CIS 550 Group Project

Fall 2016

Team and Intermediate Checkpoint due 4/9

Final Demonstration and report due during Finals Week

Introduction

In recent years, due to the "big data revolution," there has been significant interest in pooling together data resources (tables, CSV files, XML, JSON, etc.) into large collections of partly structured data – so-called data lakes. The vision is that one can use MapReduce to read these files, possibly convert them into standard formats, and/or analyze them.

https://en.wikipedia.org/wiki/Data_lake

Note especially the criticism! For the course project this year, we will be building a *data lake management system* (DLMS) – with the goals of (1) storing files and their extracted fields and subsets, (2) allowing for links among these fields, and (3) allowing for search across the fields and links. You will experiment with *structured search* and do things that Google can't, you'll know what can be done with a data lake, and you'll become familiar with open Web data.

For some example data, see:

- http://data.philly.com/
- https://nycopendata.socrata.com/
- http://www.zillow.com/research/data/
- https://www.census.gov/data.html
- https://www.wikidata.org/wiki/Wikidata:Main_Page
- http://wiki.dbpedia.org/

Though you are free to choose alternative data. Additionally, assume some of your data may be PDFs, Excel sheets, Word documents, etc.

Requirements - Data Lake Management System

Your DLMS will consist of several components:

1. A basic **back-end** with
    a. **user login** information (including username, password, and other details)
    b. A catalog of raw **data items** (likely stored in a key/value store or filesystem)
    c. **user permissions over data** (who is the data item owner and who is authorized to view the data items)
    d. Interfaces for creating users and data items, assigning permissions, etc.
    e. The ability to define edge-style links between data items (see below)
2. Web-based **account management and administrative services** to create accounts, see lists of "owned" data items, and assign permissions to others.

3. A mechanism for pointing the DLMS at a new file, then invoking one of set of content *extractors* (e.g., the Jackson JSON parser, the Tika reader for many file formats) that read the contents of the raw data items, interpret the file format, and output key/value pairs along with information about where in the file these key/value pairs were found.
    a. You will likely want at least the following libraries (the extractors) for extracting keys and values from (partly) structured files:
        i. https://tika.apache.org/ reads Word docx, PDF, ... text and headers
        ii. https://github.com/FasterXML/jackson reads JSON
        iii. https://docs.oracle.com/javase/tutorial/jaxp/ reads XML
        iv. https://commons.apache.org/proper/commons-csv/ reads comma separated
    b. The various content extractors should be possible to invoke from a standard interface, such that they are given a pointer to (or path of) a raw data item, and they return a set or list of extracted attribute-value pairs.
    c. This will require a **mapping from file extension to the appropriate content extractor.**
    d. In the simplest case you may assume that the file is already on the filesystem for your DLMS. A more interesting implementation would provide a Web interface where the user could upload the file into a temporary directory. See, e.g., http://howtonode.org/really-simple-file-uploads and http://www.moxiegroup.com/moxieapps/gwt-uploader/
4. An *indexer* that takes the key/value pairs and indexes them by key, and also creates an additional index that breaks the values into constituent keywords, and records the mapping from each keyword (search term) to the attribute containing the term.
    a. This will take the output of the content extractor, and index all key/value pairs.
    b. It will also take each key/value pair and create an **inverted index**, i.e., a table matching from each word to the specific key (and value).
5. A set of *linkers* that try to find links between raw data items, e.g., by linking items that share uncommon keywords, or where a keyword in one data item matches a filename of another data item, or where it appears that a particular field in one piece of content represents a foreign key referencing another field in a different piece of content.
    a. These linkers should have a simple interface that (1) takes pairs of raw data items, (2) iterates over all possible key/value pair combinations (one from each data item) and sees if they overlap in name or in value, (3) stores in the back-end a link between the data items and attributes if such an overlap exists.
    b. You should **at least** link two items if:
        i. You have a parent data item and a nested data item, e.g., the parent is a JSON map and the nested data item is one of its properties.
        ii. One attribute's value is the same as the key attribute of another data item
        iii. One attribute's value is the same as the file or path name of another data item
        iv. Both attributes have the same value, which is the ID of a known entity
        v. You may also want to look at measures of **approximate matches** among strings, including **string edit distance** and **n-grams**.
    c. The result will conceptually be a graph, where data items are nodes and overlapping values result in edges between the nodes. We will want searches to return parts of the graph (specifically trees).

6. A Web-accessible *search engine* that takes multi-keyword queries, matches them using the contents of the index, and returns ranked results that link nodes in the above graph
    a. If the different keyword terms match on different data items in the graph, the searcher should find paths or trees (up to length/diameter *k*) that link those matches.

       E.g., suppose we have two data items: a file F with an field "link" and value "G"; a file G.

       A search result for "F G" would match separate terms on F and G, respectively; then find a path between these of the form F -link-> G.

    b. Each result will be scored in a way that should be proportional to the strength of keyword match, and inversely proportional to the diameter of the tree (# of edges).

An Example

Suppose you are given a file **sample.json**.

[{a1: 'test', a2: 'value', a3: 10}, {a5: {nested: 'one', two: 3}, a6: 7.5}}]

The goal is to extract out all of the useful properties in the JSON, and even to preserve the hierarchy. Recall that we discussed encoding all data in a graph. Here is one approach. Your indexer should create a "node" or "attribute/value pair" for each significant piece of content. Your linker should create links or edges for each. Here we might get:

sample.json: file1

file1.content: map1

map1.content1: list1

map1.content2: list2

list1.content1: map1

map1.a1: test

map1.a2: value

map1.a3: 10

list2.content1: map1

And so on. Here we adopted a scheme where we think of the JSON structure as a tree, and every intermediate node gets a unique ID derived from its parent item (e.g., file1.content, map1.content1).

Edges from the linker would be, e.g., (file1, "contains," file1.content); (list1.content1, "contains," map1).

This isn't the only way of doing things... In fact, if you want to ignore the hierarchy and just include all of the attribute/value pairs from a structured file, that is OK. Note that common filetypes would be

comma-separated values (with multiple rows, each with specific fields), JSON, XML, and text document. You should thus have a scheme that allows for multiple values with the same key or type.

Starting Points / Building Blocks

You have seen a variety of tools and technologies that may be useful:

- **You should use a version control system and platform like bitbucket or github to share and synchronize code.**
- You can use key-value stores like Amazon S3 or BerkeleyDB to store the raw data items (objects).
- You may choose among more structured key/value stores (MongoDB, Redis, Cassandra, HBase, ...) and relational DBMSs for storing the keyword index and the attribute/value pairs
- You will need to link the "back end"(s) to a Web-based user interface (in GWT or Node.js or equivalent) to enable queries from the Web. You have seen both of these in your homework assignments.

We expect the ultimate system to be hosted on Amazon Web Services, such that you can demo the data lake search service from a standard Web browser (Chrome, Safari, Firefox).

Forming a Team and Making Role Assignments

You should have a team of 4 members. As with any real startup, you'll want to make sure there are clearly defined responsibilities. You'll need to break down roles for each member, e.g.:

- One person should define interfaces to the storage system(s), i.e., the key/value stores, relational DBMS, etc.
- One person should handle the content extraction and linking aspects.
- One person should handle the Web application.
- One person should be responsible for the keyword search tasks.

Note that the goal is to have one person lead, and ensure progress, on each component. At times the work may require contributions from several people -- but the "point person" will be responsible, both to the team, and with respect to their project grade.

Intermediate Checkpoint -- due April 9th

By **April 9,** you are required to post in a **Private Post to Piazza**:

1. Your project team **name** and **members**
2. For each member, their **assigned roles**.
3. A link to an active but **private repository** on Bitbucket or Github, with read permissions granted to zackives.

The Demo and Submission

We will circulate (via ScheduleOnce) an online sign-up form. Your team will be expected to give a **15 minute, in person demonstration** with all members present and available for questions. The demo **must be over the platform, hosted on Amazon Web Services EC2 (or equivalent alternative cloud provider).**

All of your team's code must be available, in final form, by the day of the demo. It should be in the repository you shared on April 9th.

You are also expected to include a 5-8 page **project write-up** with an overview of:

- Introduction. The basic problem and its challenges.
- Architecture and road map. The main components.
- Implementation details.
- Validation of effectiveness. You should define an **experiment with measurable outcomes** that validates the hypothesis that your data lake system can effectively find data in a timely fashion. At minimum you should validate that (1) it returns results within a timely fashion and that (2) it returns, for some set of test data, all of the correct answers (according to some objective criterion).
- Team member contributions.

Finally, you will be expected to send a follow-up **private email** to zives@cis.upenn.edu outlining (1) your overall assessment of team dynamics, (2) your own contributions to the team, (3) your assessment of your teammates' contributions. This private email, as well as questions during the demo and the details in the report, will ensure that each team member's grade reflects both the team's level of success and their own contributions.

Extra Credit Opportunities

Extra credit will be considered for any and all of:

1. Additional features, e.g., stemming, use of thesauri, innovative ranking functions.
2. Creative use of novel datasets and applications, particularly those with new kinds of file format readers.
3. Novel algorithmic contributions.
4. Linking to external data sources.
5. Superior validation showing scale-up.