Task Identification Using Search Engine Query Logs

Group report

Li Quan Khoo, Horace Li, Bartosz Adamczyk

Tutor: Ms.Emine Yilmaz

COMP3096 Research Methods

Department of Computer Science

University College London

29th March 2014

Contents

1 Abstract

Given a set of strings which are issued to a search engine, a human can deduce or guess the user(s)' motivation for issuing the queries. For instance, if a user searches for "air tickets to Australia", we can guess that he/she intends to fly there at some point. If a search engine is able to perform an equivalent operation, it might be able to provide information or services of more relevance to the user than merely displaying the pages which has the highest relevance based on that single query, such as giving information about travel-related services in the given example. This involves the search engine being able to group queries together, and to be able to understand the semantics of the query strings. This is a formal introduction to the problem which we define as *task identification using search engine query logs.* In this paper, we describe an attempt to apply various methods to solve this problem and visualize the results.

2.1 Introduction

2.1.1 The Problem

Traditionally, search engines have returned sets of web pages that may contain the information required by the user based on the search query keywords. Whilst there has been discrete attempts at allowing these queries to be friendlier to the user rather than the search engine or queries to be predicted, such as Ask.com in the 90s and search suggestions respectively, users still tend to have to analyse and deconstruct their own search goals before inputting their query into a search engine manually.

The problem currently at hand is how search engines can, using a corpus of previous search engine query logs, derive a user's search goal from a small sample of searches, and more accurately and pre-emptively predict and return the required information or relevant related topics. This would vastly reduce the cognitive burden on the user having to break up tasks manually, and potentially return more accurate results by disambiguating the context. Used in conjunction with other cutting edge areas of search engine research, such as Google's recent knowledge graph, efforts in this area could eventually revolutionize the wider area of information retrieval research.

2.1.2 The Project

The project we set out to do does not attempt to fully solve the problem *task identification using search engine query logs*, as the problem domain is huge. We can further subdivide it into several smaller problems, some of which have already been solved, and our solution only extends part-way.

Before attempting to derive semantic meaning from search strings, first, we need to group similar ones together. There are three main metrics which are used to do this. One, the closer searches are to each other in time, the more likely they are to be related. Two, the more similar they look to each other (e.g. words they have in common), the more likely they are to be related. Finally, if we are capable of analysing the possible semantic meaning of the strings' components, we can determine how close their meanings are to each other.

We make use of such features in the strings to group them into what literature calls *search sessions*, or groups of search strings issued to solve an atomic information need. This is the problem of *search session segmentation*, and it has already been solved.

Once we have search sessions, we need to identify the most likely semantic entities contained within the search strings by making use of the mappings within a knowledge base. This is what the *mapper*

component of our project aims to do. Once we have determined that a set of these semantic entities co-exist within one session, we can say that these entities are related to one another with respect to this one search session, at a certain confidence level, based on how strongly the search string maps to the entities. This is what the *clusterer* component of our project aims to do.

This is as far as our project extends. It produces clusters of entities which are determined to be related to one another because they are determined to be the meanings contained within a search session at a certain confidence. This means if we gather up all these mappings to a particular entity, we have a group of otherwise unrelated entities which are also relevant when the user issues a search which gets mapped to this particular entity.

In order to fully solve the problem of *task identification using search engine logs*, we need to perform one further step which involves running search strings through a natural language processor, and then making use of these clusters which our project produces. All this is discussed in the Future work section.

2.2 Prior Knowledge and Related Work

Most of the prior knowledge and related work has been covered in our literature review(s) in the previous term, so this section merely serves as a summary of the most important information. For concision, the term "literature review" in the rest of this report would refer to the document linked below.

For an online copy of the literature review and the full bibliographies, see http://lqkhoo.com/wiki/index.php/Task_Identification_Using_Search_Engine_Query_Logs

2.2.2 Task/Session Identification

This problem has been investigated since 1999 (*Silverstein et. al.)*, so for the purposes of this project, all the required work has already been researched for us. The literature review in the previous term covered 7 papers in this area. The major results which we used are from *Lucchese et. al. (2011)*, and *Jones and Klinkner (2008)*.

In particular, the 26-minute time segmentation threshold (called T-26), and the methodology by which we calculate syntactic similarity scores, and the way we combine them are all fully based on *Lucchese et. al.*'s work.

2.2.3 Ontologies

Much of the literature surrounding ontologies are understandably about resource extraction methods and correctness evaluations, and hence are not directly related to the project. We only use results from ontologies and knowledge bases.

The literature review investigated four ontologies: Yago (and Wordnet), Dbpedia, BabelNet, and Cyc. Yago was a natural choice to use as:

1. It was the most heavily investigated ontology / knowledge base in the first term, as the suggested papers cover Yago.
2. It had the widest coverage amongst the four listed.
3. It draws information from Wikipedia / Wordnet and other manually-maintained resources, so its factual accuracy (as investigated in the literature review) is very high.

4. It was one of Ms. Emine (our tutor)'s suggestions.

We had two members look into Freebase, as Ms. Emine mentioned that the coverage from even Yago might not be enough. None of our literature reviews covered Freebase.

Since we had four team members, we decided to have the last one look at OpenCyc as well.

3 Project outline

From the problem definition in the abstract and introduction, we claim that this project is not so much about hypothesis testing as some of the projects our peers are working on. While we have prior knowledge to work with, none of them have results directly related to the results this project aims to produce, so we do not have any baseline to compare against.

This project is more about finding the right knowledge base(s) to work with, and how we are going to make use of its data features in order to augment these search sessions with semantic meaning, such that we might use it to produce mappings between semantic entities.

We have noticed since the beginning that this project was going to be a very linear one, where one module cannot be started before its dependants have been finished. From the problem's breakdown in the introduction, it can be seen that the project can be roughly divided several stages.

See appendix 5.1 for a diagrammatic breakdown of project components.
See appendix 5.2 for group member responsibilities with respect to project components.

Our final goal is, starting from a set of search queries and a viable knowledge base, isto obtain clusters of semantic entities mapping to each other.

3.1 Assumptions

Assumptions of the project, as explained in the introduction:

1. All search strings are assumed to be in English
2. All search strings are assumed to be "as-is", with no attempt to parse grammatical features

3.2 AOL Search Logs

Contributing members: Li Quan Khoo, Bartosz Adamczyk

3.2.1 Introduction

In order to cluster user-issued search engine queries, we need a corpus of search strings to work with. There are several such corpuses, but since many of the related research we read made use of the leaked AOL search logs, so that is what we settled on.

The AOL logs were originally released into the wild by an employee who thought it would benefit research efforts. Although the release itself is unauthorized, it stands as one of the biggest publicly available bodies of search text, and it is one of the most used for research.

The data available in the logs amounts to 2.1GB, with 36.4 million search strings issued in total. The actual logs available are from a subset of individuals identified via the AnonID parameter, and the

searches themselves were conducted around the year 2006. The following is an extract from an AOL log file in TSV format:

```
AnonID  Query   QueryTime       ItemRank        ClickURL
142     timing  2006-03-01 07:17:12
142     rentdirect.com 2006-03-01 07:17:12
142     www.prescriptionfortime.com   2006-03-12 12:31:06
142     staple.com      2006-03-17 21:19:29
142     staple.com      2006-03-17 21:19:45
142     www.newyorklawyersite.com      2006-03-18 08:02:58
142     www.newyorklawyersite.com      2006-03-18 08:03:09
142     westchester.gov         2006-03-20 03:55:57    1       http://www.westchestergov.com
142     space.comhttp  2006-03-24 20:51:24
142     dfdf    2006-03-24 22:23:07
142     dfdf    2006-03-24 22:23:14
142     vaniqa.comh    2006-03-25 23:27:12
142     www.collegeucla.edu    2006-04-03 21:12:14
```

Fig 3.2.1.1 Excerpt from an AOL query log

All the queries in the logs are in lowercase, and all click-through URLs have been truncated to the root page. AnonID identifies the individual who sent the query. The fields ItemRank and ClickURL identifies how far down the list of results the user actually clicked on and what the URL is, respectively, if there happens to be a click-through.

3.2.2 Processing

We want to get so-called search sessions out of the AOL logs. A *search session* is defined as a set of search strings satisfying a user's task. A *task*, on the other hand, is defined as an atomic information need. *(Jones and Klinkner)*

In *Lucchese et al.'s* research, in order to get the session output, they tested for syntactic and semantic similarity between adjacent strings; if they exceed a certain threshold, then they belong in the same session, otherwise they belong in separate sessions. Their findings also indicated that for segmentation purely based on time, 26 minutes is the best cut-off point, so a session cannot contain queries more than 26 minutes apart. Lastly, if the searches have different AnonIDs, they belong to different sessions, as they originate from different users.

On top of that, we also need to remove *stop words*, which are words having little to no meaning on their own, and occur very commonly, and we want to run the Porter-Stemming algorithm on each of the search strings to converge inflections to the base word. Doing this before running the syntactic comparator improves its accuracy.

See appendix 5.3 for the list of stop words we used.

3.2.3 Algorithm

The ideas behind the algorithm closely mimic *Lucchese et al.'s* work, but since their semantic comparator is poorly described and we are limited on time, we left that component out and segmented the search strings purely based on time and syntactic similarity. The implementation is a single-step deterministic algorithm that does everything in one go, with no intermediate output or files.

Pseudocode:

```
var input = AOL log files;
varmaxSessionDuration = 26 minutes;
varsimilarityThreshold = x;

var time;
var stemmed;
varanonId;

varprevTime = null;
varprevStemmed = null;
varprevAnonId = null;

var sessions = [];

while( input not null ) {

    line = readLine(input);
    {searchString, time, anonId} = getData(line);
    stemmed = stem(removeStopWords(searchString));

    if( prevStemmedSearchString is null ) {

            // First string
            addToNewSession(input);

    } else if (anonId not prevAnonId or
            time - prevTime>maxSessionDuration or
            similarity(stemmed, prevStemmed) >similarityThreshold) {

            // Belongs to new session
            sessions.add(existingSession);
            addToNewSession(input);

    } else {

            // Belongs in the same session
            addToExistiongSession(input);

    }

    prevTime = time; prevStemmed = stemmed; prevAnonId = anonId;

}

sessions.serialize();
```

3.2.3 Sanitization, Stemming, and Segmentation

Lucchese et al.'s metric for syntactic similarity is Levenshtein distance based on tri-grams. For two strings, they split the string into sequences of three characters and then perform a set intersection to see how many tri-grams the strings have in common. The exact method by which they calculate semantic similarity is not clear; beyond the fact that they made use of DBpedia for the comparisons.

We implemented the tri-gram Levenshtein distance calculator for calculating syntactic similarity. Its output is a normalized Levenshtein distance i.e.

$$\frac{A \cap B}{A \cup B}$$

where A and B are sets of tri-grams generated from the two strings respectively.

The final similarity threshold we used for two strings to be considered similar is 0.3. This value is arbitrary as *Lucchese et al.* did not specify their value.

For the Porter-Stemming algorithm, we used publically-available libraries. One problem of Porter-Stemmer is that although it converges inflections of a common root, the root may not be a valid English word. This affects certain word-endings in particular. For instance, “lotteries” become “lotteri”, and “ponies”, become “poni”. It also presents problems when it stems a non-English word, such as “google”, or a URL. We face a similar problem when removing stop words. For instance, “The Sound of Music” refers to the film or musical, so we cannot remove the word “The” even though it is a stop-word.

To solve this problem, we stem the search strings in order to generate tri-grams, as that improves the accuracy of the Levenshtein distance calculations, but in the actual sessions generated, we store the original search strings, whether the stemmed search strings belong to the same session or not.The rationale is that even though the Porter-Stemmer garbles certain words, if the words were the same in the first place then the garbled words would also be the same.

The arbitrary choice of 0.3 as a similarity threshold and the fact that we do not use semantic information for segmentation impacts how large or small sessions are. All sessions get cut off at 26 minutes regardless of similarity, so there is a definite upper limit of number of strings per session for all practical purposes. A higher similarity threshold (or semantic similarity, if implemented) would mean that sessions become smaller. While this value changes the sessions we work with, it does not affect our methodology for solving our problem. Having larger sessions would simply mean we get more noise in the result. Having smaller sessions would mean we might overlook certain relationships. We ignore this as we are very limited on time, and the quality of the output is secondary to our goal of investigating a viable means of clustering tasks.

3.2 Knowledge Base

Although knowledge bases appropriate for our project number at least in the teens, we eventually shortlisted YAGO, Freebase, and OpenCyc as the ontologies on which we would build upon. YAGO was notable for its accuracy and depth of knowledge, Freebase for its wide coverage, and OpenCyc for the accuracy of the data, since the content is maintained by company for enterprise level aplications. In this section, we explain in further detail each ontology, how it was assessed, and the extent which we eventually utilized its contents (or justification against further work where applicable).

3.2.1 Yago

Contributing members: Li Quan Khoo

Yago was the first knowledge base we looked at, and it is a natural choice as it is the knowledge base we were researching as part of COMP3095 in the term before. The name Yago actually refers to a series of knowledge bases (YAGO, YAGO2, YAGO2s) which are iterations and improvements of each other. For the purposes of this project, we used Yago2.

Yago’s distribution stores data in a series of files, in either TSV or TTL (turtle) format, and there is a set of separate deployment tools which are supposed to deploy those files onto a local instance of either SQL or Postgres. However, we only managed to find the deployment tools with great difficulty,

and it fails to deploy the database to either SQL or Postgres on two out of two of the machines we tried it on; it only builds the database indices and not the data, so after several days of trying to deploy, we switched tactics and decided to build our own extractors to put the relevant data into MongoDB.

When unzipped, Yago's distribution files amounts to about 20 Gigabytes, but it turns out that we are only interested in about 33% to 50% of it. What we want are *entities*, *classes*, and class-to-class and class-to-entity *mappings*. Yago contains other information, such as longitude/latitude information, where it derived the information from, and facts, such as the age of an individual. We do not make use of this extra information.

Yago files are semi-structured, so we found it easy to construct our own classes to map components in each line of a file to what they are supposed to mean, and then inserting that data into MongoDB for querying.

```
<wordnet_person_100007846>    rdfs:subClassOf       owl:Thing
<wordnet_person_100007846>    owl:disjointWith      <wordnet_organization_108008335>
<wordnet_person_100007846>    owl:disjointWith      <wordnet_building_102913152>
<wordnet_person_100007846>    owl:disjointWith      <yagoGeoEntity>
<wordnet_person_100007846>    owl:disjointWith      <wordnet_artifact_100021939>
<wordnet_person_100007846>    owl:disjointWith      <wordnet_abstraction_100002137>
<wordnet_person_100007846>    owl:disjointWith      <wordnet_physical_entity_100001930>
```

Example of a TSV Yago file (yagoSimpleTaxonomy.tsv)

Each Yago class can have up to n superclasses and subclasses, so the mappings do not equate to a tree, although classes which have more than one superclass only account for < 0.1% of the classes. Each Yago entity can also belong to more than one class; the files actually map them starting from the leaf class all the way to one of the root classes, although strictly speaking, the class graph is not a tree.

The final output of the Yago extractors is two MongoDB collections containing entities and classes each.

Class

```
{
    "name": "<className>",
    "searchString": "classname",
    "subclassNames": [
        "(subclassName)"
    ],
    "superclassNames": [
        "(superclassName)"
    ]
}
```

Entity

```
{
    "name": "<Java_(programming_language)>",
    "cleanName": "Java",
    "disambig": "programming language",
    "searchString": "java",
    "relations": {
        "<hasWikipediaArticleLength">: [
            (value)
        ],
        "<hasWikipediaUrl>": [
            (value)
        ],
        "<linksTo>": [
            (value)
        ],
        "rdf:type": [
            (classNames)
        ]
    }
}
```

Example MongoDB BSON/JSON document schemas

Each class originally had a reference to the entities it contains, but the MongoDB default can only store 16Mb per document, and some of the most massive classes have millions of entities, which exceed this by a large margin. We had a choice of either compiling MongoDB ourselves, which we chose not to, or we store the references using an integer reference to another document storing a list of entities. We implemented the latter but we ended up not using those mappings anyway.

These two collections give us all the information we need to proceed to the mapping phase.

3.2.2 Freebase

Contributing members: Horace Li, Bartosz Adamczyk

Freebase is a large community-built knowledge base, maintained primarily by its community (in a way not dissimilar to Wikipedia's method of operation). In the same way that Wikipedia has an accuracy comparable to proprietary encyclopaedias (such as Encyclopaedia Britannica) and a coverage far surpassing any of its peers, Freebase currently possesses one of the largest coverage in its area, with a remarkable level of accuracy for a dataset that is theoretically editable by the general public. Indeed, it's currently used by Google as one of the core providers of data for the latter's much hyped Knowledge Graph.

Our original intent was to use Freebase as an alternative potential ontology from which we could derive relations between topics and perform clustering similar to what was attempted with YAGO. Whilst Google readily provides an API from which we can query the relevant graphs and topics in JSON or RDF, it was identified at an early stage that this method of access would not be easily scalable due to Google's restrictions on 100k read queries per day per access key (i.e. Google account), compared to our session query data reaching a count of over few millions.

With Freebase possessing a coverage far exceeding that of YAGO with over 40M topics and 24B facts, we eventually decided to query a selection of session data in order to decide whether this was an avenue worth pursuing further, even if that might eventually mean running queries on a proprietary cluster on over 250GB (uncompressed, 22GB when gzipped) of RDF data. This 'side project' was conducted concurrently with the primary efforts on YAGO.

Google provides various types of APIs which perform a range of functions:

- The Topic API returns all known facts and properties for a specified topic in JSON.
- The RDF API returns a subgraph of all data connected to a specified Freebase object in RDF. The results are directly filtered from the raw RDF data dump (the entire Freebase knowledge graph) that can be downloaded and disseminated manually.
- The MQL API provides both read and write access to the Freebase ontology. While it is by far the most flexible and powerful method of accessing Freebase, it requires the use of a DSL derived from JSON known as Metaweb Query Language (MQL).
- The Search API returns a list of matching entities in JSON based on arbitrary textual keywords.
- The Reconciliation API returns a list of entities in JSON whose properties' values match that specified in the query.

Although all the APIs could potentially be of varying degrees of benefit, time constraints meant that we did not have the opportunity check the coverage of Freebase using all the APIs. Certain APIs were expected to be highly useful (such as the MQL API), but it was simply unfeasible to go through the

entire MQL specification and test out the best way of formatting a query to return the most relevant results without over-narrowing the result set.

Taking under consideration easiness of the implementation, and quality of results we have decided to use the use JSON result from Search and Topic API. This allowed for easy access via almost every language and platform we would decide to use. In regards to the daily limit of 100 000 queries, the best approach was to query the knowledge base only once, for each string. Every result we received from the FreeBase we stored in the mongoDB. Firstly we used the Search API to obtain clear entities in the queries. The query is constructed via GET method containing the query string and the API key, which is what Google uses to track the daily quota.

```
https://www.googleapis.com/freebase/v1/search?query=Java&key=<API_KEY>
```

Example of query to FreeBase

Then the result formatted in JSON contain the list of best guess matches, with the score which indicate certainty. Most of the results Freebase returns have a type (class), which we can use to cluster the results. However, certain results do not have a type. In the example illustrated below, where we query about “java”, we get the best guess for which entity is most likely related to the string “java”, which is the programming language sense of the word, with score = 195.

```
{
  "status": "200 OK",
  "result": [
    {
      "mid": "/m/07sbkfb",
      "name": "Java",
      "notable": {
        "name": "Programming Language",
        "id": "/computer/programming_language"
      },
      "lang": "en",
      "score": 195.842285
    },
    {
      "mid": "/m/0j1_3",
      "id": "/en/java",
      "name": "Java",
      "notable": {
        "name": "Geographical Feature",
        "id": "/geography/geographical_feature"
      },
      "lang": "en",
      "score": 133.882019
    },
    {...},
    {...},
    {...},
    {...},
    {...}
   ],
  "cursor": 20,
  "cost": 9,
  "hits": 22407
}
```

Example response from the FreeBase Search API

FreeBase also offers the Topic API, which offers additional information (facts) about an entity. For instance, if we query for the person “Barack Obama”, we get attributes of the entity, such as gender, age, and occupation. Much like the facts in Yago, they do not contribute towards clustering / aggregration, so we do not make use of this information.


```
{
 "id": "/m/02mjmr",
 "property": {
   "/people/appointer/appointment_made": {...},
   "/people/sibling_relationship/sibling": {...},
   "/people/person/children": {...},
   "/people/person/date_of_birth": {...},
   "/people/person/education": {...},
   "/people/person/employment_history": {...},
   "/people/person/ethnicity": {...},
   "/people/person/gender": {...},
   "/people/person/height_meters": {...},
   "/people/person/nationality": {...},
   "/people/person/parents": {...},
   "/people/person/place_of_birth": {...},
   "/people/person/places_lived": {...},
   "/people/person/quotations": {...},
   "/people/person/sibling_s": {...},
   "/people/person/spouse_s": {...},
   "/people/person/weight_kg": {...}
 }
}
```


Example of the response from the FreeBase Topic API

Towards the latter half of the project, work on Freebase was dropped because YAGO's coverage proved adequate and reconciliation between YAGO and Freebase results was expected to contribute minimally to the final clustering relative to the amount of effort required to merge the code bases and results.

3.2.3 OpenCyc

Contributing members: Bartosz Adamczyk

OpenCyc is the open source components of the other proprietary Cyc project, currently developed by Cycorp. It’s primarily an attempt to build a knowledge base of everyday knowledge, with the aim of allowing AI to utilize this ontology and reason as humans do. OpenCyc contain 239,000 terms and about 2,093,000 triples. Although the commercial edition of the Cyc Knowledge Base is about two times bigger, Corporation provide free licence for the research use, which we could use.

Because of lack of time and some personal problems in equal contributions to the project, we decided to abandon OpenCYC at an early stage, to work on Freebase, which is much more powerful and useful resource. Furthermore, while the data was expected to be useful, the lack of community support and resources compared to YAGO and Freebase meant our usage of the ontology did not proceed beyond a topical analysis.

3.3 Mapper

Contributing members: Li Quan Khoo

The mapper is the module which we used to address Yago's shortcoming of not having an implicit probability mapping between a string and an entity (which Freebase has). In other words, Yago does not have information about the relative likelihood of a string mapping to its multiple possible meanings (entities). On the other hand, Freebase can tell us that at any given time, "java" is much more likely to mean the programming language than the place in Indonesia (or any other sense of the word).

We need this information to even begin mapping strings to classes or entities, as otherwise we will blindly map to all possible entities and end up with as much noise as useful results (for entity to entity mapping), or a lot of noise, even as we get more coverage (for string to class mapping).

This mapper module is the implementation of our final approach, which sacrifices coverage for high accuracy; the rationale is that since we do not know how exactly how accuracy would be impacted, whether we map blindly or not, we would rather end up with a small but usable result set rather than a large but noisy one which we cannot use.

The idea behind how to do it is simple – since Yago does not give us the probabilities of a string mapping to an entity, but it gives us all the classes which an entity belongs to, it is trivial to observe that the more classes two entities have in common, the more related the entities are to each other. Additionally, since Yago's information is based from Wikipedia articles, it also gives us information about what other entities are linked to from the Wikipedia article of any single entity. Hence, we settle on a formula:

$$similarity\ score = \ classes\ in\ common + 0.2(links\ in\ common)$$

We settled on a minimum similarity score of 8. This means two entities, when compared to the other, needs to give a score of at least 8 for the two to be considered to have a relation in the context of the single search session. The weightings 1 and 0.2 for classes and common links respectively are fairly arbitrary; we neither have the time or resources to exhaustively try for the optimal values, so we tweaked it based on the average number of links given per entity and running the mapper with the values to see how many sessions have useful information that passes this semantic similarity check. The similarity threshold of 8 is picked after running tests with integer values between 5 and 10. A lower score gives more noise, a higher one quickly limits the number of sessions we have (sessions with zero entities mapping to one another are discarded), since most entities only have a limited number of classes and we risk ignoring the minor entities altogether, although this is not really an issue if we are focusing our efforts on the main classes close to the root class.

There are caveats for running the mapper – since the mapper depends on at least two entities to compare against each other to determine the likelihood of the strings' intended meanings, only sessions with more than one entity are useful to us, and hence single-entity sessions are discarded. Despite the potential ramifications of severely cutting down on the number of sessions in the end, we found that out of 18 million sessions, about 9 million of them satisfy this criterion.

Also, consider the string "travel to java", if travel and java do not share common classes, it is going to be discarded, but there was no way around this beyond using Freebase, or by running an NLP processor beforehand (this is discussed in the results). We do not actually know how many useful classes we discarded in the 9 million sessions which we do not account for, but we can be sure that

this skews our mappings towards searches such as “great lakes, united states”, for instance, where we have entities belonging to the same classes (e.g. places) occurring together.

In the visualizer, the mapper’s equivalent implementation in JavaScript gives an output looks like this:

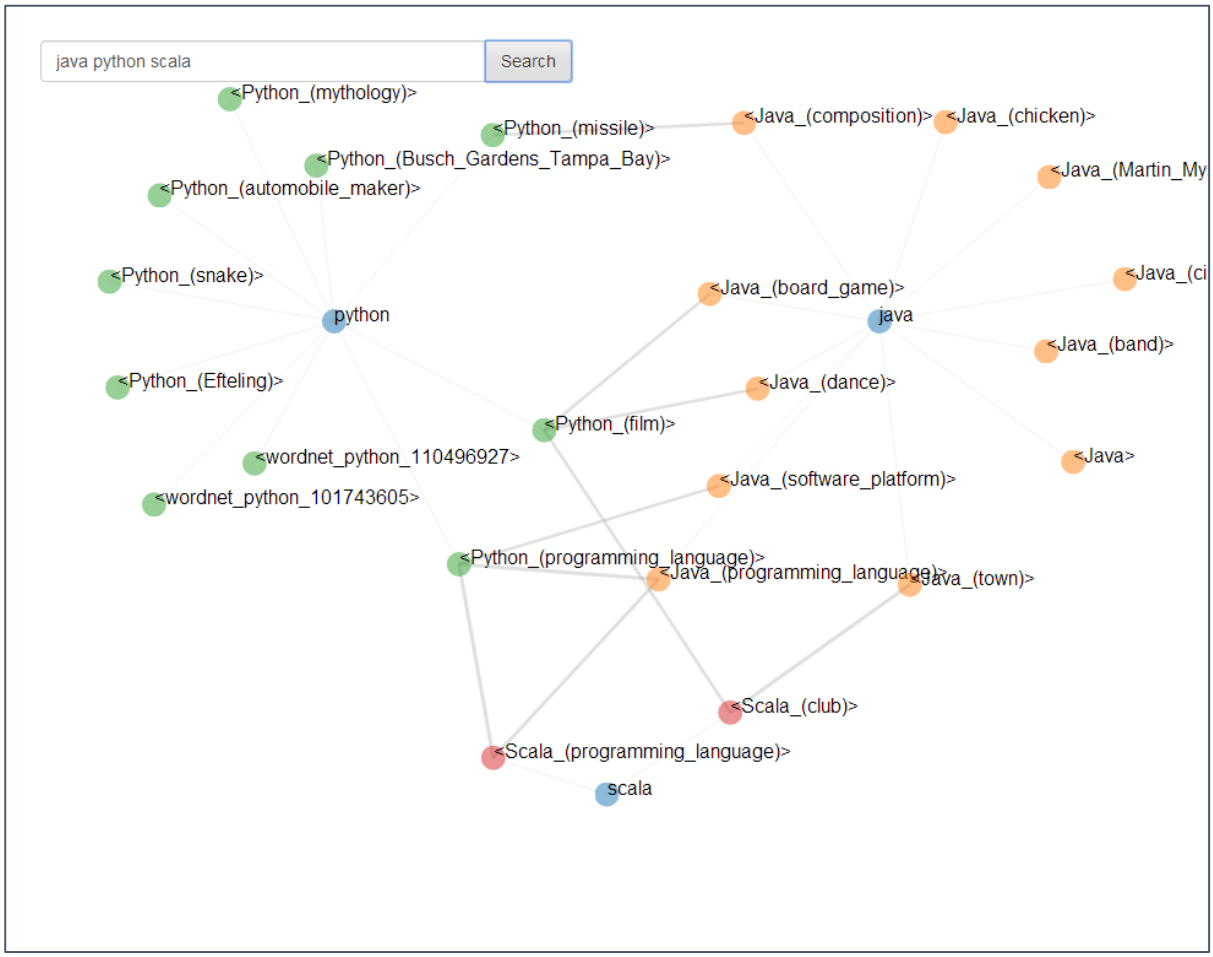


Fig 3.3.1. Visualization of mapper output. Blue nodes are strings, other colours distinguish groups of entities.

In the above screenshot from the visualizer, we can see that within the search context “java python scala”, the mapper has isolated the most likely semantic meanings of the strings, and it has determined that the programming language senses are related to each other, and it also draws associations between the senses which have positional, or cultural meanings as well. The broad links join entities which have a similarity threshold exceeding 8 when compared to each other.

The mapper is only run before we perform two clustering / aggregation procedures where we need this semantic information: entity to entity mapping, and string to class mapping.

3.4 Clusterer / aggregator

Contributing members: Li Quan Khoo

The clusterer is the module that gives us our final output. We investigated three modes of clustering:

1. String to string (baseline – naïve related searches implementation)
2. Entity to entity
3. String to class

3.4.1 String to string

Our string to string implementation serves as our baseline, to see roughly how many mappings we would get if we included absolutely everything, without performing the procedure involving the mapper module. This is our version of a "related searches" implementation, as what we would find on actual search engines.

For instance, the following are related searches returned by Google when searching for "python".

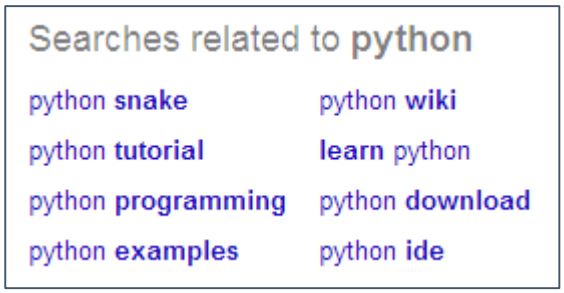


Fig 3.4.1.1. Google's related search strings for the term "python"

We can see that the results are not entity-specific; they are the most common terms queried alongside the string "python", and our implementation mimics almost exactly that, with the condition that the string in question, in the example it would be "python", must map to at least one entity in the knowledge base. Which entity, for this case, we don't care.

Our method of obtaining the mappings is best illustrated via an example. For instance, for the string "flights to java", first, we break it down into an array of strings consisting of all possible spacebar-delimited substrings. For our example, this would be ["flights to java", "flights to", "to java", "flights", "to", "java"].

Once we have the array, for each array member, we query the knowledge base, checking whether the string corresponds to an entity. If not, we run Porter-Stemmer on the string to attempt to get the root word. If the stemmed version is also not an entity, we move on. If we find that a string does indeed correspond to an entity, we copy it to an array which we'll call the *graphing array*, and we remove all substrings of that string from our original array. For instance, if "flights to" got a match, then we transform the array into ["flights to java", "flights to", "to java", "java"] by removing ["flights", "to"].

Once we reach the end of the array, we take all members of the *graphing array* and map them all to each other (by constructing a complete graph). We also map each string to themselves, as if we only have one string in the graph, without self-mapping, we get zero mappings, which means we lose the information in the current session.

To compare against Google's related searches, we check the string "python", and it looks like this in the visualizer:

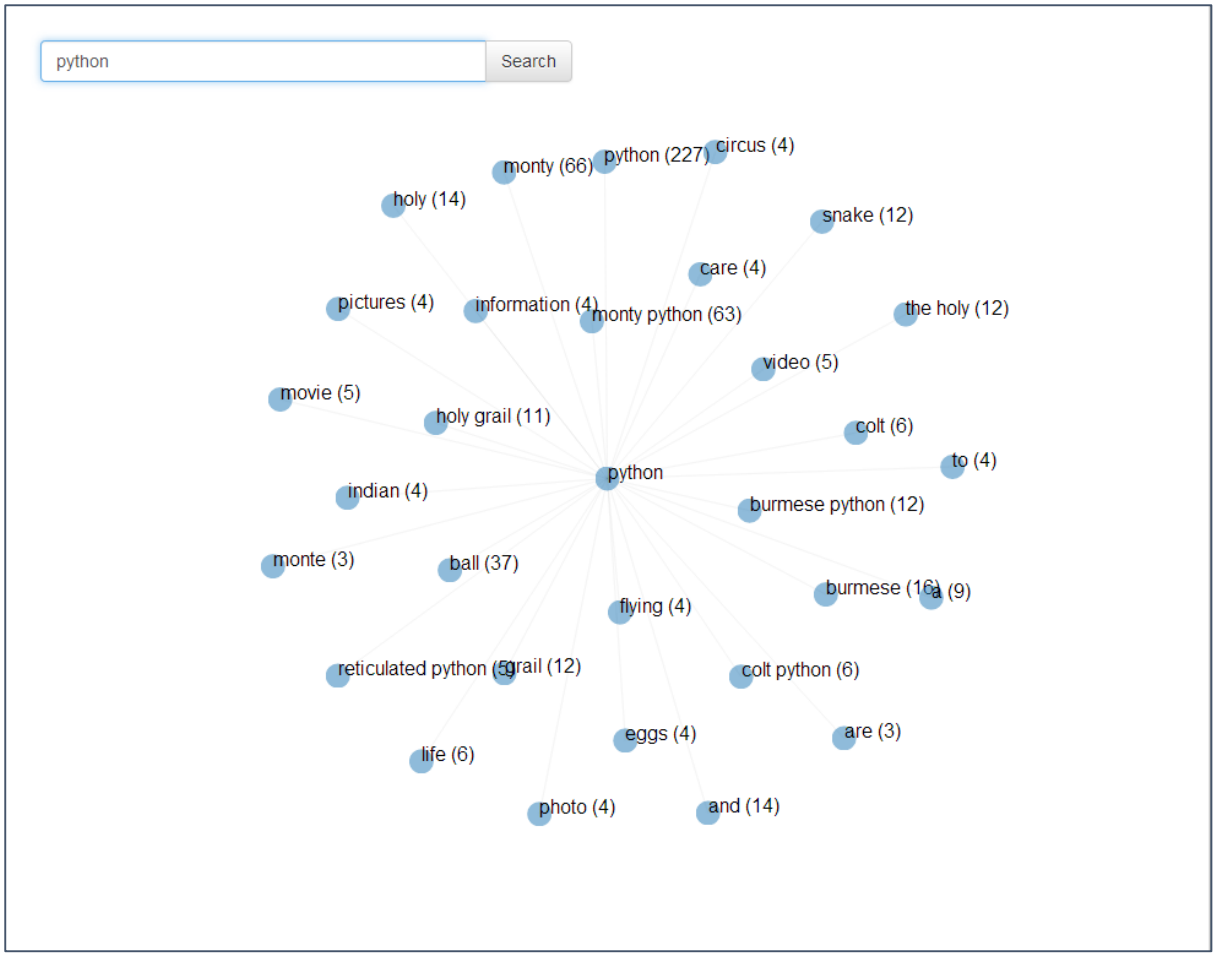


Fig 3.4.1.2. Visualization of string to string mapping of the cluster "python".

We can see that there is a python node (weighted 227) mapping to itself; this simply means there were 227 sessions containing the string "python", which happens to map to at least one entity in Yago, our knowledge base of choice. This self-mapping is important, as if the session only contains the string "python" and we disallow self-mapping, since "python" does not map to anything, then this particular instance of "python" would end up not being counted. The other nodes are self-evident – search strings occurring in conjunction with "python". For our result set, we limited such strings to a maximum length of two words, to restrict the results. Results having more words also become exceedingly unlikely to be a common query, and are very unlikely to reoccur or overlap.

3.4.2 Entity to entity

This second type of mapping is the natural progression after implementing our baseline. The line of thinking was, since we are already determining that these strings map to at least one entity, we should investigate which entity would be the most likely one in the context of the session, and map to it instead of to its raw string, and, if these strings mapping to the main entity map to at least one entity as well, we should also find out the most likely one, and so then we get mappings from entities to entities instead of from strings to strings.

This is when we ran into the problem of identifying which was the most likely entity and ended up implementing the mapper module. Using the mapper's output, referring to *Fig 3.3.1.,* in the section

discussing the mapper, it is a simple matter of registering which entities have a broad line connecting to another entity.

What we ended up with is this:

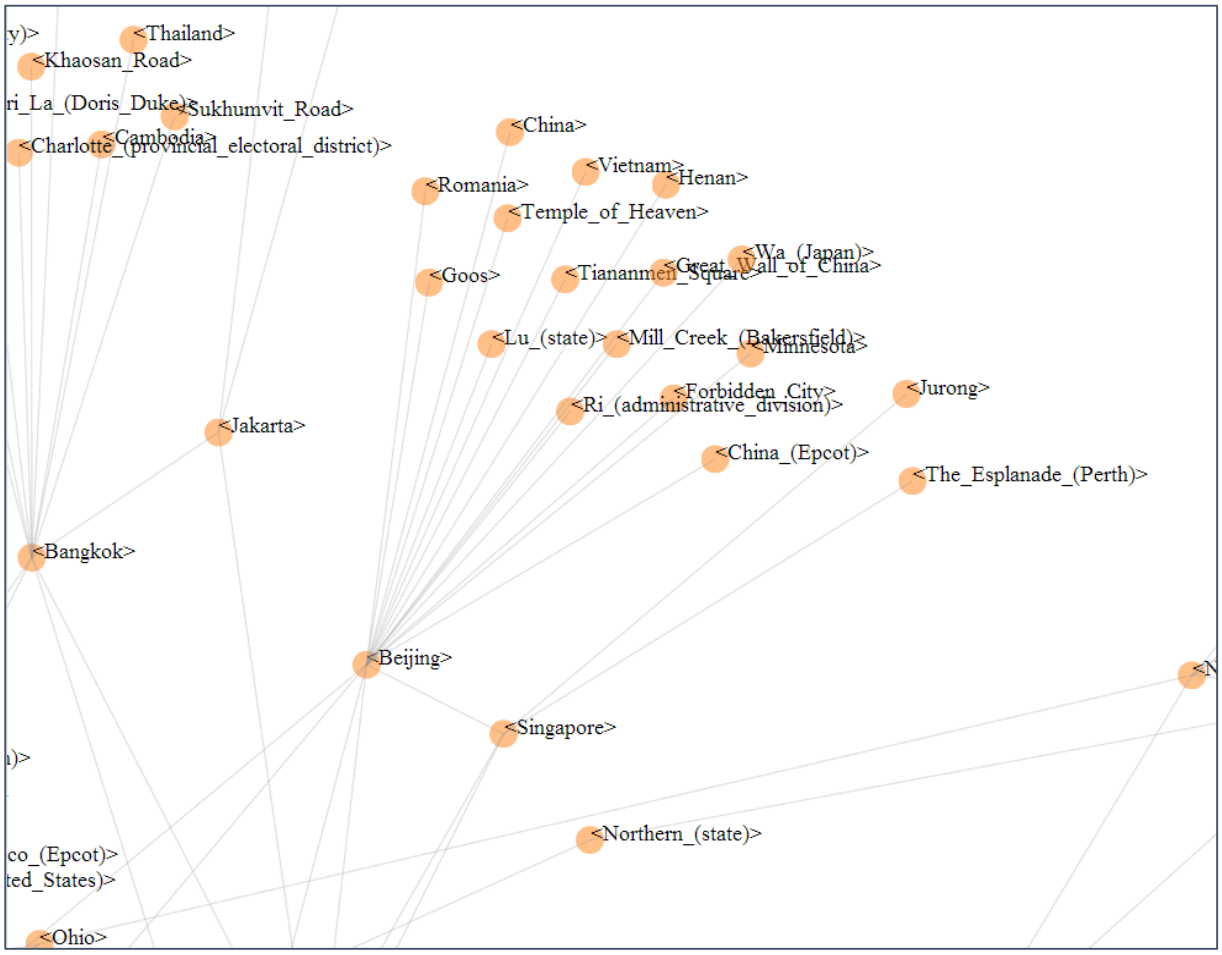


Fig 3.4.2.1 Zoomed view of entity to entity map.

The above screenshot is a partial view of the map, focusing on the entities "<Beijing>", and "<Singapore">. We can see that the result set actually makes sense – users searching for Beijing also searched for landmarks like the Great Wall of China, or Tiananmen Square, or China itself. However, we cannot make use of this information to perform aggregation, which is what we set out to do, because here, relations are specific to the entities they bind. For instance, <Great Wall of China> is related to <Beijing>, and <Beijing> is a place. <Singapore> is also a place, but we cannot say that <Great Wall of China> is related to <Singapore>; entity to entity relations are too specific to be used for aggregation. We got surprisingly good results – it's just unfortunate that we cannot use it for our purposes.

There is another feature of entity-to-entity mapping. Since we have entities mapping to one another without regard for their respective class hierarchies, we end up with a graph which is very disorderly and difficult to traverse.

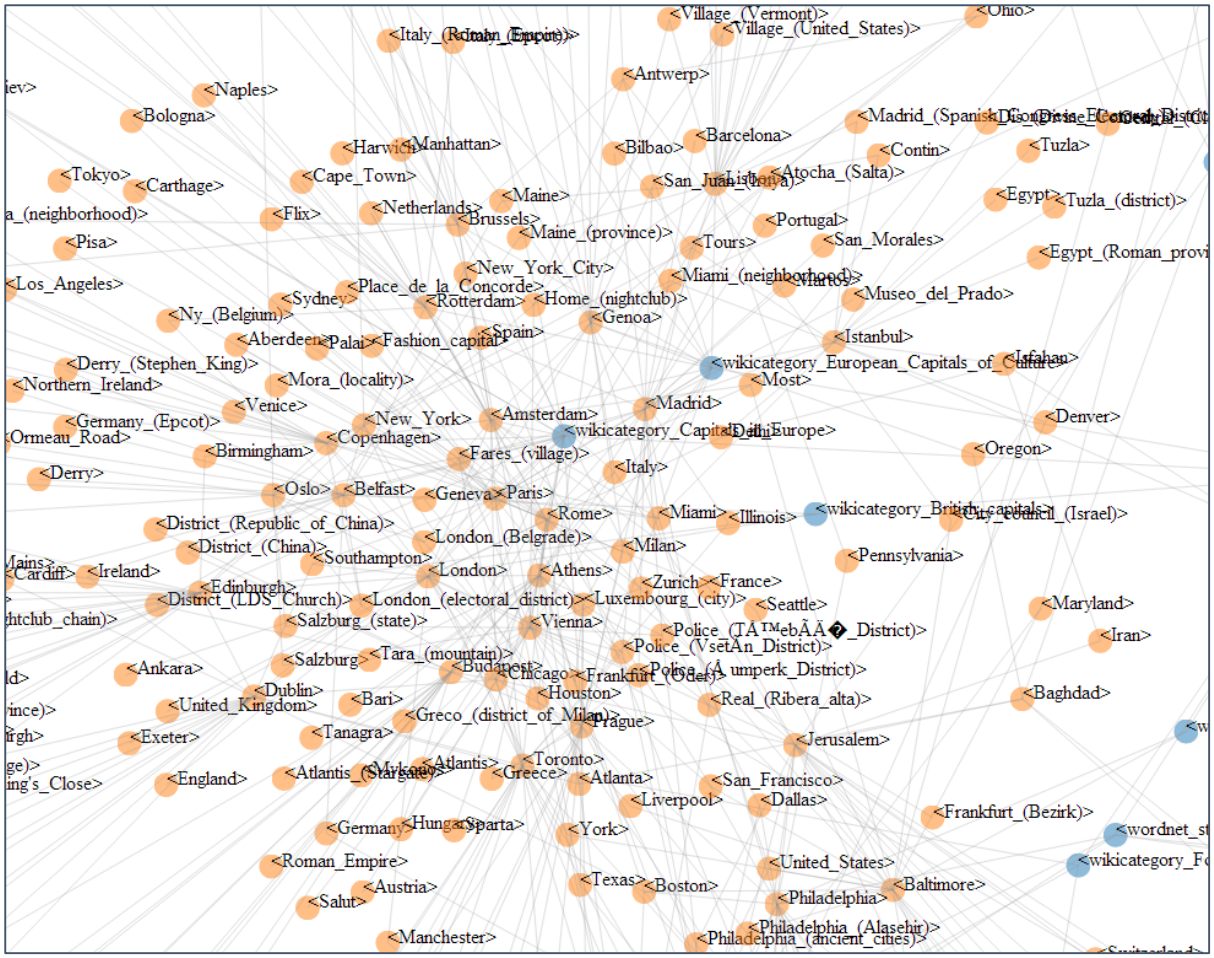


Fig 3.4.2.2 Central cluster of entity to entity mapping. Showing disorderliness of mappings.

Since the result set is not useful and the data is difficult to work with, we immediately abandoned this method of clustering. It was not a pointless exercise, however, as while doing it, we deduced that we cannot map using entities either way – they are too specific. We need to revert to our previous line of inquiry which was using strings, and since we need to aggregate results, we need to be mapping them to *classes*. This brings us to our third and final form of aggregation, which is also the one which is arguably the most successful.

3.4.3 String to class

In this method, we perform the same mapping as if we were doing entity-to-entity mapping, but we also keep the strings intact. An excerpt from our presentation slide:

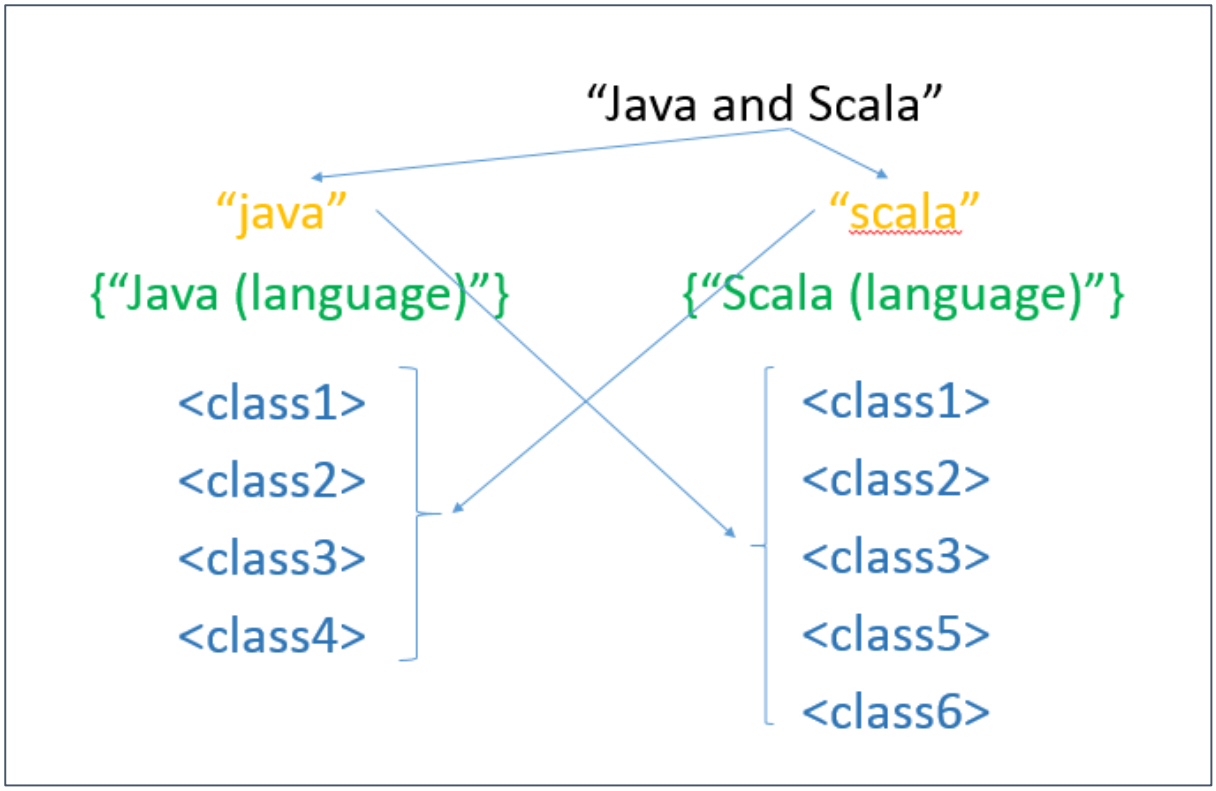


Fig 3.4.3.1 Illustration of string to class mapping procedure.

From the mapping output, we get the green text, which are the likely entities mapping to a given string. The orange text is simply the lowercase form of the original text. Now, instead of mapping the green text to one another, as we had done in entity-to-entity mapping, we use Yago's class information and perform the mapping illustrated above – mapping the original string to all other entities' classes. If a string corresponds to more than one likely entity, then all classes of all likely entities are mapped to.

The results to be discussed are quite substantial, so we shall relegate that to the results section to be discussed in full.

3.5 Visualizer

Contributing members: Li Quan Khoo, Horace Li

The visualizer component does not involve any new research or work per se. Rather, it is a tool for us to inspect the data within our database to get some inspiration of how to proceed or how to improve the data we have, and finally it is a way for us to present our findings succinctly during the presentation.

We used MongoDB as our storage database, and since MongoDB does not handle HTTP requests directly, we eventually opted to deploy our own HTTP server which runs a MongoDB driver. We ended up using a lightweight Python-based backend called Flask, which queries the local MongoDB instance for us, and returns results to the browser. The JSON results are then visualized with the D3.js framework using a force-layout graph.

We had four main visualization components; the screenshots in the sections before this are courtesy of the visualizer. The first shows how the mapper works (Fig 2.3.1), and we have one each for

showing the results for string to string mapping (Fig 2.4.1.2), entity to entity mapping (Fig 2.4.2.1), and string to class mapping (Fig 4.1.x).

4 Results and analysis

4.1 String to class mapping

This is the final result from our process. The root node looks like this:

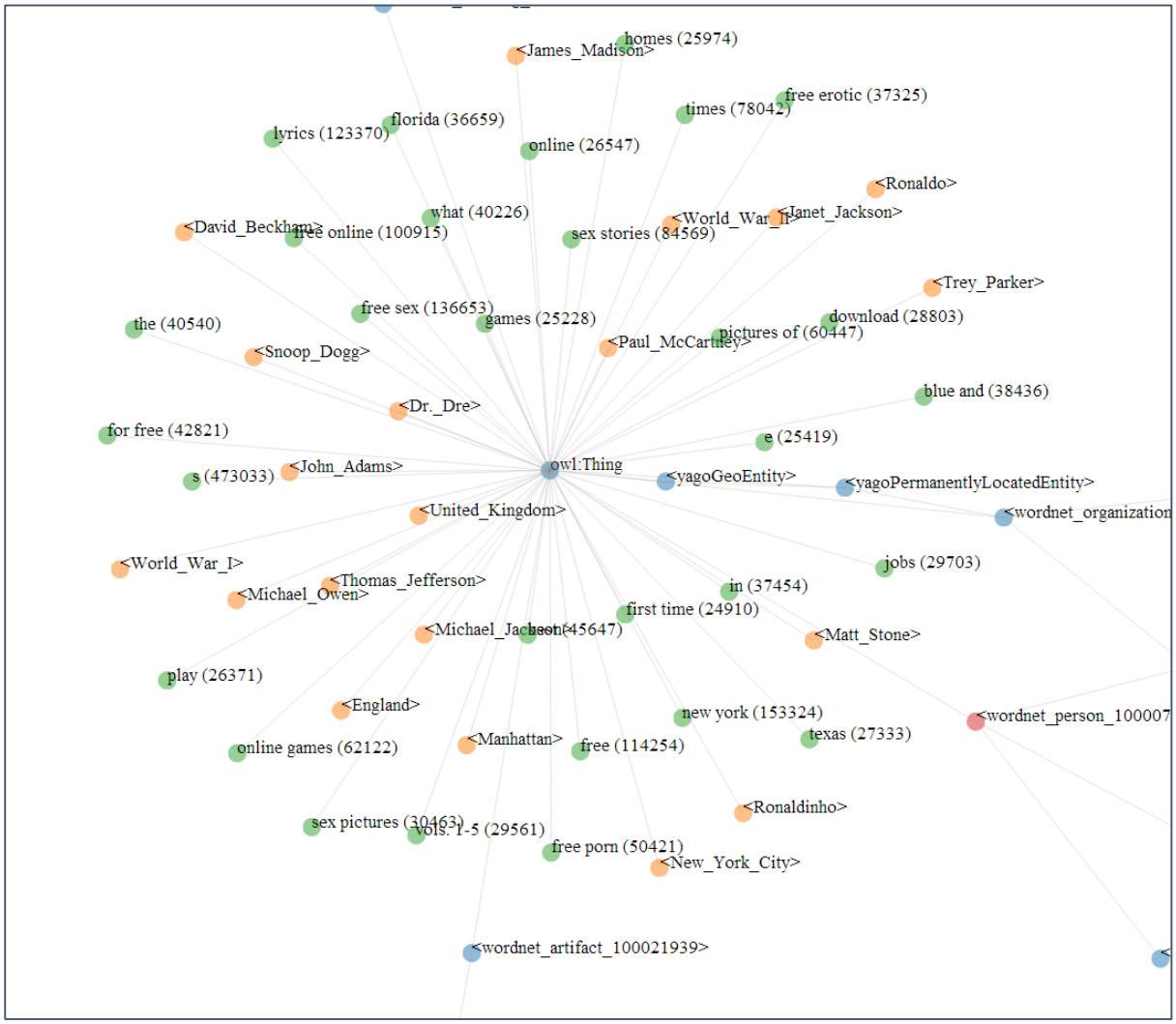


Fig 4.1.1 Root cluster of string to class mapping.

Blue, and more rarely red nodes are Yago classes (there is no distinction between the colors). Green nodes consist of 30 of the most frequently occurring strings mapping to an entity which belongs to the central node (in this case the root). Orange nodes are the entities belonging to the central node which gets mapped to the most often by strings.

Most of Yago's entities map to the class chain from their lowest class all the way up to the root node, so the root node in the above screenshot contains just about anything. The high prevalence of search strings related to the sex industry in our result accurately reflects its prevalence in the web – that gives us a degree of confidence about the reliability of strings mapped to root. The majority of the top 30 strings get between 10k to several hundred thousand mappings as well, so that's another measure of confidence.

However, when we travel one class down the hierarchy, we find something interesting. For instance, when we expand the node <yagoGeoEntity>, which is essentially the root class for all places, landmarks, and buildings, we get this:

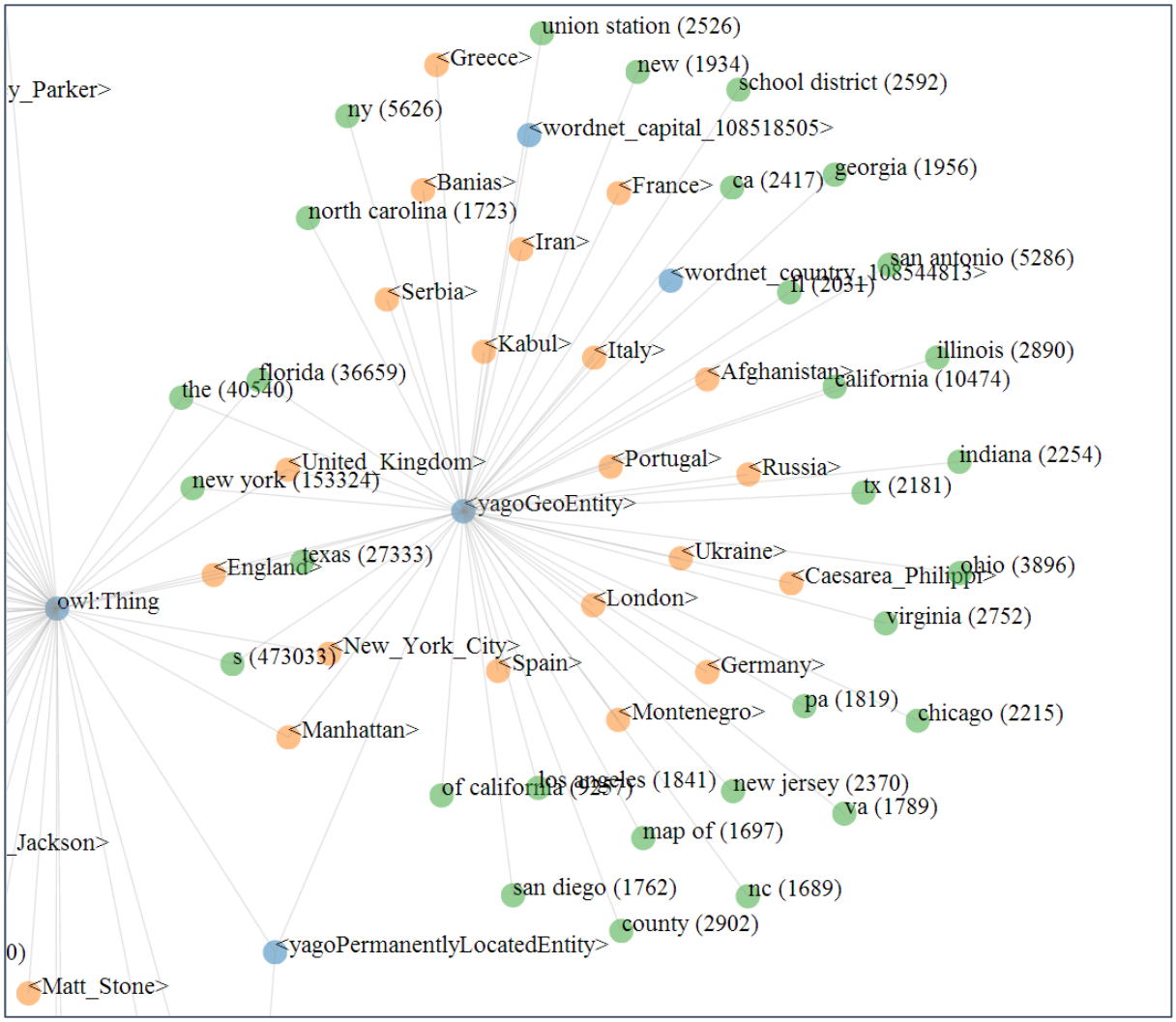


Fig 4.1.2 <yagoGeoEntity> cluster of string to class mapping.

Notice that from tens to hundreds of thousands of mappings per string, we are now in the neighbourhood of around a thousand, for the strings mapping only to <yagoGeoEntity> and not to owl:Thing. This is one of our major results – travelling down one node decreases the number of matches by 100 times – we get an exponential decay.

If we focus only on the <yagoGeoEntity> class, however, we find that among the different strings which correspond to places, we have “map of”, “school district”, and “union station”. This was what we were looking for – terms which are important to users when they search for places. The fact that we managed to find them is encouraging, but the fact that they get outnumbered by other places is not so. Perhaps in a future endeavour, we might check if a string corresponds to an entity which is of the same class it is mapping to, and ignore the mapping if that is indeed the case, to solve this problem. Even though the rest of the green nodes are strings, we are in a similar situation as we had in the entity-to-entity mapping where the mapping is too specific to be generalized. Again, this warrants further investigation, as simply omitting all places would get rid of the results “school district” and “union station” as well.

Looking at further nodes down the chain (2 nodes from root):

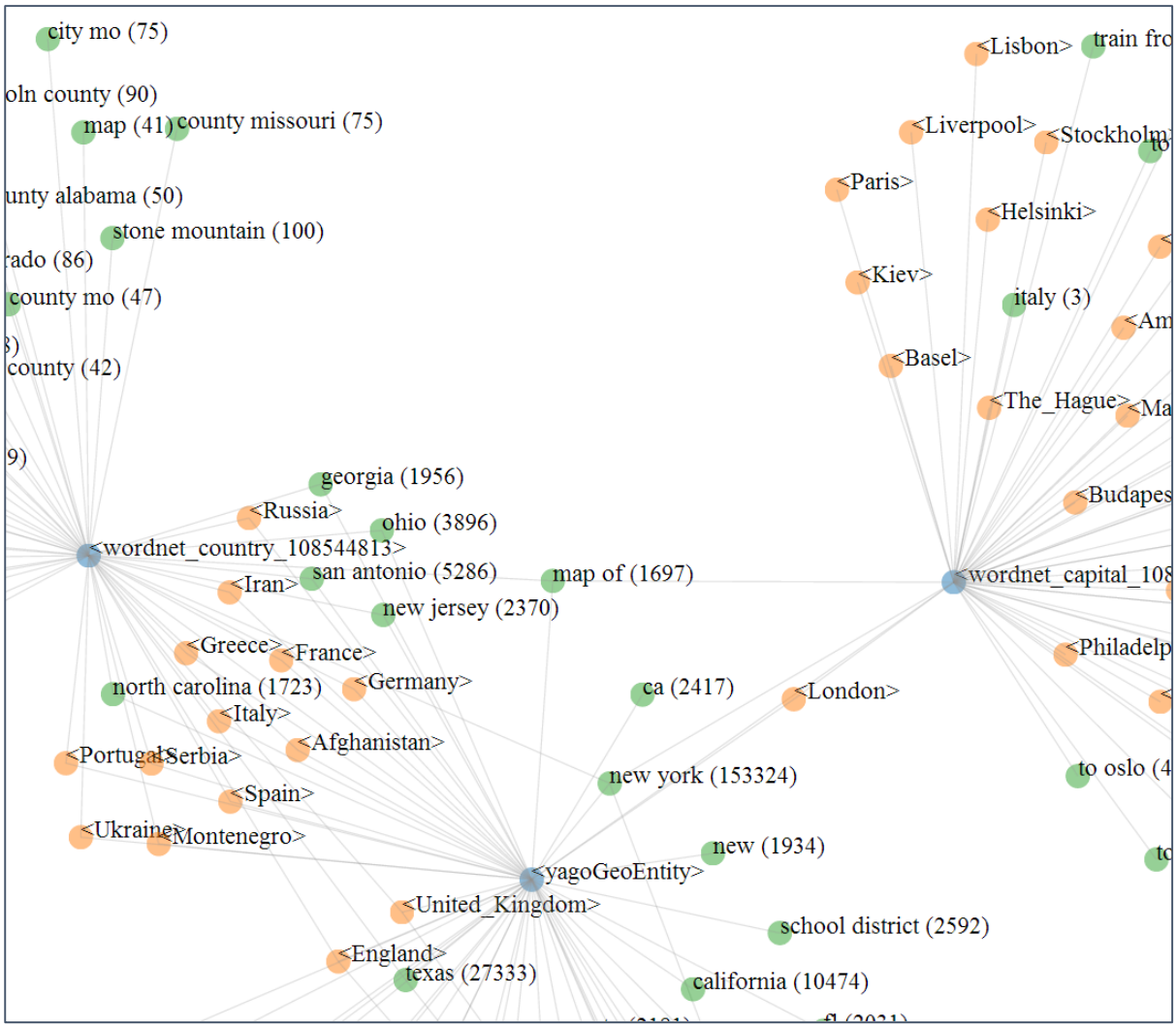


Fig 4.1.3 Order 2 nodes, children of <yagoGeoEntity>. Nodes beyond borders are unremarkable.

We can see that the same 100x decrease in number of mappings for nodes mapping to the second order classes. Although the screenshot does not have enough room to show all the results, the majority of strings mapping to the second order nodes correspond to specific places as well, but we see that at the center of the screenshot, “map of” maps to all 3 classes, and we have another green “map” node mapping to <wordnet_country> at the top left. This suggests that maps are indeed what users look for when they look for locations.

However, the overriding conclusion is this – even with 35 million search strings, we do not have enough to perform much aggregation beyond the first one or two layers of nodes below root, as we cannot distinguish the important mappings apart from noise. Considering that Yago classes can nest up to 6 to 7 deep, with a 100x decrease per node away from root, this is a trillion-fold decrease from root to a 6th order node. Since we had 35 million strings to start with, and we ended up with mappings of about 100k per string at the root node, this means we need a staggering 350 trillion strings to get reliable mappings near the bottom of the tree.

With Google’s 3.5 billion searches per day, this means they need 100k days (277 years) to get enough mappings for a 6th order node, but given the 100x decrease, this also means they only need 3 years for a 5th order node, and just 10 days for a 4th order. Classes at the bottom of the hierarchy

are usually too specific to be generalized anyway, so we strongly believe that even with this crude implementation that we came up with over three months, this method of mapping is viable – we just need a lot more strings and search sessions to work with.

4.2 Metrics

AOL Logs total size	2.1Gb
No. of search strings in AOL logs	36, 389, 567
No. of sessions (TS-26, S=0.3, stemmed)	18, 290, 428
No. of "useful" sessions	2, 417, 219
No. of "semantically-augmented" sessions (Threshold 8.0)	917, 509

The above table shows the relative sizes of the data as we filter out unusable portions of the AOL query logs. Starting from 36 million query strings, our session segmenter produced 18 million sessions based on a 26 minutes time threshold, a similarity score of 0.3, and with the Porter-Stemmer algorithm part active, so that makes the average session 2 strings long.

"Useful" sessions are sessions which contain at least one string which maps to at least one Yago entity. We found that only about 1 out of 9 sessions have that information. "Semantically-augmented" sessions are sessions which contain at least two strings mapping to their respective entities, which, when run through the mapper, give at least two entities which share common classes at a threshold score of 8. In other words, these sessions are those which the mapper has enough information to attempt to determine the most likely semantic meaning of the strings.

4.3 Information loss from mapper / role of Freebase

We have discussed the ramifications of running the mapper module when we discussed the module's details. Since it is only needed when we use Yago and not Freebase, using Freebase instead would likely alleviate the results skew which is caused by running the mapper. However, Freebase's mappings are static – it does not consider other strings in conjunction with the current one being considered. Hence, since Freebase decided that, on average, users querying for "java" means the "programming language" sense of the word, even when confronted with a session "travel to java", Freebase would map travel to the programming language. We believe a happy medium can be found by using both Yago and Freebase, but time did not allow us further investigation along this line of thought.

On the other hand, if we had an NLP module in between the search sessions and our clusterer / aggregator instead of the relatively simple mapper, we would resolve most of this issue even when using only Yago. This is further discussed in the Future Work section.

5 Conclusion and Future Work

5.1 Conclusion

We've shown that string to class aggregation compares favourably to both string to string (naïve related searches) and entity to entity mapping (which disallows aggregation).

We've also demonstrated that the 36 million queries contained within the AOL logs were only enough to get us results up to three nodes down, in the most popular search categories. Despite the

lack of data, we managed to show that about 10% of the string nodes mapping to any class except root contains the useful information we want (e.g. “map of”, “union station”, “school district” mapping to <yagoGeoEntity>) alongside other strings which correspond to entities belonging to that class. As for the method to separate the useful data out of these other strings, that is outside of the scope of the project.

Even if the number of queries required for our method to work was discouraging, we’ve managed to show that for the volume of searches modern search engine providers get, they would be able to get the most useful results within several days to three years, depending on how deep the hierarchy they want to go.

Tying the strings to the Yago class structure means we can easily aggregate these results by traversing up the class chain. Since was the goal of the project, we conclude that the research was very successful.

5.2 Future work

During several stages of the project, we considered using an NLP module to parse the search strings into syntax trees. Using NLP would enable a more accurate determination of which entity to actually map to a string, assuming correct sentence structure. We considered the possibility of constructing an incorrect syntax tree with unstructured sentences, or possibly a no-parse output where we have to handle separately. Due to the limited time we have, we left the NLP parts for future work. We did not investigate the literature behind NLP, as implementing our own NLP module is far beyond the scope of this project.

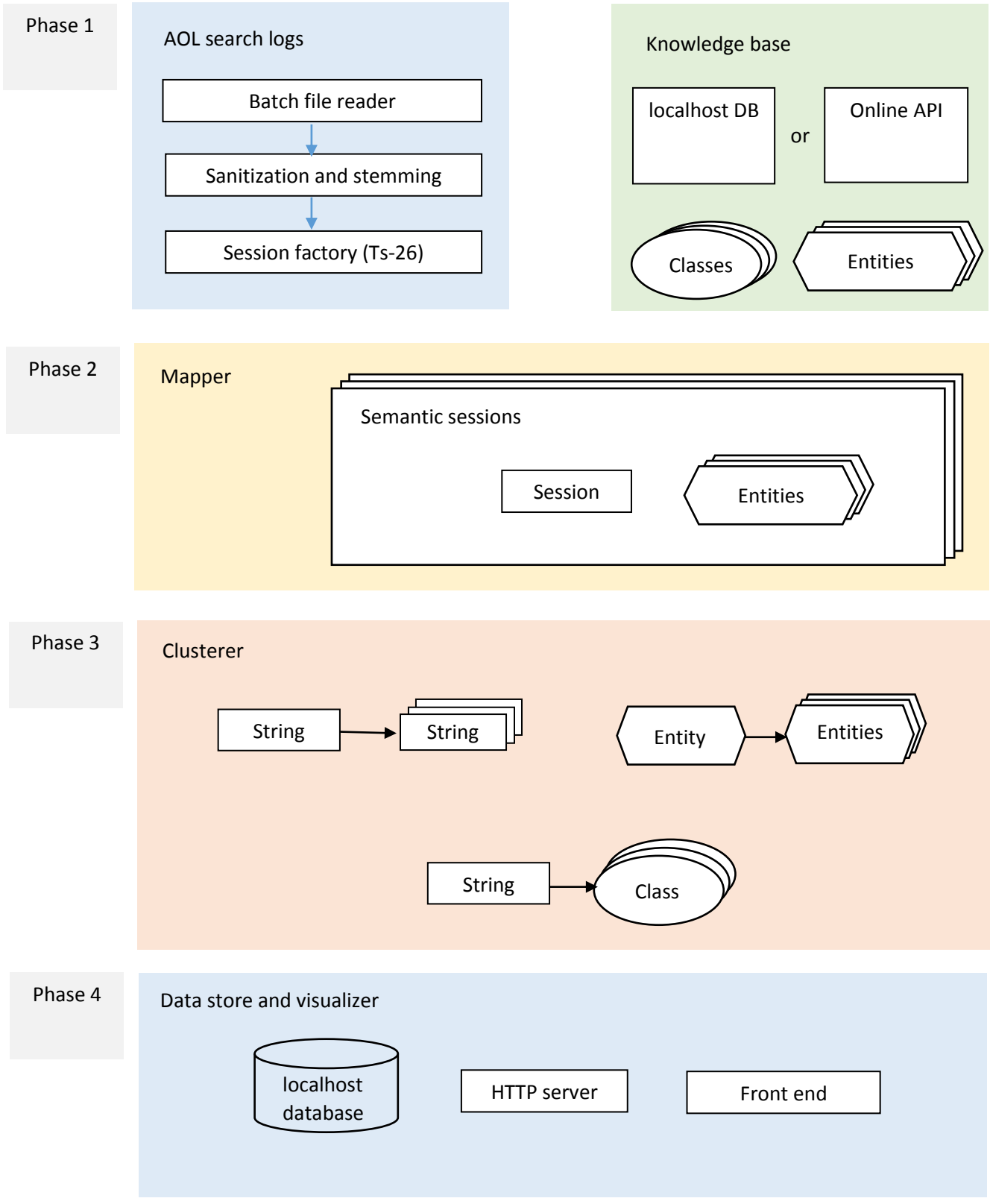
To solve the remainder of the problem, we propose the following outline:

First, we need to parse the search string using an NLP processor in order to determine the grammatical structure of the query. For instance, the string “travel to Australia” needs to be deconstructed as an understanding along the lines of “Australia has attribute ‘place’”, “Travel is an action that needs an entity which has attribute ‘place’”. Our project treats all strings as equals, and is not suited for this task. This attribute-related information can be retrieved from knowledge bases, once we have determined the proper entity which the search string maps to; this mapping has been successfully demonstrated by our project.

Once we have such an understanding, we determine “Australia” to be the primary entity and “travel” to be the subordinate, so we can display entities which are mapped to “Australia” which are applicable to “travel”, in other words, places in Australia. We can also display entities most strongly mapped to “travel”, which might also be places, or we might purposefully omit these and display non-places, which might get us travel-related services etc. All of this data is available from our clusterer’s output. Although our results set contains only a handful of the strongest mappings, as long as there is a sufficiently robust and large storage capacity, even extremely weak entity-to-entity mappings can be stored and considered, although there still remains the problem of an n x n combinatorial explosion when generating the clusters and then when considering the entities within a cluster.

6 Appendix

6.1 Project components and dependencies



6.2 Team member responsibilities and work distribution

6.2.1 Initial (projected)

	Li Quan Khoo	Horace Li	Bartosz Adamczyk	Samson Zou
AOL logs	✓	✓	✓	✓
Knowledge base	Yago	Freebase	OpenCyc	Freebase
Mapper	✓			
Clusterer	✓			
Visualizer	✓			

6.2.2 Final

	Li Quan Khoo	Horace Li	Bartosz Adamczyk	Samson Zou*
AOL logs	✓		✓	
Knowledge base	Yago	Freebase (before half term)	OpenCyc (before half term) Freebase (after half term)	
Mapper	✓			
Clusterer	✓			
Visualizer	✓	✓		

*Samson did not produce any form of contribution to the project throughout the entire term. All the other team members and the tutor were aware of this.

6.3 Stop words

Definitions of stop words vary very widely depending on context and languages in use. For simplicity, we settled on a short list optimized for English.

```
i
a
about
an
are
as
at
be
by
com
for
from
how
in
is
it
of
on
or
that
the
this
to
was
what
when
where
who
will
with
the
www
```

End of document