Development of An Interactive Job Matching System

Xixi Xiao*
Computer Science and Engineering Department
University of California, San Diego
x3xiao@eng.ucsd.edu

ABSTRACT

Selection and recommendation systems are widely used on the Internet to assist customers in finding the products or services that best fit with their individual preferences. Recently, job seeking and recruiting websites have been experiencing a striking rise. Websites like *Monster.com*, *Indeed.com* are currently dominate this market. As the amount of information grows, an accurate job searching system is necessary to help job applicant find the desired jobs.

In traditional search engines, the searching process is usually done without much involvement of the user. Full-text documents are usually analyzed and indexed by words or phrases. When a user query comes, which are often a few key words, the index helps find information relating to the query as quickly as possible. However, the order and semantic information of words are ignored.

However, job and resume data are more than just full-text documents since they have relative fixed structure. With the emergence of *AsterixDB*, a big data management system that supports ingesting, storing, managing, indexing, querying, and analyzing vast quantities of semi-structured information, we are able to index hierachical-structured data and utilize structural information in searching without knowing the exact schema of data.

Besides, by interacting with user using background and context information in the querying process, more accurate matching can be achieved.

We have built an application in the scenario of job-hunting to illustrate the ideas. The dataset is consisted by a bunch of job documents and user profiles (semi-structured), which are manually acquired from available online resources. A DSL (Domain Specific Language) is designed to facilitate the user query.

Keywords

AsterixDB, query processor, question generation, semi-structured, Solrj, user interaction

*A master student at University of California, San Diego

1. INTRODUCTION

Most of the traditional search engines utilize the same idea to index and query data. Either the data have fixed linear relational structure or they are indexed and matched regardless of the semantic and structural information. With the advent of Internet, semi-structured data is increasingly occurring. Semi-structured data is a form of structured data that does not conform with the formal structure of data models associated with relational databases or other forms of data tables, but nonetheless contains tags or other markers to separate semantic elements and enforce hierarchies of records and fields within the data. Therefore, it is also known as self-describing structure. Since this is a more flexible data form, it's widely used in data representation and exchange system. The structure itself contains semantic information that can be utilized for searching. Two typical types of semi-structured data are *XML* and *JSON*.

AsterixDB, a result of about 3.5 years of R&D involving researchers at UC Irvine, UC Riverside, and UC San Diego, is a big data management system which supports ingesting, storing, managing, indexing, querying, and analyzing vast quantities of semi-structured information. With the help of this innovative platform, we are able to index semi-structured data and retrieve useful structural and semantic information.

We use another platform to index the extracted information from AsterixDB. Solr is an open source enterprise search platform from the Apache Lucene project. Solr is written in Java and runs as a standalone full-text search server within a servlet container such as Apache Tomcat or Jetty. The powerful external configuration allows it to be tailored to many types of application. Both the extracted structural information and the original text data can be indexed within Solr and be queried efficiently. The combination of these two platforms enables efficient query on both the text information and the structural information of the semi-structured data.

Another idea we proposed here is the user interaction in the searching process. In traditional search engine, the searching process is done in a single pass, which means that user inputs a query, the search engine searches the database and gets back to the user with related information. With limited information provided by the user, it's hard to find exactly what the user is looking for. By interacting with user, it's possible to generate questions using the current context and narrow down the searching scope dynamically according to user's answer. In other words, implicit user interest and skills can be probed automatically. The information can be

inversely used for finding better fit.
We built an application in the scenario of job-hunting to illustrate these ideas. Job data and user profiles are collected and indexed. User input a query stating job constraints (company name, area, title, salary, etc.). The search engine interacts with both the database and user to narrow down the searching scope as well as discovering important attributes for the user to get the desired jobs. A personalized profile will be generated accordingly.
Since natural language is hard to process, we define a DSL (Domain Specific Language) for the user to query. The design of DSL can be changed according to the requirements.
The rest of the paper is organized as follows. In section 2, we give a brief introduction to the related work. Section 3 describes the system structure while section 4 details the design and implementation of each module. Future work and possible improvements are states in section 5. Finally section 6 concludes the work.

2. RELATED WORK

Multiple job selection systems have been researched to improve the accuracy.
Some researches focus on the representation of knowledge in the given domain. In [6], an ontology-based mathod has been proposed for knowledge representation. The use of ontological descriptions for representing concepts according to a hierarchical structure helps exploit the semantic relationship between both job descriptions and user profiles. Based on the representation, a ranking algorithm allowing for qualification comparison has been designed. Maryam Fazel-Zarandi and Mark S. Fox presented an ontology-based hybrid approach in [2] to effectively match job seekers and job postings as well. A deductive model has been proposed and a similarity-based ranking algorithm has been designed for recommendation.
Personalized and dynamical recommendation system draws a lot of attentions as well. In [1], Ioannis Paparrizos, B. Barla Cambazoglu and Aristides Gionis trained a machine learning model by exploiting all past job transitions as well as the data associated with employees and institutions to predict an employee's next job transition. Wenxing Hong, Siting Zheng and Huan Wang [8] researched on a dynamic user profile-based job recommender system. Features are updated and extended dynamically based on the information and behaviors on the recruiting website which belong to the job applicants. Then a similarity calculation algorithm is proposed to match jobs with job applicants.
Many text mining techniques has been used for discovering more useful information from resumes(user profiles). For example, [5] classify the job applicants by using their resume in the method of text mining for improving the resource utilization and demand fulfill in the Resource Planning system that they build.
The recommendation algorithm is also the core of the system and vital to the accuracy. Basic approaches tend to be content-based. Features are extracted and similarity is computed, which can be used for ranking results. Collaborate filtering approach has been introduced to improve the performance. Hybrid approach has been proved to have a higher accuracy [4].

3. SYSTEM OVERVIEW

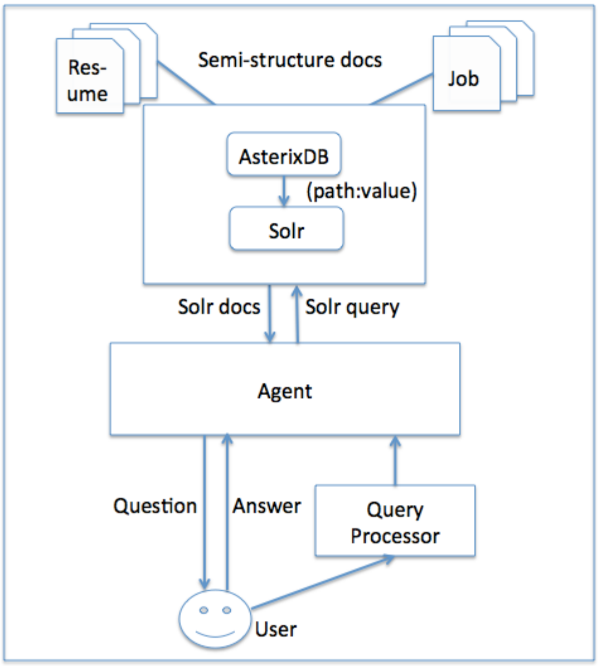

Figure 1: The system structure

The structure of the system is shown in Figure 1.
The semi-structured data, job data and resume data, are stored and indexed using *AsterixDB*. The JSON-like semi-structured data is actually a hierarchical data structure. It can be easily transformed to a tree structure. Each leaf node corresponds to a text field value while the root-to-leaf path is consisted of all the attributes, or tag names in *XML*, from the outer-most level to the inner-most level. We thus design an algorithm to extract all root-to-leaf paths and their corresponding text values from *AsterixDB*. In this way, the hierarchical data has been transformed to linear-structured data on which efficient indexing and searching methods can be performed. The path-value pairs are then organized and indexed in *Solr*. Each document in *solr* is consisted of all related path-value pairs and all theses pairs are indexed as key-value pairs.
A *DSL* has been designed for the query process. The syntax will be introducted in section 4. By parsing the user query, predicates(job constraints) can be acquired to form solr query sentences. Performing those queries in solr gives us a bunch of jobs the user may be interested in. If the number of jobs is within a reasonable range, the system can respond to the user query immediately. However, the number of jobs may either be too few, due to the strict constraints from user, or be too many if the user query is too general.
If the user query is too strict, *query relaxation* will be performed by the agent. The query relaxation techniques are usually used to expand query result set. Many algorithms have been designed to do query relaxation. For example, in [3], they focus on relaxing query conditions on numeric attributes. Say we are seeking for job-candidate pairs, such that the job and the candidate are in the same area (same

zip code), the job's annual salary is at most $95K$, and the candidate has at least 5 years of working experience. The result set for this query may be empty if no such pair of records satisfy both selection conditions. However, by relaxing both selection conditions on *salary* and *work year*, we can get a nonempty answer. For instance, for a condition *salary* <= 95, we could relax it to *salary* <= 100 or *salary* <= 120. In our work, the query relaxation is completed by dropping some predicates. Dropping predicates may lead to a large query result set, the same thing that will happen when the query conditions are too general. In this case, the interaction process will begin. During the interactive process, certain questions will be dynamically generated based on the current context and then delivered to the user. According to the answers provided by the user, a personal profile will be dynamically generated. User answers will be used to filter out non-related jobs as well since new predicates could be extracted. In a nutshell, the input of the system should be a user query that conforms to the syntax of the designed *DSL* while the output of the system are a personal profile and a resonable number of jobs. The content of personal profile and the jobs being recommended are closely related to the user answers.

4. ALGORITHMS AND IMPLEMENTATION DETAILS

The system is mainly consisted of three modules: data module, query processor and interactive agent. The rest of this section will detail the implementation of each module.

4.1 Data module

As mentioned in Section 1, two existing platforms, *AsterixDB* and *solr* are used in this module to store and index data. We use AsterixDB to manage original semi-structured documents and we use solr to store and index the information extracted from AsterixDB for efficient information retrieval in the subsequent interactive process.

AsterixDB

There are three basic concepts-*dataverse*, *datatype* and *dataset* in AsterixDB. The top-level organizing concept in the AsterixDB world is the dataverse. A dataverse-short for "data universe"-is a place (similar to a database in a relational DBMS) in which to create and manage the types, datasets, functions, and other artifacts for a given AsterixDB application. A datatype tells AsterixDB what you know (or more accurately, what you want it to know) a priori about one of the kinds of data instances that you want AsterixDB to hold for you. A dataset is a collection of data instances of a datatype, and AsterixDB makes sure that the data instances that you put in it conform to its specified type. Since AsterixDB targets semistructured data, you can use open datatypes and tell it as little or as much as you wish about your data up fron. Here is an example for the definition of datatypes *Job* and .

```
create type Job as open{
    id: int16,
    company_name: string,
    title: string,
    requirement: requirementType,
    location: string?
}

create type requirementType as open{
    minimum_degree: string?,
    discipline: [string]?
}
```

In this example, datatype *Job* and *requirementType* with several known fields is created. Some fields are of primitive types such like string, fields can also be of self-defining ADM(AsterixDB Data Model) types such like the requirementType here. There is a special dataverse in AsterixDB called *Metadata* in which there is a special dataset named *Dataset*. This dataset contains the AsterixDB system catalog. There is a record in this dataset for any dataset in any dataverse. For example, if we use a dataverse *searchAgent* to manage the job documents and user profiles, and we use two datasets-Jobs and Resumes-to store the corresponding data, then there will be a record in *Dataset* corresponds to the dataset Jobs in dataverse searchAgent and there will be another record in *Dataset* corresponds to dataset Resumes in dataverse searchAgent. Therefore, as long as we know the name of the dataverse used for managing our data, we are able to acquire the information of all datasets in that dataverse.

AsterixDB has provided us a *XQuery*-like query language to talk to it. With the information of datasets and the REST API provided by AsterixDB, we can get a JSON Object for each record in each dataset. By analyzing this JSON Object, several path-value pairs can be acquired, which we can use to index with solr. Say we have the following data record in the dataset that conforms to the Job datatype we defined above.

```
{
  "id":int16("101"),
  "company_name":"LinkedIn",
  "title":"Software Engineer",
  "requirement":{"minimum_degree":"Master","discipline":["CS"]},
  "location": "mountain view",
  "compensation": "$66/hour",
  "career level": "entry level"
}
```

Based on this record, we are able to extract several path-value pairs such as Pair(dataset.requirement.discipline, "CS"). I use *google-gson*, a JAVA library to help analyze the JSON Object got from the AsterixDB server. The algorithm for extracting information from AsterixDB and converting JSON Object to corresponding path-value pairs is described in Algorithm 1.

The algorithm used in step 6 is very much like a DFS(depth-first-search) algorithm. A JSON object is actually a tree sructured object, therefore by starting from the outer-most level(just like the root of the tree) and keeping track of the current path, which is actually a prefix of the final paths of all sub-objects, we can get all root-to-leaf paths by traversing the tree back and forth. The recursive algorithm is de-

Algorithm 1: Extract path-value pairs

Require: Dataverse name D
Ensure: A set of path-value pairs P
Initialize $P \leftarrow \emptyset$,
Query Metadata of AsterixDB, $S \leftarrow$ get all datasets within the dataverse D.
for $d \in S$ **do**
$R \leftarrow$ All records(JSON Object) of dataset d
for each JSON Object $js \in R$ **do**
$P \leftarrow P\cup$ DFS-JSON(js, $d.name$)
end for
end for
return P

scribed in Algorithm 2.
Solr

Algorithm 2: DFS-JSON

Require: A JSON Object js, a string p denotes the current path prefix
Ensure: A set of path-value pairs P
Initialize $P \leftarrow \emptyset$
for $\{key, value\} \in js$ **do**
$p \leftarrow p$ + "." + key
if $value$ is a JsonArray **then**
for $jse \in value$ **do**
if jse is a JsonObject **then**
DFS-JSON(jse, p)
else
$P \leftarrow P + Pair(p, value)$
end if
end for
else if $value$ is a JsonObject **then**
DFS-JSON($value$, p)
else
$P \leftarrow P + Pair(p, value)$
end if
$p \leftarrow$ drop the last part of p
end for
return P

Solr is a powerful and flexible standalone enterprise search server with a REST-like API. It supports many types of documents and various methods can be used to import data to Solr. We use *Solrj*, a java client, to programatically create documents to send to Solr.
The document sent to solr is very much like a key-value store, however in Solr, key is called *field*. There is a very important configuration file in Solr - the *schema.xml* file-which contains all of the details about which fields your documents can contain, and how those fields should be dealt with when adding documents to the index, or when querying those fields. This configuration file gives solr great flexibility. A fantastic feature we employed when indexing the path-value pairs is the *Dynamic Fields* feature of solr. This feature enables on-the-fly addition of new fields. Since we extracts all path-value pairs and insert them into solr on-the-fly, this feature is exactly what we are looking for. For each job document, we construct a corresponding solr document and insert all related path-value pairs into it. We use the path as field name, which is stored and indexed, and the value is linked to the corresponding field. For user profiles, however, we use two fixed fields-path and word-to store the path-value pairs. The different designs are determined by the information retrieval demands in the subsequent interactive process. For job documents, we need to extract corresponding words of a given path, therefore we can use path as key to index text. Given a path value p, we can simplely query solr with the condition "p:*" to retrieve all related documents. By iterating these documents and retrieve the values of field p, the text values corresponding to path p can be gathered. However, for user profiles, the interaction algorithm requires that given a specific text value, all corresponding paths can be retrieved. Some people may argue that we can inversely use text as indexed field. Unfortunately, the given text value may not be a single word, such like "software engineer", which cannot be indexed properly according to our experiments. An alternative way to index the path-value pairs is to use two fixed fileds and use "AND" to join the two conditions. For example, we have a path-value pair Pair(Resumes.name, "Sissy"), the way we index it is to create a solr document {path: Resumes.name, word: "Sissy"}(there may be other fields in this documents). If we want to retrieve all corresponding path of the word "Sissy", we can use the query "word: Sissy" to retrieve all documents that contains the keyword "Sissy". By iterating these documents we can retrieve all values of the field path.

4.2 Query processor

The most ideal query and interactive language between human and computer is natural language. However, understanding natural language and extracting accurate inforamtion is challenging. For many dominant job hunting websites like *Monster*, *glassdoor*, a simple way to gather query information is to ask user to provide values for several fixed fields. For example, Monster.com asks user to fill in three text boxes-keywords, city and state-before starting matching jobs.
In this project, we have provided a more flexible query method. We have designed a DSL to facillitate the query process. It's more flexible and user friendly since the design of the syntax of DSL can vary from the actual requirements. We are able to design a DSL with the same grammar of natural languege and we are able to inject logical operations between query conditions by special design. Here is the basic grammar for the user query language.

Query $\rightarrow$ Seeking *Job*

Job $\rightarrow$ *Title Company Area Salary* | *Job* or *Job*

Title $\rightarrow$ *TitleDictionary*

Company $\rightarrow \epsilon$ | in *CompanyName* | in *CompanyType*
| preferably in *CompanyName* | preferably in *CompanyType*

CompanyName $\rightarrow$ *StringConstant*
| *CompanyName* or *CompanyName*

CompanyType $\rightarrow$ *CompanyTypeDictionary*
| CompanyType or CompanyType

Area $\rightarrow \epsilon$ | in *AreaName* | around *AreaName*
| preferably in *AreaName* | preferably around *AreaName*

AreaName $\rightarrow$ *AreaDictionary*
| *AreaName* or *AreaName*

Salary $\rightarrow \epsilon$ | with a salary of *Number* per *Unit*

StringConstant
$\rightarrow ([\text{"}a\text{"} - \text{"}z\text{"}, \text{"}A\text{"} - \text{"}Z\text{"}, \text{"\_"}, \text{"} - \text{"}, \text{"0"} - \text{"9"}])+$

Number $\rightarrow ([\text{"0"} - \text{"9"}])+$

Unit $\rightarrow$ "month" | "hour" | "year"

We omitted the definition for *TitleDictionary*, *CompanyTypeDictionary* and *AreaDictionary* here, which are lists of candidate words. The above grammar helps to check whether an input user query has the correct syntax or not. With the help of *JavaCC*, we have implemented a parser for this DSL. When a valid user query comes, the parser will generate an AST(Abstract Syntax Tree). By navigating along the AST, query conditions and logical operations can be extracted, which can be then transformed to solr query sentences.
For example, here is a valid user query:

"seeking a software engineer position preferably around San Francisco or Mountain View or San Diego"

By parsing this query, several path-value pairs can be extracted:

Pair("Jobs.title", "software engineer")

Pair("Jobs.location", "San Francisco")

Pair("Jobs.location", "Mountain View")

Pair("Jobs.location", "San Diego")

Besides these pairs, the keywords "preferably", "around" will be recorded. The corresponding Solr query will be:

Jobs.title: "software engineer" AND (Jobs.location:"San Francisco" OR Jobs.location:"Mountain View" OR Jobs.location:"San Diego")

The keywords "preferably" and "around" here can be utilized for query relaxation.

4.3 Interactive agent

A collection of qualified job documents will be retrieved according to the input user query. As we discussed before, a large result set makes little sense. By interacting with user may provide us more useful information and help us figure what exactly the user is looking for.
The interaction process involves asking user questions and collecting their answers. The questions should be generated dynamically and the user answers should be used to narrow qualified job documents set. Meanwhile, a personal profile for the user will be automatically generated based on user's answers to those corresponding questions.

Question generation
Before generating questions, we need to figure out what attributes are essential for the user to get the job he or she desires. The algorithm we designed to get those attributes is described in Algorithm 3.

Algorithm 3: Extract attributes

Require: A collection of job documents J, a frequency threshold θ
Ensure: A set of paths and their corresponding histograms $R = \{p, h\}$

Initialize $P \leftarrow \emptyset, R \leftarrow \emptyset, T \leftarrow \emptyset$
for $d \in J$ **do**
$T.wordlist \leftarrow T.wordlist+$ all field values of d
Update the number of occurrences of all field values of d
end for
According to T, construct a histogram H to record the number of occurences of each word/phrase. All related paths to a specific word/phrase should be recorded as well.
Sort the words/phrases by their occurrence frequencies
$W \leftarrow \emptyset$
for $u \in H.wordlist$ **do**
if $u.frequency > \theta$ **then**
$W \leftarrow W + u$
end if
end for
$P \leftarrow \emptyset$
for $w \leftarrow$ each word in W **do**
$P \leftarrow P+$ all paths related to w in any user profiles indexed in Solr
end for
for $p \in P$ **do**
Construct a histogram h to record the number of occurrences of all values related to p in any user profiles, $R \leftarrow R + \{p, h\}$
end for
return R

The basic idea of the algorithm is that if a path in a user profile corresponds a word/phrase that appears several times in the job documents the user may be interested in, then the path, or we can say the attribute, may be essential. For example, the word "JAVA" appears many times in several qualified job documents, and we find that people usually mention the word "JAVA" when they stating their "programming language", then it's reasonable to say that "programming language" is an important attribute for job matching. Therefore, a proper question to ask the user is "What programming language are you familiar with?".

Job refilter
User answers provide us with additional information, which contributes to more accurate job matching. As we described

above, a question is generated based on a path and for each path, we have constructed a histogram to record expected values for that path. If the user answer matches one of those high frequency words/phrases, we'll add this information to user's profile and use this answer for filtering jobs. Otherwise, questions will be designed to further explore the user's abilities. Algorithm 4 describes how we utilize user answers to form more query predicates and refilter candidate jobs.

Algorithm 4: Job refilter

Require: A set of words W, a collection of current qualified jobs J, a set of words and all paths related to each word $S = \{w, P\}$
Ensure: A collection of jobs R
1: Initialize $R \leftarrow \emptyset, C \leftarrow \emptyset$
2: **for** $u \in W$ **do**
3: **for** $\{w, P\} \in S$ **do**
4: **if** $u = w$ **then**
5: **for** $p \in P$ **do**
6: $C \leftarrow C + Pair(p, w)$
7: **end for**
8: **end if**
9: **end for**
10: **end for**
11: Form solr query according to C and J, get new job collection R
12: **return** R

Profile generation
A set of question-answer pairs, in other words, path-value pairs can be acquired during the interactive process. These pairs record the information that are essential for the user to be qualified for the jobs he or she is looking for. Therefore, we automatically organized and formatted the information to create a personal profile for the user, which is similar to a resume.
The interactive agent is just like a state machine. It maintains the current context, generate questions based on the current context to interact with user and then update the context according to user's feedback.

5. DISCUSSION
The idea of this interactive search agent is novel and the implementation is a big challenge. The system actually involves many computer science topics.
The first part relates to the representation of the knowledge. As we stated in section 2, serveral methods have been proposed including ontological representation. The advantage of ontology-based knowledge representation is obvious. The hierarchical structure itself contains semantic information. Machine learning algorithms can be easily applied to it. It is possible to integrate the idea with our system and apply related machine algorithm to pre-process attribute values.
There are actually a lot of text-processing related work we can do to improve the matching accuracy. We are now doing the exact word matching. However, the existence of synonyms asks for clustering words that have similar meaning. For example, "software engineer" and "programmer" should be considered in the same group. Many researchers in the data mining field have dealt with this issue.
Besides, the DSL provides great flexibility for the searching and matching process. For example, the keyword "preferably" can be used in multiple ways. One way I could think about is to asssign a weight to the corresponding query conditions. The weight information tells the agent which features are more important and which are less, which we believe can be utilized to design suitable algorithms to improve performance.
A big problem we faced during the design process is how to generate proper easy-understanding questions. [9], [7] proposed some ideas to deal with the question generation problem. Unfortunately, none of those works address our problem. Firstly, in our system, we need to generate a question given a specific path. It's much harder than transforming a narrative sentence to a corresponding question. Different sentence patterns should be applied to different vocabularies. Besides, the order of the question should be concerned as well. For example, we may have two paths, "Education.university_name" and "Education.graduation_date", based on which we need to generate questions. In the correct logic, the question should be "Could you tell me more about your education, what university have you attended? When did you graduate from the university?" rather than "Could you tell me more about your education, when did you graduate from the university? What university have you attended?". This is a crossing area of linguistic and computer science, which makes it really challenging.

6. CONCLUSION
In this paper, we presents an interactive job searching and recommendation system.
Combined with the content-based information, we utilized the structural information of user profiles to improve matching accuracy. We employed two powerful databases-AsterixDB and Solr-to store and index data. We designed the algorithm to retrieve structural information from AsterixDB as well as configuring the proper indexing schema for Solr according to the searching demands.
We have proposed an interactive model to improve matching accuracy. The interactive process between user and agent helps discover important attributes and prob implicit features of the user. This is actually a bilateral selection process. Users express their preferences for the job, by analyzing job data and user constraints, important vocabularies are extracted, based on we are able to figure out essential attributes for the user to get the desired job. The user answers in return help the agent find better fitted jobs for them.
The automatically generated personal profile is a bonus of this system. The entries in the profiles are exactly those essential attributes the user should have to be competent for the jobs they desired.

7. REFERENCES
[1] Ioannis Paparrizos, B. Barla Cambazoglu, Aristides Gionis. Machine Learned Job Recommendation. *RecSys*, 2011.
[2] Maryam Fazel-Zarandi, Mark S. Fox. Semantic Matchmaking for Job Recruitment: An Ontology-Based Hybrid Approach. *In Proceedings of the 3rd International Workshop on Service Matchmaking and Resource Retrieval*, 2009.

[3] Nick Koudas,Chen Li, Anthony K. H. Tung, Rares Vernica. Relaxing Join and Selection Queries. *VLDB '06 Proceedings of the 32nd international conference on Very large data bases*, pages 199–210, 2006.

[4] Prem Melville, Raymond J. Mooney and Ramadass Nagarajan. Content-Boosted Collaborative Filtering. *In Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 187–192, 2002.

[5] Tere Gonzalez, Pano Santos, Fernando Orozco, Mildreth Alcaraz, Victor Zaldivar, Alberto De Obeso, Alan Garcia. Adaptive Employee Profile Classification for Resource Planning Tool. *In Proceedings of SRII Global Conference 2012 Annual*, pages 544–553, 2012.

[6] Valentina Gatteschi, Fabrizio Lamberti, Andrea Sanna, Claudio Demartini. A Semantic Matchmaking System For Job Recruitment. *Proceedings of I-KNOW 2010*, pages 50–59, 2010.

[7] Wei CHEN, Gregory AIST, Jack MOSTOW. Generating Questions Automatically from Informational Text.

[8] Wenxing Hong, Siting Zheng, Huan Wang. Dynamic User Profile-Based Job Recommender. *The 8th International Conference on Computer Science & Education (ICCSE 2013)*, 2013.

[9] Xuchen Yao, Gosse Bouma, Yi Zhang. Semantics-based Question Generation and Implementation. 2012.