

Fall 2015

# Relationship based Entity Recommendation System

Rakhi Poonam Verma  
*San Jose State University*

Follow this and additional works at: [https://scholarworks.sjsu.edu/etd\\_projects](https://scholarworks.sjsu.edu/etd_projects)

Part of the [Computer Sciences Commons](#)

---

## Recommended Citation

Verma, Rakhi Poonam, "Relationship based Entity Recommendation System" (2015). *Master's Projects*. 454.  
DOI: <https://doi.org/10.31979/etd.xvv9-vy26>  
[https://scholarworks.sjsu.edu/etd\\_projects/454](https://scholarworks.sjsu.edu/etd_projects/454)

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact [scholarworks@sjsu.edu](mailto:scholarworks@sjsu.edu).

# Relationship based Entity Recommendation System

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Rakhi Poonam Verma

December 2015

Copyright © 2015

Rakhi Poonam Verma

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Relationship based Entity Recommendation System

by

Rakhi Poonam Verma

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2015

Dr. Thanh Tran	Department of Computer Science
----------------	--------------------------------

Dr. Robert Chun	Department of Computer Science
-----------------	--------------------------------

Mr. Ronald Mak	Department of Computer Science
----------------	--------------------------------

## **ABSTRACT**

### **Relationship based Entity Recommendation System**

by Rakhi Poonam Verma

With the increase in usage of the internet as a place to search for information, the importance of the level of relevance of the results returned by search engines have increased by many folds in recent years. In this paper, we propose techniques to improve the relevance of results shown by a search engine, by using the kinds of relationships between entities a user is interested in. We propose a technique that uses relationships between entities to recommend related entities from a knowledge base which is a collection of entities and the relationships with which they are connected to other entities. These relationships depict more real world relationships between entities, rather than just simple “is-a” or “has-a” relationships. The system keeps track of relationships on which user is clicking and uses this click count as a preference indicator to recommend future entities. This approach is very useful in modern day semantic web searches for recommending entities of user’s interests.

## **ACKNOWLEDGEMENTS**

I would like to thank my advisor Dr. Thanh Tran, for his continuous guidance and support throughout the project and providing me such an opportunity to work on this interesting project. I would also like to thank my committee members, Dr. Robert Chun and Mr. Ronald Mak, for their valuable time and feedback. Lastly, I would also like to convey my thanks to my family, and friends for their help and support.

# TABLE OF CONTENTS

## CHAPTER

<b>1. Introduction.....</b>	<b>10</b>
<b>2. Background and Related Work.....</b>	<b>11</b>
2.1 Background.....	11
2.1.1 Entity.....	11
2.1.2 Related Entity.....	11
2.1.3 Relationship.....	11
2.2 Related Work.....	12
2.3 Drawbacks in existing systems.....	13
<b>3. Problem Definition and Proposed Solution .....</b>	<b>15</b>
3.1 Problem Definition.....	15
3.2 Observations from the existing systems.....	15
3.3 Hypothesis derived for the proposed solution.....	16
3.4 What is different in the proposed solution? .....	16
<b>4. Algorithm of Proposed Solution .....</b>	<b>17</b>
4.1 Proposed Solution.....	17
4.2 Pseudocode of the proposed solution.....	18
<b>5. Implementation Details .....</b>	<b>20</b>
5.1 Solr Overview.....	20
5.1.1 Some key terminologies used in Solr.....	20
5.1.1.1 Solr Document .....	20
5.1.1.2 Core or Collection.....	21
5.1.1.3 Shard .....	21
5.1.2 How Solr works?.....	21
5.2 Implementation of the proposed approach.....	24
5.2.1 Client side implementation using SolrJ API calls.....	24
5.2.2 Extending Solr by adding a new SearchComponent in Solr .....	25
5.2.2.1 Different Solr cores created in the system .....	26
5.2.2.2 Solr schemas created.....	27
5.3 An overview of the dataset used .....	30

5.4 Ranking.....	32
5.4.1 Proposed ranking algorithm.....	33
<b>6. Experiments and Results.....</b>	<b>34</b>
6.1 Query results for different users with different user profiles .....	35
6.2 Query results for same user in presence and absence of user context .....	40
6.3 Improvements made after Experiments .....	41
6.3.1 Move implementation from client side to a plugin in solr server .....	41
6.3.2 Ranking related entities by applying appropriate weights .....	42
<b>7. Conclusion .....</b>	<b>44</b>
<b>8. References .....</b>	<b>45</b>



## LIST OF TABLES

Table 1: User profile and User Context for user “rakhi” with click counts of relationships .....	35
Table 2: User profile and User Context for user “poonam” with click counts of relationships ...	37
Table 3: User profile and User Context for user “newuser1” with click counts of relationships.	39
Table 4: Execution time of searching two entities on the two mentioned implementations .....	42

## LIST OF FIGURES

Figure 1: Example of Entity Relationship Graph .....	11
Figure 2: A Solr Document.....	21
Figure 3: An example of HTTP GET Request for Solr Server [7] .....	22
Figure 4: Handling of request from a client to Solr [7] .....	23
Figure 5: Definition of /select request handler in solrconfig.xml [7] .....	24
Figure 6: solrconfig.xml file of CS298-project core.....	27
Figure 7: Snippet of schema for CS298-collection.....	27
Figure 8: Snippet of schema for CS298-userprofile .....	28
Figure 9: Snippet of schema for CS298-usercontext .....	29
Figure 10: Snippet of schema for CS298-project .....	29
Figure 11: Snippet of a solr document belonging to dataset.....	30
Figure 12: Graphical view of a portion of the dataset .....	31
Figure 13: Snippet of an entry in the user profile .....	31
Figure 14: Snippet of an entry in the user context .....	32
Figure 15: Total documents in the dataset .....	34
Figure 16: Recommendations for Queried entity “The Dark Knight” for user “rakhi” .....	35
Figure 17: Relationships of Christian Bale.....	36
Figure 18: Relationships of Heath Ledger.....	36
Figure 19: Recommendations for Queried entity “The Dark Knight” for user “poonam” .....	37
Figure 20: Relationships of Christopher Nolan.....	38
Figure 21: Relationships of Heath Ledger.....	38
Figure 22: Recommendations for Queried entity “The Dark Knight” for “newuser1” .....	39
Figure 23: Result in presence of user context.....	40
Figure 24: Result in absence of user context .....	40
Figure 25: Execution time of searching two entities on the two mentioned implementations .....	41
Figure 26: Results for user “poonam” before and after applying updated weighing scheme.....	43

# CHAPTER 1

## Introduction

As the usage of Internet to search for things, knowledge about concepts, history about events etc. increases, the importance of the information returned by search engines to be highly relevant to user's query increases. Showing the same results to two users whose interests are completely different may lead to dissatisfaction of one of them. It has now become imperative for a search engine to provide personalized search results for each user to keep user interested in using the same search engine again.

In this project, we propose a personalized entity recommendation system that uses different kinds of relationships (not just "is-a" or "has-a") existing between entities for recommending them to the user based on his interests [1]. The system will keep track of what kinds of existing relationships is the user clicking on most and store it in the user's profile. The system will use this profile and return information which is related to user's query through these relationships that depict the user's interests.

The main goal of this project is to implement an effective semantic matching algorithm that would match the given entity with a set of entities present in the huge knowledge base and return highly relevant entities to the given entity based on the user profile and user context. A weighing semantic is used to rank related entities based on their relationship to the queried entity, before suggesting them to the user.

Hereafter, this paper has been organized into following sections. Chapter 2 provides information about the related work done so far in existing systems for recommending entities and drawbacks of these systems. Chapter 3 provides the problem definition, our hypothesis, and how we derived it based on the things observed from the existing recommendation systems. It also explains how our proposed solution overcomes the drawbacks present in existing solutions. In Chapters 4 and 5, we discuss our algorithm for the proposed solution and the implementation details. Chapter 6 follows with discussions about the results of our implementation on a large dataset. The paper then provides a conclusion and the future work that could be done to improve the system.

## CHAPTER 2

### Background and Related Work

#### 2.1 Background

There are a lot of terminologies used in the paper in order to explain the existing systems, the proposed hypothesis and its implementation in the paper. So, it's important to understand these terminologies before proceeding further. The following text explains the meaning of most of the terms used in the paper:

##### 2.1.1 Entity

An entity can be defined as something which is real or existing and is surrounded by a lot of information. It could be anything like a person, a company, city, college etc.

##### 2.1.2 Related Entity

An entity related to a given entity via any means is said to be a related entity of the given entity.

##### 2.1.3 Relationship

This is the means of connection between two entities. One entity can be connected to another entity through a relationship. For example – friend of a friend, lives in, is director of, etc.

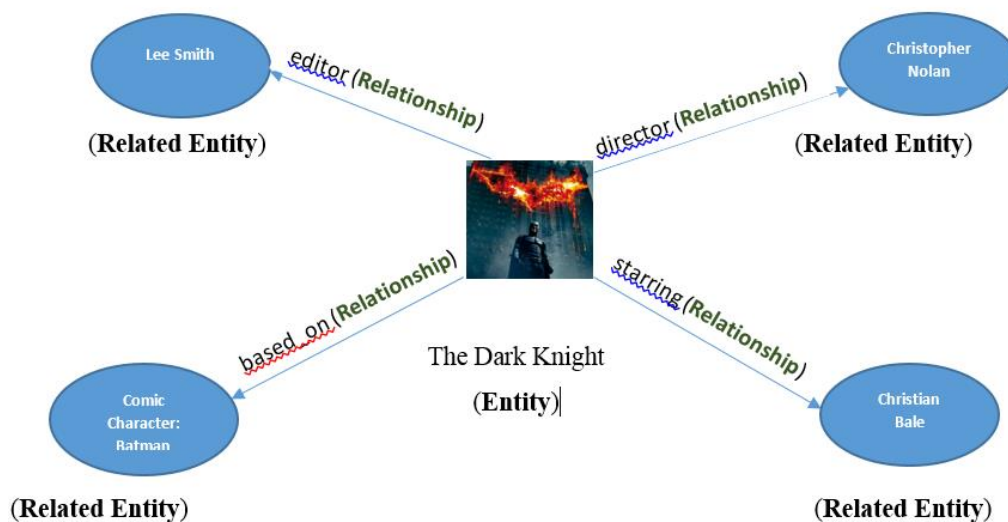


Figure 1: Example of Entity Relationship Graph

## 2.2 Related work

A number of systems exists that attempted to provide entity recommendation based on various criteria identified by these systems. A few such recommendation systems that use semantic information present in datasets are discussed below:

**Spark** is a semantic search assistance tool that exploits the public knowledge bases and Yahoo!'s proprietary data to provide related entity suggestions for queried entity on the web [2]. It uses three kinds of features for recommending entities extracted from the dataset(s), as discussed below:

- **Co-occurrence** – Authors found out all the entities that occurred together in old query logs, Flickr photo tags, tweets etc. This information is used by Spark to recommend an entity when one of the entities from the pair was queried.
- **Popularity** – Spark uses popularity of the entities from Wikipedia and Yahoo! search results to recommend entities. This information is used to select the expected meaning of the query, when there could be multiple meanings or multiple entities pointed by the query. This information can also come in handy when User Profile does not have sufficient data (i.e., it is a sparse user profile). Therefore, this can be used to recommend default entities for any search query.
- **Graph-theoretic features** – Spark uses two types of graphs to find common neighbors or related entities between any two entities. The first type of graph is an entity relationship graph in which vertices represent entities and edges represent relationships between these entities. The second graph is a hyperlink graph that is obtained from a large web page collection. The authors run a page rank algorithm on both graphs to find out common neighbors of two given entities.

The authors then assigned different weights to these features and ranked the entities based on total value of weights each entity has. For example, frequency of tags of co-occurring entities in Flickr photos are given a weight of 70.3, while those entities that occurred together in past search queries are given a weight of 54.8.

**News@Hand** is a news recommending system that uses semantic technology to provide news items related to user's query and interests [3]. It uses an automatic annotator to annotate the data with classes from an ontology and then uses these classes called concepts to suggest highly relevant and personalized news items. It suggests that the current context of the user's interaction with the system can help improve the personalization of recommended entities and the results would be highly relevant since they would be more focused towards what user is searching in the current session.

**Aethna** is a news item recommendation system that uses ontology and user profile to recommend news items which are related to user's query [4]. It suggests that a user profile can be a good way to make the system understand the likes and dislikes of a user and can be used to personalize recommendations based on this understanding. It assigns concepts from the ontology to news items and stores these concepts in the user profile whenever the user clicks on any news item. It then employs different similarity metrics like Binary Cosine, Jaccard Similarity, etc. to find similarity between news items and the preferred concepts stored in the user profile. The authors also suggest giving a weight of 1 to the concepts present in user profile and entity, a weight of 0.5 to the concepts in user profile which are directly related to concepts of the entity and a weight of -0.1 to all the other concepts in user profile which are not related to concepts of the entity.

**ODAS**, a Domain Ontology for Adaptive Hypermedia Systems, suggested a formal rule-based system that would allow users to personalize their profile [5]. The authors present many rules that can be used to select or reject related entities from the result in order to provide high level of personalization in the recommendation. The system also provides a way to the user to change the rules to further improve the personalization of the recommendations.

## **2.3 Drawbacks in existing systems**

- **Spark** recommendation system provides recommendations based on data it extracted from several social networking sites. It uses a machine learning algorithm to train the system and then provide recommendations based on it. It doesn't take a user's preferences or past and current interests into consideration. So, entities recommended by Spark would more or less be

the same for all the users if they enter the same query. This reduces the level of personalization in recommendations.

- **News@Hand** performs matching of news items based on the ontological annotations on the items and the ones saved in user's profile. The matching is done based on similarity between these annotations. So, the system recommends news items which are ontologically similar to queried concept. But, this type of matching ignores or overlooks the relationships between dissimilar entities like a person "drives" a car.
- **Aethna** uses direct relationships between concepts to provide personalized news recommendations, but it doesn't take into account the current context of the user. Hence, the recommendations generated by the system may be relevant to user's past interests, but they might not be that much relevant to the user at the time he is making the query.
- **ODAS** provides a flexible way to define rules in order to provide personalization in the recommendations. It also provides an easy, scalable and a very flexible way to user to edit the rules according to his linking to improve the personalization of recommended entities according to their choice. Although this flexibility can be very useful to the user in many cases, a flaw in this approach is that the user should be aware of the rules of the system, and should be knowledgeable enough to tweak the rules in correct ways. This requires the users to be fairly technically sound in using the system, which is not really the case most of the time.

## **CHAPTER 3**

### **Problem Definition and Proposed Solution**

#### **3.1 Problem Definition**

In this project, our goal is to implement an effective semantic matching algorithm that would match the given entity with a set of entities present in the huge knowledge base and return highly relevant entities related to the given entity based on the user profile and the current context of the user.

As an input, the system will take an entity that the user is looking for, the profile of the user that contains the history of the searches done by the user, and also the current context of the user.

As an output, the system will suggest a set of entities that are related to the given input entity in the knowledge base on the basis of relationships the user was interested in the past, taking the current context of the user into account.

#### **3.2 Observations from the existing systems**

Looking at the existing systems discussed above, we can make some general observations for systems that aim to recommend entities that are relevant to user's interest in the past and in the current session. A few such observations are:

- Keeping track of relationships of entities can give us an idea of the user's "preferences".
- The higher the click count of a relationship, the higher is user's liking or preference towards it.
- A user's preferences can change temporarily based on his requirements. Thus, recommending entities in which the user is currently interested is important to improve relevance.



### 3.3 Hypothesis derived for the proposed solution

- **Relationship-based relevance:** Given an entity type, the user has preferences towards some particular relationships. Entities are relevant if they are connected to the given entity via these “preferred” relationships.
- **The number of user clicks observed for a particular relationship:** It can be used to measure the user preference towards that relationship and its intensity.
- **Context of user's current session:** It can be used to better recommend entities that are more related to user's current interest as compared to his past interests.

### 3.4 What is different in the proposed solution?

In our proposed solution, we tried to overcome various drawbacks which were noticed in the existing systems. The suggested solution provides recommendations ranked in the order based on the user’s current context first, and then based on user’s past interests. The solution also takes into consideration different relationships that exist between entities for recommending related entities rather than just recommending similar entities. It also checks for semantic similarity between entities on which user clicked in the past and the entities the system is going to recommend, to eliminate entities that are unrelated to the user’s interests.

## **CHAPTER 4**

### **Algorithm of Proposed Solution**

#### **4.1 Proposed Solution**

As discussed in the previous sections, a personalized search engine can be implemented using a personalized entity recommender system that takes into account a user's interests and recommends entities that align with them. We propose an entity recommender system that works on a knowledge base (K) having entities and relationships such that each entity is related to several entities via relationships. The relationships can be generic relationships and not limited to just "is-a" or "has-a" relationships. The system stores the information about the relationships a user clicks while interacting with the system and looking through the recommended entities. Whenever a user clicks on a recommended entity, details of the relationship by which it is connected to the queried entity is stored in the user profile and the user context. Over time, the user profile would grow and would start depicting user's preferences through all the relationships stored in user profile and the total number of times they were clicked. The count of user clicks observed for a particular relationship can be used to measure the user's preference towards that relationship and its intensity. The system will also store user's current context that would portray the user's current interests or things that user has searched for in the current session. Using the user's current context to rank recommendations would increase personalization of recommendations because sometimes users might be temporarily interested in entities that do not align with their past interests or intents.

## 4.2 Pseudocode of the proposed solution

```
List<Entity> getRelatedEntities(String queriedEntityName, String userId)
{
    Entity queriedEntity = findEntityInKB (queriedEntityName);
    Set<Entity> recommendedEntites = new HashSet<Entity> (); //declaration of result set
    Map<Entity, Integer> relatedEntityToRelCount = new HashMap<Entity, Integer> ();
    Map<Relationship, List<RelatedEntityRelClickCount>
    mapOfRelationshipToOrderedEntitiesCount = new HashMap<>();

    /* Step 1: Get Map of all relationships and all the entities related to queried entity via those relationships from KB*/

    Map<Relationship, List<Entity>> relationshipEntitiesMap =
    queriedEntity.getRelationshipEntitiesMap();

    /* Step 2: Get Matching relationships for that user context based on click count in decreasing order */

    List<Relationship> matchedOrderedRelFromContext =
    getOrderedMatchingRelFromContext();

    /* Step 3: Get Matching relationships for that user Profile based on click count in decreasing order */

    List<Relationship> matchedOrderedRelFromProfile =
    getOrderedMatchingRelFromProfile();
```

**/\* Step 4: Merge relationships in the order - first from User context then from user profile and then the remaining \*/**

```
List<Relationship> mergedOrderedRelationships = merge  
(matchedOrderedRelFromContext, matchedOrderedRelFromProfile);
```

**/\* Step 5: Sort the entities which are related to 1 relationship based on their matching total relationship count with user profile \*/**

```
for (Relationship rel : mergedOrderedRelationships){  
    List<Entity> entities = relationshipEntitiesMap.get(rel);  
    List<RelatedEntitiesPerClickCount> orderedEntitiesPerRelation = sort(entities);  
    mapOfRelationshipToOrderedEntitiesCount.put(rel, orderedEntitiesPerRelation);  
}
```

**/\* Step 6: Calculate score for each related entity and then recommend**

**Two types of weights:**

**- Relationship weight: for ordered relationships associated to queried entity starting from 1, 1/2, 1/3 and up to 1/n.**

**- Entity Weight: for the list of ordered entities associated with “same relationship”. This would start from 1, 1/2, 1/3 and up to 1/m.**

**Total score of an entity = (rel weight \* entity weight \* entity rel count)**

**If 1 entity is connected to queried entity via multiple relationships, score for that entity is updated by adding the scores calculated above for all the relationships through which it is connected to queried entity."**

**\*/**

```
recommendedEntites =  
applyWeightsAndSort(mapOfRelationshipToOrderedEntitiesCount);
```

```
}
```

## **CHAPTER 5**

### **Implementation details**

#### **5.1 Solr Overview**

Apache Solr is a popular tool used for indexing huge datasets and allowing flexible, scalable and efficient ways to perform simple and complex queries on it. It is a Java based framework which uses Apache Lucene in the background to perform the work of indexing and searching on that index. Solr can be visualized as a sophisticated user interface to interact with the Lucene index. Lucene is a Java based library which builds and manages the index of the whole dataset, which is ultimately used for searching. By default, it uses an inverted index algorithm to index the documents added in it. Inverted index is a kind of data structure helpful for matching the queried terms with the documents in Solr.

##### **5.1.1 Some key terminologies used in Solr**

Before going into depth, we should understand the basic terminologies of Solr which have been used frequently and would be used to understand the rest of the approach and implementation.

###### **5.1.1.1 Solr Document**

Solr stores data in the form of solr documents. Each solr document contains one or more fields. If we compare this with relational database, then we can say that every Solr document is equivalent to a row in the table and field values are equivalent to columns of a table [6].

When we add documents to Solr, it takes the information from the fields and updates the index accordingly. So, when the query is performed, it refers to the index and quickly returns the documents matching the query term. A solr document can be in any format like XML, JSON, CSV. In xml format, a solr document can be viewed as follows:

```

<doc>
<field name="uri">http://dbpedia.org/resource/%C3%80_Nos_Amours</field>
<field name="owl_sameAs_resource">http://rdf.freebase.com/ns/m.0fn4s2</field>
<field name="rdfs_label_text">À Nos Amours</field>
<field name="dbpprop_director_resource">http://dbpedia.org/resource/Maurice_Pialat</field>
<field name="dbpprop_producer_resource">http://dbpedia.org/resource/Michelien_Pialat</field>
<field name="dbpprop_writer_text">Maurice Pialat</field>
</doc>

```

**Figure 2: A Solr Document**

Every Solr document contains multiple fields with their values associated with it. Every field has a name and a value. Before we add documents or data in Solr, we need to specify the schema. A schema contains what fields are there in data, unique key, what fields are required etc. This schema is stored in a file called schema.xml.

#### **5.1.1.2 Core or Collection**

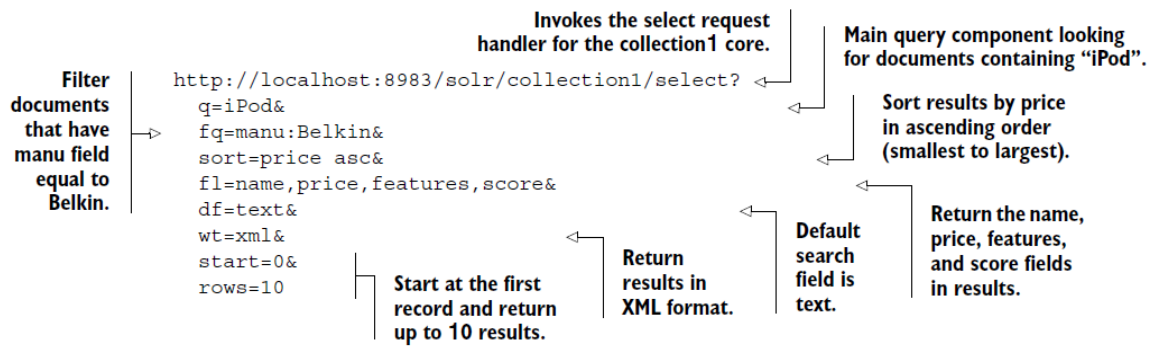
A core or collection is an index with a schema that holds a set of documents. Every core has its own schema.xml and solrconfig.xml.

#### **5.1.1.3 Shard**

A shard is nothing but a part or whole of the collection and are non-overlapping.

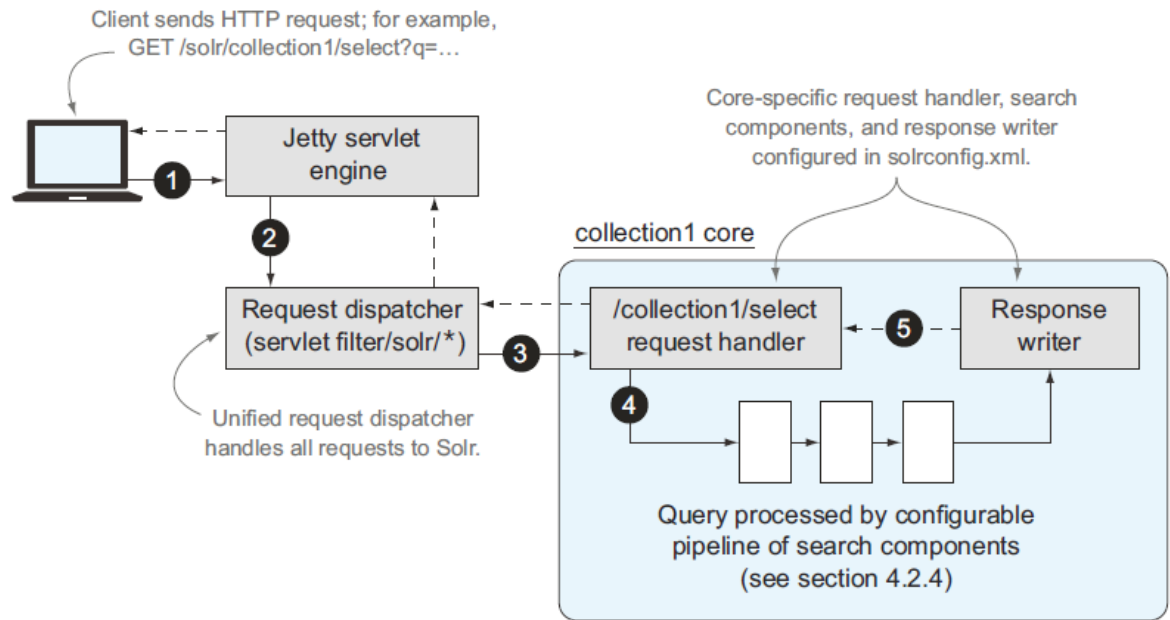
### **5.1.2 How Solr works?**

A solr server can be reached for running queries via a web browser. The default implementation of solr starts it on port 8983. A query to a running instance of solr server is generally made as a GET HTTP call. The URL to this call contains information about the data which has to be searched. The query looks as follows:



**Figure 3: An example of HTTP GET Request for Solr Server [7]**

Solr is a web application, based on Java and can run on any application server like Apache Tomcat, JBoss etc. Solr by default comes with a Jetty application server in its distribution. So, when a GET request is sent from a client, it goes to the application server that handles the request and routes it to a specific handler. All these handlers are Java classes written at server side. In the above query, client has given the query term as “*iPod*” and is looking for documents where manu field is equal to “*Belkin*”. The resultant documents are requested to be sorted based on the price in ascending order. From these resultant documents, fields labelled as Name, Price, Features and Score will be returned to the client since the query has a filter to retrieve only these fields in the result.



**Figure 4: Handling of request from a client to Solr [7]**

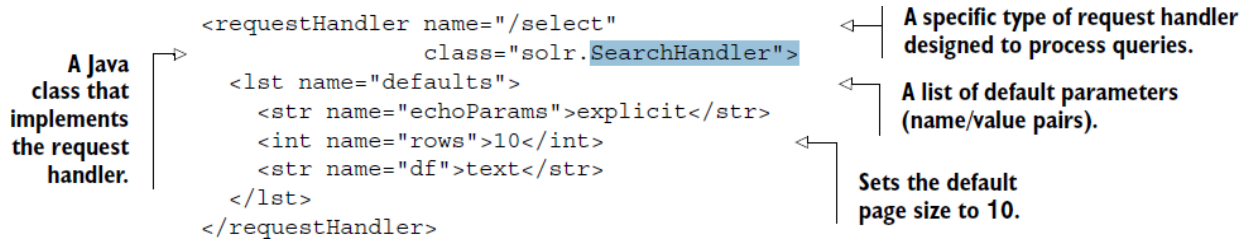
Figure 4 above, illustrates the workflow of how a search query is processed by Solr. Here are the steps:

- When the client sends a query, it will be sent as a GET request to the application server. The GET request would be in the form of `/solr/collection1/select?q=<Query>`
- Server receives the GET request and routes it to the request dispatcher of Solr using `/solr` context in the query path.
- Request dispatcher identifies the collection from the query path and looks for `/select` request handler in `solrconfig.xml` file of the identified collection.
- `/select` request handler processes the request using the underlying `SearchComponent` – a class written in java.
- After the query is processed, response writer sends the results to the client.

So, search components are responsible for doing the search and their specific type to be used can be specified in `solrconfig.xml`. Each search component extends from the class `SearchComponent` which is an abstract class that list methods required to be implemented



by child classes in order to complete the search on data successfully. Search Component contains the logic which is used by SearchHandler in order to execute a query [8].



**Figure 5: Definition of /select request handler in solrconfig.xml [7]**

SearchHandler is a subclass of Request Handler which is responsible to respond to search requests. This is the default request handler which comes pre-configured in Solr and is responsible for all incoming searches. A SearchHandler is usually registered in solrconfig.xml with the name "/select" and has different parameters [9].

## 5.2 Implementation of the proposed approach

There could be multiple ways of implementing the proposed approach for the relationship based entity recommendation system (RBERS):

1. Client side implementation using SolrJ API calls
2. Extending Solr by adding a new SearchComponent in Solr

### 5.2.1 Client side implementation using SolrJ API calls

Client side implementation can be done by making Solr server calls for each and every query done on the dataset. The client side implementation involves wiring a lot of logic in the client site web app which would make all the calls to the solr server on the client's behalf. A Java library named SolrJ is created for this purpose. The library provides methods to make GET method calls to the

solr server and methods to display the corresponding results. For the complete execution of our proposed algorithm, it would take a lot of queries (~ 1 query per related entity) to be made to solr over the internet. Since these calls go over the HTTP protocol of remote method calling, this method can prove to be significantly slow. The implementation using SolrJ APIs can be easier because it does not require one to understand the internals of solr and allows the flexibility to search any kind of query, along with all the parameters supported by solr. But this way of implementation will not be very efficient, since we will have to make a lot of queries to solr to find out all the details we need, and it will take a lot of networking and memory resources to find out the result and recommend related entities to the user.

### **5.2.2 Extending Solr by adding a new SearchComponent in Solr**

As we have seen above that client side implementation could be slow and less efficient, there is a different way to implement the same logic in solr server itself. This is possible by extending the Solr's abstract class SearchComponent and providing the details of unimplemented method. We just have to provide the jar file with our custom implementation and provide a path to jar file and name of class to be used in the solrconfig.xml. This child class of SearchComponent can run in the solr server itself, so that, a lot of calls that were made over the Internet in the previous approach can now be made directly in memory. This increases the speed of query execution by a significant number. So we decided to extend Solr for entity recommendation and provide our custom implementation of SearchComponent to accomplish the task of recommending entities.

There are different abstract methods in SearchComponent class in Solr, which are used to carry out the logic of performing a search query in solr. Some of these methods are [10]:

- `init (NamedList args)`: This method is called when the plugin is first loaded.
- `getDescription()`: This returns the description given for the SearchComponent.
- `prepare (ResponseBuilder rb)`: This method is called before the process method and is called for every incoming request. All the variables which are not dependent on the incoming request get initialized here.
- `process (ResponseBuilder rb)`: This contains the logic of processing the request.

- `distributedProcess (ResponseBuilder rb)`: This method is called to process the request when the search is performed in a distributed manner.
- `handleResponses(ResponseBuilder rb, ShardRequest sreq)`: This method is called after all responses for a single request were received

### 5.2.2.1 Different Solr cores created in the system

In our implementation, we have made different cores or collections in one server and used them as shards with the help of which, we are performing distributed search. When we perform a search for an entity, Solr server talks to these shards and get the result and send it back to the user.

Solr cores in our system:

- **CS298-collection**: Core which contains the dataset in the form of solr documents
- **CS298-userprofile**: This contains click count of relationships for all users in the form of solr documents
- **CS298-usercontext** – This contains click count of relationships for all users in current context in the form of solr documents
- **CS298-project**: This is the controller solr core of RBERS. The query comes to this and it internally connects to every other shard for the results as required. The information of all the shards are given in the `solrconfig.xml` file of the controller.

```

For processing Search Queries, the primary Request Handler
provided with Solr is "SearchHandler" It delegates to a sequent
of SearchComponents (see below) and supports distributed
queries across multiple shards
-->
<requestHandler name="/select" class="solr.StandardRequestHandler">
  <!-- default values for query parameters can be specified, these
       will be overridden by parameters in the request
  -->
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <int name="rows">10</int>
    <str name="shards">localhost:8983/solr/cs298-collection,localhost:8983/solr/cs298-userprofile,localhost:8983/solr/cs298-usercontext</str>
  </lst>
  <!-- <arr name = "last-components"> -->
  <arr name = "components">
    <str>xberssearchcomponent</str>
  </arr>
</requestHandler>

<searchComponent
  class = "xbers.solr.RbersSearchComponent"
  name="xberssearchcomponent">
</searchComponent>

```

Figure 6: solrconfig.xml file of CS298-project core

As shown in Figure 6 above, two attributes have been given between `<searchComponent>` and `</searchComponent>` tags. Attribute “class” tells the class name of our implemented search component along with its package details. The “name” attribute of the searchComponent tells the name of our implemented SearchComponent which is being referred inside requestHandler.

### 5.2.2.2 Solr schemas created

#### Dataset schema:

```

<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="true" 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>

```

Figure 7: Snippet of schema for CS298-collection

Figure 7 shows the knowledge base schema that defines the structure of the solr documents. Each entity will be represented as a solr document and will be identified by a unique URI. In our dataset, we have two types of fields associated with each entity. One ending with “\_text” and the other ending with “\_resource”. Fields ending with “\_text” contain the literal values associated with the entity and the fields ending with “\_resource” contain the URIs of related entities, related to this entity via some relationship.

Every field is associated with an attribute called indexed. We make this attribute as true if we want to apply the inverted index to it and want to make that field content searchable. We created two more fields called text and relatedDocs. The field called as text field will contain all the fields ending with “\_text” and relatedDocs field will contain all the fields ending with “\_resource”. These fields have been created to make our search in knowledge base efficient. In our implementation, instead of searching all fields, we can search only the required fields.

## User Profile Schema

```
<schema name="things" version="1.5">

  <!-- common fields, every doc has these -->
  <field name="_version_" type="long" indexed="true" stored="true"/>
  <!-- Typically should be concatenated string of user_id and relationship field values using "_"-->
  <!-- <field name="id" type="string" indexed="true" stored="true" multiValued="false" /> -->
  <field name="user_id" type="string" indexed="true" stored="true" multiValued="false" />
  <field name="relationship" type="string" indexed="true" stored="true" multiValued="false" />
  <field name="click_count" type="long" indexed="true" stored="true" multiValued="false" />

  <field name="uri" type="string" indexed="true" stored="true" required="true" multiValued="false"/>

  <fieldType name="string" class="solr.StrField" sortMissingLast="true"/>
  <fieldType name="long" class="solr.TrieLongField" precisionStep="0" positionIncrementGap="0"/>

  <!-- unique fields -->
  <!-- <uniqueKey>id</uniqueKey> -->
  <uniqueKey>uri</uniqueKey>
```

**Figure 8: Snippet of schema for CS298-userprofile**

User profile contains the documents with the structure given in the user profile schema. Every document will have a user\_id of the user using the system, the relationship he was interested in the past and its click count. URI is the unique key in the schema.

## User Context schema

User context shows the current context of the user using the system. It is a subset of user profile and will have the same schema as the user profile schema.

```
<schema name="things" version="1.5">

  <!-- common fields, every doc has these -->
  <field name="_version_" type="long" indexed="true" stored="true"/>
  <!-- Typically should be concatenated string of user_id and relationship field values using "_" -->
  <!-- <field name="id" type="string" indexed="true" stored="true" multiValued="false" /> -->
  <field name="user_id" type="string" indexed="true" stored="true" multiValued="false" />
  <field name="relationship" type="string" indexed="true" stored="true" multiValued="false" />
  <field name="click_count" type="long" indexed="true" stored="true" multiValued="false" />

  <field name="uri" type="string" indexed="true" stored="true" required="true" multiValued="false"/>

  <fieldType name="string" class="solr.StrField" sortMissingLast="true"/>
  <fieldType name="long" class="solr.TrieLongField" precisionStep="0" positionIncrementGap="0"/>

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

Figure 9: Snippet of schema for CS298-usercontext

## Master core schema

```
<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="true" 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"/>

  <field name="user_id" type="string" indexed="true" stored="true" multiValued="false" />
  <field name="relationship" type="string" indexed="true" stored="true" multiValued="false" />
  <field name="click_count" type="long" indexed="true" stored="true" multiValued="false" />
  <field name="id" type="string" indexed="true" stored="true" multiValued="false" />

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

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

Figure 10: Snippet of schema for CS298-project

### 5.3 An overview of the dataset used

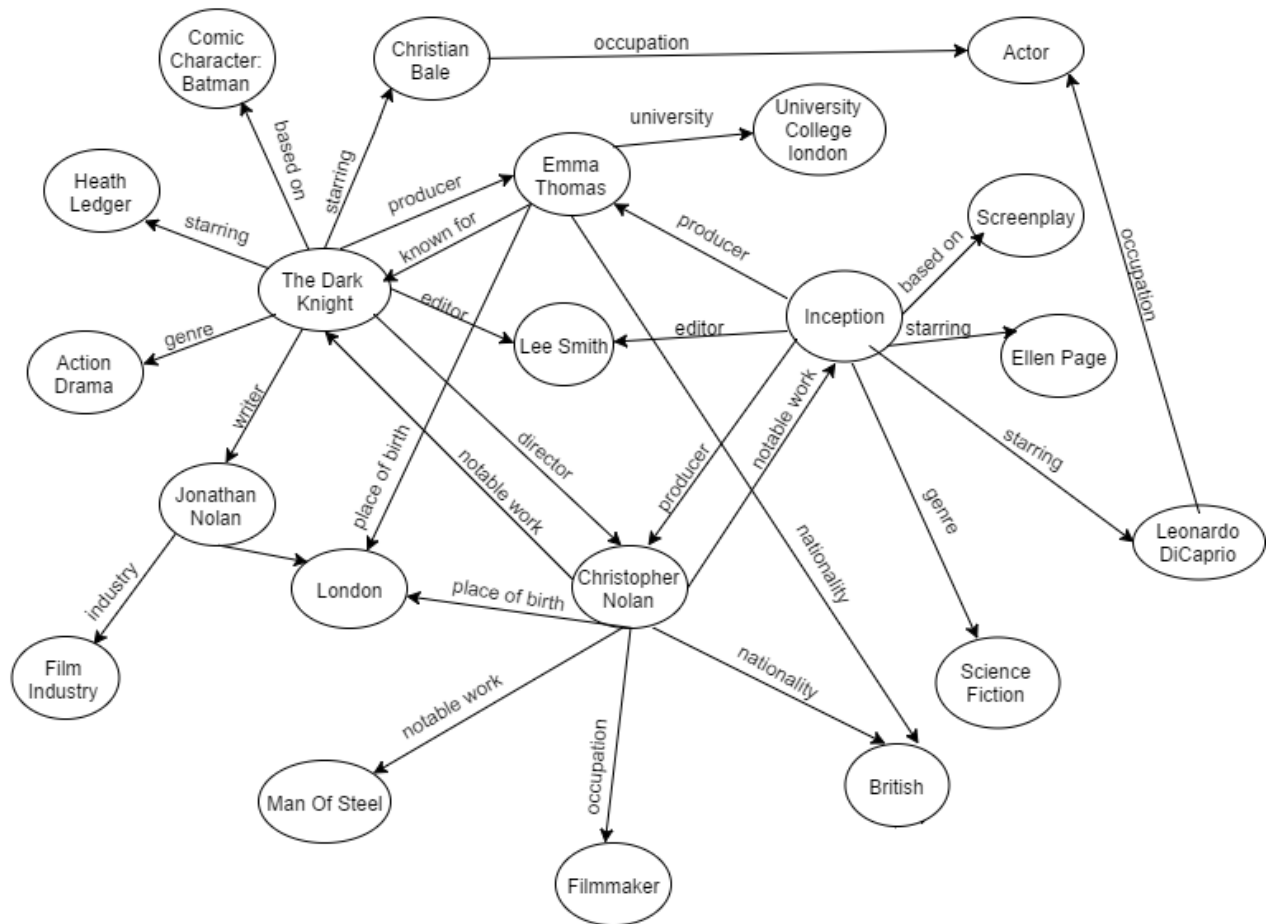
We have used dbpedia dataset to support our implementation. Originally, data was in the form of RDF documents that was later converted to Solr documents by one of the previous students, who did Master's Project under the supervision of Dr. Tran. We have directly used the data in form of Solr documents provided by him [11].

In the dataset, every document represents an entity of given URI and the entities related to this entity via different relationships.

```
<doc>
<field name="uri">http://dbpedia.org/resource/%C3%80_Nos_Amours</field>
<field name="owl_sameAs_resource">http://rdf.freebase.com/ns/m.0fn4s2</field>
<field name="rdfs_label_text">À Nos Amours</field>
<field name="dbpprop_director_resource">http://dbpedia.org/resource/Maurice_Pialat</field>
<field name="dbpprop_producer_resource">http://dbpedia.org/resource/Michelien_Pialat</field>
<field name="dbpprop_writer_text">Maurice Pialat</field>
</doc>
```

**Figure 11: Snippet of a solr document belonging to dataset**

As can be seen from the above screenshot, there are two types of field names in each Solr document in the dataset. One ending with “\_text” and the other ending with “\_resource”. The field names ending with “\_text” are the text fields and the field names ending with “\_resource” are the relationships, through which the entity is connected to other entities. Here, every entity is uniquely identified by its uri. In the above screenshot, the entity identified by uri: [http://dbpedia.org/resource/%C3%80\\_Nos\\_Amours](http://dbpedia.org/resource/%C3%80_Nos_Amours) is related to entity having uri: [http://dbpedia.org/resource/Maurice\\_Pialat](http://dbpedia.org/resource/Maurice_Pialat) through relationship director. At the time of indexing, Solr creates a field “text” and copy the fields ending with “\_text”. It also creates a field “relatedDocs” and copy the fields ending with “\_resource”. The field “relatedDocs” will have the values containing URIs of different entities related to the current entity.



**Figure 12: Graphical view of a portion of the dataset**

```
</doc>
<field name="uri">user1_dbpprop_producer_resource</field>
<field name="user_id">user1</field>
<field name="relationship">dbpprop_producer_resource</field>
<field name="click_count">150</field>
</doc>
```

**Figure 13: Snippet of an entry in the user profile**

Figure 13 shows an entry from the user profile at a time. The field “uri” identifies the entry uniquely by combining the user\_id and relationship field. The fields “user\_id” and “relationship” tells, to which user this entry belongs, and the field “click\_count” shows how many times the user



has clicked on the relationship shown by “relationship” field. Each user will have one such document for each relationship on which they have clicked on.

```
<doc>
<field name="uri">rakhi_dbpedia-owl_starring_resource</field>
<field name="user_id">rakhi</field>
<field name="relationship">dbpedia-owl_starring_resource</field>
<field name="click_count">10</field>
</doc>
```

**Figure 14: Snippet of an entry in the user context**

The entries in user context are similar to user profile. The only difference is that they represent the click count of relationships clicked by the user in the current session. So, once the session is closed or ended, the entries in user context will be deleted. The entries in User profile are never deleted though.

## 5.4 Ranking

Ranking is an important concept in showing the results to the user. There could be multiple entities related to a given input entity through the same or different relationships. For example, there could be multiple entities related to the input entity via user’s preferred relationship “friend of a friend” and there could be other entities related to the input entity via different relationship for example “is director” which is less preferred by user but is comparable to the user’s most preferred relationship. So, without ranking, the user will only get related entities which are related via most preferred relationship(s). Therefore, we came up with an approach of providing weights to the related entities so that user get the recommendations in a ranked way such that the entities in which the user is most interested in comes first as compared to entities a user is less interested in.

In this approach, we use the click count of the relationships of the related entities in user profile. The approach is in complete unison with our hypothesis of recommending entities based on click count of relationships present in user profile. We use the premise that higher the number of times a user has clicked on relationships an entity may have, higher will be the chance that the user will be interested in that entity. So, for all the entities that are related to the queried entity via same relationship, we applied a decaying factor to the total click count of relationships matching to user

profile. The decaying factor is introduced to include diverse results in recommended entities, without which the result might end up having related entities from same relationship only.

#### **5.4.1 Proposed ranking algorithm**

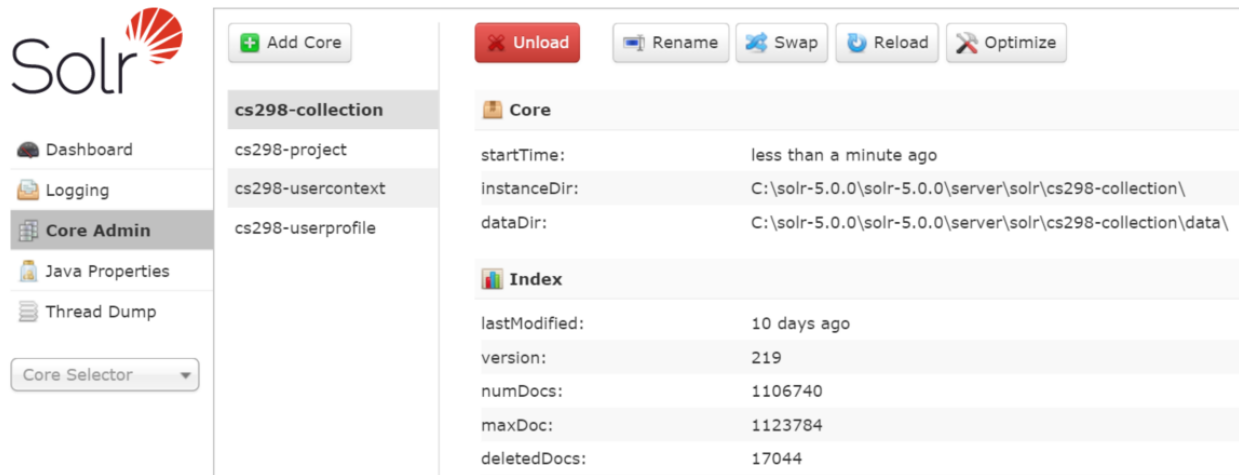
- First, we rank the relationships associated with the queried entity based on the user's preferences in the current context and from the past history. For this ranking, we assign a decaying weight to all these relationships.
- For example, if we have "n" relationships from the queried entity which match with the user profile, we assign a weight of  $1, 1/2, 1/3 \dots 1/n$  to these entities in decreasing order of click count. So, the relationship which has most clicks will have highest weight.
- Now, there could be multiple entities which will be related to the queried entity via the same relationship. So, we decided to provide a similar decaying weight to all these entities related via the same relationship. To assign the weights, we first calculate the total click count of relationships of these entities from the user profile and then assign a weight of  $1, 1/2, 1/3 \dots 1/m$  based on decreasing order of click count.
- After finding ordered entities for each relationship, we apply weights of relationship and entities on each related entity's total matched relationship count found in Step 3.
- We merge the score of the entities which are related to the queried entity by more than one relationships, by adding the entity's score of all the individual relationships through which it is related to the queried entity into one entry.
- After calculating scores for all the unique entities, we sort them based on the score in decreasing order and recommend these entities in that order.

In this way we stick to our hypothesis that user will be more interested in entities that has relationships that he has clicked on before.

## CHAPTER 6

### Experiments and Results

In this section, we present the results of some of the experiments that we did to test the correctness as well as performance of the algorithm by running it on a large dataset. As mentioned above, we have used the dataset of movies and people related to movies, which we got from dbpedia.org. The dataset contains 1106740 solr documents, where each solr document represents an entity and contains URIs of its related entities and the relationships between them.



**Figure 15: Total documents in the dataset**

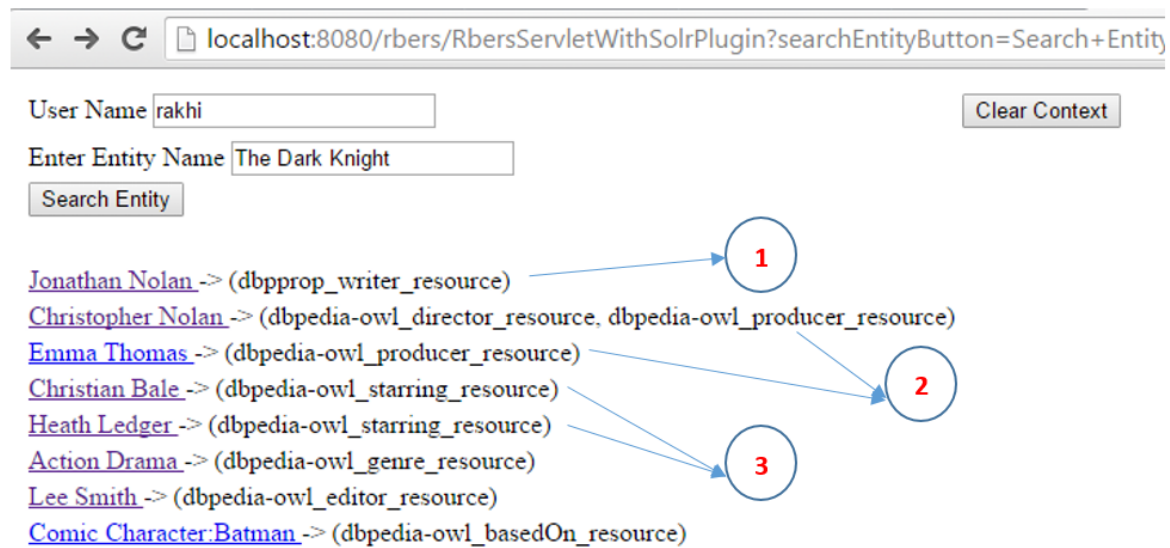
To test the implementation of our hypothesis, we created some sample user profiles that shows interests of different users in the system. The user profiles contain the total count of clicks the user made on different relationships, while interacting with the user. To ease the process of testing the algorithm, as well to improve the visualization of results, we have created a very basic web based user interface using Java servlets and JSP. In the following sections, we show the details of user profiles of three users having the click count of relationships stored in it and the user context.

## 6.1 Query results for different users with different user profiles

**Username: rakhi**

User Profile		User Context	
Relationship Name	Click count	Relationship Name	Click count
dbpedia-owl_director_resource	108	dbpprop_writer_resource	5
dbpedia-owl_producer_resource	57	dbpedia-owl_notableWork_resource	3
dbpprop_placeOfBirth_resource	40		
dbpedia-owl_notableWork_resource	25		
dbpprop_writer_resource	12		
dbpedia-owl_genre_resource	1		

**Table 1: User profile and User Context for user “rakhi” with click counts of relationships**



**Figure 16: Recommendations for Queried entity “The Dark Knight” for user “rakhi”**

Explanation of the order of recommended entities in Figure 1:

All the links with underline in figure 15 are entities related to the queried entity “The Dark Knight” in our dataset. The entities are sorted based on ordering of recommendation for the given user

“rakhi”, in decreasing order of rank from top to bottom. The text between “()” shows the relationship with which the recommended entities are associated with the queried entity.

**Entity marked with 1 (Jonathan Nolan):** This entity is ranked the highest among all the related entities of “The Dark Knight” because it is connected with the relationship “dbpprop\_writer\_resource” which is the most clicked relationship by the user in her current context (session) of interaction with the system.

### Entities marked with 2 (Christopher Nolan & Emma Thomas):

The two entities ranking 2nd and 3rd in the recommendation from the system are ranked based on the click count of their relationships with the queried entity retrieved from the user profile. As can be seen, Christopher Nolan is related to “The Dark Knight” with two user preferred relationships “dbpedia-owl\_director\_resource” & “dbpedia-owl\_producer\_resource” while Emma Thomas is only related via “dbpedia-owl\_producer\_resource”. Thus the total score of Christopher Nolan for “The Dark Knight” will be derived from scores of 2 relationships while that of Emma Thomas will be equal to score of only one of those relationships. Therefore Christopher Nolan is ranked higher than Emma Thomas.

### Entities marked with 3 (Christian Bale & Heath Ledger):


As can be seen, the 4th and 5th ranked entities are related to the queried entity via same relationship “dbpedia-owl\_starring\_resource”. So, to rank these kinds of entities which are related via same relationship to the queried entity, I sorted them based on the total click count of their relationships present in the user profile, in decreasing order.



The screenshot shows a web interface with a search bar. The 'User Name' field contains 'rakhi' and the 'Enter Entity Name' field contains 'Christian\_Bale'. Below the search bar, there is a 'Search Entity' button. The results are listed as follows:

- [Batman Begins](#) -> (dbpedia-owl\_notableWork\_resource)
- [American Psycho](#) -> (dbpedia-owl\_notableWork\_resource)
- [Haverfordwest UK](#) -> (dbpprop\_placeOfBirth\_resource)
- [Los Angeles](#) -> (dbpprop\_city\_resource)
- [United States](#) -> (dbpedia-owl\_country\_resource)

**Figure 17: Relationships of Christian Bale**



The screenshot shows a web interface with a search bar. The 'User Name' field contains 'rakhi' and the 'Enter Entity Name' field contains 'Heath\_Ledger'. Below the search bar, there is a 'Search Entity' button. The results are listed as follows:

- [Perth](#) -> (dbpprop\_placeOfBirth\_resource, dbpprop\_city\_resource)
- [Monash University](#) -> (dbpedia-owl\_university\_resource)
- [Australia](#) -> (dbpedia-owl\_country\_resource)

**Figure 18: Relationships of Heath Ledger**

Figure 17 and 18 above show the relationships of Christian Bale and Heath Ledger and their corresponding related entities, ranked after matching to user profile and context. Christian Bale has “dbpedia-owl\_notableWork\_resource” and “dbpprop\_placeOfBirth\_resource” relationships while Heath Ledger has only “dbpprop\_placeOfBirth\_resource” relationship matching to user’s interests. Therefore, Christian Bale is ranked above Heath Ledger in the recommendation for “The Dark Knight” for the user “rakhi”.

All the relationships that are not present in user profile are ranked at the bottom with a similar approach as above.

### **Username: poonam**

User Profile		User Context	
Relationship Name	Click Count	Relationship Name	Click count
dbpprop_writer_resource	120	dbpedia-owl_starring_resource	10
dbpedia-owl_starring_resource	65	dbpprop_placeOfBirth_resource	3
dbpedia-owl_notableWork_resource	40		
dbpedia-owl_university_resource	27		
dbpprop_placeOfBirth_resource	17		

**Table 2: User profile and User Context for user “poonam” with click counts of relationships**

← → ↻ localhost:8080/rbers/RbersServletWithSolrPlugin?searchEntityButton=Search+Entity&

User Name

Enter Entity Name

[Christian Bale](#) -> (dbpedia-owl\_starring\_resource)  
[Jonathan Nolan](#) -> (dbpprop\_writer\_resource)  
[Christopher Nolan](#) -> (dbpedia-owl\_director\_resource, dbpedia-owl\_producer\_resource)  
[Heath Ledger](#) -> (dbpedia-owl\_starring\_resource)  
[Emma Thomas](#) -> (dbpedia-owl\_producer\_resource)  
[Lee Smith](#) -> (dbpedia-owl\_editor\_resource)  
[Comic Character:Batman](#) -> (dbpedia-owl\_basedOn\_resource)  
[Action Drama](#) -> (dbpedia-owl\_genre\_resource)

**Figure 19: Recommendations for Queried entity “The Dark Knight” for user “poonam”**

Figure 19 shows the recommended entities for the queried entity “The Dark Knight” for user “poonam” with decreasing rank from top to bottom.

**Entity with rank 1 (“Christian Bale”)** is ranked top-most, since it is related to the queried entity via “dbpedia-owl\_starring\_resource” relationship, which is the most clicked relationship in the current context of the user “poonam”.

**Entity with rank 2 (“Jonathan Nolan”)** is ranked second due to its relationship “dbpprop\_writer\_resource” and its click count in the user’s profile.

**Entities ranked 3<sup>rd</sup> and 4<sup>th</sup> (“Christopher Nolan” and “Heath Ledger”):** The interesting part of ranking by the system is depicted in ranking of these entities. The entity “Christopher Nolan” is ranked above “Heath Ledger” even when Heath’s relationship is the most clicked relationship in user’s context while Christopher’s relationships are not even present in the user’s context and profile.

User Name

Enter Entity Name

[The Dark Knight](#) -> (dbpedia-owl\_notableWork\_resource)  
[Inception](#) -> (dbpedia-owl\_notableWork\_resource)  
[London](#) -> (dbpprop\_placeOfBirth\_resource, dbpprop\_city\_resource)  
[Man Of Steel](#) -> (dbpedia-owl\_notableWork\_resource)  
[University College London](#) -> (dbpedia-owl\_university\_resource)  
[Filmmaker](#) -> (dbpprop\_occupation\_resource)  
[Film Industry](#) -> (dbpedia-owl\_industry\_resource)  
[Director](#) -> (dbpedia-owl\_knownFor\_resource)  
[Producer](#) -> (dbpedia-owl\_knownFor\_resource)  
[Science Fiction](#) -> (foaf\_isPrimaryTopicOf\_resource)  
[British](#) -> (dbpprop\_nationality\_resource)  
[American](#) -> (dbpprop\_nationality\_resource)  
[United Kingdom](#) -> (dbpedia-owl\_country\_resource)

**Figure 20: Relationships of Christopher Nolan**

User Name

Enter Entity Name

[Perth](#) -> (dbpprop\_placeOfBirth\_resource, dbpprop\_city\_resource)  
[Monash University](#) -> (dbpedia-owl\_university\_resource)  
[Australia](#) -> (dbpedia-owl\_country\_resource)

**Figure 21: Relationships of Heath Ledger**

This is because Christopher Nolan is related through two relationships to the queried entity, while Heath Ledger is related via only one relationship. Also, from figures 20 & 21, the total click count of relationships in user profile for Christopher (“dbpedia-owl\_notableWork\_resource”, “dbpedia-owl\_university\_resource” and “dbpprop\_placeOfBirth\_resource”) is  $40 + 27 + 17 = 84$  while that

for Heath (“dbpedia-owl\_university\_resource” & “dbpprop\_placeOfBirth\_resource”)is **27 + 17 = 34**.

*This difference in sum of click counts shows that the user will be more interested in Christopher Nolan as compared to Heath Ledger even though the direct relationships of Christopher is not present in user profile.*

### **Username: newuser1**

User Profile		User Context	
Relationship Name	Click Count	Relationship Name	Click count
<None clicked yet in profile>		<None clicked yet in context>	

**Table 3: User profile and User Context for user “newuser1” with click counts of relationships**

User Name

Enter Entity Name

[Lee Smith](#) -> (dbpedia-owl\_editor\_resource)  
[Christopher Nolan](#) -> (dbpedia-owl\_director\_resource, dbpedia-owl\_producer\_resource)  
[Christian Bale](#) -> (dbpedia-owl\_starring\_resource)  
[Heath Ledger](#) -> (dbpedia-owl\_starring\_resource)  
[Emma Thomas](#) -> (dbpedia-owl\_producer\_resource)  
[Comic Character:Batman](#) -> (dbpedia-owl\_basedOn\_resource)  
[Action Drama](#) -> (dbpedia-owl\_genre\_resource)  
[Jonathan Nolan](#) -> (dbpprop\_writer\_resource)

**Figure 22: Recommendations for Queried entity “The Dark Knight” for “newuser1”**

Since there is no information about user’s preferences for relationships, the entities are ranked randomly.



## 6.2 Query results for same user in presence and absence of user context

To show the results of how our hypothesis improves the relevance of results, we conducted an experiment by running a query to search for the entity “The Dark Knight” for the user “rakhi”, both in the presence and absence of user context. We use the same user profile and context as shown in table 1. It shows that the user is currently interested in “dbpprop\_writer\_resource” and “dbpedia-owl\_notableWork\_resource” relationships, which have comparatively lower click count in the user profile.

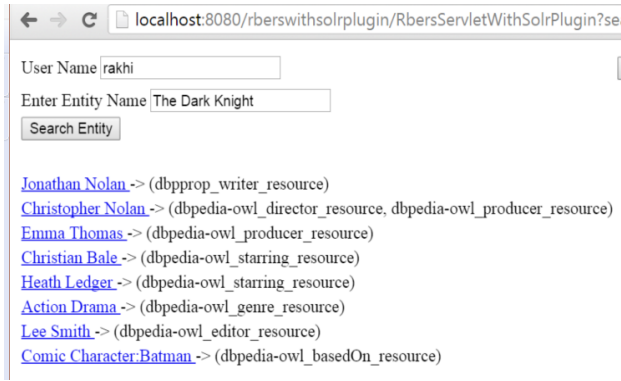


Figure 23: Result in presence of user context

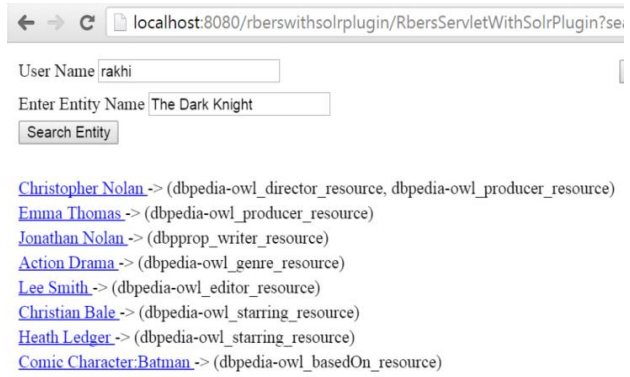


Figure 24: Result in absence of user context

Figure 23 and 24 show the results of searching of the entity “The Dark Knight” for user “rakhi”. Figure 23 shows the results when the user context contains relationship click count as shown in Table 1. Figure 24 shows the results when the system does not take user context into account (we achieved this by clearing the context). Since, the user is currently interested in looking at “writer” and “notable work” relationships, figure 23 shows results with higher relevancy as compared to results in figure 24, because it shows “Jonathan Nolan” before “Christopher Nolan” and “Christian Bale” before “Action Drama”. This is because “Jonathan Nolan” is related to “The Dark Knight” via the “writer” relationship, which makes it important for us to show it before the “director” and “producer” relationships, since the user is currently more inclined towards it. Similarly, “Christian Bale” has some notable work listed in our dataset, while the entity “Action Drama” does not have any notable work. So, showing “Christian Bale” before “Action Drama” makes it relatively more relevant for the user. The other entities like “Christopher Nolan” and “Emma Thomas” are shown before “Christian Bale” because they also have “notable work” listed in the dataset, plus they are

also related to “director” and “producer” relationships in which the user has shown more interest as compared to “starring”, as shown in user profile in table 1.

## 6.3 Improvements made after Experiments

After performing some experiments, we made some improvements and optimizations in our approach based on the analysis of the results. Following are the two major changes we did in our system to improve its performance and relevance of the results:

### 6.3.1 Move implementation from client side to a plugin in solr server

As discussed in sections 5.2.1 and 5.2.2, there are two ways to implement our proposed algorithm. First is implementing it on client side using SolrJ APIs, and second is to extend existing Solr classes and implement a plugin that runs in Solr server itself.

```

Java - rbers-with-solr-plugin/src/rbers/web/RbersServletWithSolrPlugin.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Tomcat v7.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jre7\bin\javaw.exe (Nov 30, 2015, 11:40:24 PM)
Total Running time for user: rakhi for entity Les_Anges_de_Satan is : 58 ms for webapp rbers
Total Running time for user: rakhi for entity Les_Anges_de_Satan is : 24 ms for webapp rberswithsolrplugin
Total Running time for user: rakhi for entity Les_Anges_de_Satan is : 52 ms for webapp rbers
Total Running time for user: rakhi for entity Les_Anges_de_Satan is : 25 ms for webapp rberswithsolrplugin
Total Running time for user: rakhi for entity Les_Anges_de_Satan is : 58 ms for webapp rbers
Total Running time for user: rakhi for entity Les_Anges_de_Satan is : 31 ms for webapp rberswithsolrplugin

Total Running time for user: rakhi for entity Leo_Beuerman is : 80 ms for webapp rbers
Total Running time for user: rakhi for entity Leo_Beuerman is : 23 ms for webapp rberswithsolrplugin
Total Running time for user: rakhi for entity Leo_Beuerman is : 89 ms for webapp rbers
Total Running time for user: rakhi for entity Leo_Beuerman is : 26 ms for webapp rberswithsolrplugin
Total Running time for user: rakhi for entity Leo_Beuerman is : 76 ms for webapp rbers
Total Running time for user: rakhi for entity Leo_Beuerman is : 25 ms for webapp rberswithsolrplugin

```

**Figure 25: Execution time of searching two entities on the two mentioned implementations**

Figure 25 above, shows execution times of the two implementations of our approach. The webapp named “rbers” represents the implementation on the client side, and the webapp “rberswithsolrplugin” represents the implementation by extending the solr classes. We randomly selected 2 movies – “Les\_Anges\_de\_Satan” and “Leo\_Beuerman” - from our data set and searched for them in both of our implementations. We searched for the same entity multiple times using both implementations alternatively for 3 times for each entity.

Entity Name	Query #	Client side (Time in ms)	Solr plugin (Time in ms)
Les_Anges_de_Satan	1	58	24
	2	52	25
	3	58	31
Leo_Beuerman	1	80	23
	2	89	26
	3	76	25

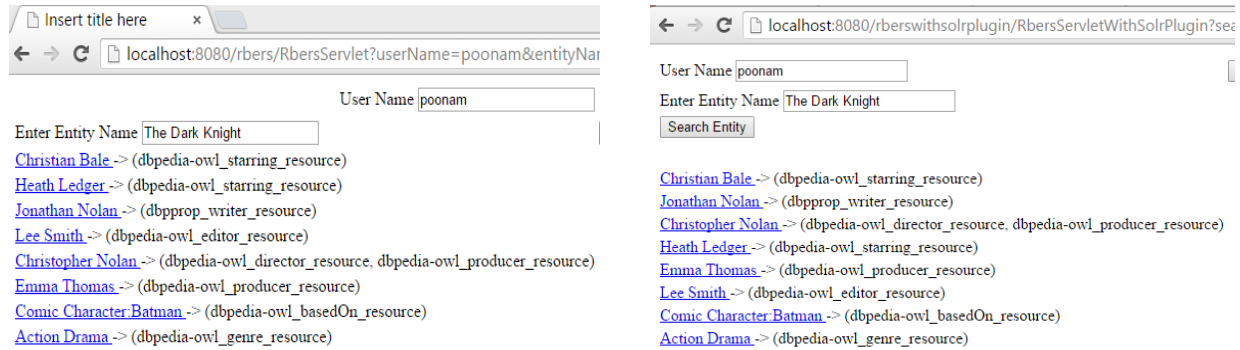
**Table 4: Execution time of searching two entities on the two mentioned implementations**

As can be seen from Table 4, the system that extends solr classes and runs our algorithm as a custom solr plugin has better running time for each query execution of both entities. Therefore, the above results prove that implementing the algorithm as a solr plugin is a better approach, than implementing as a client side application.

### 6.3.2 Ranking related entities by applying appropriate weights

In the initial implementation of our algorithm, we recommended the related entities for a queried entity based on the click count of relationships present in the user profile. So, all the entities which were related through same relationship to the queried entity were grouped together. As we ran some queries on the dataset with this approach, we found out that there are 2 flaws in the approach. First, if an entity has large number of entities related to itself via same relationship, all those entities will show up together. This creates an issue for the system, because if the user is not interested in entities for a particular relationship at a given time, a large portion of the result might be considered as irrelevant by the user, thus reducing the overall satisfaction of the user. Second, if there are some related entities which matches user's interest but are connected via less clicked relationships than some less matching related entities which are related to highly clicked relationships to the queried entity, then there is no way for the system to show those entities above the less relevant entities. This observation helped us to find a new issue in the approach, that ranking entities based on just the click count of relationships of the queried entity, can lead to less relevant results. So, with these 2 findings from our experiment, we decided to use a weighing technique to rank entities

not only based on the click count of relationships of the queried entity, but also on the click count of relationships of the related entities.



**Figure 26: Results for user “poonam” before and after applying updated weighing scheme**

Figure 26 shows an example of the change in the results before and after the change suggested above was made to the algorithm. As can be seen, the place of the entity “Christopher Nolan” has changed and it has moved up than the entity “Heath Ledger” as compared to the result returned by the initial approach. This change can be understood by looking at figures 20, 21 and table 2 above, which shows the relationships of “Christopher Nolan” and “Heath Ledger” and the details of user profile for user “poonam” respectively. The figures and the tables show that the click count of relationships of “Christopher Nolan” is higher than “Heath Ledger”, which indicates that the user might be more interested in “Christopher Nolan” than “Heath Ledger”. So, with the updated approach, we were able to increase the relevance level of results from our algorithm by using extra information that we have on the related entity’s relationships.

## **CHAPTER 7**

### **Conclusion**

In this paper, we discussed the importance of the content of search results and how they can be improved by adding some kind of personalization to the results, so that each user will get recommendations based on his interests. We examined different approaches followed by many different existing systems, and analyzed their solutions and their drawbacks in recommending relevant related entities to user's interest. With the help of this investigation, we derived our hypothesis and developed an algorithm that overcomes the issues found in the existing systems. While implementing our algorithm, we concluded that a high degree of relevancy cannot be achieved by just using the click count of relationships of the queried entity. We soon realized that, a second round of ranking should be applied, by taking the click count of relationships of the related entities into consideration. This helped to provide better representation of the degree of interest of the user in those entities. We measured the performance of different approaches to use the solr server and deduced that the performance of the overall system is far better when the algorithm runs as a plugin in the solr server itself as compared to the approach when the algorithm runs in the client side application. In the previous section, the attached results show that even when two users search for the same entity, different entities or different ordering of entities are shown based on each user's preferences. So, in the end, we would like to conclude that using relationships which are more sophisticated than just "is-a" and "has-a" relationships and a user profile that stores user's interest in relationships can be very useful to recommend highly relevant entities to the users.

## CHAPTER 8

### References

- [1] Pound, J., Mika, P., Zaragoza, H., Ad-hoc object retrieval in the web of data. New York, NY, USA: 19<sup>th</sup> international conference on World wide web, 2010.
- [2] Blanco, R., Cambazoglu, B., Mika, P., Torzec, N., Entity Recommendations in Web Search.
- [3] Cantador, I., Bellogín, A., Castells, P., Ontology-based Personalized and Context-aware Recommendations of News Items.
- [4] IIntema, W., Goossen, F., Frasincar, F., Hogenboom. F., Ontology-Based News Recommendation.
- [5] Tran, T., Cimiano, P., Ankolekar. A., A Rule-based Adaption Model for Ontology-based Personalization.
- [6] Tan K, SolrTutorial.com, <http://www.solrtutorial.com/basic-solr-concepts.html>
- [7] Grainger T, Timothy Potter. Manning publications, 2014. Solr in Action
- [8] Apache Solr Reference Guide,  
<https://cwiki.apache.org/confluence/display/solr/RequestHandlers+and+SearchComponents+in+SolrConfig#RequestHandlersandSearchComponentsinSolrConfig-SearchComponents>
- [9] Solr Wiki, <http://wiki.apache.org/solr/SearchHandler>
- [10] SearchComponent Javadoc, [http://lucene.apache.org/solr/5\\_3\\_1/solr-core/org/apache/solr/handler/component/SearchComponent.html](http://lucene.apache.org/solr/5_3_1/solr-core/org/apache/solr/handler/component/SearchComponent.html)
- [11] Nguyen H., San Jose State University, URL: [http://scholarworks.sjsu.edu/etd\\_projects/398/](http://scholarworks.sjsu.edu/etd_projects/398/)