

Team Nessie Writeup

Contributors:

Alex Harellick

Anjali Khetan

Joey Raso

Max Tromanhauser

Code Location: <https://github.com/aharellick/cis550-project>. The code used in our demo can be found in the *nessie-webapp* folder of the *deployed* branch (master is not up to date).

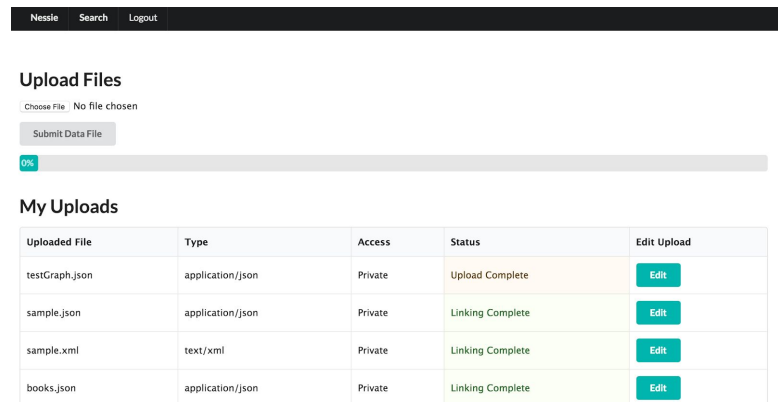
Introduction

The goal of this project was to build a database that stored information from various file types. Users can upload .xml, .csv, or .json files, and then search the database as well.

Architecture and Roadmap

User Interface:

Our user interface was a webapp built with Node.js and Express. We made use of the Semantic UI library in order to style our user interface. The interface consisted of a login screen, a search screen, and an upload management screen. From the upload management screen, a user could upload files stored on their computer, and track the progress of the upload via a progress bar. The interface provided a clean visualization of a user's uploads that included the upload name, permissions, content type, and upload status. The search screen was made up of a search bar and a <canvas> element that the search results were rendered on.



The screenshot shows the Nessie web application interface. At the top is a navigation bar with links for 'Nessie', 'Search', and 'Logout'. Below this is the 'Upload Files' section, which includes a 'Choose File' button (currently showing 'No file chosen') and a 'Submit Data File' button. A progress bar is visible below these buttons, showing 0% completion. Below the upload section is the 'My Uploads' section, which contains a table listing uploaded files.

Uploaded File	Type	Access	Status	Edit Upload
testGraph.json	application/json	Private	Upload Complete	Edit
sample.json	application/json	Private	Linking Complete	Edit
sample.xml	text/xml	Private	Linking Complete	Edit
books.json	application/json	Private	Linking Complete	Edit

Document Parsers:

Our system was able to support three different document types; xml, json, and csv. We decided that the best route to take with our parsers was to first convert all input documents into json objects. That way we could parse through all documents with the same code. In order to accomplish this we used two third party libraries, the first being an xml to json parser called xml2js, and the second being a csv to json parser called csvtojson. If we were given a json file as

an input, then we could trivially parse it to a json object using the JSON library that is included with Node. Once we had the document as a json file, we had a single method that would parse through the object and store it as an in memory graph that we would later persist to our database.

Document Linker:

The linker simply looked for exact value matches between nodes. In the end, we found that our linker became somewhat of a bottleneck on the system because many nodes contained exact matches on stop words such as 'the' or 'a'. Thus, we ended up with an extremely dense graph that took quite a bit of time to process when inserting new nodes or performing a search. In the future, we could remedy this situation by ignoring stop words and keeping track of word frequencies to only match on uncommon values. Some other options that could have made our searches more interesting would have been stemming or making use of n-grams. However, these approximate matching methods again would have increased the density of the graph, which we were trying to avoid.

Document Searcher:

The document searcher (contained within `routes/search.js`) exports a single function that takes a query, and produces a list of paths. It begins by splitting the search query into terms and finding all nodes in the inverted index that are associated with the term. Once it has a collection of terms and their nodes, it produces the Cartesian product pairs, so every node has a pair with every other node *not* associated with the same search term. Each pair then has a limited search to find all possible minimal paths between them. Only the k shortest paths found will be returned, where we set k to be five for display purposes.

Results Display:

Once the document searcher executes and returns the best shortest paths, the resulting nodes and paths are displayed to the user. We used Sigma.js to aid in the graph visualization. The results are passed through a JSON translator that reads in paths containing our db nodes, and creates nodes and edges formatted to comply with the Sigma API.

Implementation details

Two different implementations for this project were executed after discovering that our first resulted in some resolute obstacles. The first attempt was a Node app with Java code that was deployed on AWS Lambda. We made use of MongoDB and the Mongoose ODM library for all of the storage related to user profiles, S3 to store all of our raw files, and DynamoDB to store the graph structures that we would search on. The Node app handled uploading items to S3,

keeping track user profile details, and managing your past uploads. The code that we deployed to AWS Lambda was triggered by an upload hook on S3 and would execute anytime a new file was uploaded. This Java code parsed the incoming data file, created an in memory graph out of it, executed the linker on these nodes, and finally persisted this data to a DynamoDB database. The biggest obstacle we faced in this attempt was scalability. While using Lambda allowed us to instantaneously spin up workers to handle file uploads, each Lambda function was still responsible for parsing and linking an entire file (which could be quite large in size). Uploading anything larger than a few kilobytes to our dense DynamoDB tables took a large amount of time due to throughput constraints and made the app extremely slow. We attempted to raise the throughput on our tables, and while this did slightly improve our situation, the app's response time was not demo-ready. In the end we decided to try and turn away from our DynamoDB/Lambda stack.

We felt pretty confident that the large number of query requests over the network was the bottleneck in our previous approach. To remedy this we decided to port all of our java code to native javascript and move from DynamoDB to a local MongoDB instance that could be quickly accessed from our Express application. However, after rewriting our parsing/linking/searching logic and making sure we were employing Node's asynchronous event-driven model to eek out all of the performance that we could, we still found that we were having trouble scaling. As a final attempt, we utilized a priority job queue that communicated over a hosted Redis instance to distribute the heavy lifting of our linking process. We were able to greatly improve the speed by splitting the input file into multiple sections and having each worker only handle a portion of an uploaded data item. We deployed the web application to heroku which gives us the flexibility to scale our worker pool size easily to meet all of our theoretical demand.

Validation of effectiveness

There were a few different things that we checked in order to determine that our search was performing as desired. We were interested in single-word queries, multi-word queries, and cross-document links. In order to understand whether or not our single-word queries were working, we tested searching for words that occurred once, and words that occurred multiple times within a single document. The document we tested with, books.json, had information about several books including title, author, and year and the node structure for the document can be seen in Figure 1. Before searching for the term "Hobbit", we hypothesized that we should get a path from Hobbit to the root of the file, books.json, as displayed in Figure 2.

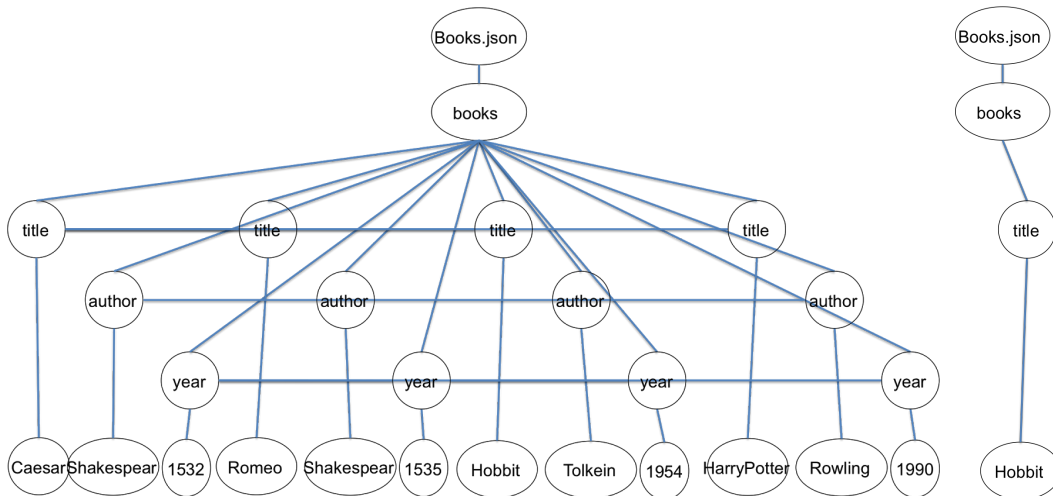


Figure 1

Figure 2

The output was as expected for this specific single-word, single-file query and several others that we tried. Once we felt comfortable with that execution, we moved on to testing multi-word queries within a single file. Using the same file, we searched for “Hobbit Shakespear” expecting that the two terms would be connected through the “books” node, as in Figure 3. That is what ended up being returned so we were satisfied with that test case after trying a couple other pairings.

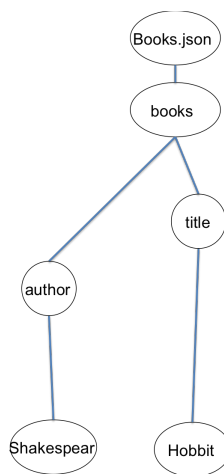


Figure 3.

After that we ensured that the linker and searcher would allow for finding connections between nodes between documents. By uploading an .xml document with a title tag alongside books.json, we would expect that the ‘title’s in both would be linked by the linker. By using the searcher, we found that by searching terms in book.json with the sample.xml, we could find a path through the linker’s ‘title’ link. This confirmed that not only did the linker successfully find meaningful links between documents, but that the searcher successfully searches over these edges.