Spring 5-21-2015

CONTEXT-BASED AUTOSUGGEST ON GRAPH DATA

Hai Nguyen
San Jose State University

CONTEXT-BASED AUTOSUGGEST ON GRAPH DATA

A Thesis

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Hai H. Nguyen

May 2015

The Designated Thesis Committee Approves the Thesis Titled

CONTEXT-BASED AUTOSUGGEST ON GRAPH DATA

by

Hai H. Nguyen

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2015

Dr. Thanh Tran, Department of Computer Science Date

Dr. Chris Pollett, Department of Computer Science Date

Dr. Suneuy Kim, Department of Computer Science Date

ABSTRACT

CONTEXT-BASED AUTOSUGGEST ON GRAPH DATA

by Hai H. Nguyen

Autosuggest is an important feature in any search applications. Currently, most applications only suggest a single term based on how frequent that term appears in the indexed documents or how often it is searched upon. These approaches might not provide the most relevant suggestions because users often enter a series of related query terms to answer a question they have in mind. In this project, we implemented the Smart Solr Suggester plugin using a context-based approach that takes into account the relationships among search keywords. In particular, we used the keywords that the user has chosen so far in the search text box as the context to autosuggest their next incomplete keyword. This context-based approach uses the relationships between entities in the graph data that the user is searching on and therefore would provide more meaningful suggestions.

ACKNOWLEDGEMENTS

I would like to thank my project advisor, Dr. Thanh Tran, for introducing me to this interesting project that allows me to get to know more about the field of information retrieval and to work on open-source software such as Solr. In addition to constantly providing me with insightful feedbacks, Dr. Tran's knowledge and experience in Big Data have encouraged me to go beyond what I myself could imagine. I am very grateful for all of that.

I also would like to thank Dr. Chris Pollett and Dr. Suneuy Kim for spending your valuable time reviewing my report and being my committee members.

Finally, I would like to thank my parents for giving me the opportunity to study in the U.S. and for their continuing support and encouragement throughout my studies here. They have been my great source of inspiration.

Table of Contents

1. Introduction **9**
1.1 An Overview about Graph Data and RDF *10*

2. Related Works **12**

3. Problem Definition, Existing Solution, and Proposed Solution **14**

4. Context-based Autosuggest High-level View Implementation **16**

5. An Overview about Solr and its Existing Suggester **17**
5.1 Solr Document *17*
5.2 Solr Inverted Index *18*
5.3 Indexing *19*
5.4 Querying *21*
5.5 Solr Existing Suggester *23*

6. Obtain, Process, Writing schema and Index Graph Data **24**
6.1 Obtaining Data From Dbpedia *24*
6.2 Write schema.xml for the obtained Graph Data *26*
6.3 Transform RDF XML Documents to Solr Documents *27*

7. Smart Solr Suggester Implementation **29**
7.1 How to Suggest Meaningful Second Search Keywords *29*
7.2 Dictionary Implementation *31*
7.2.1 SmartDocumentDictionary - Single Solr Instance 32
7.2.2 SmartDocumentDictionary - Cluster Mode 33
7.3 Lookup Implementation *34*
7.3.1 Comparison between AnalyzingLookup and FSTLookup 35

8. Results **36**

9. Conclusion and Future Work **39**

List of Figures

Figure 1: Autosuggest Timeline 9
Figure 2: Informal Graph of Triples 11
Figure 3: Example Graph Data 13
Figure 4: AGGREGO SEARCH Autocomplete Grammar 13
Figure 5: Solr Example Document 18
Figure 6: Inverted index - a key data structure supporting information retrieval 19
Figure 7: Main Components of Solr 4 20
Figure 8: An example Search Query GET Request 21
Figure 9: An example of /select handler configuration 22

List of Tables

Table 1: Indexing and Building Suggestion Index Time 37
Table 2: Single Term Lookup Time (in milliseconds)..... 38
Table 3: Two-term Lookup Time (in milliseconds) 38

1. Introduction

Inspired by Dr. Thanh Tran's paper about query rewriting on graph data [1], this project aims to implement an autosuggest feature that provides meaningful keyword suggestions for users searching for entities on graph data. Currently, as Figure 1 demonstrates, Solr, a scalable search engine optimized to handle a large amount of text data [3], can only suggest a single term based on some predefined weight or alphabetical order by default. However, users normally input multiple terms into the search text box to find a specific entity. For example, a user searching for George Lucas who directs the Star Wars movie is likely to type "George Lucas Star Wars" into the search text box. Autosuggestions for the first term the user types in vary by applications, depending on how they rank terms. After the user has selected the first suggested search keyword, autosuggesting semantically meaningful second search keywords by using the first one as a context is the ultimate goal of this project.

Figure 1 also shows that the suggestion list keeps changing as the user continuously adds more characters to the search text box. Therefore, in addition to providing meaningful suggestions, this implementation also aims to return results fast enough to keep up with the user's input.

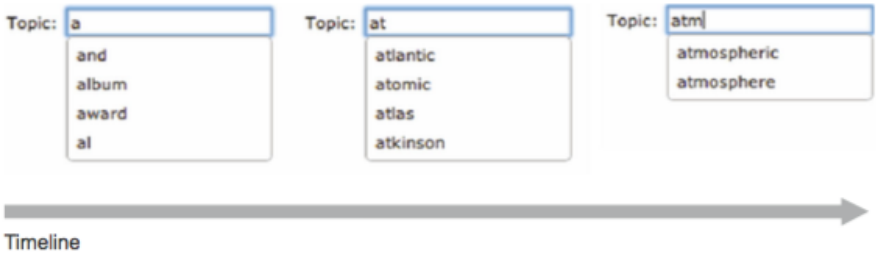


Figure 1: Autosuggest Timeline

To satisfy those two requirements, we decided to implement this autosuggest feature, which we call *Smart Solr Suggester*, as a plugin to the Solr search engine. We then used a large volume of graph data obtained from http://dbpedia.org/ to test the performance of this implementation. The next two subsections give more details about graph data.

1.1 An Overview about Graph Data and RDF

Before discussing about graph data and RDF, it is important to talk about the Semantic Web. Everyday we come across hundreds, if not, thousands of pages connected via hyperlinks. At that level, the web is constructed by interconnected documents. And the Semantic Web, a term coined by the World Wide Web Inventor Tim Berners-Lee, is an effort led by W3C and many organizations to make the web become "a web of data." The Semantic Web achieves that goal by letting different applications share and reuse data using a common framework [4]. That means there will be relationships among pieces of data just like the way documents are connected with each other. Data in the Semantic Web can be accessed using the general Web architecture. That is using the URI to define and access resources. There is a wide range of applications that Semantic Web technologies can be used on, and resource discovery is one of them. In resource discovery, one can use Semantic Web technologies to help improve search engine capabilities, and that is the area where this project wants to contribute to.

Graph data (also referred to as Linked Data) is the content of the Semantic Web, and RDF (Resource Description Framework) is a framework for describing information about resources in graph data. Resources can be anything from documents to people, physical objects to abstract concepts [5]. Each resource is uniquely defined by a URI (Uniform resource identifier). For example, the URI http://www.example.com/bob#me can be used to provide data about Bob. Retrieving data from that URI tells us facts about Bob and his relationships with other entities. Data about Bob can also include other information such as his friends, interests, etc. By providing such a common framework, RDF makes it possible for us to publish and interlink data on the Web in a way that different applications can understand and process the information, giving ways to link different graph datasets.

RDF achieves its goal by defining a data model that lets us make statements about resources. RDF statements are required to have the following structure:

```
<subject> <predicate> <object>
```

There are three kinds of nodes in an RDF graph: URIs, literals, and blank nodes, which refer to anonymous resources. The subject is a URI or a blank node. The predicate is an URI. The object is an URI, literal or a blank node. Object doesn't necessarily mean object in English because a subject in one RDF statement can be an object of a property in another. From there, we can see that an RDF statement demonstrates a relationship between two resources, the subject and the object. The predicate represents their relationship, which goes from the subject to the object. A predicate is also called a property in RDF. Since RDF statements are comprised of three elements, they are often referred to as triples.

Figure 3 shows a small graph data. Such a graph can be constructed from the following RDF statements:

"<Bob> <is a> <person>.
<Bob> <is a friend of> <Alice>.
<Bob> <is born on> <the 4th of July 1990>.
<Bob> <is interested in> <the Mona Lisa>.
<the Mona Lisa> <was created by> <Leonardo da Vinci>.
<the video 'La Jocode a Washington'> <is about> <the Mona Lisa>." [5]

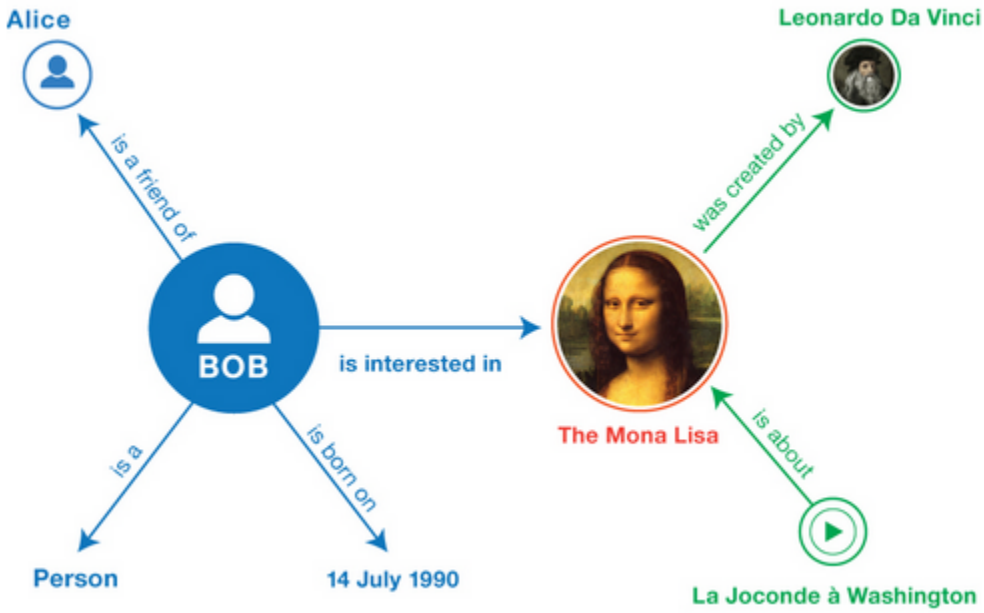


Figure 2: Informal Graph of Triples

A data graph includes nodes and arcs. Nodes represent the subjects and objects of the triples, and arcs represent the relationships/predicates. One can use graph data query language such as SPARQL (Simple Protocol and RDF Query Language) to retrieve and manipulate data stored in RDF format. For example, the following SPARQL query can be used to list all episodes of "Game of Thrones" on HBO ordered by airdate.

```
SELECT *
 WHERE
 {
   ?e <http://dbpedia.org/ontology/series>     <http://dbpedia.org/resource/Game_Of_Thrones> .
   ?e <http://dbpedia.org/ontology/releaseDate>    ?date .
   ?e <http://dbpedia.org/ontology/episodeNumber>  ?number .
   ?e <http://dbpedia.org/ontology/seasonNumber>   ?season
 }
 ORDER BY DESC(?date)
```

Though SPARQL is powerful, an average user is unlikely to use it to retrieve data. He/she would prefer to enter some keywords into a simple search text box and expect the search engine to return the piece of information they're looking for. In this project, we tested the *Smart Solr Suggester* plugin against a portion of the DBpedia dataset, a large multi-domain ontology that has been derived from the Wikipedia.

2. Related Works

In his paper on query rewriting [1], Dr. Tran and his colleagues tackled the problem of rewriting keyword search queries on graph data. For example, using the graph in Figure 4, the original keyword query "Publication John McCarty Tuning Award" can be rewritten to the following possible queries:

"Article John McCarthy Turing Award",
"Article John McCarthy Tuning Award",
"Article John McCarty Turing Award", and
"Article John McCarty Tuning Award" [1]

Out of those possible query rewrites, there's one query matching a connected graph, which connects three nodes *Article, John McCarthy,* and *Turing Award.* That is the query "Article John McCarthy Turing Award," so it becomes the best candidate query.

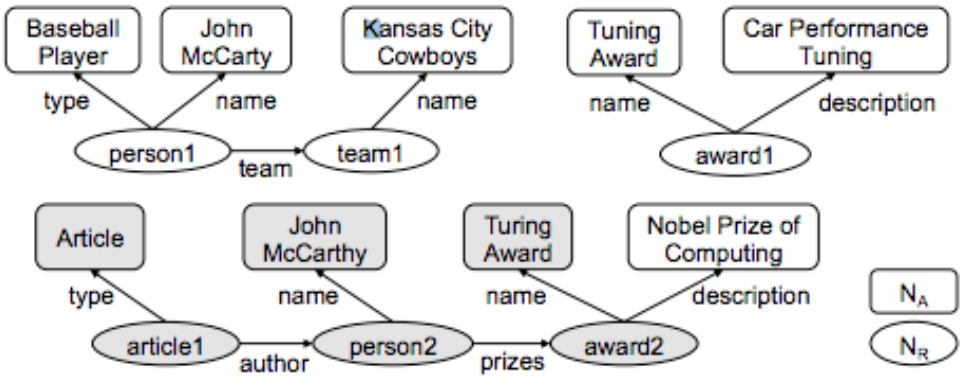


Figure 3: Example Graph Data

Instead of taking into account "all possible segments of a (sub-) query rewrite" like other approaches, they used a noble probabilistic model that computes query rewrite scores that focuses only on "the previously observed context" [1]. They showed that their approach performs 3-4 times faster when testing on the IMDb dataset and 2 times faster when testing on the Wikipedia dataset than the existing solutions in query rewriting. In addition, it also improves keyword search on large datasets, producing 2-3 times faster keyword search performance together with higher precision and recall of keyword search results compared to existing methods [1].

Another example that validates the use of previously chosen search keyword as a context is demonstrated in AGGREGO SEARCH [2]. In this work, the authors utilized an autocomplete strategy that suggests completions relevant not only to the first letters typed by the user but also to the structure of the query whose construction is in progress. Their autocomplete algorithm only suggests keywords that strictly follow the grammar in Figure 5.

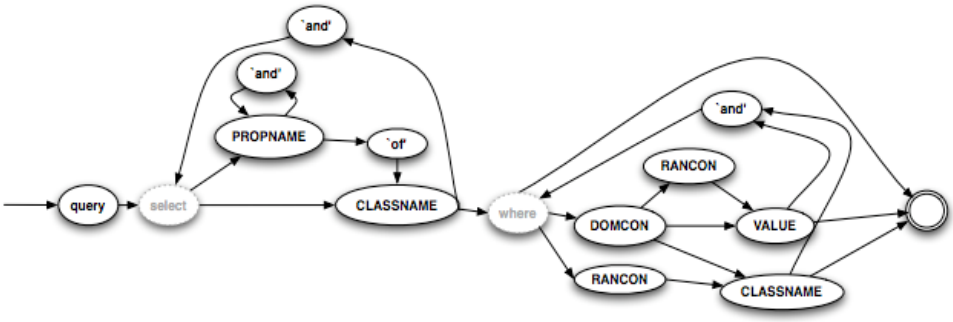
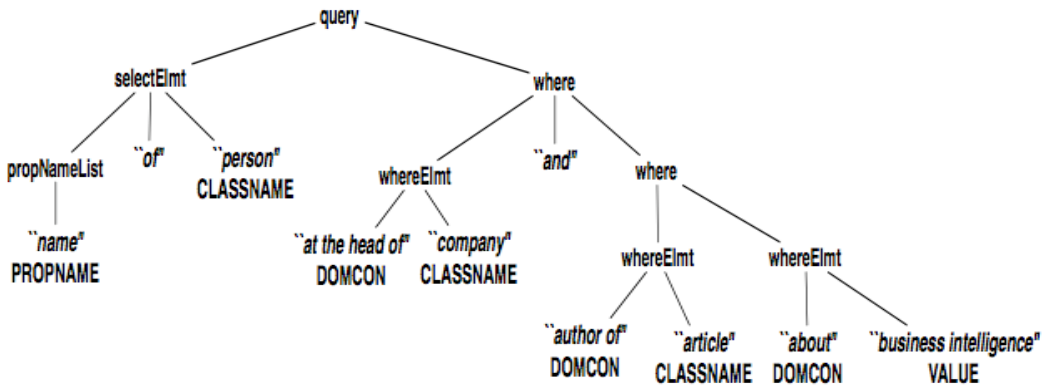


Figure 4: AGGREGO SEARCH Autocomplete Grammar

An example of a query constructed by such grammar looks as follows:

*"name **of** person **at the head of** company **and author of** article **about** 'business intelligence'"*

This query can be interpreted into the following tree:



Note that after a suggested element is selected by the first letters the user types in, the autocomplete algorithm moves to the neighbor nodes of the current one. While the autocomplete process is taking place, a SPARQL query is being constructed in the background. The final SPARQL query is sent to a triple store to retrieve the data once the user submits his query.

These two works both deal with semantic search on graph data. While the first work concerns more about labels of actual entities, ignoring properties' labels, the second work enforces the use of default connectors such as "of", "and", and treats properties' labels equally as that of actual entities. Although each solves a different problem and uses a different approach, they both have demonstrated a common theme that users typically enter search keywords related to each other. Since this is happening on graph data, it means that these search keywords/terms belong to entities that are neighbors in the graph.

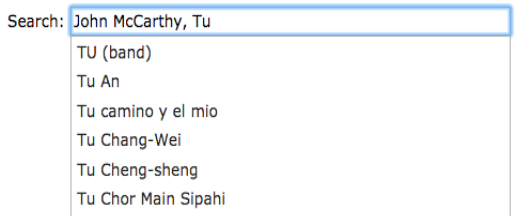
3. Problem Definition, Existing Solution, and Proposed Solution

A user searching for an entity named "John McCarthy" who won the "Turing Award" is likely to enter the following to the search textbox:

```
John McCarthy Turing Award
```

Our goal is to autosuggest search keywords that the user is likely to enter first, thereby saving the user some keystrokes and improving their search experience. Existing

autosuggest solutions rank suggested terms in alphabetical order or how often a term is searched upon. The following figure describes how autosuggest based one alphabetical order works. The keyword "Turing Award" might not even make it to list at all. These solutions work fine when autosuggesting the first search keyword ("John McCarthy"). However, for the second search keyword, can we do better?



We proposed a semantically context-based approach that takes advantage of relationships between entities in graph data. After the user has chosen the first search keyword from the suggestion list, we use that keyword as the context to autosuggest the next meaningful keywords. The context is "John McCarthy" in our example.
Assume the user continues to type in the first characters of the next search keyword. In our example, that could be "t," "tu," "tur," depending on what keyword the user is typing. Our approach is to autosuggest search keywords that start with "t," "tu," or "tur, and are in the same entities that contain the context, or in the entities that are related to the ones containing the context. In particular, each entity is represented by an RDF document. Thus, finding search keywords in the same entity means finding terms starting with "t," "tu," or "tur," in the same RDF document. RDF statements that have the value of the object as an URI represent relationships between one entity and another.

For example, suppose the user chose "John McCarthy" from the suggestion list as the first search keyword. Thus, the context is now "John McCarthy". The term "John McCarthy" appears in entities A and B. Entity A is related to entity X, and entity B is related to entity Y. Suppose the user continues to type in "t" as the first character of the second search keyword. Our approach is to make terms that begins with "t" and are in entities A, B, X, or Y appear first in the suggestion list for the second search keyword. The context and first terms in the suggestion list are more likely related to each other because they are in the same entities or in entities related to each other.

4. Context-based Autosuggest High-level View Implementation

Below is the conceptual view of context-based autosuggest:

Let
C = set of entities that contain the context
R = set of entities related to one or more entities in set C
q = first characters of the 2nd keyword that the user has entered so far
L = list of LookupResult
(each LookupResult has a **term** that begins with **q**; a set **D** of entities that contain its **term** , and a **score** originally set to 0)

for each **lookupResult** in **L**:
lookupResult.score = 10 * intersectSize(C,lookupResult.D) +
5 * intersectSize(R,lookupResult.D)
sort **L** based on the **score**
return **L** as the final suggestion list

Search keywords appearing in the same documents *containing* the context are more related to the context than search keywords in documents *related to documents* containing the context. Therefore, we add 10 points to the score of a `LookupResult` if its term and the context are the same entity and add 5 points if its term is in an entity related to an entity containing the context. We multiply those points by the sizes of the corresponding intersected sets. Since the ranking is based the relative order of scores between `LookupResults`, the absolute score of `LookupResults` does not really matter. We chose 10 and 5 so that suggested terms in the same entities with the context weigh twice as much suggested terms in related entities.

Speed is very important in autosuggestion because users tend to lose patience if it takes longer than 1 second to display suggestion results. Therefore, we aimed to implement this conceptual algorithm as efficient as possible. The time performance of this implementation depends on how fast we can retrieve `C, R, L, D,` and how efficient the `intersectSize` function is. We were looking for different tools that satisfy these requirements and came up with the conclusion that Solr is the best-fit technology. Given a term or a phrase, Solr can retrieve documents that contain that term or phrase very quickly. An entity can be represented as a Solr document with relationships preserved. Solr provides an existing suggest component that suggests search keywords in alphabetical order.

5. An Overview about Solr and its Existing Suggester

Autosuggest only makes sense when implemented as part of a search engine so that it can help improve user experience in both keyword input and relevant search results. We decided to implement our autosuggest feature in Apache Solr, a specific NoSQL technology, because it is open-source, fast, scalable, designed to deal with large indexes with millions of documents. Solr is also optimized to search on a large amount of text data. In addition to providing a basic keyword search functionality, Solr also provides a great user experience by returning results very quickly, including spelling correction when the user misspells some of the query terms, recognizing synonyms of query terms, handling phrase queries and queries containing common words such as “a,” “an,” etc. very well. It also supports geospatial queries and faceting.

Solr is a search engine built on top of Apache Lucene, a popular, open-source, Java-based information retrieval library. In short, Solr uses Lucene to provide the core data structures for indexing documents and executing searches to find documents. Lucene is a library for building and managing an inverted index, which is the main data structure for matching query terms to text-based documents. Solr indexes text-based documents and returns documents based on search queries.

5.1 Solr Document

The input to and output from Solr are called Solr documents, which contain content of different fields. Solr documents can be in XML, JSON, or CSV formats. An important restriction is that Solr documents must have a flat structure, which means we cannot store nested fields. Below is an example of a Solr document in XML format.

```
<doc>
  <field name="id">9885A004</field>
  <field name="name">Canon PowerShot SD550</field>
  <field name="manu">Canon Inc.</field>
  ...
  <field name="inStock">true</field>
</doc>
```

Figure 5: Solr Example Document

A Solr document consists of a series of fields. Each field has a name associated with it. We need to choose a field to be a unique field. For example, the `id` field in the above document can be used as a unique field like primary key in relational database. A pre-defined schema must recognize any field in a Solr document because Solr uses that schema to perform text different combinations of analysis steps before sending terms into the inverted index. Each Solr index, also called core or collection, has an `schema.xml` file that specifies field names, field types, and field type definitions for all the fields in any document. We talk more about the specific `schema.xml` that we use for this project in the next section.

5.2 Solr Inverted Index

The main data structure of a search engine is a inverted index. Below is a visual presentation of an inverted index, in which keys are the terms and values are the ids of documents containing those terms. In addition to storing the document ids, the inverted index used by Solr also stores other useful information such as term frequency and term position. The inverted index in Solr is actually built by Lucene core components. Lucene inverted index is composed of a number of files, each of which stores different information. Data such as the term being indexed, the documents that contain the term, the frequency of that term, and its position are encoded in bit values using Lucene existing conventions. Storing which documents containing the term is useful to find those documents given a search keyword. However, that's not enough because searching also involves retrieving documents in a relevant order. Therefore, information such as term

frequency is stored to compute tf-idf in relevance computation. In addition, term position is stored to support functionality such as phrase search.

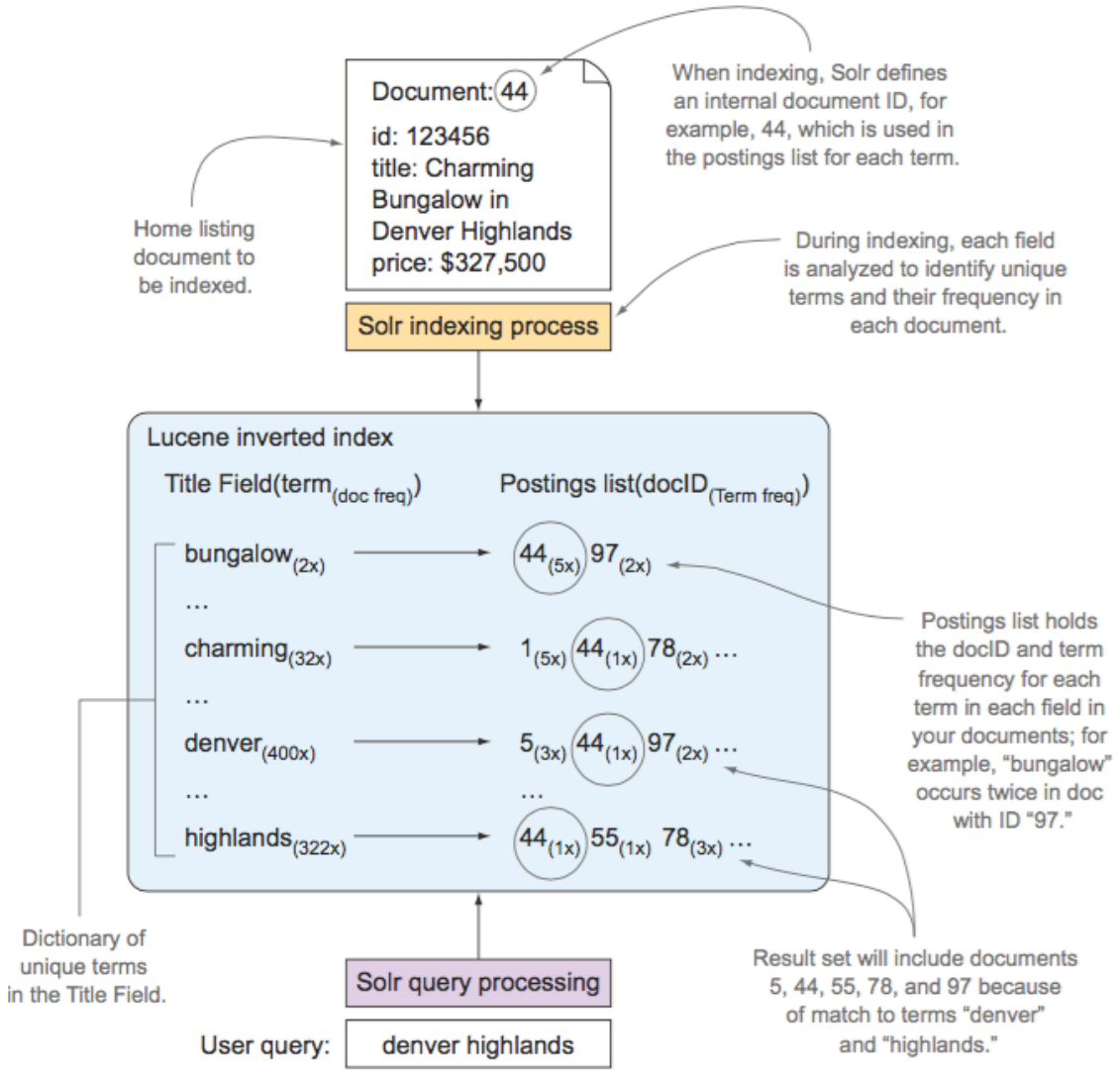


Figure 6: Inverted index - a key data structure supporting information retrieval

5.3 Indexing

Solr is a search engine, but technically, it is a Java web application that runs in any modern Java Servlet engines like Jetty or Tomcat. Figure 8 on next page describes the main components of Solr 4. Each of those components can be configured easily using the `solrconfig.xml` file. Basically, each main component consists of multiple sub-

components. We can use different kinds of sub-components and change their order to meet our needs by specifying the names of the corresponding factory classes.

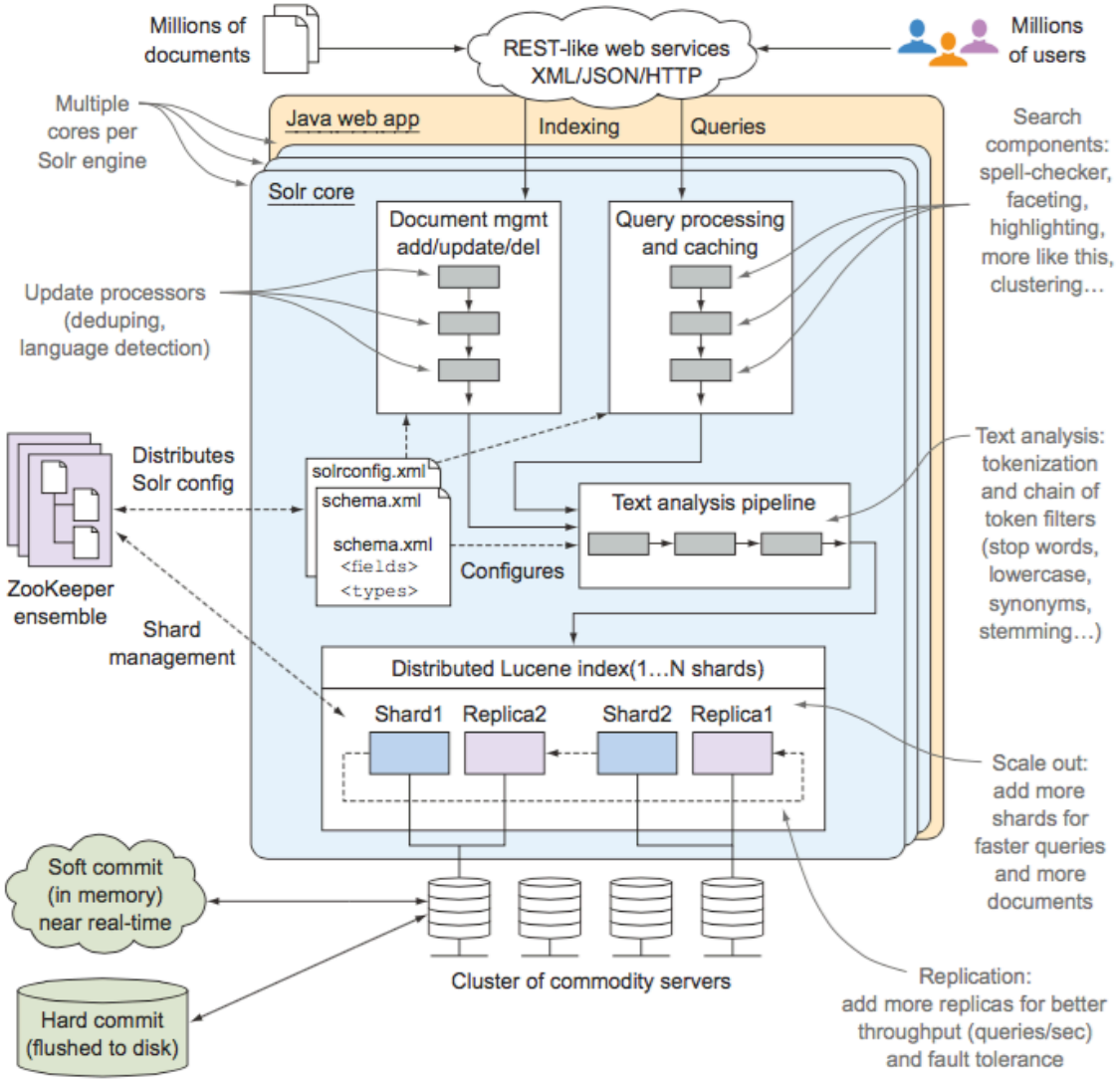


Figure 7: Main Components of Solr 4

Solr provides RESTful services built on Web standards such as XML, JSON, and HTTP, making it very convenient to access and use Solr's core services such as pagination and sorting, faceting, autosuggest, spell-checking, hit highlighting, and geospatial search. In order to perform search on Solr, we first need to index the documents we want to search from by posting them to the Solr server's `/add` or

`/update` endpoints. Requests sent to each endpoint are handled by a handler, whose components are specified in the `solrconfig.xml` file as well. In the case of indexing documents, each document will go over a processing pipeline and then be added into the inverted index, which is then saved to disk on a hard commit.

5.4 Querying

Since Solr is a search engine, it's important to understand how search queries are sent to Solr and how Solr processes these queries. To submit search queries, client applications typically send GET requests that look as follows:

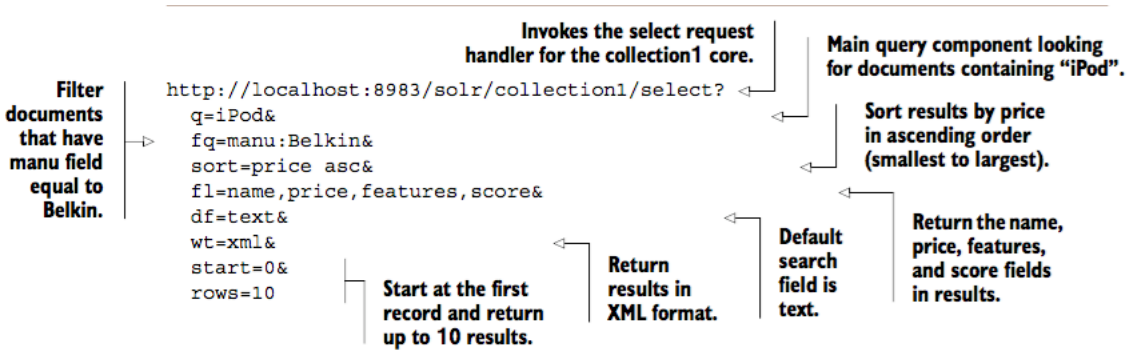


Figure 8: An example Search Query GET Request

With the above example request, the client application expects to get back a response in XML format that contains 10 documents that have the term "iPod", have the value of the "manu" field to be "Belkin". When returning the result documents, Solr returns all of the *stored fields* if we don't specify which fields we want to get back. *Stored fields* are the fields that have their values stored exactly as the original format. *Stored fields* can be indexed or not, depending on the need of the application. For instance, in the above example, we request Solr to only include the name, price, features, and scores fields in the returned documents.

Query requests are submitted to a Servlet engine such as Jetty, the default servlet that comes with the Solr distribution. Each request is routed to a specific handler, which is a Java class. Following is an example of the `/select` handler provided along with the Solr

example collection. In this case it's the `SearchHandler` class. A search handler consists of a sequence of components. In a normal search query request, the Query component is always required, and its results become input for standard components such as facet, more like this, hit highlighting, statistics and debug. In the `/select` handler example below, since we are not changing the components section, it uses the default search components in the `SearchHandler` class.

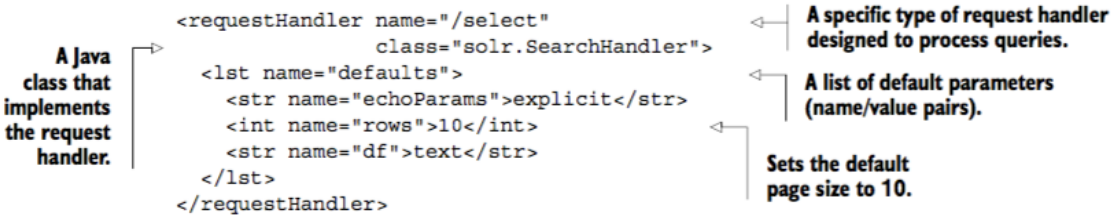
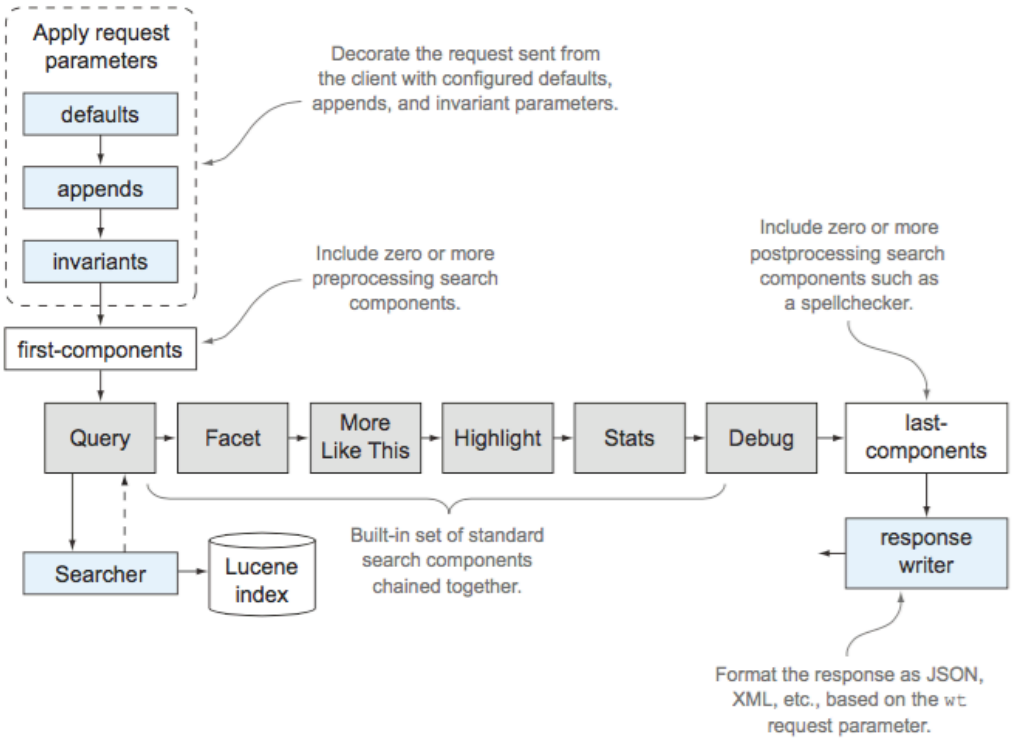


Figure 9: An example of /select handler configuration



Query parameters from the query requests are first extracted and used to create a list request parameters along with the default parameters (used these values when the client application doesn't set them), appends parameters (these values are appended to the final

parameter list), and invariant parameters (these values are always used and will overwrite the values sent from client applications.) After obtaining the final search request parameters, the query is then passed to the Query component, which is always required. The Query component has access the `SolrIndexSearcher`. Any Solr instance always has one and only one active `SolrIndexSearcher` instance, which has a snapshot of the Lucene inverted index. When new documents are added into Solr, they are not immediately include in the active `SolrIndexSearcher` yet. Solr needs to open a new searcher in order to include the new updates. The process of opening a new searcher takes some time, therefore we need to be careful on how often we want to schedule a hard commit, which triggers opening a new searcher.

In addition to configuring request handlers and search components, the file `solrconfig.xml` also includes a lot other configuration options such as commit policy, caching policy, warming new index searcher policy, and so on.

5.5 Solr Existing Suggester

Solr version 4.10.3 comes with a suggest component, which can be configured as below. Requests sent to the `/suggest` end point are handled by a search handler. We modified the list of components used by this search handler. In particular, we only included a search component named "suggest" in the processing pipeline. We also set the `suggest` parameter to true to enable the suggest component. A suggest component contains a list of suggesters. Each suggester is responsible for suggesting terms from a particular field. The example below suggests terms from the `text` field.

A Solr suggester takes several configuration parameters. The three main parameters are `lookupImpl`, `dictionaryImpl`, and `field`. A `LookupImpl` component defines how terms are found in the suggestion dictionary. A `DictionaryImpl` component defines how terms are stored in the suggest dictionary. `Field` defines which field from each document to build the suggestion index from. The current implementation of Solr Suggester ranks suggested results either alphabetically or by the value of the `weightField`, and we wanted to improve that in *Smart Solr Suggester* by ranking

suggestion results by the level of meaningfulness or how close they are related to the first search keyword.

```
<requestHandler name="/suggest" class="solr.SearchHandler" startup="lazy">
  <lst name="defaults">
    <str name="suggest">true</str>
    <str name="suggest.count">100000</str>
  </lst>
  <arr name="components">
    <str>suggest</str>
  </arr>
</requestHandler>


<searchComponent name="suggest" class="solr.SmartSuggestComponent">
  <lst name="suggester">
    <str name="name">textSuggester</str>
    <str name="lookupImpl">FuzzyLookupFactory</str>
    <str name="dictionaryImpl">DocumentDictionaryFactory</str>
    <str name="field">text</str>
    <str name="weightField"></str>
    <str name="payloadField">id</str>
    <str name="suggestAnalyzerFieldType">string</str>
    <str name="buildOnCommit">true</str>
  </lst>
</searchComponent>
```

In addition to these parameters, one can also specify a parameter named, “storeDir”, which will save the suggester index to a file after it’s built. It’s highly recommended to use this parameter because it takes some time to build the suggester index. If you have to restart the Solr server but didn’t save the suggester index into a file, when the server restarts again, it will have to go through the same build process. If the suggester index was saved, the system only takes a second to reload the suggester index from the saved file.

Before discussing about the implementation of the *Smart Solr Suggester* plugin, it’s worth mentioning about the data that we used to test the performance and validity of our context-based approach. We also discuss about how processing data is done in a clever way that helps us tackle one of the issues mentioned at the end of Section 4. That is how to efficiently retrieve the set **R** of entities that are related to one or more entities in set **C**, which consists of entities that contain the context.

6. Obtain, Process, Writing schema and Index Graph Data

6.1 Obtaining Data From Dbpedia

Dbpedia.org is one of the most popular publicly available graph data providers. It has a total of almost 5 million entities including everything from people to things to abstract concepts. The data for each entity is pulled from Wikipedia. Dbpedia.org provides a RDF triples store for the public. We queried and downloaded about 1.5 million people, 74,000 movies, 20,000 films, 30,000 bands, and 3,200 American directors. Data about each entity is stored in an RDF document. Because Solr doesn't take RDF/XML format, we need to transform these RDF documents into Solr flat documents in XML.

Below is an example of a RDF document about John McCarthy, an ice hockey player from the San Jose Sharks team. The value of the `rdf:about` attribute of the `rdf:Description` element is the URI of the current entity. Sub elements of the `rdf:Description` element are the properties of this entity. A property either has a literal value (number, string, date) or has an `rdf:resource` (an entity) attribute. Strictly speaking, the value of the `rdf:resource` attribute is a URI that represents the entity that this entity is related to. In the example below, "John McCarthy" has a property named `dbpedia-owl:team` , which has the `rdf:resource` attribute equals to http://dbpedia.org/resource/San_Jose_Sharks. `dbpedia-owl` is the name space of the dbpedia ontology. Basically, we can translate this RDF triple into the following relationship: John McCarthy is in team San Jose Sharks.

```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <rdf:RDF
3    <rdf:Description rdf:about="http://dbpedia.org/resource/John_McCarthy_(ice_hockey)">
4      <rdf:type rdf:resource="http://dbpedia.org/ontology/IceHockeyPlayer" />
5      <rdf:type rdf:resource="http://schema.org/Person" />
6      <rdfs:label xml:lang="en">John McCarthy</rdfs:label>
7      <rdfs:label xml:lang="fr">John McCarthy (hockey sur glace)</rdfs:label>
8      <rdfs:label xml:lang="de">John McCarthy (Eishockeyspieler)</rdfs:label>
9      <dbpprop:birthPlace rdf:resource="http://dbpedia.org/resource/Massachusetts" />
10     <dbpprop:shortDescription xml:lang="en">American ice hockey player</dbpprop:shortDescription>
11     <dbpprop:team rdf:resource="http://dbpedia.org/resource/San_Jose_Sharks" />
12     ...
13     ...
14     ...
15     <dbpprop:prospectTeam rdf:resource="http://dbpedia.org/resource/Worcester_Sharks" />
16     <dc:description xml:lang="en">American ice hockey player</dc:description>
17     <dbpedia-owl:team rdf:resource="http://dbpedia.org/resource/San_Jose_Sharks" />
18     <foaf:isPrimaryTopicOf rdf:resource="http://en.wikipedia.org/wiki/John_McCarthy_(ice_hockey)" />
19   </rdf:Description>
20 </rdf:RDF>
```

When transforming an RDF/XML document into a Solr XML document, we need to decide what field name, field type, field value, whether the field needs to be indexed or stored or both. In addition, all fields in a Solr document needs to be recognized by the fields specified in schema.xml, and we cannot manually check and add all property names. This is the reason why we decided how `schema.xml` should look like before even thinking about how to transform the downloaded RDF documents. In addition, Solr `schema.xml` provides some neat tools that greatly support the data preprocessing tasks.

6.2 Write schema.xml for the obtained Graph Data

Below is the schema.xml that we used for the dbpedia data. The way this schema is laid out largely affects the XSLT template that we used to transform RDF XML documents to Solr documents. Note that `uri` is specified as `uniqueKey`. We use dynamic fields and copy fields in order to cover every field name possible and add some useful field names that we need to use later.

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema name="things" version="1.5">
  <!-- common fields, every doc has these -->
  <field name="_version_" type="long" indexed="true" stored="true"/>
  <field name="uri" type="string" indexed="true" stored="true" required="true" multiValued="false"/>

  <dynamicfield name="*_text" type="textSpell" indexed="true" stored="true" multiValued="true"/>
  <dynamicField name="*_resource" type="string" indexed="false" stored="true" multiValued="true"/>

  <dynamicField name="*" type="string" indexed="false" stored="true" multiValued="true"/>

  <field name="text" type="textSpell" indexed="true" stored="false" multiValued="true"/>
  <field name="relatedDocs" type="string" indexed="false" stored="true" multiValued="true"/>

  <!-- copy fields -->
  <copyField source="*_text" dest="text"/>
  <copyField source="*_resource" dest="relatedDocs"/>

  <!-- unique fields -->
  <uniqueKey>uri</uniqueKey>
  ...
  ...
</schema>
```

For dynamic fields, we decided that there are two types of field names: ending in "_text" and ending in "_resource". As the names imply, field names ending in "_text" correspond to properties that have literal value, and field names ending in "_resource"

correspond to properties that have a resource attribute. The former has field type "textSpell" and the latter has field "string". Fields with field type "textSpell" are indexed and stored, but fields with field type "string" are stored only. We don't want to modify the resource's URI in anyway, so string is the best field type to use for them. The "textSpell" field type only performs minimal text analysis: keyword tokenization and transformation to lower-case. Other field names that don't match any of these patterns are treated as string and won't be indexed. A field is indexed means that its content is analyzed and stored in the inverted index. A field is stored means that its content is stored on disk with no modification at all, and its value won't be stored into the inverted index. We index a field if we want to search upon its content, and we store a field if we really need to restore its original value.

We then added two new fields: `text` and `relatedDocs`. The `text` field is a general-purpose indexed field. Instead of having to search for every field, we copy the content from every field ending in "_text" to the `text` field. When we perform a search query, we only search against this field for simplicity. Similarly, we copy the content of every field ending in "_resource" to the `relatedDocs` field, so it contains a list of every entity/resource's URI that this entity is related to. These two fields make the retrieval of sets **C** and **R** (mentioned at the end of section 4) become instant.

6.3 Transform RDF XML Documents to Solr Documents

We wrote an XSLT template to transform these RDF documents into Solr XML documents. Each RDF/XML file might contain multiple entities. However, we are only interested in entities that have the `/rdf:RDF/rdf:Description/rdfs:label` element. The content of each entity is marked by the `/rdf:RDF/rdf:Description` element, so if it indeed has a `/rdfs:label` element, we will create a Solr document. The `@rdf:about` attribute of the `/rdf:RDF/rdf:Description` element is the URI of that entity, so we extract and use it as the URI field. Below are a snippet of the XSLT file that we used and an example of the transformed Solr XML document.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
   <xsl:template match="*|@*|text()">
      <xsl:apply-templates />
   </xsl:template>

   <xsl:template match="/">
      <xsl:for-each select="rdf:RDF/rdf:Description">
      <xsl:if test="rdfs:label">
         <doc>
            <field name="uri">
               <xsl:value-of select="@rdf:about" />
            </field>
            <xsl:apply-templates />
         </doc>
      </xsl:if>
      </xsl:for-each>
   </xsl:template>

   <xsl:template match="/rdf:RDF/rdf:Description/rdfs:label[not(@*)]|
                  /rdf:RDF/rdf:Description/rdfs:label[@xml:lang='en']">
      <field name="rdfs_label_text">
         <xsl:value-of select="." />
      </field>
   </xsl:template>
   <xsl:template match="/rdf:RDF/rdf:Description/owl:sameAs[@rdf:resource]">
      <field name="owl_sameAs_resource">
         <xsl:value-of select="@rdf:resource" />
      </field>
   </xsl:template>
   <xsl:template match="/rdf:RDF/rdf:Description/dbpedia-owl:creator[@rdf:resource]">
      <field name="dbpedia-owl_creator_resource">
         <xsl:value-of select="@rdf:resource" />
      </field>
   </xsl:template>
   ...
   ...
</xsl:stylesheet>
```

```xml
<doc>
<field name="uri">http://dbpedia.org/resource/John_McCarthy_(ice_hockey)</field>
<field name="owl_sameAs_resource">http://fr.dbpedia.org/resource/John_McCarthy_(hockey_sur_glace)</field>
<field name="owl_sameAs_resource">http://wikidata.org/entity/Q1701008</field>
<field name="owl_sameAs_resource">http://yago-knowledge.org/resource/John_McCarthy_(ice_hockey)</field>
<field name="owl_sameAs_resource">http://de.dbpedia.org/resource/John_McCarthy_(Eishockeyspieler)</field>
<field name="owl_sameAs_resource">http://dbpedia.org/resource/John_McCarthy_(ice_hockey)</field>
<field name="owl_sameAs_resource">http://sv.dbpedia.org/resource/John_McCarthy_(ishockeyspelare)</field>
<field name="owl_sameAs_resource">http://rdf.freebase.com/ns/m.09v2ylm</field>
<field name="owl_sameAs_resource">http://wikidata.dbpedia.org/resource/Q1701008</field>
<field name="rdfs_label_text">John McCarthy</field>
...
...
<field name="dbpprop_placeOfBirth_resource">http://dbpedia.org/resource/Boston</field>
<field name="dbpprop_placeOfBirth_resource">http://dbpedia.org/resource/Massachusetts</field>
<field name="dbpprop_placeOfBirth_resource">http://dbpedia.org/resource/United_States</field>
<field name="dbpprop_shortDescription_text">American ice hockey player</field>
<field name="dc_description_text">American ice hockey player</field>
<field name="foaf_givenName_text">John</field>
<field name="foaf_name_text">John McCarthy</field>
<field name="foaf_name_text">McCarthy, John</field>
<field name="dbpedia-owl_resource">http://dbpedia.org/resource/San_Jose_Sharks</field>
<field name="foaf_surname_text">McCarthy</field>
<field name="foaf_isPrimaryTopicOf_resource">http://en.wikipedia.org/wiki/John_McCarthy_(ice_hockey)</field>
</doc>
```

The transformed Solr XML documents have a flat structure as required by the Solr indexing system. There are two types of field name; one ending in "_text" and one ending

in "_resource". The ones ending in "_text" are text fields, and the ones ending in "_resource" are considered as relationship fields. For example, the field:

```
<field name="dbpedia owl_resource">http://dbpedia.org/resource/San_Jose_Sharks</field>
```

implies that the current entity, identified by the URI:

http://dbpedia.org/resource/John_McCarthy_(ice_hockey) is in team San Jose Sharks. During indexing time, Solr creates a new field named "text" and copies all fields ending with "_text" to it. In addition, Solr also creates a new field named "relatedDocs" and copies all fields ending with "_resource" to it. Therefore, the value of the "relatedDocs" field in each document contains the URIs of all of the entities related to the current entity.

7. Smart Solr Suggester Implementation

7.1 How to Suggest Meaningful Second Search Keywords

After the user has chosen the first term from the alphabetical suggestion list, the client application marks it as the `context`. When the user continues to type the first letters of the second term, the client application sends requests to Solr server. Specifically, we send the request to the `/suggest` end point in this project. In those requests, we add the `context` parameter to the query string (`&context=first_term`).

When the `context` parameter is not null, *Smart Solr Suggester* triggers the mechanism that suggests meaningful second search keywords on the top of the suggestion list. This mechanism includes two steps.

The first step, as mentioned at the end of section 4, is to retrieve sets **C** and **R** given a **context**. This step can be done very quickly because this Solr is very fast at finding documents that contain a term. In addition, when indexing the documents in the previous section, we already included a **relatedDocs** field that contains the URI of all related entities. However, this step's implementation for a single Solr instance is different from that for a cluster.

The second step is to compute the score for each lookup result from the default alphabetical suggestion list. Each of these lookup results has a set **D** that consists of the URIs of all entities containing the lookup result's term. This set exists because we encode

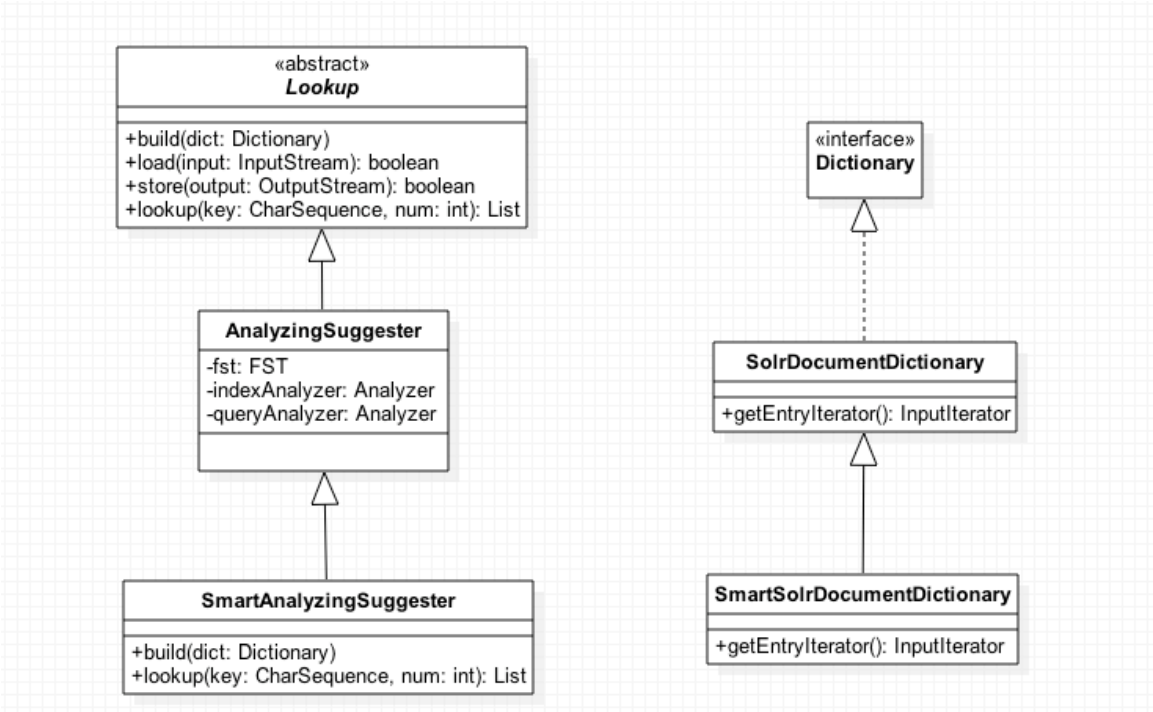
it when building the suggester index. The next subsection gives more details on how building the suggester index is done on single instance and cluster mode. Below is the snippet of code that shows where we check for the context and execute the Smart Solr Suggester.

```
1  public SuggesterResult getSuggestions(ContextSuggesterOptions options) throws IOException {
2    SuggesterResult res = new SuggesterResult();
3    List<LookupResult> suggestions =
4          ((SmartAnalyzingSuggester)lookup).lookup(options.token, null, false, options.count);
5
6    if (options.context != null) {
7      this.setContainingAndRelatedDocs(options.context.toLowerCase());
8      contextContainingDocsArr = convertIntegers(containingDocsList);
9      contextRelatedDocsArr = convertIntegers(relatedDocsList);
10
11     List<LookupResult> zeroList = new ArrayList<LookupResult>();
12     List<LookupResult> nonZeroList = new ArrayList<LookupResult>();
13
14     for (LookupResult curResult : suggestions) {
15       curResult.score = this.getScore(curResult.containingDocsBytesRef);
16       if (curResult.score == 0) {
17         zeroList.add(curResult);
18       } else if (curResult.score > 0) {
19         nonZeroList.add(curResult);
20       }
21     }
22
23     ScoreComparator scoreComparator = new ScoreComparator();
24     Collections.sort(nonZeroList, scoreComparator);
25
26     nonZeroList.addAll(zeroList);
27     suggestions = nonZeroList;
28   }
29
30   if (suggestions.size() > 25) {
31     suggestions = suggestions.subList(0, 25);
32   }
33
34   res.add(getName(), options.token.toString(), suggestions);
35   return res;
36 }
```

Note that in order to speed up the performance, we put `LookupResult` with score equal to 0 to one list and put `LookupResult` with score greater than 0 to another list. We then only sort the non-zero list and append the zero-list to it. This can be explained by the fact that the majority of `LookupResult` instances have score equal to 0, so it doesn't make much sense to compare all of them against those having nonzero score. In addition, when sending `LookupResult` to client applications, we limit number of suggested terms to 25 because users don't normally scroll down the suggestion list to find the search keyword they want to type but continue to type instead. By limiting the number of suggested terms, we save time transferring the data to client applications.

7.2 Dictionary Implementation

Dictionary is an important sub-component of the suggest component. There are a number of different dictionary implementations in Solr: `DocumentDictionary, DocumentExpressionDictionary, HighFrequencyDictionary, and FileDictionary`. For our purpose, we found that `DocumentDictionary` suits our needs the most because it provides an easy access to the term and high flexibility to encode additional information into the suggester index. `SmartDocumentDictionary` extends `DocumentDictionary`. Its responsibility is to provide the lookup with a `DocumentIterator`, which includes a current term and any other information associated with that term. In this project, we encoded the URI of the related documents as a sequence of bytes and attach them to the end of the term. Below are the class diagram and a snippet of `SmartDocumentDictionary` implementation.



```
public BytesRef next() throws IOException {
  while (currentDocId < docCount) {
    currentDocId++;
    if (liveDocs != null && !liveDocs.get(currentDocId)) {
      continue;
    }
    Document doc = reader.document(currentDocId, relevantFields);
    int[] tempContainingDocs = null;
    IndexableField fieldVal = doc.getField(field);
    if (fieldVal == null || (fieldVal.binaryValue() == null
                             && fieldVal.stringValue() == null)) {
      continue;
    }

    if (fieldVal.stringValue() != null) {
      setContainingDocs(fieldVal.stringValue().toLowerCase());
      tempContainingDocs = convertIntegers(containingDocsList);
    } else {
      return null;
    }

    tempTerm = (fieldVal.stringValue() != null) ? new BytesRef(fieldVal.stringValue())
                                                : fieldVal.binaryValue();
    curentContainingDocs = tempContainingDocs;

    return tempTerm;
  }
  return null;
}
```

Note that we set the `currentContainingDocs` list to be the set of all entities that contain the current term. The implementation for `setContainingDocs` is different between single Solr instance and cluster mode and is discussed next.

7.2.1 SmartDocumentDictionary - Single Solr Instance

For index that uses one single Solr instance, the active index searcher knows about all documents in the index, so we use it to directly query for documents that contain the current term, thereby obtaining their corresponding URIs of the related entities. These queries are really fast because it uses an internal object to execute the queries, so I/O overhead is very minimal. As a result, building the suggester index in a single Solr instance mode is faster compared to that in a cluster mode. In addition, when query for the documents that contain the current term, we query set the querying field to be “text”. As discussed in the section about data and schema.xml, “text” is the field that contains data copied from every text field in the documents. A document might have a lot of text fields, and it takes a long time to iterate over all of them to find a certain term. Therefore, by aggregating all of them into one single field, we saved the iteration time.

```
1  public void setContainingDocs(String queryTerm) throws IOException {
2    Query query = new TermQuery(new Term("text", queryTerm));
3    DocSet tempDocSet  = searcher.getDocSet(query);
4
5    containingDocsList = new ArrayList<Integer>();
6
7    int[] localContainingDocs = null;
8    Document doc = null;
9
10   if (tempDocSet instanceof BitDocSet) {
11     BitDocSet bitDocSet = (BitDocSet)tempDocSet;
12     int numDocs = bitDocSet.size();
13     localContainingDocs = new int[numDocs];
14     Iterator iter = bitDocSet.iterator();
15     int curIdx = 0;
16
17     while (iter.hasNext() && curIdx < numDocs) {
18       localContainingDocs[curIdx++] = (Integer)iter.next();
19     }
20   } else if (tempDocSet instanceof SortedIntDocSet) {
21     localContainingDocs = ((SortedIntDocSet)tempDocSet).getDocs();
22   }
23
24   for (int i = 0; i < localContainingDocs.length; i++) {
25     doc = reader.document(localContainingDocs[i], getRelevantFields(new String [] {"uri"}));
26     String uri = doc.get("uri");
27     containingDocsList.add(uri.hashCode());
28   }
29
30   Collections.sort(containingDocsList);
31 }
```

7.2.2 SmartDocumentDictionary - Cluster Mode

With a large amount of documents or entities, sometimes one Solr instance cannot handle all of the requests or even hold all of the data. As a result, Solr introduces SolrCloud, is a technology to run Solr in a cluster mode. In Solr, an index is also called a collection. A collection can be hosted on multiple shards, each of which is essentially a Solr server. When indexing a document to a Solr cluster, a document router decides which shard that document is indexed into. The document router does so by hashing the value of the unique field of that document. The hash function that the document router uses is MurmurHash 3, an efficient hash function that outputs a value within 32-bit. Each shard in the cluster is assign a range of number within 32-bit. The hash range is equally divided among shards. The document router routes a document to a shard if the hashed value of that document falls into the assigned range of this shard. Consequently, the local active `SolrIndexSearcher` in one Solr instance does not know anything about documents in other instances. Thus, we cannot use it to directly query for all documents that contain a particular term. To overcome this problem, we actually sent an HTTP request to the `/select` handler to query across all shards and find documents that

contain the current term when building the suggestion index. Though it only takes a millisecond or two to complete a request, this becomes a huge I/O overhead when have to build the suggestion index for millions of documents. The size of the overhead is proportional to the size of the number of documents in the inverted index. Below is a snippet of code that shows how we handled this situation.

```
public void setContainingDocs(String queryTerm) throws IOException {
  containingDocsList = new ArrayList<Integer>();
  String url = "http://localhost:8983/solr/collection1/select";
  String charset = "UTF-8";
  String q = "\""+ queryTerm + "\"";
  String fl = "uri";
  String wt = "csv";

  String queryString = String.format("q=%s&fl=%s&wt=%s",
                       URLEncoder.encode(q, charset),
                       URLEncoder.encode(fl, charset),
                       URLEncoder.encode(wt, charset));

  String finalUrl = url + "?" + queryString;

  URLConnection connection = new URL(finalUrl).openConnection();
  connection.setRequestProperty("Accept-Charset", charset);
  BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInputStream()));
  String line;
  while ((line = in.readLine()) != null) {
     if (!line.equals("uri")) {
        containingDocsList.add(line.hashCode());
     }
  }
  in.close();
  Collections.sort(containingDocsList);
}
```

7.3 Lookup Implementation

Lookup is another crucial sub-component of the suggest component. There are a number of existing lookup implementations: `AnalyzingLookup`, `FuzzyLookup`, `AnalyzingInfixLookup`, `BlendedInfixLookup` (an extension of `AnalyzingInfixLookup`), `FSTLookup` (an automaton-based lookup), and `TSTLookup` (a simple compact ternary trie based lookup). Solr documentation claims that `FSTLookup` is the fastest implementation and consumes the least amount of memory. However, we found that `AnalyzingLookup` performance is not too far behind, and it also provides some useful configurations for indexing and querying text analysis. Below is a quick comparison between the two implementations.

7.3.1 Comparison between AnalyzingLookup and FSTLookup

`FSTLookup` is said to provide the lowest memory cost. However, it comes as a cost that `FSTLookup` doesn't analyze the term or the query string, so the term "John McCarthy" will be indexed exactly as it is. Thus, a query for "jo" would not return that term. For each term, `FSTLookup` constructs an `int[]` that starts with the weight of that term and UTF-8 encoding for each character of the term. In our example, since we don't specify a numeric field to be used as the weight, it's always 0. So the term "John McCarthy" will be transformed to the following

`int[] = [0,4a,6f,68,6e,20,4d,63,43,61,72,74,68,79]`

`FSTLookup` then adds that `int[]` to a finite state automaton builder. Each term is added the same, and eventually `FSTLookup` has an automaton that represents the suggester index. Since we're not utilizing the weights, it doesn't matter much in this case. But the weights are actually arranged into buckets so that the `FSTLookup` would look for the arc that has highest weight. If all weights are the same (like in this case), it will fall back to have suggest results sorted alphabetically. The automaton that the `FSTLookup` built represents each term's `int[]` in its graph, with each int represents a node. There's an arc between two nodes if the two nodes are in the same `int[].`

`FSTLookup` saves memory because it doesn't store any output or value in the nodes. During lookup time, the automaton traverses its graph, and if a path matches the query's characters, then it collects all the paths from the sub-graph at the point to become suggestions. `FSTLookup` then simply translates those UTF-8 `int[]` back to String, and those are the suggestion results.

`AnalyzingSuggester` uses the same automaton builder and lookup mechanism as `FSTLookup` except that it does store value in the leaf's nodes. For the term "John McCarthy", if we choose to apply a lowercase filter in the index analyzer, we'll have the analyzed form as an

`int[] = [6a,6f,68,6e,20,6d,63,63,61,72,74,68,79]` (Note the difference between this array and the one above).

The items in the analyzed form will be used as the nodes to build the automaton, and the value of the leaf's node for that sequence is a `byte[]`, which encodes the weight, payload, and surface form. Surface form is the original text before it was analyzed ("John McCarthy" in this case). The idea is that a suggest query for "jo" would return terms start with "Jo" like "John".

Overall, these two FST (Finite State Transducer) based suggesters are the foundation for other lookup implementations such as `FuzzyLookup`, `AnalyzingInfixLookup`, and `BlendedInfixLookup` to extend from. `AnalyzingSuggester` and `FSTLookup` should have roughly the same performance because they use the same indexing and lookup mechanism. Using `AnalyzingSuggester` might cost us more memory, but that is worth it because we can suggest text in their original forms while suggest queries don't have to match the original forms. Therefore, we decided to extend `AnalyzingSuggester` to become `SmartAnalyzingSuggester` as a lookup implementation for our *Solr Suggester Plugin*.

8. Results

Below are the snapshots that show how Smart Solr Suggester works. All of them indicate that the user has chosen "John McCarthy" as the first search keyword. In the first snapshot, the user continues typing the letter "t". Smart Solr Suggester automatically suggests "Turing Award" at the top of the list followed by alphabetical order terms. That's because there is a relation in one of the "John McCarthy" entity that links to an entity having the term "Turing Award." Similarly, in the second snapshot, Smart Solr Suggester automatically displays "Lisp (programming language)" on the top of the list when the user enters "l" for the second search keyword. The last snapshot shows similar result when the user enters "s" as the first character of the second search keyword. It suggested "San Jose Sharks" because that term is in an entity (San Jose Sharks hockey team) that is related to the entity containing the context "John McCarthy."

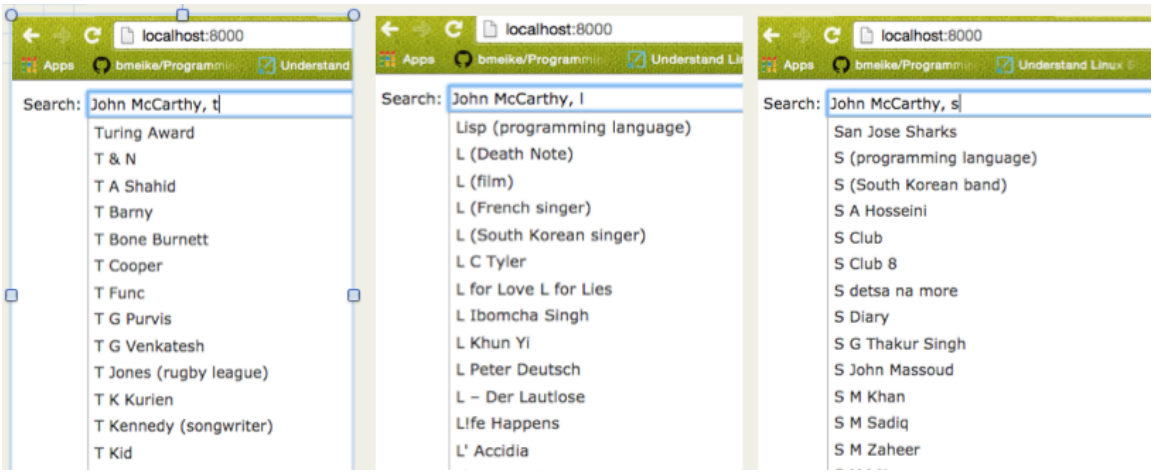


Table 1 below shows the average time to index and build the suggestion index for 1,253,425 documents. If we computed the indexing time alone, the cluster mode might takes less time to index because each shard only needs to handle a half amount of the documents. However, it takes roughly 5 times longer to build the suggestion index in the cluster mode because we need to query across all shards to find the documents containing the current term. This cluster consists of only 2 shards hosted on the same machine, so the communication time is reduced as much as possible. However, in a situation where we have more shards and they are located far away from each other, this could be an issue. Therefore, we hope to find a better solution for building the suggestion index in the cluster mode.

Number of documents	Single Solr Instance	Cluster with 2 shards
1,253,425	10 minutes 12 seconds	49 minutes 18 seconds

Table 1: Indexing and Building Suggestion Index Time

Table 2 below shows the time it takes to retrieve the suggestion list for the first search keyword. We noticed that the cluster mode takes a little longer to retrieve the suggestion list. That is expected because after finding all of the suggested terms, each shard has to send their results to the shard where the original request was sent to, and then results from all these shards are merged to create the final suggestion list. The communication time between shards and the merging time make the cluster mode slower.

Term	Single Solr Instance		Cluster with 2 shards	
	Lookup time	Total time	Lookup time	Total time
“g”	173	184	188	200
“ge”	61	66	74	85
“geo”	40	45	61	73
“geor”	36	43	51	55
“georg”	43	55	66	78
“george”	35	40	55	58
“george l”	4	8	15	27
“george lu”	3	7	18	23

Table 2: Single Term Lookup Time (in milliseconds)

Finally, table 3 shows time performance of Smart Solr Suggester between a single Solr instance and a Solr cluster of 2 shards. The time measured here is how long it takes to retrieve the suggestion list after the user has entered the first characters of the second search keyword. In this case, the cluster seems to outperform the single Solr instance. That is because the single Solr instance has to compute the score for all of the terms, rearrange, and sort them. However, in the cluster mode, that work is split into half for each shard to handle. Therefore, even though there’s communication and merging overheads in the cluster mode, those are compensated by the reduced amount of work that each shard has to do.

Two terms	Single Solr Instance		Cluster with 2 shards	
“context, first letters”	Lookup time	Total time	Lookup time	Total time
“George Lucas, s” → Star Wars (known for)	275	287	265	268
“George Lucas, st” → Star Wars	51	57	67	71
“John Lasseter, c”→ Cars (film)	201	205	157	161
“John McCarthy, t” → Turing Award (award)	222	235	144	148
“John McCarthy, l”→ Lisp (known for)	165	177	149	161
“John McCarthy, a” → Artificial Intelligence	302	315	205	208
“John McCarthy, s” → San Jose Sharks (team)	308	313	179	182
“Steve McQueen, t” → The Sand Pebble	339	351	160	163
“Steve McQueen, 1” → 12 Years a Slave	6	18	17	21
“James Cameron, t” → The Abyss, Titanic	236	248	172	184
“John Lasseter, t” → Toy Story, Tin Toy	210	215	150	154

Table 3: Two-term Lookup Time (in milliseconds)

9. Conclusion and Future Work

In this project, we attempted to implement and test the context-based autosuggest approach as a suggest plugin in Solr. With over 1 million documents, the Smart Solr Suggester seems to perform reasonably well in terms of speed. We were indeed able to retrieve more meaningful second search keywords using the first keyword as a context. There are still some areas in Smart Solr Suggester that can be improved, but that depends on the needs of the specific applications. Smart Solr Suggester has proved that it is possible to improve the existing Solr Suggester in a way that meets your needs, and in this case, it is searching for entities. In addition, we also found that performing autosuggest in a Solr cluster has both advantages and disadvantages. While it takes longer to build a suggester index in the cluster mode, the benefits come when it takes less time to retrieve the suggestion list compared to running autosuggest on a single Solr instance. For future direction, we hope to combine the implementation of this project with another project directed by Dr. Tran that uses the output of Smart Solr Suggester as it its input to actually return highly relevant entities.

Literature references:

[1] L. Zhang, T. Tran, and A. Rettinger. *Probabilistic Query Rewriting for Efficient and Effective Keyword Search on Graph Data.*

[2] Gregory Smits, Olivier Pivert, et al. *AGGREGO SEARCH: Interactive Keyword Query Construction*

[3] Grainger, Trey and Timothy Potter. *Solr In Action.* Print.

[4] "W3C Semantic Web Frequently Asked Questions." *W3C Semantic Web FAQ.* Web. 01 May 2015.

[5] "RDF 1.1 Primer." *RDF Primer.* Web. 01 May 2015.