Project Report

CIS 450/550

Shuai Shao, Zimeng Yang,Karan Hiremath, Anthony Janocko

5/11/2016

Introduction

A data lake is a large scale storage repository and processing engine that pools together data in various forms via partly structured data collections. Data lakes can support different forms of data including tables, CSV files, XML, JSON, PDF, Word documents, Excel sheets and others. The goal of this project is to create a Data Lake Management System (DLMS). The primary goals of our DLMS are:

1. To store files and their extracted fields subsets.
2. To allow for links between these extracted fields and subsets.
3. To allow for a search functionality across these extracted fields and subsets and links between them.

Architecture

Back-End:

- User login information
- Catalog of raw data items
- User permissions over data
- Interface for users
- Interface for data items
- Interface for assigning permissions
- Links between data items

Account Management and Admin Services:

- Create accounts
- See lists of “owned” data items
- Assign permissions to others

DLMS Pointer:

- Points DLMS at a new file

- Invokes content extractor
- Reads content
- Interprets file format
    - Mapping from file extension to appropriate content extractor
- Outputs key/value pairs and info about where in the file pairs were found

Indexer:

- Indexes key value pairs by key
- Creates additional index that breaks the values into constituent keywords
- Records mapping from each keyword to attribute containing the term
- Creates an inverted index (table matching from each word to specific key and value)

Linkers:

- Finds links between raw data items
    - Parent and nested data item
    - One attrib vlue is the same as key attrib of another data item
    - One attrib value is the same as the file or path name of another data item
    - Both attribs have the same value which is ID of known entry
- Takes pairs of raw data items
- Iterates over all possible key value pair combinations and checks for overlap in name or value
- Stores in the back-end a link between the data items and attributes if such an overlap exists

Web Accessible Search Engine:

- Takes multi keyword queries
- Matches them using the index
- Returns ranked results that link nodes in a graph (trees)
- Graph
  - Data items are nodes and overlapping values result in edges between the nodes
- Results ranked in a way that should be proportional to the strength of keyword match and inversely proportional to diameter of tree

Implementation Details

Our DLMS is a Node.js web app that hosts the server on an EC2 instance which is also running our local MongoDB. We are additionally connecting to an Amazon Aura Instance. We use a combination of MongoDB and MySQL models and a Node webapp. The MongoDB database is used to store the File, Inverted Index and User data items (File.js, InvertedIndex.js and User.js respectively). The actual raw data items are stored in an S3 bucket. The MySQL database is used to store the index and vertex data items (index.js and vertex.js respectively).

Extractor and Linker

This part extracts a json structure out of each uploaded file, parses the json and turns it into vertices. Each vertex is assigned a UUID and is stored in MySQL. Meanwhile, it maintains an inverted indexer in MongoDB in which keyword of leaf vertex are keys and the values are vertex that has the keyword as its value. Edges connecting parent and children and vertices with the same keyword are inserted to MySQL.

We support csv, json, and xml file format for now. The data structure gets updated once a new file is added to the datalake.

When a search is called. We use the keyword to look up in the inverted indexer to find all vertices

An optimization is that we only store edges that connect the vertices in the same file (the json tree structure). When searching for the edge connecting vertices with the same keyword, we query from the inverted indexer in mongoDB and add all the vertices into the current search frontier. In this way we can get rid of a large number of edges in our mysql and effectively speed up the search.

Search Engine

1. Single Keyword

Implemented by bottom-to-top tree search from lead node to the file's root. A keyword might have multiple related nodes (different nodes have same value) in data-lake. Each node will generate a path to file's root node independently.

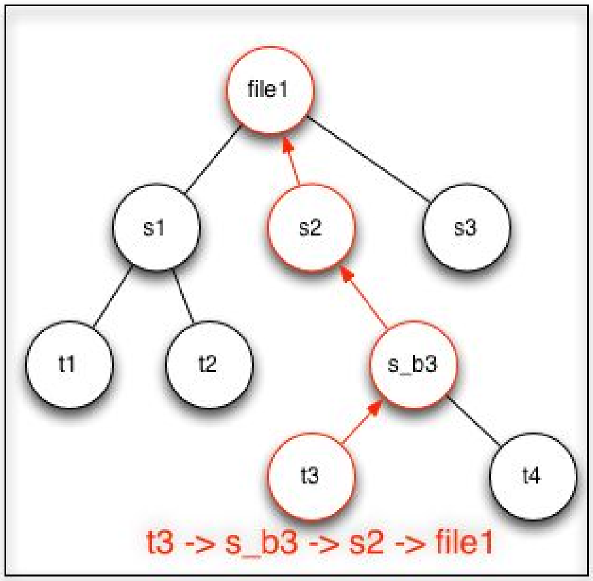


Figure - Single Keyword Search

2. Double Keywords - Breadth First Search

Naive and simple BFS is used for double keyword searching. Take the datalake connection as a graph, starting from a node set, nodes of which have the same keyword value as input keyword1, to travel in the graph to reach the node set, nodes of which have the same keyword value as input keyword2.

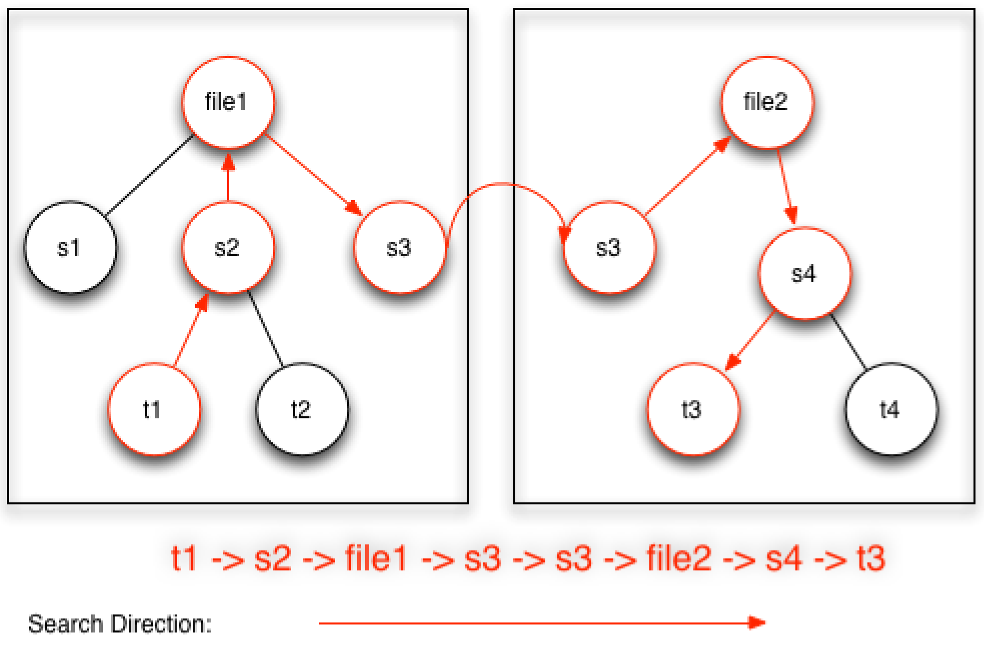


Figure - Double Keyword Breadth First Search

3. Double Keywords - Bidirectional BFS

For optimizing search performance, bidirectional breadth first search algorithm is implemented based on BFS. Search will start from both node sets, which include both keyword1 and keyword2 node sets, and when the two searches intersect with each other, a path will be concatenated together to form a path.

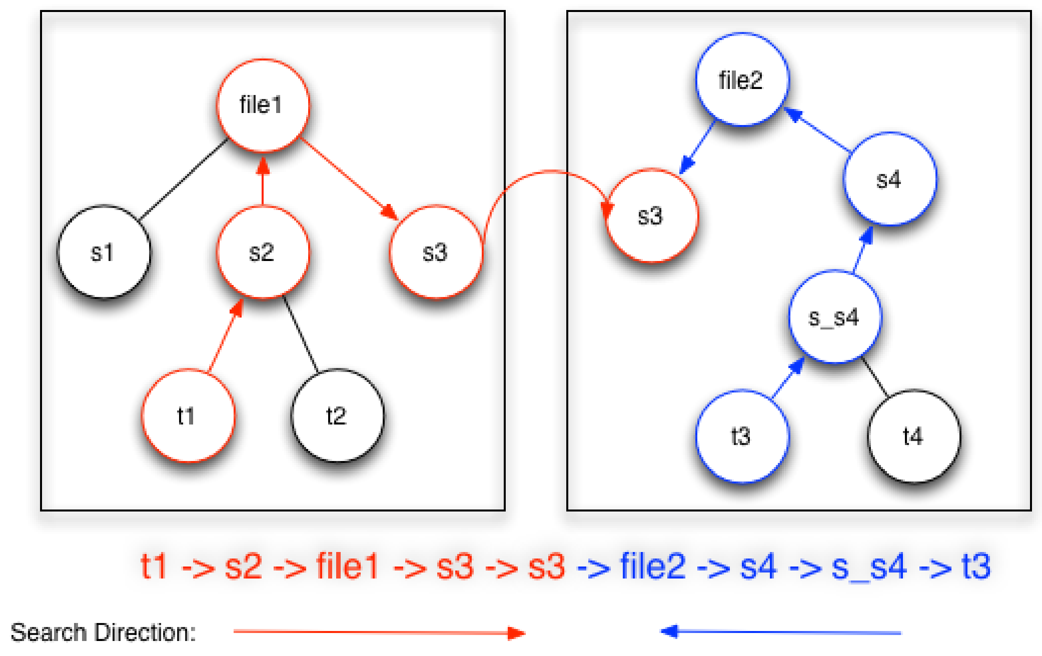


Figure - Double Keyword Search: Bidirectional BFS

Our search algorithm uses python for faster speed.

Validation of Effectiveness

We have a file that records the information of the football club, including both their full name and abbreviated code.

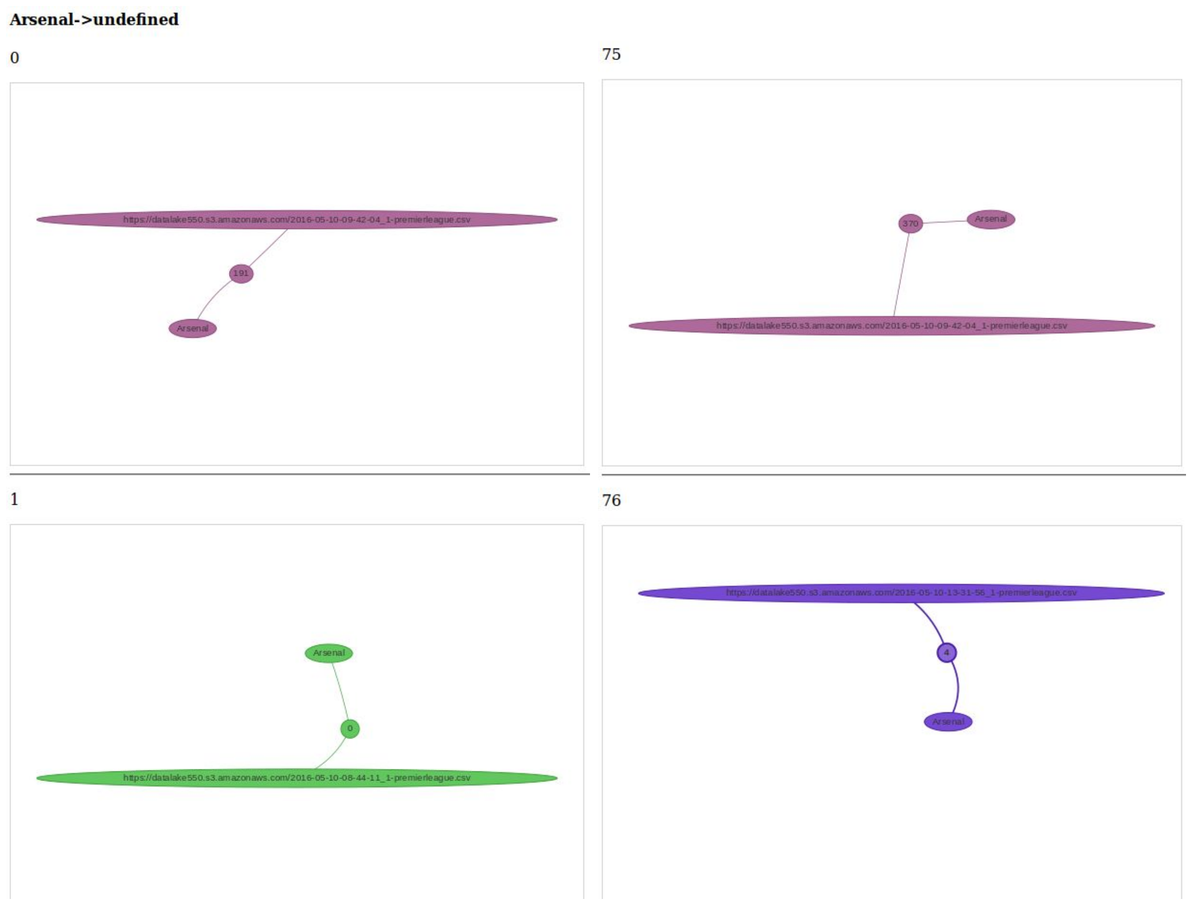
We also have another file that contains all the matches in a season but the team names are not abbreviated, they are written out.

For the visualization of search results, we render nodes from the same file with same color, nodes from different files will be rendered with various colors.

Final search results will be ranked according to the length of paths.

One keyword search [Arsenal]:

Arsenal being a name of a team, exists in the match file of different seasons. It also exists in the clubs file. We return a path to the file root for each of the vertex whose value = ‘Arsenal’

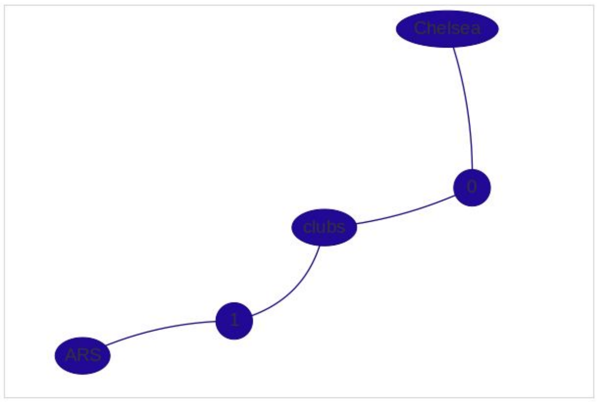

Arsenal->undefined
0
https://datalake550.s3.amazonaws.com/2016-05-10-09-42-04_1-premierleague.csv
191
Arsenal
75
370
Arsenal
https://datalake550.s3.amazonaws.com/2016-05-10-09-42-04_1-premierleague.csv
1
Arsenal
0
https://datalake550.s3.amazonaws.com/2016-05-10-08-44-11_1-premierleague.csv
76
https://datalake550.s3.amazonaws.com/2016-05-10-13-31-56_1-premierleague.csv
4
Arsenal

Two keywords search [ARS, Chelsea]:

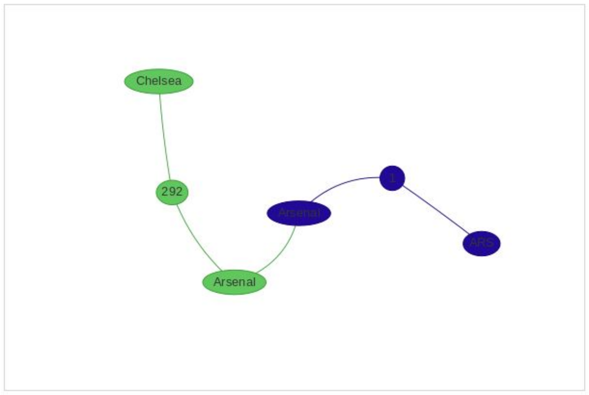
[ARS] is an abbreviated team code which only exists in the club file. [Chelsea] is a full team name which both appears in the match file through all seasons and also in the club file. So the shortest path is the one connecting [ARS] and [Chelsea] in the club file (the dark blue nodes). While other paths across file exists because the are connected via the shared keywords (in this case the full name of the team [ARS], i.e. [Arsenal])

ARS->Chelsea

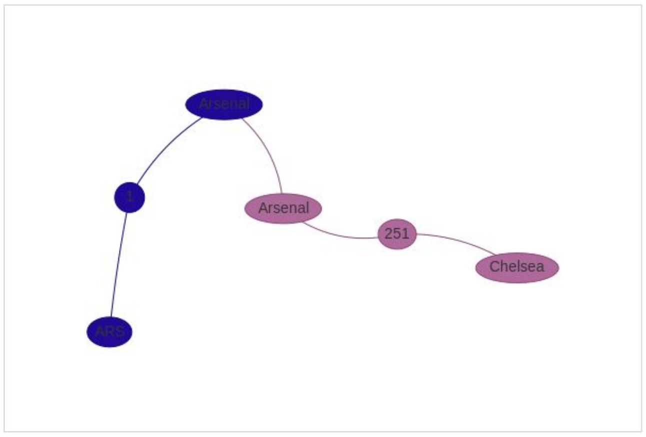
0



1

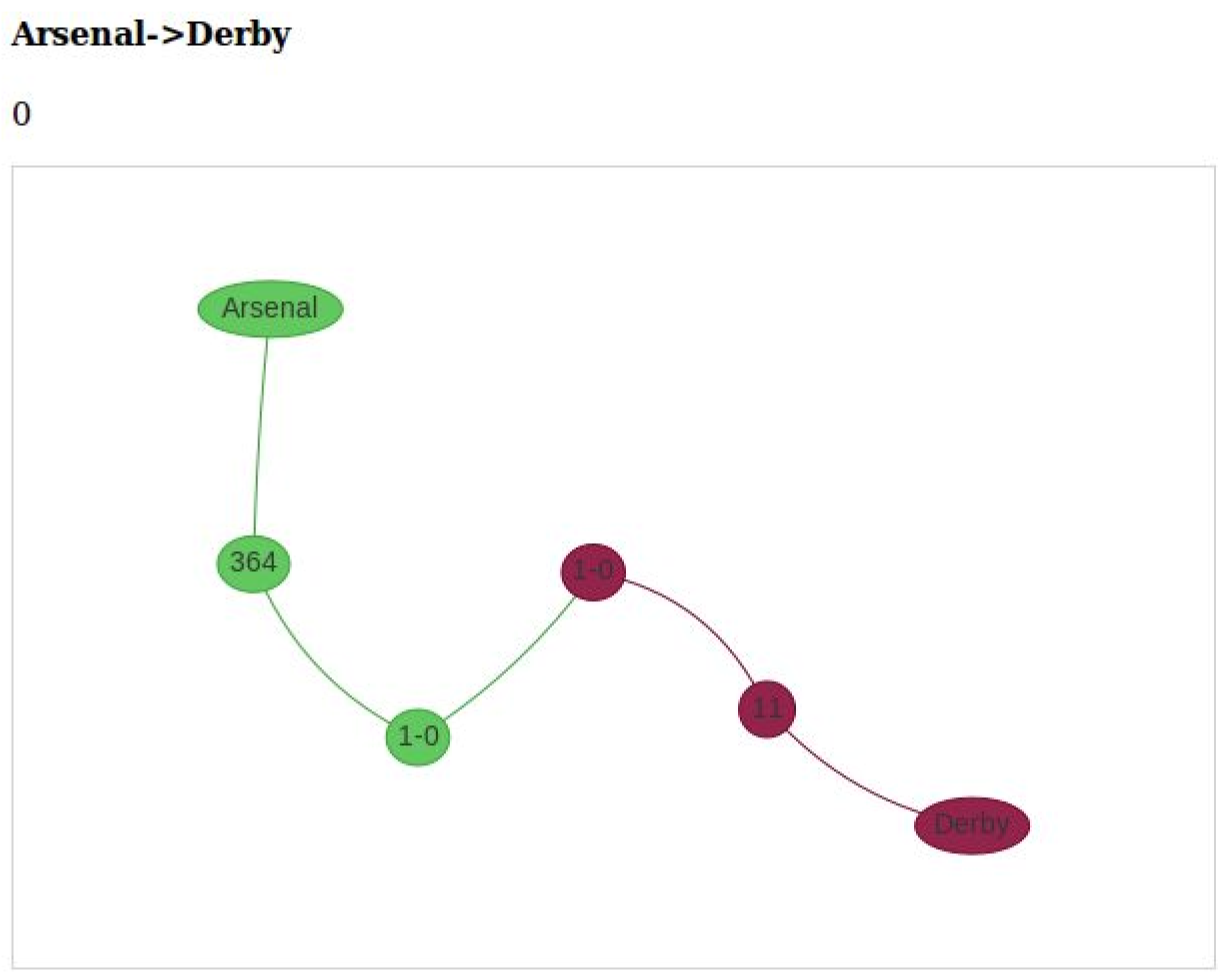


2



Two keywords search [Arsenal, Derby]:

[Arsenal] is a team stays in premier league for our current files. While [Derby] is a team in in championship league. They will not appear in the same file. Also, they might don't share the same opponents since they are in different level. The are connected by the most common score in football: [1-0]



Arsenal->Derby

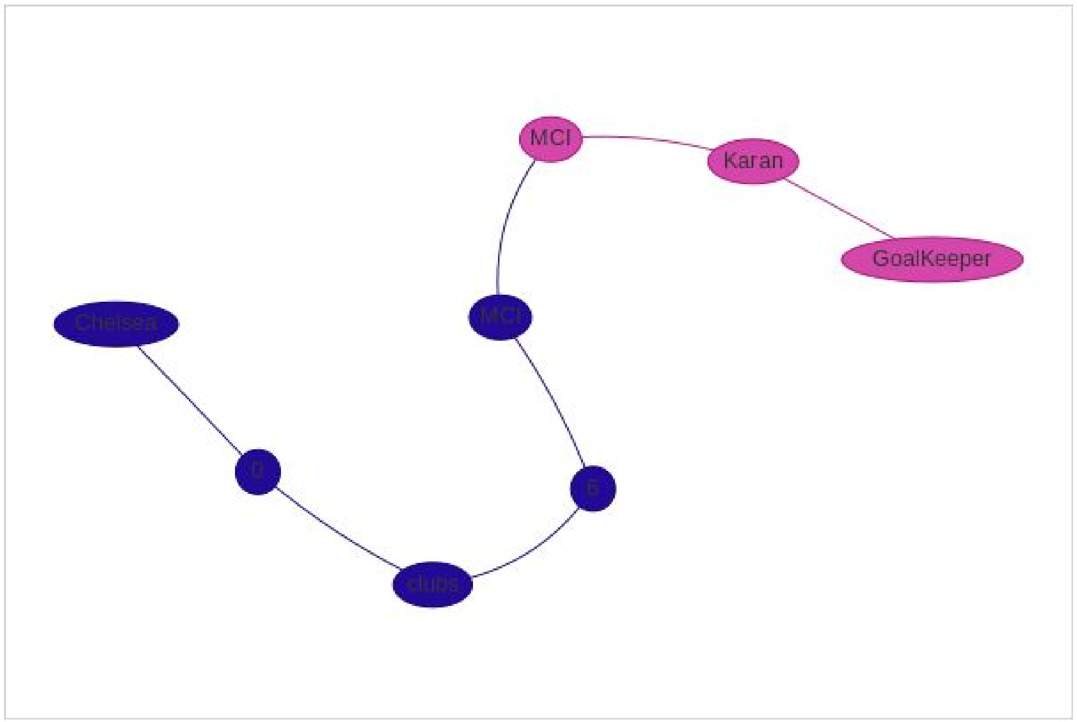
0

Search before and after a file added [GoalKeeper, Chelsea]:

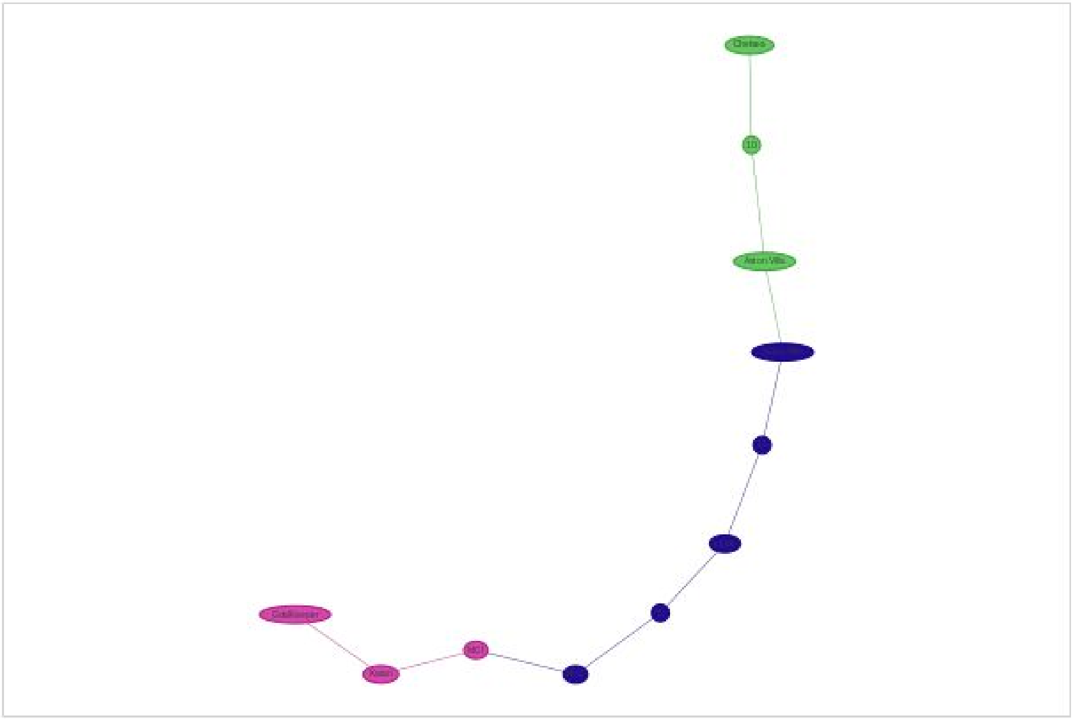
Our system do the extracting and linking every time a new file is added to the system to keep the vertices and edges up-to-date. There is no player info in the datalake. So a search of [GoalKeeper, Chelsea] will return nothing. So we upload a test player file which includes a [GoalKeeper] from Man City. And search for [GoalKeeper, Chelsea] again. Now they are linked bridged by [MCI] (Man City) or other crazy stuffs.

GoalKeeper->Chelsea

0



1



Contributions

Anthony - defined interfaces to the storage system, MongoDB/MySQL Model design and creation, report

Karan - Web Application(User system, file uploading), MongoDB setup, EC2 setup, S3 storage for file , integration

Shuai - Extractor and linker. MySQL for vertex and edge. Inverted-Index

Zimeng - Keyword(s) Search Engine Algorithm and Implementation