CIS 550: Database and Information Systems

Data-Lake Management System

10 May 2016

Team Name: DataLakers

Jitesh Gupta, Mani Mahesh, Sajal Marwaha, Sanidhya Tiwari

Table of Contents

Introduction

As per wiki, a data lake is a large-scale storage repository and processing engine which provides "massive storage for any kind of data, with enormous processing power". The aim of this project is to create a smaller version of a data lake management system which can store both semi-structured data (CSV, XML, JSON) and unstructured data (TEXT, PDF etc.) and extract key value pairs from them, independent of their format, in order to link different documents with each other. Thus, the application provides a search interface which returns the links between different documents for a particular query string. The application will also allow users to upload & share files using an authentication management system so that they can find other documents in the corpus linking with their own.

Web Interface

NodeJS

The front end interface was developed on node.js with the help of https://scotch.io/ tutorials comprised of the following modules:-

- Express: for the web application framework
- Mongoose: for querying and storing user login and signup information on mongodb
- Passport: for user authentication and password safety using by-crypt, salt and hash functions
- S3FS and Connect Multiparty: for connecting to the AWS S3 database, roles were assigned in aws with FullS3Access.
- Connect-flash: for flashing error messages
- Other modules like body-parser, express-session, bycrypt-nodejs were also used

Embedded javascript (ejs) was used as the view engine and along with the local CSS, bootstrap CSS was also referred for styling. The top 10 search engine result paths were displayed using d3.js tree structures. The final NodeJS app was deployed on Heroku.

Client-Server Communication

Our application required integration of NodeJS (javascript) and Java on the server side but we found it very difficult and cumbersome. Hence, we separated these two components by making the NodeJS a client application and using Java Servlets on the back end server side. The different processing components of the back end can now be easily instantiated using the servlets with only a small overhead of data communication between the client and the server. Not only this design choice provided a great abstraction, but it also made our application completely centralized as the database was located on the server, and hence multiple nodes can access it at the same time. The data between the front end (client) and the back end Apache server was communicated in the JSON format. We used JSON as it is easy to parse and customizable according to the particular application.

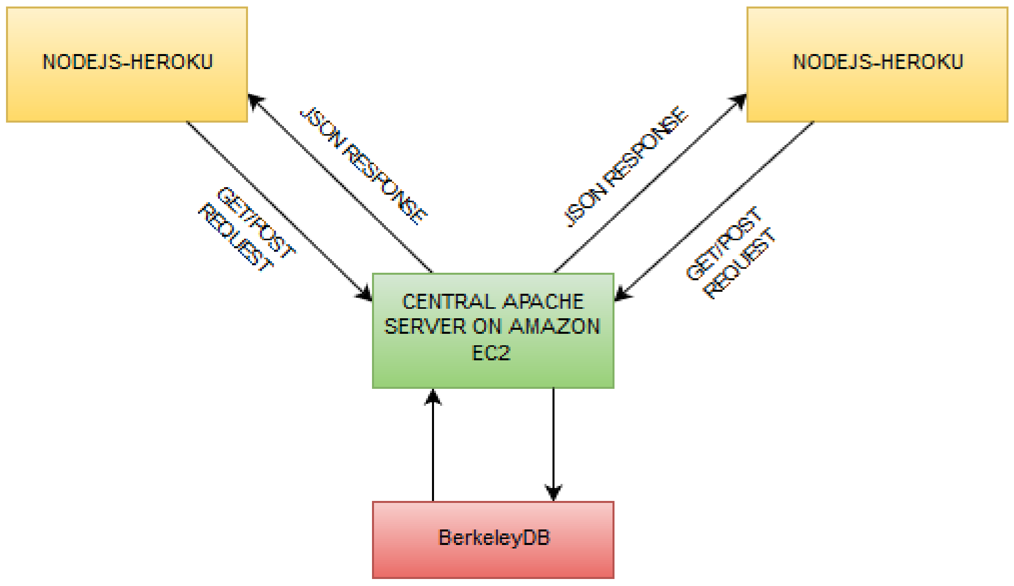


Figure 1: Front End Architecture

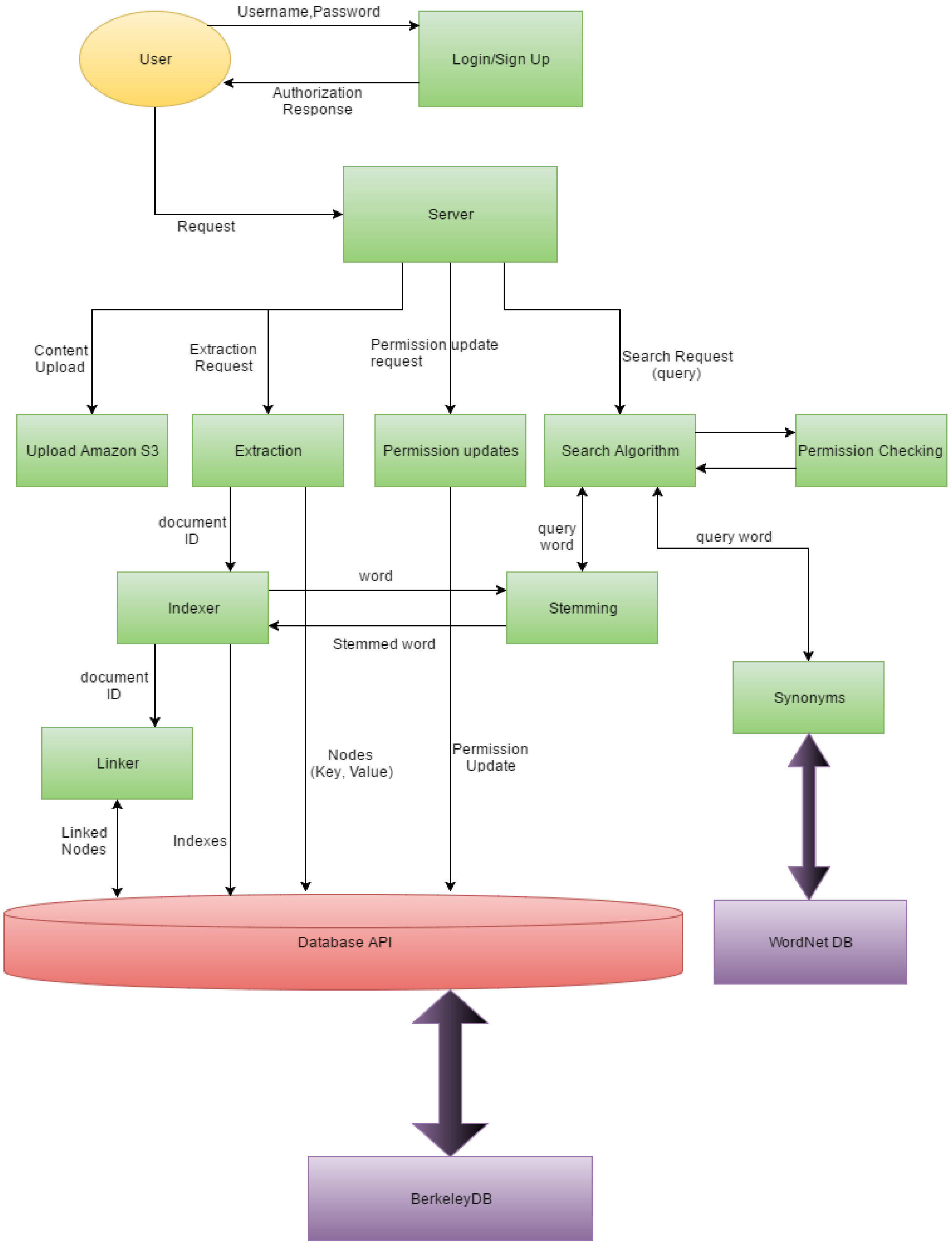
Servlets

As mentioned above, we used Java Servlets on the server side to receive different requests from the clients. We hosted the server on Amazon EC2 using jetty servlet container. The different request, process calls & response types handled by the server are shown below:

Request (GET/POST)	**Process Called**	**Response (JSON)**
Upload File	Extractor->Indexer->Linker	Success/Error report
Assign Permission	Update Database	Success/Error report
Documents List of a User	Query Database	List of <Doc ID, Doc Name>
All Users	Query Database	List of Users
Search for query words	Run Search Engine	List of <List of <Linked Paths>>

Table 1. Request and Response Types

Architecture



Data Extraction

We are supporting 4 types of data formats in our datalake management system, to collect information from these data sources of different types into a uniform format, we parse all these files to extract a graph structure using various parsers as shown in Table 2. The structure of our graph node is:

```
public class Struct {
      private int id;
      private int documentID;
      private String name;
      private String type;
      private String value;
      private int parent;
      private ArrayList<Integer> children = new ArrayList<Integer>();
}
```

While extracting the graph, we are always preserving the hierarchy structure of the data. As can be seen from the structure of our node, each node has a parent and list of children, which refers to the node ids. Each node also contains a key (`name`) and `value` to store any key value relationship in data, as in case of xml, json and csv files. Each node also maintains information about it type and which document it belongs using `type` and `documentID` respectively.

The package `edu.upenn.cis550.extractor` in our project implements the extraction logic. The nodes are pushed into the database using our `storageAPI` in key-value pairs <*node id, Node Object*> resulting in a graph, a uniform data structure of all supported data types, which can be queried later to retrieve information.

File format	Parser Used
Plain text or PDF	TIKA + Stanford PTB Tokenize
XML	Java API for XML Processing (JAXP)
JSON	Jackson JSON Processor
CSV	Apache Commons CSV

Table 2. Data Formats and Parsers

Data Indexing

After the extraction of graph for a particular document is complete, we maintain two index on our data, first forward index, which is stored as key-value pairs in database <documentID, List<nodeIDs>>, second inverted index, which is also stored as key-value pairs in database <node.name, List<Node Occurrence in either name or value>>. These two index are particularly helpful in linking similar nodes and querying over the data. All the strings for each node's key(name) and value are converted to lowercase and stemmed using Porter Stemmer before pushing into the database.

Graph Linking

The linking algorithm starts when a single document ID is provided to it. This way it can be called for a batch of data as well as it can be called on the fly for a single document ID. It also requires the object of the `storageAPI` to be passed along with to be maintain ease of access from the calling function. If no object is passed, then it creates a new one for its use.

When it receives a document ID, it hits the forward index and extracts all nodes from the database in the form of `List<GraphNode>`. For each of these nodes, it extracts `name` and `value` and hits the inverted index to get all linked nodes. It then removes all nodes from this dataset that have the same document ID. This way we ensure that we do not link nodes that are in the same document.

Search Engine

The search algorithm is called from the GUI and it takes in 2 parameters : the userID and the search query. We first check if the search query is single term or two term (any more is not allowed) Both of these are handled in different ways as explained below.

For a single term query, we extract the search term and lowercase it. Then, we check if Spell Checker is enabled and If so, we get the corrected word. We perform our search on these two words and if synonyms/stemming is enabled then we extract all the synonyms/stem all these words. Finally, for the single term search query, we get a list of words that we need to search upon. Hence, we query the inverted index and get all the related nodes (which include its parent and all its children) and check if the user has access to the nodes that we extracted. We remove the nodes on which the user does not have permissions upon. We add all these nodes to a list (maintaining order) and get all the linked nodes of this set of nodes. We check for permissions and update our list of nodes with these. Note these auxiliary nodes are in the list after the the directly matched nodes. For each of these nodes, we get the root of the document and get path from root to this node which is then returned to the servlet.

For two term query, we extract the two search terms, lowercase them and do a spell check which returns us with a most one corrected word each. We then find synonyms and perform stemming. At this point, we have a list of strings related to first and second search term. Of these two lists we generate their corresponding list of node IDs by hitting the inverted index (and also add its parent and all its children to the list). Once the two lists of node IDs are generated we remove from them the node IDs that the user does not have access to. At this point, we start from each node ID from the first list and do a Depth First Search until a certain depth (configurable value `DEPTH`) or if we find one of the nodes in the second list. Once all such paths have been generated, we sort all of these on their length, i.e. the shorter the path, the higher rank it has.

In both types of queries, we return `List<List<Integer>>` to the servlet.

Database

In the project 3 different databases as listed below were used for very different purposes:

- Amazon S3: Reliable database for storing and accessing large data files on cloud
- MongoDB: For storing usernames and encrypted passwords as key value pairs for fast retrieval.
- BerkeleyDB: We used Berkeley DB as our primary storage for our data lake management system. The most important reason behind using Berkeley DB was fast concurrent update and response. The complete Berkeley DB interface was wrapped into a Storage API to ease the process of storing and retrieving data. Following are all the tables stored in the database: -

Table Name (Entity)	Primary Key	Other Fields
Graph Node	nodeID	docID, name, type, value, parent, List<children>
Forward Index	docID	List <node ID> (all nodes in a document)
Inverted Index	word	List <node ID> (all nodes with name/value as word)
Document	docID	docName, docType, List <users>
User	userName	List <doc ID> (all docs user has permission to)
Linked Nodes	nodeID	List <node ID> (all nodes linked with the key node)
Stemmed Index	stemmed word	List <node ID> (all nodes with name/value as word)
Data Log	“log”	lastNodeID, lastDocID (persistent map)

Table 3: Tables in BerkleyDB

Special Features

Ranking

Ranking is done separately for single word and two-word query. For a single word query, the direct matches are presented the first and all auxiliary nodes are presented next. For a two-word query, the shorter the path the higher precedence it is given.

Stemmer

For Stemming, we are using Porter Stemming Algorithm (CITATION: The Porter Stemming Algorithm <http://tartarus.org/martin/PorterStemmer/>). Stemming is useful while querying the data to retrieve all results related to a particular word query.

Synonyms

For Synonyms, we are using WordNet (CITATION: Princeton University "About WordNet." WordNet. Princeton University. 2010. <http://wordnet.princeton.edu>). This is an extensive library of synonyms of words in the form of synsets. We are extracting words from each possible synset of a given word.

Spell Check

We picked up the spell checker from the website http://norvig.com/spell-correct.html. It uses naive bayes to determine the best spelling for a given word. It has its own corpus of a million words to test upon. It finds out *c* for `argmax`$_c$ `P(c|w)` (For all values of *c*, find the once the maxes out Prob(corrected word given original word))

Clickable GUI

Whenever a user uploads a file, it is added to their list of files which we display in the `AllFiles` tab. The user can download the file directly from the application.

Linking On-The-Fly

The Linker code has a function call that takes in a `StorageAPI` object and document ID. This way the linker can be called as and when we need it, providing us the much needed flexibility to not have a corpus already in the system and create it manually by users uploading files.

Conclusion and Performance

Analyzing our performance on different datasets was the key feature that we employed for deciding algorithms to use in `Indexer`, `Linker` and `Search`. Looking at initial results we decided that Berkeley DB would be the best fit for our application. Berkeley has extremely good concurrent update handling which was necessary for our application. The code was threaded in many places so that we save on time since the number of nodes to extract, index and link were way too many.

On our initial code, uploading and linking 50 resumes (6mb data) took around 2 hours which was much more than we could afford. We tweaked around our code and made use of concurrent updates in Berkeley which reduced our time to 20min. At this point, 40mb of data took just over 2 hours to get indexed and linked.

Primary challenge we faced was how to make search faster in case it returned too many links. To counteract this issue, we utilized the inverted index, implemented modified Depth First Search and limited our results to take only paths of a certain configurable length. At the end, we managed to cut down the search time for around ten files to under 10 seconds.

As more and more files are added with similar search terms and with synonyms enabled the number of possible paths and time grows exponentially. Perhaps using an in memory distributed environment like Spark would make our algorithm much faster since we do not have to extract data every time from the DB and waste time in I/O and network latency.

The amount of modifications and tweaking possible are a lot out of which we were just able to implement a few to our data lake management system run fast.

Division of Labor

Team Member	Participation
Mani Mahesh	Web Development :- Node JS, AWS S3
Sajal Marwaha	Linking, Searching, SpellChecker
Jitesh Gupta	Database Storage, Integration
Sanidhya Tiwari	Extraction, Stemmer, Synonyms

Appendix and Search Results

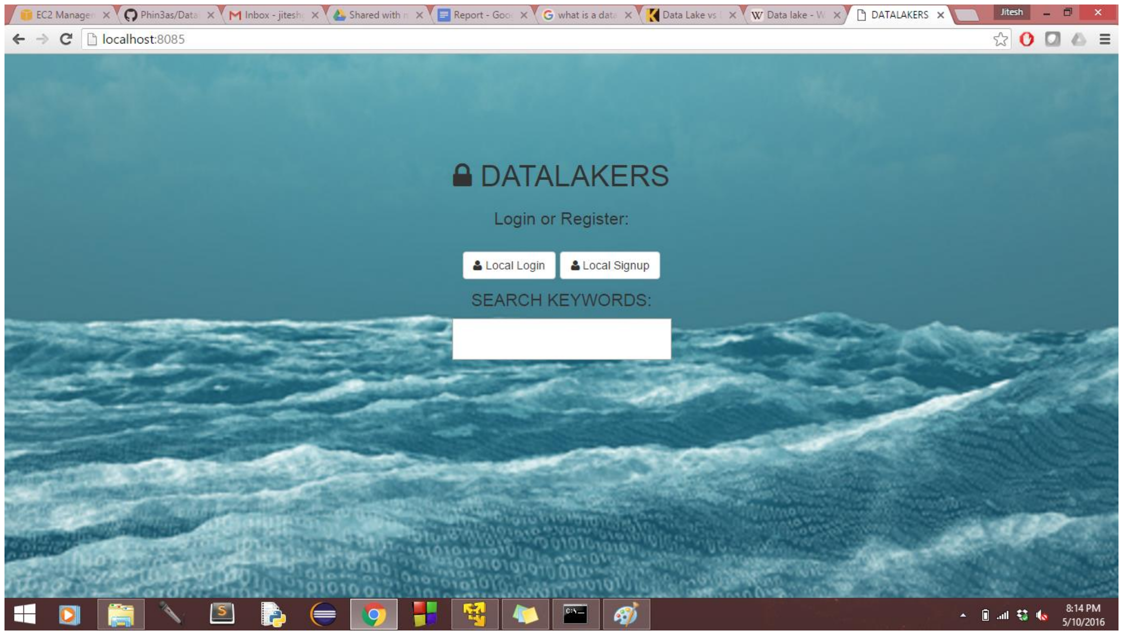
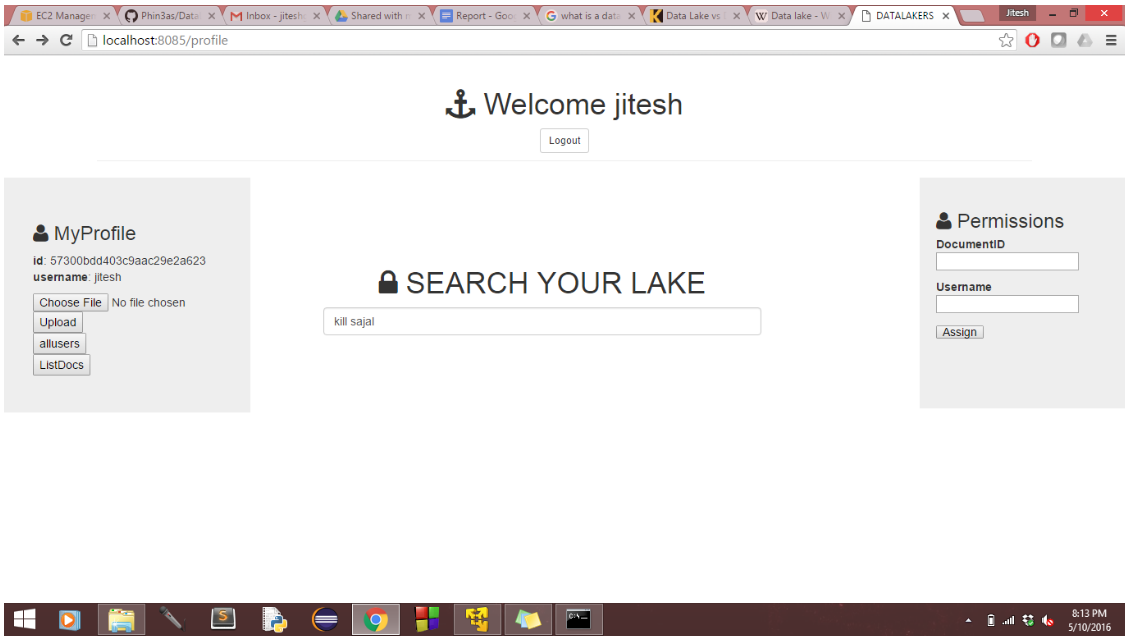


Figure 1: Home Page of our application



id: 57300bdd403c9aac29e2a623
username: jitesh
kill sajal

Figure 2: Login Home for user "Jitesh"

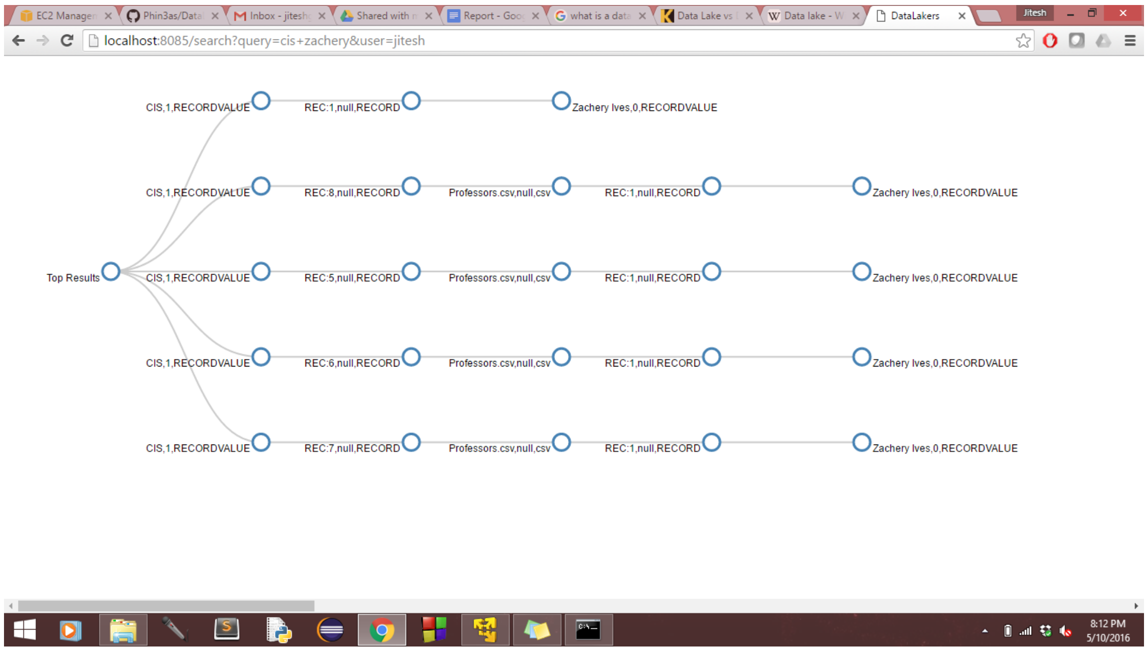


Figure 3: Search Result for query “CIS Zachery”

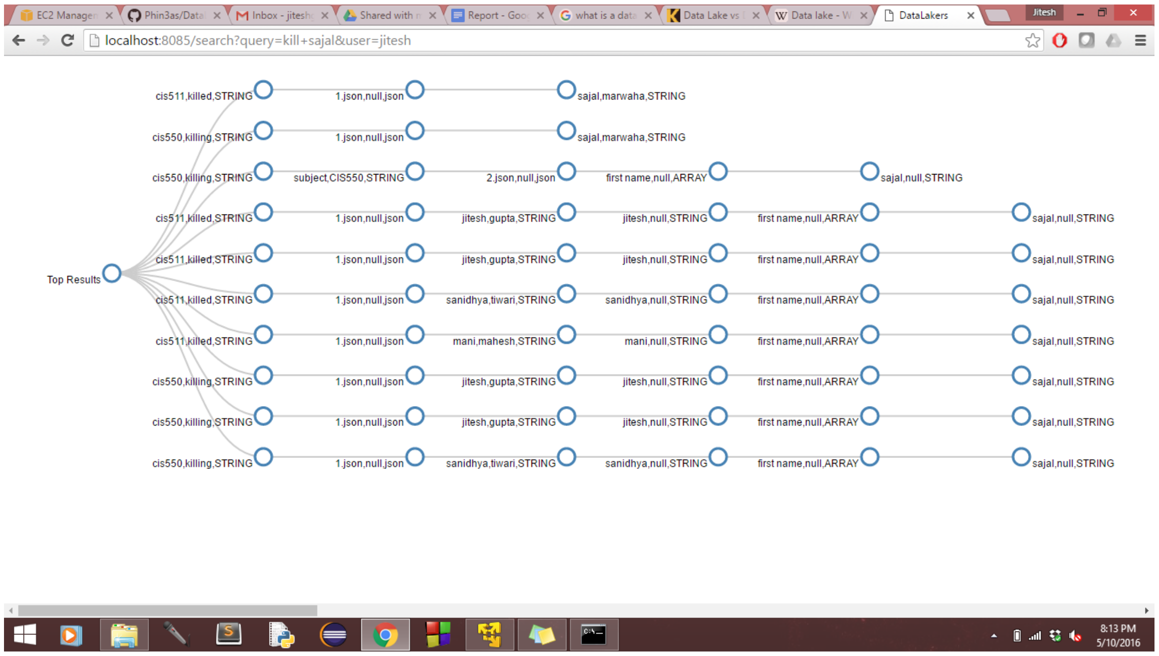


Figure 4: Search Result for Query “Kill Sajal” with Sysnonyms