

Supporting Entity Search: A Large-Scale Prototype Search Engine

Tao Cheng, Xifeng Yan, Kevin Chen-Chuan Chang
University of Illinois at Urbana-Champaign
{tcheng3, xyan, kcchang}@cs.uiuc.edu

ABSTRACT

As the Web has evolved into a data-rich repository, with the standard “page view,” current search engines are increasingly inadequate. While we often search for various data “entities” (e.g. phone number, paper PDF, date), today’s engines only take us indirectly to pages. Therefore, we propose the concept of *entity search*, a significant departure from traditional document retrieval. Towards our goal of supporting entity search, in the *WISDM*¹ project at UIUC we build and evaluate our prototype search engine over a 2TB Web corpus. Our demonstration shows the feasibility and promise of a large-scale system architecture to support entity search.

Categories and Subject Descriptors

H.3.3 [INFORMATION STORAGE AND RETRIEVAL]: Information Search and Retrieval; H.3.5 [INFORMATION STORAGE AND RETRIEVAL]: Online Information Services

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Entity search, Web search, large-scale, association mining

1. INTRODUCTION

The immense scale and wide spread has rendered the Web as an ultimate information repository— as not only the sources where we *find* but also the destinations where we *publish* our information. These dual forces have enriched the Web with all kinds of *data*, much beyond the conventional *page view* of the Web as a corpus of HTML pages, or “documents.” Consequently, the Web is now a rich collection of *data-rich* pages, on the “surface Web” of static URLs (e.g., personal or company homepages) as well as the “deep Web” of database-backed contents (e.g., flights from aa.com).

While data is proliferating, however, we are currently not able to effectively access such data (e.g. finding the customer service number of amazon) using traditional document retrieval techniques. Often times, we have to think hard in formulating the right keywords

¹ *WISDM* Project (Web Indexing and Search for Dynamic Mining): <http://wisdm.cs.uiuc.edu>

for hitting promising documents. Furthermore, we have to sift through the returned documents one by one to identify the *data* we are looking for. The lack of an effective search mechanism over data or entities on the Web motivates us to study entity search in our *WISDM* project at UIUC. We proposed the concept of entity search in [3]. We now briefly describe the data model and the abstract problem of entity search.

Data Model: Entity View.

In the standard page view, the Web is viewed as a set of documents (or pages) $D = \{d_1, d_2, \dots, d_n\}$. In contrast, our data model takes an *entity view*: We consider the Web as primarily a repository of entities: $E = \{E_1, E_2, \dots, E_n\}$, where each E_i is an entity type (e.g. phone, email, etc). Further, each entity type E_i is a set of *entity instances* that are extracted from the corpus, i.e. literal values of entity type E_i that occur in documents in D . We use e_i to denote an entity instance of entity type E_i . Take phone entity as an example, by recognizing phone-number patterns (say, in a regular expression of digits) from D , we may extract $\#phone = \{\text{“800-201-7575”}, \text{“244-2919”}, \text{“(217) 344-9788”}, \dots\}$ where the $\#$ sigh is used to distinguish entity types from keywords.

The Entity Search Problem:

Given: An entity collection $E = \{E_1, E_2, \dots, E_N\}$ over a document collection $D = \{d_1, d_2, \dots, d_n\}$
Input: Query: $\beta - \alpha(E_1, E_2, \dots, E_m, K_1, K_2, \dots, K_l)$
Output: $t = \langle e_1, \dots, e_m, K_1, \dots, K_l \rangle$: sorted by $score(t)$, the *tuple score* of t with respect to β and α .

- As *input*, an entity search query allows users to specify entity types, $E_1, \dots, E_m$, in addition to keywords $K_1, \dots, K_l$. Optionally, in a complete form, a query can also specify a *matching pattern* α to restrict when an occurrence of entity instances and keywords is considered a matching tuple, and a *scoring measure* β , to specify how all the matching tuples are ranked.
- As *output*, each matching result is a *tuple*, $t = \langle e_1, \dots, e_m, K_1, \dots, K_l \rangle$, i.e. an instantiation of entity instances with keywords. Note that while the keywords are always the same for all result tuples, the specific entity instances in the tuple may differ.
- The *objective* is to find, from the space of $E_1 \times \dots \times E_m \times K_1 \times \dots \times K_l$, the matching tuples in ranked order by how well they match the query, i.e. how well entity instances and the specified keywords associate. Tuples are matched by pattern α and ranked by measure β , which produces the tuple score $score(t)$. $\square$

With the problem of entity search introduced, we now discuss the important characteristics of entity search.

Requirements of Entity Search:

Entity search has to consider the *uncertain* nature of entity extraction, the main technique that enables our entity view of the Web.

Entity search has to assess the *contextual* relationship between entities and keywords, *i.e.* how they appear together in documents. Entity search should leverage the redundancy of the Web in being able to aggregate information *holistically* from various sources. Entity search has to rely on testing *association* between entity instances and keywords for effective scoring. Entity search should also take into account the *popularity of documents* as supporting evidence of scoring the matched tuples in them.

While a user is free to specify his own ranking measure β , as system default we take into consideration all the important characteristics of entity search in designing our ranking model. We now briefly describe our conceptual ranking model. The concrete system architecture, which contains the ranking model, in supporting entity search will be presented in Section 2.

Ranking Model of Entity Search:

The ranking model of entity search consists of two primary steps. *Locally*, tuples are instantiated within documents with their local scores examined by considering the uncertainty of extracted entities as well as the contextual relationship between entity instances and keywords in the matching document. Then *globally*, the local scores of an entity tuple should be aggregated across documents. In this step, global statistics, such as the popularity of documents, the frequency of individual keywords and entity instances should be taken into consideration. Tuples are finally ranked according to the aggregation score derived through these two steps.

2. SYSTEM ARCHITECTURE

In this section, we discuss the architecture of an entity search system, whose components could be divided into two categories: offline processing and online processing (*i.e.* entity ranking). The core components are shown in Figure 1.

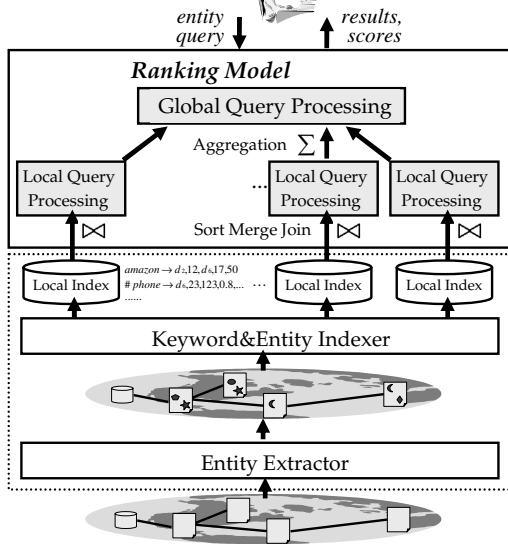


Figure 1: System Architecture

2.1 Offline Processing

Entity extraction and indexing are the two main offline processing modules for entity search, as shown in the dashed box in Figure 1.

Entity Extraction: Entity extraction, which has been extensively studied, aims to extract entity instances from documents. It can be either straightforward, *e.g.* using regular expression for email address extraction, or very sophisticated, *e.g.* using statistical classifiers for street address identification. As entity extraction is inherently imperfect, which is unavoidable for complicated extraction

tasks, our entity search must essentially deal with uncertainty. By this offline extraction, we will recognize, for each entity instance e_i , all its *occurrences* in the corpus. To facilitate query matching, we will record the “features” of each occurrence e_i — These local occurrence features will facilitate our local scoring to quantify the strength of local matchings.

- Position $e_i.pos$: the document id and word offset of this instance occurrence, *e.g.* instance e_1 may occur at $(d_6, 23)$.
- Entity ID $e_i.id$: the unique id that represents the string value of an entity instance, *e.g.* instance e_1 with ID 123.
- Confidence $e_i.conf$: the estimated probability that shows how likely this entity instance occurrence is regarded as an instance of entity type E_i , *e.g.* instance e_1 with extraction confidence 0.80.

An important aspect in entity extraction is entity disambiguation, for instance, figuring out that two phone numbers “213-4545” and “805-213-4545” actually refer to the same phone number. Entity disambiguation is beyond the scope of this study. We believe this is an orthogonal issue to the task of entity search. However, relying on large scale document analysis where much redundancy exists, we can afford not performing sophisticated entity disambiguation. For example, the phone number of “805-213-4545” appears frequently on the Web and could provide enough evidence for ranking.

Indexing: To support entity as a first-class concept in search, we index entities in the same way as indexing keywords.

To index the extracted entity instances, the indexer builds an inverted index of entities, in addition to the traditional keyword inverted index. Given an entity type E_i , the *entity inverted index* will return a list containing all the information regarding the extracted instances of E_i . Specifically, the *entity inverted index* records for each occurrence of entity instance e_i , the position $e_i.pos$ of the extraction in the documents, the entity instance ID $e_i.id$, and the probability of extraction accuracy $e_i.conf$, *e.g.* an occurrence of phone instance “805-213-4545” as $(d_6, 23, 123, 0.8)$. Such information is stored in an ordered list according to the document ID, similar to a keyword inverted index. All the occurrences of keywords and entity types are recorded in their respective inverted indices. Therefore, given a query consisting of keywords and entity types, our system only need to load and process the inverted lists for each keyword and each entity type, respectively.

In addition to entity extraction and indexing, there are other tasks that need to be done offline before online queries are executed. For example, the popularity score of each document should be computed offline. The individual frequency of keywords and entity instances in a corpus should also be computed in an offline fashion.

2.2 Online Processing

In this subsection, we discuss the two main online processing modules corresponding to the local scoring and global scoring of our ranking model, as shown in the solid box in Figure 1.

Local Query Processing:

First, we use a pattern matcher to instantiate tuples by matching pattern $\alpha(K_1, \dots, K_l, E_1, \dots, E_m)$. The matcher will retrieve the inverted lists of all the entity types E_i and keywords K_j , and then perform one pass of “sort-merge join” to match tuples. That is, the matcher will iterate through all the lists in parallel. For every document d in the intersection, the matcher will check the actual positions of every E_i instance e_i and keywords k_j , to determine whether pattern α is satisfied. If it is, a tuple is instantiated and its local score is calculated according to the extraction confidence of its entity instances and the contextual relationship between its

entity instances and keywords (more concretely their positions in the document). Then the matched tuple, with its local score, is sent to the global query processing module for aggregation.

In terms of computation, in spirit, it is the same as what needs to be done in traditional document retrieval. The operation, “sort-merge join”, used in answering almost all the document retrieval queries, is well known to be easy to compute in a very fast manner due to its linear computational nature.

This local query processing module operates on a document basis, and therefore could be fully parallelized by partitioning the whole document collection into sub-collections. This query processing feature enables supporting entity search in large-scale, by utilizing the power of distributed computing.

Global Query Processing:

The global query processing module, upon getting a user’s entity search query, sends the query to distributed local query processing nodes. It then waits for the local processing nodes to produce matched tuples with their local scores. After receiving all the tuples with their local scores, it will perform global aggregation, by considering factors such as the popularity of pages, the frequency of individual keywords and entity instances, etc in the aggregation process. The goal is to derive a final score for each distinctive tuple that truly captures the association between entity instances and keywords in the tuple. Finally the tuples are ordered by their global score and output to the user. For concise presentation purposes, only entity instances of each tuple need to be shown, as the keywords for every tuple are the same.

Standard network protocol could be used for the communication between the global processing unit and the local processing units. It is worth noticing that almost all the expensive computations, the IO intensive ones that access indices, are done locally. The global processing workload is light, given all the information is readily available and simple aggregation could be performed very quickly.

3. DEMONSTRATION

Toward our goal of supporting indexing and search over entities on the Web, we have built our prototype system for entity search. It is implemented in C++ on Red Hat Linux with gcc 3.2.2. More specifically, we built our entity search system by morphing the Lemur Toolkit (version 2.2), an information retrieval engine [1]. We implemented our own local query processing model and global query processing module under this Lemur framework.

To verify our design of the system architecture for supporting entity search on large-scale, we decide to use the Web, the ultimate information source, as our corpus. Our corpus is directly obtained from the Stanford WebBase Project², collected as a result of general crawl in Aug, 2006. The total corpus size (uncompressed) is around 2TB, containing 48974 websites and 93 million pages.

In order to process such large-scale data set, we ran our indexing and query processing modules on a cluster of 34 nodes, each with Celeron 2.80GHz CPU, 1 GB memory and 160 GB disk. We evenly distribute the whole corpus across these nodes, which means each node is in charge of around 60GB data.

On this corpus, we tested two general entity types: phone and email, extracted based on a set of regular expression rules. Around 8.8 million distinctive phone entity instances and around 4.6 million distinctive email entity instances are extracted. We choose to use the UIMA [2] framework as our underlying entity extraction layer, as it allows flexible plugin of various extraction components.

The extracted entities are indexed using the modified indexer of the Lemur Toolkit [1]. More specifically, the entity indexer records,

Query	Telephone	Rank
Citibank Customer Service	800-967-2400	1
New York DMV	800-342-5368	2
Amazon Customer Service	800-201-7575	1
Utah IRS	801-297-2200	1
Query	Email	Rank
Bill Gates	bgates@microsoft.com	4
Oprah Winfrey	oprah@aol.com	2
Elvis Presley	elvis@icomm.com	5
Arnold Schwarzenegger	governor@governor.ca.gov	4

Table 1: Telephone Number and Email Address Queries

for each occurrences of an entity instance, its entity ID and extraction confidence, in addition to its position information.

The results of example queries, *e.g.* finding the customer service number of Amazon, are shown in Table 1, where the first column lists keywords used in the query, the second column lists the correct answer for each query (manually verified) and the third column lists the rank of the correct answer in the result returned by our ranking model. As we can see, our entity search system is highly effective, returning right answers within top 5 places in most cases.

Our entity search system is both space efficient, in terms of the overhead created by indexing entities in addition to keywords, and time efficient, in terms of query processing time. First of all, indexing email and phone entities only incurs less than 0.1% overall space overhead in the indices. The size of the inverted index of email and phone is comparable to the size of a stopword’s inverted index. This overhead is proportional to the number of entity types supported. However, we believe the space overhead will remain insignificant, since useful “data” is only a small proportion of the corpus where syntactic “sugar” abounds.

Second, The highly parallel nature of query processing enable high efficiency. Most queries are returned within seconds. The query response time is inversely proportional to the number of cluster nodes. Given more cluster nodes, we can further divide the corpus into smaller chunks and therefore get higher query efficiency.

In our demonstration, we will show a few example scenarios of using the system. We now use a concrete example to illustrate the interaction with our system. First, users fill in their query in our query interface, by specifying keywords and the entity types they are looking for, plus optional pattern information. For instance, the query could be “amazon customer service #phone” with ordered window pattern. Our system will then perform the ranking algorithm for instantiation and ranking of results. Finally, a ranked list of tuples (in this example, a list of phone numbers) are returned to the user. Users can directly see what he is asking for from the returned tuples (each with its supporting URLs), and moreover has the freedom to check the pages using the supporting URLs of each tuple (the documents where the tuple is identified) for further verification. We plan to generate snippet using the surrounding context of a tuple for each of its supporting URLs to alleviate or avoid the need for checking supporting documents in the future.

4. REFERENCES

- [1] Lemur toolkit for language modeling and information retrieval. <http://www-2.cs.cmu.edu/~lemur>.
- [2] Unstructured information management architecture. <http://www.research.ibm.com/UIMA>.
- [3] T. Cheng and K. C.-C. Chang. Entity search engine: Towards agile best-effort information integration over the web. In *CIDR*, pages 108–113, 2007.

²<http://www-diglib.stanford.edu/~testbed/doc2/WebBase/>