

Data Lake Management System

Introduction

Data Lake Management System is a large collection of partially structured data in “data lakes”. In a world where the amount of data we accumulate continues to grow rapidly, the need to develop better tools to manage data is greater than ever before. Through a DLMS, large sets of documents and data can be rapidly searched through in order to find similarities and effectively filter for keywords. Unlike a generic search tool, DLMS focuses more on specific keywords in relation to the documents they are in, as opposed to their general existence across various forms of data. One of the main reasons DLMS are used at a large scale is because of the ability to employ MapReduce algorithms to perform operations on a huge subset of data concurrently, by using multiple servers / computers. Data Lakes also offer the option to store data in its native form, allowing inputs to be structured, semi structured, or unstructured. With a large focus on agility, flexibility, and efficiency, Data Lakes are becoming a prominent way of storing large data - this project focuses on a basic implementation of a DLMS.



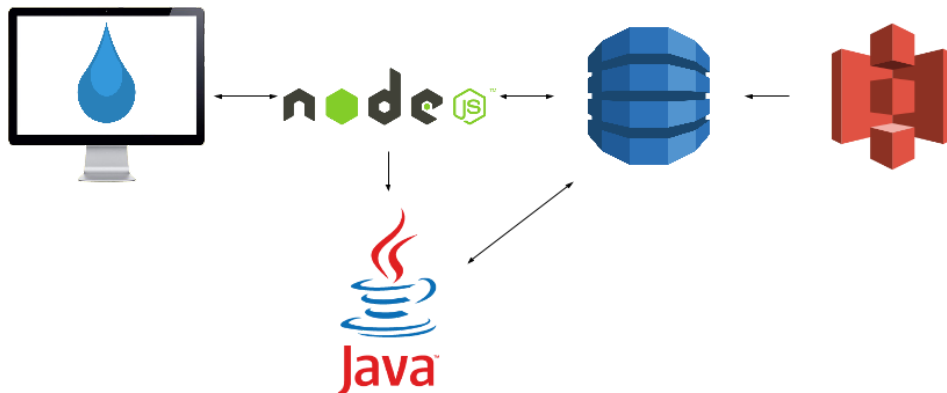
Architecture

The DLMS application consists of a front end written in HTML, CSS, and jQuery. The front end communicates with a RESTful server written in Node.JS. This server stores user information (email, username, password, document list) and information about raw data (id, owner username, list of viewers, whether it is public, and a URL to the file on S3) in DynamoDB tables. The server also allows file uploads, which are stored on S3.

On upload, a .jar file is called by the server to extract the uploaded file and to store key, value pairs on DynamoDB tables.

When a search query is performed, the server, again, calls a .jar file which performs the search and prepares a JSON with the results to pass to the client.

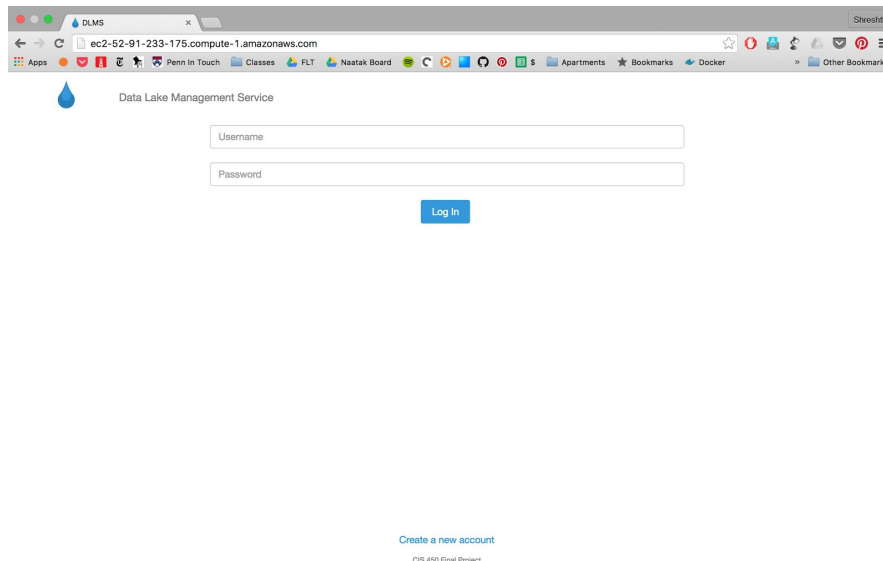
An interaction diagram can be found below



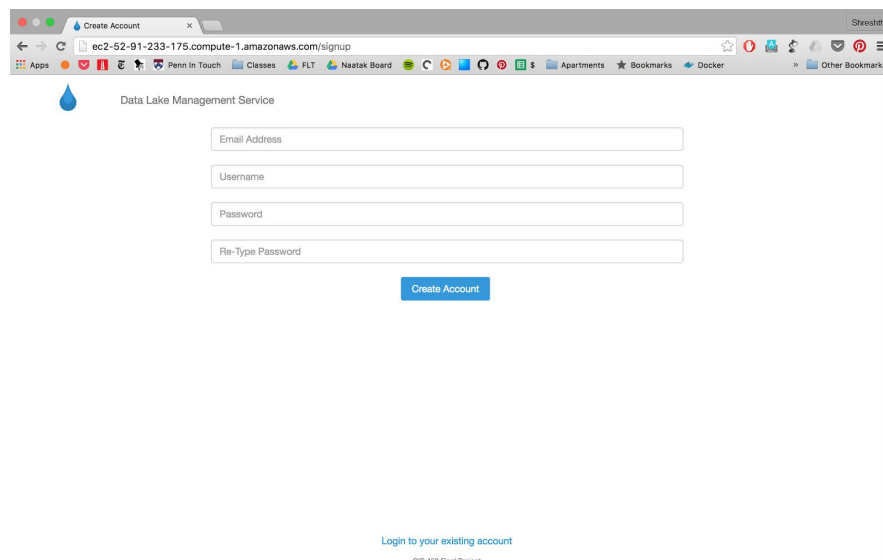
The Web Application

The back end for the web app is written in Node.JS and hosted on an Ubuntu EC2 instance. The features it supports are as follows:

- Account creation and login/logout: Users can create new accounts, login, and logout from the system.

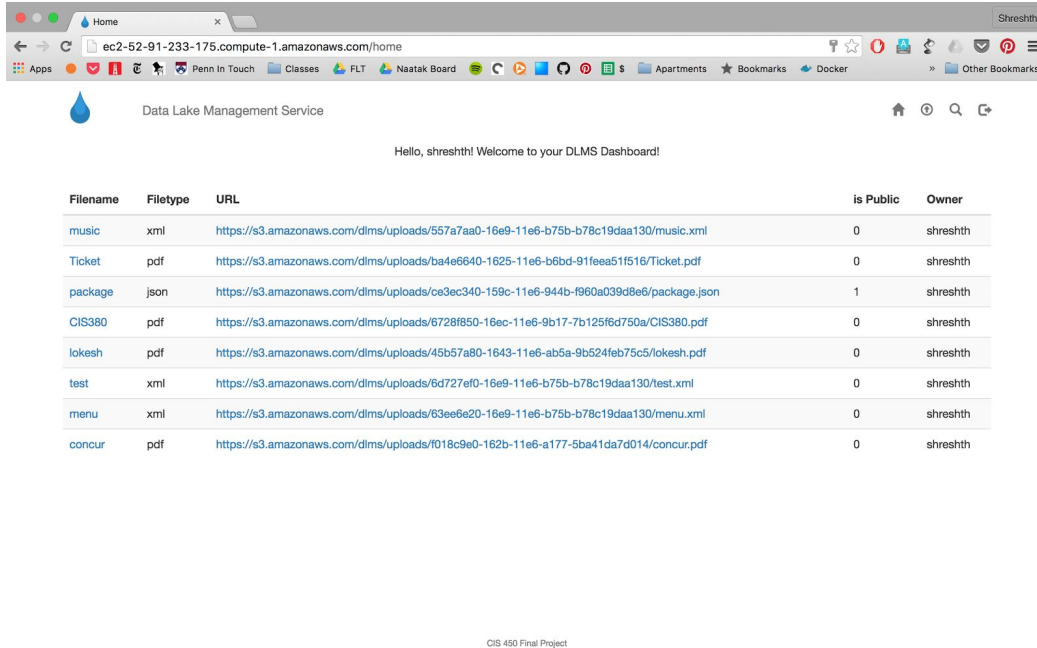


The screenshot shows a web browser window with the address bar displaying "ec2-52-91-233-175.compute-1.amazonaws.com". The page title is "Data Lake Management Service". The login form consists of two input fields: "Username" and "Password". Below these fields is a blue "Log In" button. At the bottom of the page, there is a link that says "Create a new account" and a small text "© 2018 450 Final Project".

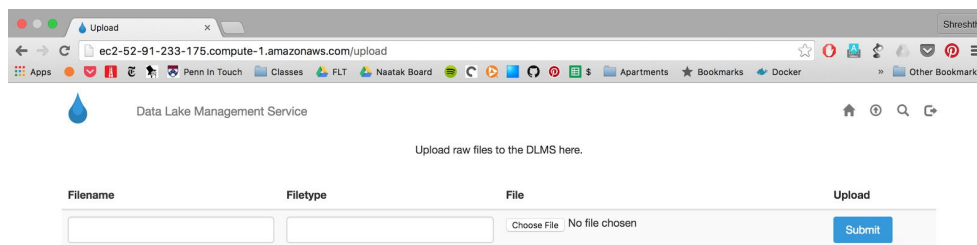


The screenshot shows a web browser window with the address bar displaying "ec2-52-91-233-175.compute-1.amazonaws.com/signup". The page title is "Data Lake Management Service". The create account form consists of four input fields: "Email Address", "Username", "Password", and "Re-Type Password". Below these fields is a blue "Create Account" button. At the bottom of the page, there is a link that says "Login to your existing account" and a small text "© 2018 450 Final Project".

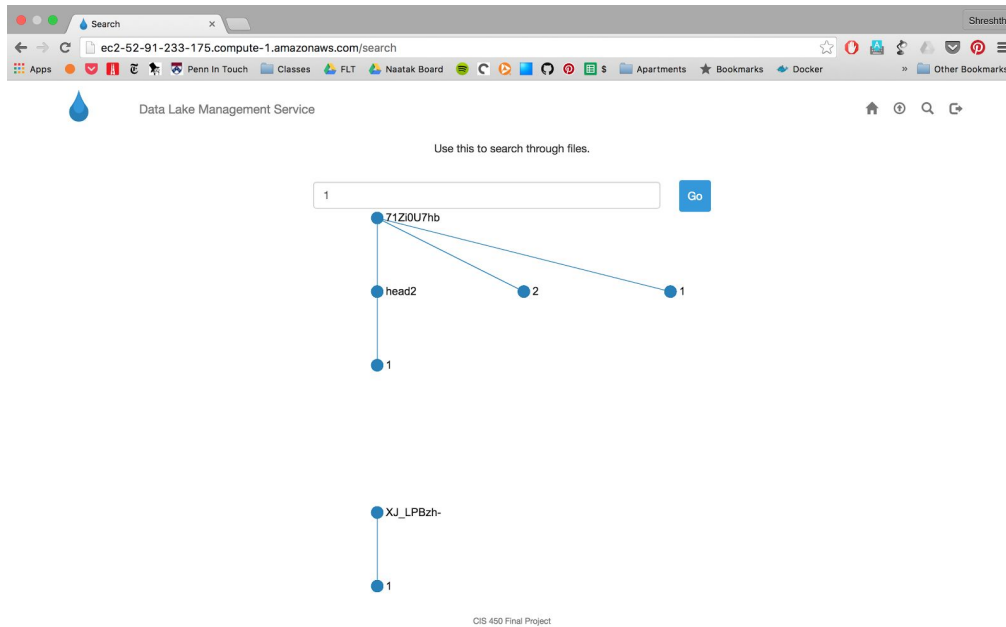
- Viewing list of files the user has access to: On the home page, a user can see a list of the files he has access to (either owns or has view permissions). For each file, filename, filetype, S3 URL, is public, and owner can be seen. Filenames can be clicked on to view permissions and a preview of the file.



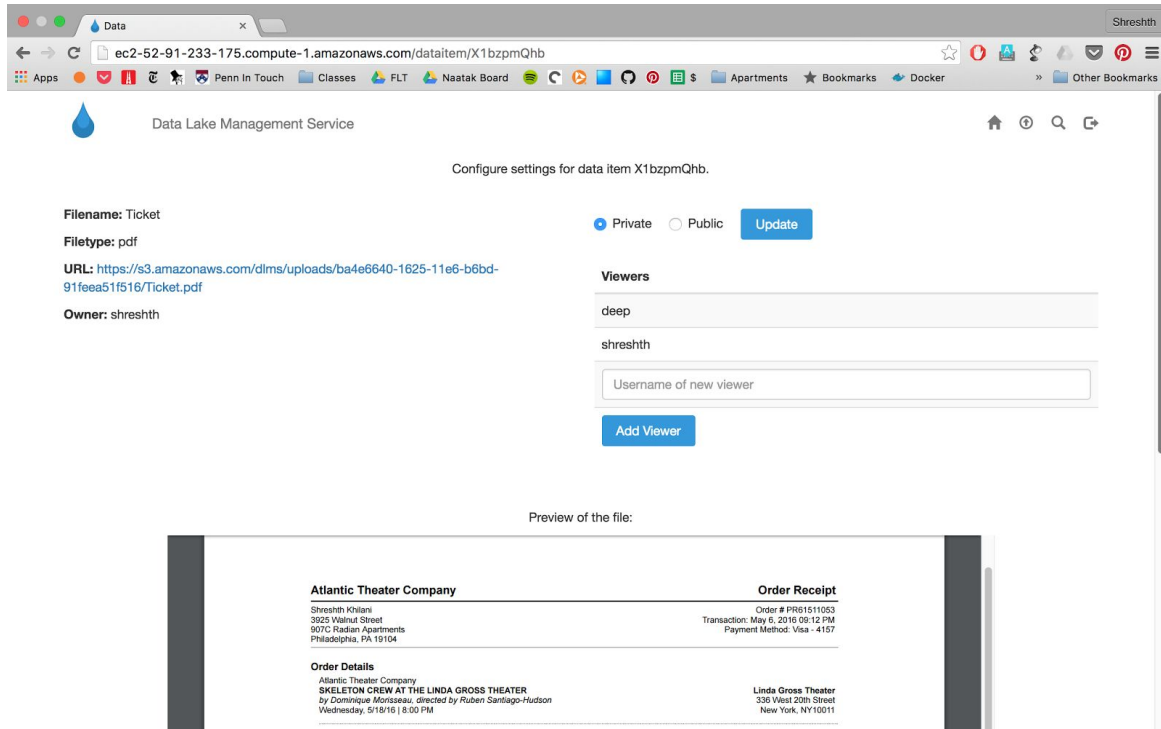
- Upload files: User can click an upload button on the navbar to be directed to a page which allows the user to upload new files. Every time a user uploads a file, the Node server calls a “extract.jar” with the id of the file. This .jar extracts the information from the file and stores it into key-value pairs.



- Search queries: Another item on the navbar is the search feature. A query can be typed out and results are displayed as a forest. For this, the server calls a “search.jar” with the search keyword and the username of the current user (for permissions). This .jar writes a JSON to the server which sends this to the client to represent as a forest.



- Viewing and changing file permissions: The last feature of the application is the permissions. A user can click on a filename on the home screen to go to a detailed view of the file, the permissions, and a preview. There are two types of permissions: fine and coarse. The coarse permissions allows a user to toggle a file between private and public. Private files are only searchable by the users who have access to them. Public files are searchable by everyone. The fine permissions allow a user to (when the coarse permissions are on Private) select a list of viewers who they want to have access to the file. Permissions can only be changed by the file owner.



Extraction

The system has designed to take in any kind of file, and extract the necessary keywords from them. This was achieved through using a variety of open source platforms that allow the program to break down XMLs, JSONs, PDFs, Word docs, HTML docs, etc. The website calls on a separate .jar which uses the correct method to extract data, which means that the overall model is quite scalable if further file support is necessary in the future.

Algorithms - indexing, linking, searching

Indexing was a straightforward hashing of the key-value pairs generated by the extractor. The index as grown in parallel to the data extraction. A reverse indexer was also simultaneously created

mapping values to their keys. The linking algorithm used a Ngram method to find similar documents to link to, finding a weight factor between 0 and 1 based on string differences in keywords, to help rank the closely linked documents.

The search engine was modeled off personalized content selection by content providers, such as YouTube or Spotify. First the search model discovers paths between each query token and document roots. These paths are then folded between different tokens to create shortest walks between the tokens. An initial constant k is used to specify the max length of paths that will be considered. The initial constant k scales proportionally to the number of unique tokens in the shortest walk between the tokens. The logic behind this increase is that as the complexity search queries grows through increasing query size the scope of the search needs to expand as well.

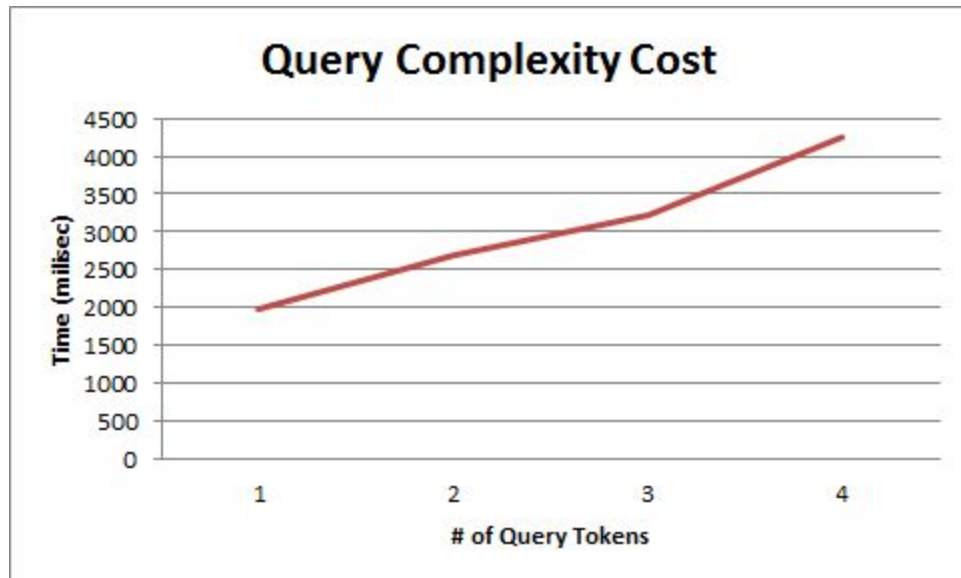
Once the walks have been created, they are amalgamated in trees representing each document. Each document's token walk trees are then ranked by summing the the weight of each walk connecting different tokens in the tree, where the weight can be calculated as a constant m taken to the power of the level of the walk divided by the walk's distance. The level of the walk is defined as the number of different tokens on the walk. So a walk with 3 tokens on it would have a level of 3. The reasoning behind this formula is that as the the number of different query tokens on a given path increases, that path experience a disproportionate increase in its likelihood of being relevant to the user. However as the walk length expands the associations between the tokens in the original query.become increasingly unlikely to have been maintained. Furthermore, additional token walks in the document should increase the overall documents strength relative to the query since higher token path frequency signals that the document might have a high focus on the query or closely related subject. The additional weight should depend on the weight on the walk itself, since a document with a lot of strong walks should be weighted higher than the a document with a lot of weak walks.

The document with the highest pagerank is used as the starting point of the search. Similarly to how YouTube and Spotify would want to initially return the result that is most relevant to the user query. The the algorithm then returns the return the token path representation of the doc. The purpose of this return type is that a company like YouTube would like to return the strong content to the user for their viewing purposes. However, companies could benefit from knowing where exactly the content is in the video so that they can allow users to jump directly to it. This especially, useful for companies like Twitch which host live streams that can store hours of content, making it difficult for users to manually locate their desired content. In order to move to new content, the search algorithm uses inter-document links to determine, which documents with the token paths were most related to the current document. Document similarity was determined by n-grams as stated above. The reason for heavily weighting document similarity is that entertainment consumption search doesn't have a general target goal like information search does. Instead they can generally be bucketed into similar "moods". For example, someone watching a video about Obama's policies, would most likely prefer to watch a newstrip on current events that occasionally mentions Obama over a comedy show dedicating an entire episode to making fun of Obama.

Validation of Effectiveness:

The search algorithm scales poorly due to its reliance on DynamoDB. Just connecting to a Dynamo table once can take several seconds, which heavily penalizes algorithms like search which have a greater than proportional growth in necessary queries as the document database expands. Storing the data locally using a different database service could tremendously enhance the scalability of our search rank. Running 15 random queries per query size ranging from 1 to 4 on a dataset of random subset of the the database consisting of 90 key-value pairs, yielded a roughly linear relationship related to query length and runtime. Therefore, the database would be able to scale to in this aspect fairly well. However, if query length does become an issue it is possible to improve database scalability by distributing parts of the search algorithm.

For example, the task of finding paths to the root node can be divided parallelly across multiple machines, then sent back to the master machine, which can then fold them into shortest token walks.



Team member Breakdown

Web application (Node.JS backend and all frontend development), AWS setup: Shreshth Khilani

Indexing and Search Engine: Vijay Prabakaran

Extraction and Linking: Deeptanshu Kapur