier for further processing.
5. **Frontier:** The search engine has a frontier that is executed as a priority queue. The elements in the queue are nodes in the graph that are represented as an object of a custom class *"WeightedPath"* that contains the node, the path that found the node and the cost/weight of the path. The nodes in the frontier are prioritized on cost so shortest path is at the head of the queue.

2.4.2. Working

The search engine was implemented as a bidirectional, depth-limited, multi-threaded graph search, where both directions are implemented as a Breadth First Search. It supports both one word and two word queries.

Bidirectional Search:

When the search engine receives as input, a 2 word query, it starts two threads to search the graph. One thread starts from the first word (forward thread) and the other thread starts from the second word (backward thread), and the search finds a result when an intersection is found between the two threads. Each thread starts its own thread pool which is what does all the work.

1. Finding a Path: Each thread keeps track of the nodes that it has seen as well as the nodes seen by the other thread. When a node is picked up from the frontier, the first check is to see if this node has already been seen by the other direction thread. If so, this would mean that one path has been found. In that case, the 2 paths are merged to get the full path and added to the result. If “k” results have been found, the search is done. Otherwise, it continues.
2. Exploring Nodes: If the node isn’t an intersection point, the depth of that search is determined, by the path length and if it exceeds the limit “d”, this path isn’t explored further. If the depth is less than “d”, the linker table is queried, to get all the links for that node. The links are filtered so as to ignore paths with cycles. The cost of the new paths is calculated and the new WeightedPath object is added to the frontier, to be explored further.
3. Completing the Search: The search is complete either when the frontier is empty, and no new nodes are being explored, or the top “k” shortest paths have been found.

One Word Queries:

When the search engine gets a one word query, the shortest path from the node to its root is returned. This is also implemented as a multi-threaded, depth-limited search.

1. Finding paths: The search engine queries the inverted index to get all occurrences of the word. The cost of the total path for each occurrence is computed. Then, each occurrence, with it’s path is added to the frontier, based on it’s cost. The top “k” shortest paths are the top “k” paths in the frontier.

Permissions:

Before a node is explored, its accessibility is checked. If the file it belongs to is “Private”, and isn’t owned by the current user, then the node isn’t added to the frontier. Otherwise, it is added.

Graph: The final results are displayed as a graph showing the path from word1 to word2 or from word to root. To create this graph, it takes an input of list of paths, represented as JSON Objects. The graph was built using D3.

2.5. Web Application

2.5.1. Architecture and Working

The web application is built on Nodejs. It interacts with the client and acts as a middleware between the client browser and different components of the Data lake Management System.

The web app uses MongoDB to store account details and document metadata. When a client makes a request, the server first verifies whether the user is registered by querying User table on mLab. In addition to that, the server maintains sessions for each logged in user. The max-age of a session is 10mins.

A logged-in user can do following tasks:

1. Upload and view files
2. Change permissions on files she owns
3. Search the DLMS and view graphical representation of the search results

As shown in the system architecture figure, the server communicates with different components of the system to serve the user’s requests. The web app uses node-java library to run the respective Java applications on the node side.

When a request to upload a file comes in, the server makes an asynchronous call to Extractor.extract(). The extractor in turn initiates the linker upon completion of extraction. Once linking of the uploaded document has been done, the linker returns to

the callback function on the node side. The asynchronous nature of this call does not block the server and the user can keep using the DLMS.

The server makes synchronous calls to the Search Engine since the user needs to see the results immediately. The output of the search engine is further processed at the server side to display just the path in following format: *node1 -> node2 -> node3*, where the search query is ***"node1 node3"***.

Each search result can be viewed in a graph format. The graph API uses D3 to convert the result of the search engine into clickable graph. The clickable nodes of the graph refer to the raw documents store on Amazon S3.

3. Validation of Effectiveness

3.1. Experiment

We used the following files for testing the accuracy of linking and search:

Test1.json

```
{
  "data":
    {
      "id": "1234",
      "from": {
        "name": "Tom Hardy ",
        "id": "X12"
      },
      "message": "Will play in 2016!"
    }
}
```

Test2.json

```
{
  "new_data":
    {
      "id": "5000",
      "from": {
        "name": "Tom Brady",
        "id": "X11"
      },
      "message": "Will Act in 2016"
    }
}
```

1. **Response Time:**

For testing the response time of the search algorithm, we extracted and linked the above json files and searched for the following keywords:

Search Query	**Average Response Time (mSec)**
Tom	~350
Tom Brady	~1300
Hardy Brady	~1350
Data new_data	~1370

2. **Accuracy:**

Based on the two documents we searched for sample keywords to test the accuracy of the search engine:

Search Query	**Top Search Results**
Tom	Tom->name_0->from->new_data->demo_generated_brady.json tom->name_0->from->data->demo_generated_hardy.json
Tom Brady	tom->name_0->brady brady->name_0->from->name_0->tom brady->name_0->from->id_0->from->name_0->tom brady->name_0->from->id_0->x11->id_0->from->name_0->tom
Hardy Brady	hardy->name_0->tom->name_0->brady hardy->name_0->from->name_0->tom->name_0->brady hardy->name_0->from->id_0->from->name_0->tom->name_0->brady hardy->name_0->new_data->name_0->tom->name_0->brady