CIS 550, Spring 2016

Alex Fiuk, Josh Kessler, Kelley Loder, Ryan Smith

Datalake Management System

Team "I did not have relations with that schema"

Introduction

Many data management systems only support search over its contents using specific syntax, and they return results in a predictable manner. We built a data management tool that supports more free-form searches, which can be more user-friendly and can also reveal connections among data that people might not think to look for. Each data file is parsed into a tree structure, and these disjoint trees are then linked based on similarities in keywords within the trees. Our system allows users to search for connections among data sources and manage their data. Users can register an account to use the system, upload data files, set permission levels for files, and perform keyword searches across data items. The results of a search are displayed as a series of nodes representing pieces of a data file and how they're connected.

The challenges in building such a system include that the system is meant to be robust enough to handle different file types with no advanced knowledge of the structure of the data they contain; datasets can be vast, and finding connections, especially meaningful ones, becomes an exponentially larger task as the corpus of documents stored in our system grows; and traversing a path through those connections to return results for different search terms also becomes more challenging as the number of documents grows. In the rest of this report, we discuss how we addressed these challenges.

Architecture

Shared Tables

We use several tables in a MySQL database deployed on Amazon RDS to coordinate among the different pieces of the datalake management system. These tables are as follows:

Document Table: Holds data on each document, including a doc ID, doc name (which also serves as a key to access the doc from S3 storage), foreign key to the owner of the doc and permission setting.

User Table: This holds first name, last name, username, password and permission setting info for a basic site user. There are currently 2 permission groups: standard or elevated. Elevated includes site administration or those that have been given additional access to protected documents (those marked elevated, private docs are always confidential).

Inverted Index Table: Holds data associating individual words with all the places within files where they're found.

Node Table: Holds data for each node in the tree structure representing each file, including node ID, parent node ID, doc ID, and strings for each node's key and value.

Edge table: Holds data representing edges between nodes, including the doc IDs of the edges being connected and the type of edge (for example, an edge within a tree structure or an edge between two different files).

Front end

A web interface was created to provide a straightforward, visual tool to allow users to access the functionality of the datalake. Through the webpage, users can upload documents, perform searches, and adjudicate permissions. This is based off the Spring MVC and uses all tables present in the MySQL database. It uses S3 as a document store and ActiveMQ for transmitting messages between Web and Extractor processes.

Search Engine

The search engine is packaged with the web MVC. It operates as a separate standalone tool which, with trivial changes, could be run from a main method if separated from the website code. It produces results for a search using search terms and User object (to check permissions) and returns results and paths in JSON format.

Extractor

When the extractor program is invoked, it determines the file type and parses the file into a tree structure. As it traverses this tree structure, it stores information in the node table, the edge table, and the word table.

Linker

The linker exists as a standalone Java program that is invoked after the extractor. It retrieves the extracted data from the database (mainly, the words, nodes, and inverted index tables), runs a linking algorithm to identify and create new edges between nodes in different document trees, and uploads the newly discovered edges to the edge table.

Implementation

Front End

Uploading Documents

Users can create password-protected accounts via the website. Once logged in, they have the option of uploading data sources to the data management system that will be parsed and included in

searches. The extractor executes as a separate process from the web MVC and is linked via a JMS instance using ActiveMQ. When a document is uploaded, the web back end identifies and saves metadata pertaining to the file and then uploads the file to a protected S3 bucket. Next, an entry is created for the file in the *Document* table of the database and permissions are set to the general setting. Once these steps are completed successfully, the web MVC submits a message to the queue that a document is ready to have the extractor run on it. This message includes all needed info to identify, fetch and begin working on the new document.

Setting Permissions Over Data

Permissions for each piece of data are set directly on a document's entry in the *Document* table. Each entry in the table includes a foreign key to a single user who is designated the "owner" of the data. Only this user can alter the permissions for the given document and can only do so while logged in. The site uses sessions to make the login experience as easy as possible (i.e., to prevent the need for multiple authorizations, a session cookie is returned identifying the user as authenticated).

There are three permissions settings: 'A', 'E', and 'P'. 'A' stands for *all* and allows all users to access the data. 'E' stands for *elevated*. This setting in its current state pertains to a global group of site administrators but in future iterations could be set to pertain to working groups relevant to the document. For example, if a user is a member of The Penn Museum X Department Group, the elevated setting may allow them to upload documents to be shared with the group with which they are affiliated without submitting them for general public consumption. 'P', as one might guess, stands for *private* and restricts the document to the user who uploaded it.

Permissions can be altered in the user's account page. Permission changes take effect immediately, and will be shown in page refreshes of searches completed before the permission change.

Downloading Documents

All documents are available for download. A simple URL syntax has been created corresponding to "<website url>/downloadDoc/{doc_id}" so that any user familiar with a document's assigned ID can download the document directly. Links to download documents are included as dropdown buttons following each search. Documents to which a user doesn't have permission will not be included in that user's search results, so downloads are obviously also restricted to authorized users.

Search Engine

The search engine performs allows for two distinct operations: single-keyword search and two-keyword search.

In single-keyword search, the user enters a one-word search term. When single-keyword search is performed, the search engine uses the inverted index table to identify all nodes that contain the requested keyword. It fetches those nodes and, using the associated document IDs, all other nodes included in the corresponding document trees. Next, it fetches all edges pertaining to the selected documents but ignoring those edges generated by the linker because, for single-keyword search, we do not wish to see paths that cross from one document to another. It then rebuilds the tree for each document and identifies the root node. Starting from the root, the search engine performs a BFS

search until the identified target node is encountered. Finally, it displays the path from root to target node for each document identified as having a matching node.

Two-word search begins by identifying the nodes containing each keyword by using the inverted index. It then identifies the corresponding documents and fetches all nodes and edges pertaining to these documents (this time, linker edges are included). Next the engine builds a graph and represents it as an adjacency matrix. Because each edge in the edge table has a node_1 and a node_2, while edges added by the linker might have no inherent direction, using just the edge table would give us a directed graph. Because we want to be able to walk in both directions across any edge, each edge's inverse is added to the adjacency matrix as well (e.g. if an entry with node_1 A and node_2 B exists in the edge table, add edge A->B and B->A to the adjacency matrix). The engine then identifies all starting nodes and all target nodes. It loops through each starting node and begins a BFS search from that point until it identifies any of the target nodes. It will continue until it has either a) collected as many paths as there are target nodes or b) exceeded a depth of 25 nodes in a given path. In this way, it is able to limit the number of paths that will be walked and avoid exceedingly long paths where connections between data become potentially less meaningful. A final additional constraint we added was to limit the number of nodes examined in BFS to 25,000--in this way, we still retrieved a large number of search results, but for searches that involved paths with very common search terms, the searches terminated in a reasonable amount of time.

Permissions are used in both types of searches. After identifying relevant documents, permissions are used to verify which documents the user has access to before fetching the nodes/edges from the database and building the graph.

Extractor

When the extractor receives a message from the ActiveMQ message queue that a document is ready to be parsed, it passes in a single string as the argument to the extractor's main method. We parse this string to get the document ID (as associated with the file in the Document table), the document name, and the path to the file in S3. The extractor determines which of several parser subclasses to use based on the file extension, uses the file path to access the file, and passes it to the appropriate parser. We used Commons CSV for CSV files, JAXB for XML files, and Jackson for JSON files. We did not end up building out support for other file types, but we did include a class for an Apache Tika parser, and it would be simple to pass all other file types to that parser.

Depending on the document type, we use placeholder nodes to give each document a tree structure. These placeholder nodes may have a key but no value, or neither a key nor a value. For CSVs, for example, we create a root node representing the entire document (with no key or value), a node representing each row in the CSV with no key or value, and for each row node, child nodes for each cell in the row, with keys corresponding to the column headers and values corresponding to the values in the row.

As we parsed each file, we queried the highest node ID in the node table, then had the parser increm ID of the next node to be added each time we added a node. We did this rather than using an auto-incrementing node ID because the parsing was recursive, and nodes had to know the IDs of their parents. We added nodes, edges, and inverted index entries to separate lists, and when a list's

capacity reached 500, we extracted the contents of each node and, using Prepared Statements, sent batch transactions to the corresponding table. At the end of parsing each file, we sent batches with any remaining entries in each list to the database, as well.

As we added each node to the node list, we updated the inverted index, as well. Though keys and values in files can have multiple words, we split the keys and values into individual words and stored those in the inverted index. To try to keep data as meaningful as possible, we didn't add words shorter than three characters; we didn't add strings that were purely numeric; and we didn't include words that appeared in a small list of stopwords. Additionally, we converted all characters to lowercase. For example, a node with the key "This is a unique key" would have entries in the inverted index associating the word "unique" and the word "key" with the node ("this" was a stopword).

Linker

The implementation of the linking pipeline can be thought of as three main components:

1. Pull/push interactions with the database
2. Representing/storing (in-memory) tuples as objects
3. Linking algorithms

The Driver class has a single method – drive() – which manages the linking pipeline from start to end.

Pull/push interactions with the database

The Database class is responsible for managing interactions with the MySQL database. It has static methods that are used to initiate the data pull at the beginning of the pipeline as well push the newly discovered links at the end. A single shared connection is created and separate private helper methods handle retrieving tuples from each table.

Representing/storing tuples as objects

As tuples are read in, they are translated into objects for easy manipulation by the linking algorithms. We create sets of objects of type Node, Word, IndexEntry, and Document. We use a Dataset class to store these sets for use throughout the program. As links are identified, we store the edges as a set of LinkedEdge objects, which are later passed to the Database pushLinks method to update the MySQL database.

Linking Algorithms

This is the core component of the linking pipeline and consists of the following classes:

1. Linker - an abstract base class with...
    a. A single abstract method – link() – to be implemented by subclasses
    b. A static inner class – StopWordFilter – which is used to dynamically prune the word set by identifying and removing the most frequently appearing words from consideration during the linking process. This is done to ensure that the links we create are more meaningful, which leads to 1) more efficient searching, since the BFS

will not be inundated with superfluous edges to consider, and 2) more meaningful search results.
    c. Several helper methods used by subclasses
2. SimpleLinker - the first of three implementations of Linker
    a. This is the most basic Linker implementation. After filtering out stop-words, it creates links between nodes where any word matches, regardless of whether it appears in the key or value.
3. DocumentNameLinker - the second of three implementations of Linker
    a. This implementation focuses on creating links in the case where one node's key or value contains the name of one of the other documents. The node containing the document name is linked to each node within that document.
4. StrictLinker - the last of three implementations of Linker
    a. This implementation was created as an alternative to SimpleLinker with the aim of achieving even more meaningful links. Here, we only create links when there is an exact string match (ignoring whitespace).
5. LinkerFactory - factory class used to create the desired Linker object
6. LinkerType - an enum used in Linker object creation
7. InvertedIndex - data utility class
    a. This class helps to enable efficient linking. It holds a subset of the dataset, several hashmaps for efficient lookup, and methods that provide the linkers with quick and easy access to the data needed throughout the linking process.
    b. This class also ensures data integrity by minimizing/controlling the amount of calls needed to the Dataset class (where the entire dataset lives)

Validation

Front end

Validation for the front end was largely anecdotal; the provided views were requested and their display and status codes were verified. Logging on the server was used to make sure all was in order.

Search Engine

Search engine results were validated using unit tests. A separate test MySQL database was created and individual document trees and links were inserted node by node. The same results could have been produced using mock objects but, in the interest of time, the outside database served our purposes for extensive testing. This testing included:

1. Single word search for word appearing in multiple nodes in multiple documents.
2. Single word search for word appearing in single node in multiple documents.
3. Single word search for word appearing in multiple nodes in single document.
4. Single word search for word appearing in single node in single document.
5. Single word search for word appearing at a depth of greater than 25.
6. Two word search - both words appearing in single document.

7. Two word search - words appearing in separate documents.
8. Two word search - words appearing in both single document and separate documents.
9. Two word search with result paths greater than 25 in length.

Testing was also conducted with live data. Due to the size of the data, this testing pertained to verifying expected links were generated between keywords and that searches produced results for words we knew to be included in the data.

Extractor

We created sample CSV, XML, and JSON files, each representing the same data. The files were small enough that we could manually determine the expected tree structure. We created test instances of our tables that had the same schemas as the live versions and parsed documents. We then printed the contents of the table and confirmed that the expected values were in each table.

Initially, we also noticed that parsing files was unacceptably slow. By printing the time elapsed before and after each method call, we determined that our database calls were the bottleneck. Though we'd written our code to use batch statements, with some research, we learned that MySQL doesn't support batches and, instead, uses extended inserts. We learned we could enable rewritable batch statements in the ConnectorJ MySQL JDBC driver by appending rewriteBatchedStatements=true to our connection URL. We toyed with different batch sizes and didn't see consistent differences, so ultimately, we decided to use batches of 500--this batch size gave us acceptable performance but also allowed us enough granularity to determine roughly when problems occurred if we got SQL exceptions.

Linker

While testing and optimizing the code, we were surprised to find that data retrieval, object creation, and the linking algorithm all executed extremely quickly (around 10-20 seconds even when row counts and links created were on the order of >100k). However, the pushing of links was considerably slower (several minutes to upload the links for that same dataset). We optimized by using prepared statements and batch uploading, and found that this helped improve upload times.

Given that our actual dataset was on the order of hundreds of thousands of entries in each table, testing of the correctness of linking algorithms was done on smaller mock datasets where we could see whether links were being created correctly. Our testing focused on ensuring that:

1. Strings were being matched as expected, depending on which linking algorithm was being used
2. Links were not created between nodes within the same document tree, since this would lead to redundant edges and potential cycles
3. Stop-words were correctly filtered out and did not show up in links

While testing, we found that linking on any word match was producing a relatively large number of links. This inspired us to implement StrictLinker, which would limit the number of links and produce more meaningful results.

Time permitting, our goal was to take linking/searching even further by adding edge weights to our links based on how strongly linked the nodes were. Our plan was to match on n-grams to determine these weights. Unfortunately, we ran out of time.

Lessons Learned

With our current setup, because of how we increment node IDs, we can only parse files sequentially. However, the web MVC is currently configured to handle file upload requests asynchronously and, through the use of the ActiveMQ message queue, as many Extractor consumers as desired could be added. Therefore, in a future iteration, we might do something such as make node IDs include doc IDs, a delimiter, and then an incrementing number. This would be a simple change to implement and would greatly increase our capacity to handle large data sources from multiple users efficiently.

We set a limit on the depth of returned results from the search engine to 25. We thought this would both make the returned results more relevant and prevent long delays in processing searches that drew upon large files. Through testing, we found that this maximum depth was more relevant to documents that contained significant nesting (some JSONs we used) and was less useful with very wide and shallow documents (some CSVs we used). In the latter case, large documents can still generate an exorbitant number of paths to be evaluated without crossing the 25 node threshold. In a future version, this would be an interesting avenue to explore. It might be possible to probe the shape of the tree by running investigatory BFS and DFS and then adjusting our limits to match the shape and structure.

Similarly, we only explored 25,000 nodes at a time during our search. We imposed this limit when we saw that some searches seemed to be caught in an infinite loop; it turned out that, when searches included nodes with very common words, there were just too many paths to explore in a reasonable amount of time (even if they were short paths). We tested a few different numbers and found that 25,000 gave us a satisfactory balance between returning results quickly and showing enough results. In the future, we might consider pagination as an alternative to a somewhat arbitrary limit.

With structured data, there are many repeated keys. On the one hand, these data points can be useful--for example, if we have data about budgets in different cities, a “department” column header might be useful for linking two different data sources. If there are thousands of entries, the word “department” will appear thousands of times, making the job of linking and finding paths more difficult and less meaningful. Because CSV files are parsed into shallow trees, one option would be to associate all the words in each of the keys with the root node. For JSON and XML data, though, it is difficult to tell whether keys will be repeated. One other idea we considered was storing the entire string in a key or value in the inverted index to support more complex searches.

Contributions

Josh and Kelley worked on the extractor. Alex worked on the linker. Ryan worked on the front end and the search engine.