Entity Search Engine: Towards Agile Best-Effort Information Integration over the Web

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

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* information. The 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." The Web has thus become 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), as Figure 1(a) shows. The richness of data, while a promising opportunity, has challenged us to effectively find data we need, from one or multiple sources.

In particular, we are motivated by, when building the MetaQuerier at UIUC, the need of large scale on-the-fly integration for online structured data. The *MetaQuerier* system, as we reported in CIDR 2005 [4], aims at finding and querying data sources on the deep Web. However, as the "last mile" for metaquerying, when users can query multiple sources on the fly [8], or when data is automatically "crawled" from sources [6], how do we identify and integrate the structured data embedded in unstructured result pages? (*e.g.*, the query results of *amazon.com* and *bn.com*).

Further, we realized, as observed earlier, *beyond* our MetaQuerier experience, such data richness is pervasive– from the depth as well as the surface of the Web. While data is proliferating, however, we are currently not able to effectively access such data. To motivate, consider a few scenarios, for user Amy:

Scenario 1: To call Amazon.com for her online purchase, how can Amy find their "phone number"? To begin with, what should be the right keywords for finding pages with such numbers? A query like "amazon customer service phone" may not work; often a phone is simply shown without keyword "phone" (*e.g.*, customer service: 800-717-6688). On the other hand, "amazon customer service" could be too broad. Then, for each querying, she must sift through the returned pages to dig for the phone numbers. This overall process can be unnecessarily time consuming. □

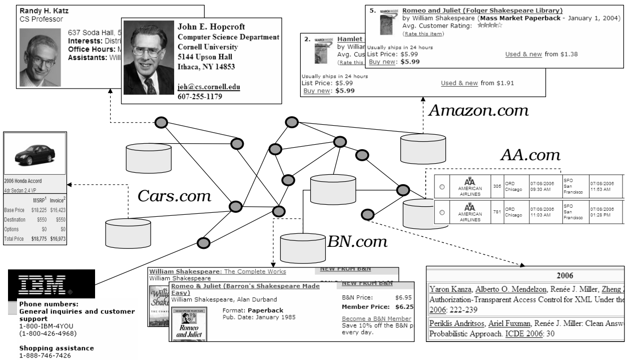


Figure 1: The data-rich Web.

Scenario 2: To apply for graduate schools, how can she find the *list* of "professors" in the database area? She has to go through all the CS department pages or, even worse, check faculty's homepages one by one– a very laborious process. □

Scenario 3: As a graduate student, Amy needs to prepare a seminar presentation for her choice of recent papers. Can she find papers that come readily with presentations, *i.e.*, a "PDF file" *along with* a "PPT file," say from SIGMOD 2006? □

Scenario 4: Now Amy wants to buy a copy of Shakespeare's Hamlet to read; how can she find the "prices" and "cover images" of available choices from, say, Borders.com and BN.com. She would have to look at and compare the results from multiple online bookstores one by one. □

In these scenarios, like every user in many similar situations, Amy is looking for particular *types* of data, which we call *entities*, *e.g.*, a phone number, a book cover image, a PDF, a PPT, a name, a date, an email address, etc. She is *not*, we stress, looking for pages as "relevant documents" to read, but entities as data for her subsequent tasks (to contact, to apply, to present, to buy, *etc.*). We are facing a dilemma:

• On one hand, for accessing over the sheer size of the Web, we are mostly relying on *search engines*, such as Google, Yahoo, or MSN, which search for pages by keywords. On this extreme, while being "IR-style" with a scalable text processing framework, they are not *data aware*.

• On the other hand, integration services, such as Expedia.com or PriceGrabber.com, exist online for specific domains. On this extreme, while providing "DB-style" precise querying, they can hardly scale the amount of data and the number of sources on the Web.

As our solution, which this paper will propose, we believe the two extremes must meet, with a synergistic "marriage" in

This article is published under a Creative Commons License Agreement (http://creativecommons.org/licenses/by/2.5/).
You may copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but you must attribute the work to the author and CIDR 2007.

the middle. From the "search" perspective: Can we build a search engine with *data awareness*? From the "integration" perspective: Can we develop an integration system with *scale awareness*, even with limited "best-effort" semantics? As we will discuss in Section 3, the two perspectives together lead to our objective of *entity search*– As a search system, we propose our *EntitySearch* engine to search *directly* for target information "entities," and holistically across their occurrences in all pages. As an integration system, it will assume only a lightweight off-the-shelf entity extraction layer, and thus scalable to many sources; further, it will construct "derived relations" as the search output, on the fly, and thus flexible for ad-hoc user needs.

In summary, this paper proposes the concept of entity search and its initial implementation as an agile best-effort framework for realizing large scale information integration over the Web. We will motivate and define entity search in Section 2 and Section 3 respectively. Section 4 will then discuss our initial "pilot" implementation for exploring research issues. Section 5 will demonstrate the usefulness of entity search using real world scenarios. We will conclude in Section 6.

2. THE DILEMMA

To access the Web, we must resort to search engines or integration services. The state of the art, however, presents a clear dilemma between data and scale, and we are thus lacking large scale data-based access to the Web.

2.1 Search Engines: Data Awareness?

On one hand, our information access is mostly relying on current search systems, such as popular engines like Google, Yahoo, and MSN Search– However, while reaching extensive parts of the Web, with their inherent "page view" of the Web, they are lacking even minimal data awareness and thus incapable for such tasks as finding "data" entities.

That is, current search systems are not *data aware*. In their rather simplistic *page view*, with a text retrieval system as their backend engine, they naturally view the Web as simply a repository of interlinked documents, or *pages*, each containing some keywords upon which searches can be performed. Our information access amounts to finding relevant pages by keywords– regardless of what data entities we are looking for. In other words, from this "search" perspective, *we are lacking a search system that is aware of data*. Specifically, this lacking leads to two major limitations of current search systems:

• **Indirect Input and Output.** In terms of the *input and output*, current engines are searching *indirectly*. To begin with, to formulate queries as input, users cannot directly describe what they want. Amy has to formulate her needs indirectly as keyword queries, often in a non-trivial and non-intuitive way, with a hope to hit "relevant pages" that may or may not contain target entities. For Scenario 1, will "amazon customer service phone" work? Then, as query output, users cannot directly get what they want. The engine will only take Amy, indirectly, to a list of pages, and she must manually sift through them to find the phone number. Can we help users to *search directly* in both describing and getting what they want? □

• **Singular Matching Mechanism.** In terms of the *matching mechanism*, current search engines are finding each page *singularly*. Our target entities often come from multiple pages. In Scenario 1, the same phone number of Amazon may appear in the company's Web site, online user forums, or even blogs. In this case, we should collect, for each phone, all its occurrences from multiple pages as supporting evidences of matching. In Scenario 2, the list of professors probably cannot be found in any single page. In this case, again, we must look at many pages to come up with the list of promising names (and each name may appear in multiple pages, like the earlier case). Can we help users to *search holistically* for matching entities across the Web corpus as a whole, instead of each individual page? □

Thus, from the "search" perspective, we are facing the challenge of building data awareness into Web search systems. As data proliferates on the Web, the classic view of the Web as a page repository is increasingly inadequate. A step forward, can we search for *data*, or specific entities, as our target directly and across all pages holistically? That is, while keyword-based search is scalable to the Web and easy to use, can it be morphed to be *data aware*?

2.2 Integration Systems: Scale Awareness?

On the other hand, there already emerged many integration systems, such as the various specialized engines in comparison shopping (*e.g.*, PriceGrabber.com, NextTag.com and BizRate.com) and vertical search services (*e.g.*, Realtor.com for real estates, Expedia.com for airfares, and Google Base for various domains)– However, while offering precise data-based access, with their inherent "database view," they are not not designed with "scale" in mind, for the large amount of data-rich pages on the Web and the large diversity of user needs.

That is, current integration systems are not *scale aware*. In their rather rigid *database view*, they naturally assume and can only query data prepared in a structured relational format– in certain prescribed schemas– upon which SQL queries can be performed. Our information access is thus significantly limited by the availability of data in such rigid schemas, as well as the types of queries that the prescribed schema can support. In other words, from this "integration" perspective, *we are lacking an integration system that is aware of scale– the proliferation of Web data and online users*.

In particular, because their database view requires data in well-structured format, an integration system must build *wrapper* for each source, which works as a *relation extractor* for each site, to precisely extract its unstructured text into structured DBMS, upon which to perform SQL queries. We believe that, while achieving structural rigor, this technique is not viable, with two limitations:

• **Limited Sources.** In terms of *sources*, it lacks scalability. As repeatedly reported [7, 5], per-source wrappers are not only laborious to build but also fragile to maintain, and thus a cost barrier[1]. To incorporate myriad potential sources (and pages), we shall not rely on per-source training or construction. □

• **Limited Queries.** In terms of *queries*, it lacks flexibility: A wrapper must make a hard decision on the schemas of the relations to extract–*e.g.*, for *book*, to extract a relation with format, publisher, price, cover_image, or all? The more complex a schema is, the more likely its extraction may fail, while a simpler scheme may be useless for many users. □

Thus, from the "integration" perspective, we are facing the challenge of building scale awareness into Web integration

[1] This is perhaps why today's Web integration systems work by assuming a small set of pre-configured sources, by user submission (*e.g.*, Google Base), or by relying on a centralized database, *e.g.*, Sabre [3] for *airfares* and MLS [2] for real estate

rank	phone number	score	urls
1	800-201-7575	0.9	amazon.com/support.htm myblog.org/shopping
2	800-988-0886	0.8	Dell.com/supportors
3	800-342-5283	0.6	xyz.com
4	206-346-2992	0.2	hp.com
...	...	...	...

rank	PDF	PPT	score	urls
1	sigmod6.pdf	sigmod6.ppt	0.8	db.com,sigmod.com
2	surajit21.pdf	surajit21.ppt	0.7	ms.com
...	...	...	...	...

Figure 2: Query Results of $Q1$ and $Q3$.

systems. While the prevalence of data on the Web presents novel opportunities for integration, this "Web-scale" scenario contrasts traditional settings and defeats current techniques– We must rethink not only new techniques but also realistic *objectives*. To bring myriads heterogeneous sources to meet ad-hoc users, as the scalability and flexibility mandate, can we develop agile integration without rigid schemas, even with "best effort" semantics?

3. OUR PROPOSAL: ENTITY SEARCH

Approaching the current barriers from the dual perspectives, we are inspired that they seem to converge to the same solution: The dual perspectives represent the two extremes of the spectrum: simple keyword *search* to fully transparent information *integration*. Meeting in the middle, they suggest the "marriage" of scale-aware search and data-aware integration. Towards searching directly and holistically, for finding specific types of information, we thus propose to support *entity search*.

• First, as *input*, users formulate queries to directly describe what types of data they are looking for: She can simply specify what her *target entities* are, and what keywords may appear in the *context* with a right answer. To distinguish between entities to look for and keywords in the context, we use a prefix #, *e.g.*, #phone for the phone entity. Our scenarios will naturally lead to the following queries:

$Q1$: (amazon customer service #phone)
$Q2$: (#professor #university #research='database')
$Q3$: ow(sigmod 2006 #pdf_file #ppt_file)
$Q4$: (#title='hamlet' #image #price)

In the queries, there are two components: (1) *Context* pattern, how will the target entities appear? $Q1$ says that #phone will appear with these keywords in the default pattern of "near" (or proximity). We may also explicitly specify the pattern, *e.g.*, $Q3$ uses ow to mean order-window (the keywords must appear before #pdf_file and then #ppt_file. (2) *Content* restriction: A target entity will match any instances of that entity type, subject to optional restriction on their content values– *e.g.*, Q_1 will match every phone instance, while Q_2 will only match research area "database." (In addition to equality "=", other restriction operators are possible, such as "contain.")

• Second, as *output*, users will get directly the entities that they are looking for. That is, as a query specifies what entity types are the targets, its results are those entity *instances* (or literal values) that match the query, in a ranked order by their matching scores. (We will discuss this matching next.) Figure 2 shows some example results for $Q1$ and $Q3$.

• Third, as *search mechanism*, entity search will find matching entities holistically, where an instance will be found and matched in all the pages where it occurs; *e.g.*, a #phone 800-201-7575 may occur at multiple URLs as Figure 2 shows. For each instance, all its matching occurrences will be aggregated to form the final ranking; *e.g.*, a phone number that more *frequently* occurs at where "amazon customer service" is mentioned may rank higher than others. Thus, entity search will not only find *entities* as the primary result, but also return *pages* where each entity is found as the secondary result and supporting "evidences." □

To support entity-based querying, the system must be fundamentally *entity aware*: As a departure from current search engines built around the notions of pages and keywords, we must generalize them to support entity as a first-class concept. With this awareness, as our data model, we will move from the current page view, *i.e.*, the Web as a document collection, to the new *entity view*, *i.e.*, the Web as an entity repository. Upon this foundation, as our query mechanism, we can then develop entity search, where users specify what they are looking for with keywords and target entities, as $Q1$ – $Q4$ illustrated. We now formalize these notions.

Data Model: Entity View. How should we view the Web as our database to search over? In the standard page view, the Web is a set of documents (or pages) $D = \{d_1, d_2, \ldots, d_n\}$.

Our data model takes an *entity view*: We consider the Web as primarily a repository of entities: $E = \{E_1, E_2, \ldots, E_n\}$, where each E_i is an entity type. For instance, to support Scenario 1 (Section 1), the system might be constructed with entities $E = \{E_1 : \text{#phone}, E_2 : \text{#email}\}$. 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 occurs somewhere in some $d \in D$. We use e_i to denote an entity instance of entity type E_i. In our example, *e.g.*, by recognizing phone-number patterns (say, in a regular expression of digits) from D, we may extract #phone = {"800-201-7575", "244-2919", "(217) 344-9788", ...}

As the starting point for data-aware search, this "tagging", or *entity extraction*, is to recognize each entity from Web pages. Entity extraction has been well studied in the context of general information extraction, and techniques abound, from simple syntactic pattern matching to sophisticated statistical taggers, with many off-the-shelf tools available. We stress that, however, while implementations may differ, such extractors are inherently imperfect, and entity search must essentially deal with *uncertainty*. Used as a black box, in abstraction, an entity extractor will return, for each occurrence, the extraction confidence probability and the position of extraction. We thus transform the page view into our entity view, $E = \{E_1, E_2, \ldots, E_n\}$.

The Entity Search Problem:

Given: An entity collection $E = \{E_1, E_2, \ldots, E_N\}$
Input: Query: $\beta\text{-}\alpha(K_1, K_2, \ldots, K_l, E_1, E_2, \ldots, E_m)$
Output: $t = \langle e_1, e_2, ..., e_m \rangle$: sorted by $score(t)$,
the *tuple score* of t with respect to β and α.

• As *input*, as Section 3 discussed, an entity-search query is similar to standard keyword queries, but now users can specify entity types, E_1, E_2, ..., and E_m, in addition to keywords K_1, ..., K_l. Optionally, in a complete form, a query can also specify a *matching pattern* α (*e.g.*, ow in $Q3$), to restrict when an occurrence of an instance $t = \langle e_1, e_2, ..., e_m \rangle$ is considered a matching tuple, and a *scoring measure* β, to specify how all the matching instances are ranked. Depending on the implementation and application settings, the choices of the β pattern and the α can be system built-in or user specified, and they together determine the ranking scores.

- As *output*, for a query with m target entities, each matching result is a m-ary *entity tuple*, *i.e.*, $t = \langle e_1, e_2, ..., e_m \rangle$, *i.e.*, a combined instance. Note that an entity tuple may contain one or multiple instances, each of which is associated with one entity type. For example, ⟨David, david@hotmail.com, 315-673-9091⟩ is an entity tuple of type ⟨#name, #email, #phone⟩, and ⟨Canada, Ottawa⟩ of type ⟨#country, #capital_city⟩.
- The *objective* is to find, from the space of $E_1 \times \ldots \times E_m$, the matching tuples in ranked order by how well they match the query, *i.e.*, how well entity tuple t and the specified keywords associate, as matched by pattern α and ranked by measure β, which result in the tuple score $score(t)$. □

To summarize, and to put our proposal in perspectives, we examine it form both the search and integration point of views:

First, as a *search* framework, entity search is data aware. With the probabilistic tagging of entities, we view the Web as a repository of entities. Users directly specify their target entities, and the system holistically evaluates the matchings.

Second, as an *integration* framework, entity search is scale aware, towards agile requirement and best-effort semantics for large scale deployment: On one hand, as deployment requirement, it is agile, requiring only minimal tagging of the data domain of interest– It can be uniformly applied to many sources, without requiring building wrappers. We stress that each entity is *independently* extracted in a "soft" probabilistic sense, and thus requires no rigid full relation extraction that often mandates per-source wrappers. It can adapt to diverse information needs, without fixed pre-scribed schemas. The desired entities are only associated by ad-hoc queries at query time– Thus, users may ask #phone with "ibm thinkpad" or "bill gates", or they ask to pair #phone with, say, #email for "white house". Supporting such online matching and association is exactly the challenge (and usefulness) of entity search.

On the other hand, as query semantics, it is only best effort, returning the result in ranked, best-first manner. In essence, the system assumes "integration in a probabilistic sense": Given independent entities with probabilities, entity search finds matching instances that "most likely" form a desired tuple.

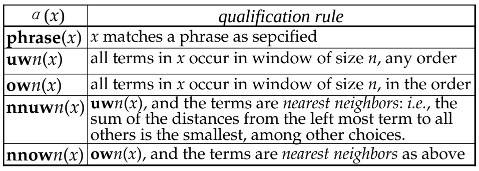
To understand the promise of the system and the practical issues, we decided to start with a quick "Version 0.1" pilot implementation, which is the focus of our demonstration.

4. THE PILOT SYSTEM

To realize our proposal in Section 3, we implemented a "pilot" system for empirical study.

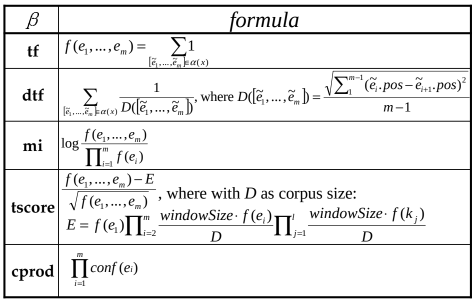
First, as we abstracted in the Entity Search problem, we need to come up with specific *matching patterns* α as well as *scoring measure* β. Figures 3 and 4 summarizes our currently implemented α and β. To specify a tuple function operator, the user or the application will choose the exact measures to use.

The α Measure: An α measure *qualifies* if match of occurrences of entities match the desired tuple, in the matching pattern $\alpha(x)$. In principle, any Web presentation features can be incorporated for pattern expression, *e.g.*, linguistic, proximity, or visual features. In this work, we treat webpages as linear documents. Consequently, we implemented several simple position-based, page-bounded α measures as Figure 3 summarizes.



$\alpha(x)$	*qualification rule*
phrase(x)	x matches a phrase as sepcified
uw$n(x)$	all terms in x occur in window of size n, any order
ow$n(x)$	all terms in x occur in window of size n, in the order
nnuw$n(x)$	**uw**$n(x)$, and the terms are *nearest neighbors*: *i.e.*, the sum of the distances from the left most term to all others is the smallest, among other choices.
nnow$n(x)$	**ow**$n(x)$, and the terms are *nearest neighbors* as above

Figure 3: Pattern measures.



β	*formula*
tf	$f(e_1, \ldots, e_m) = \sum_{[\tilde{e}_1, \ldots, \tilde{e}_m] \in \alpha(x)} 1$
dtf	$\sum_{[\tilde{e}_1, \ldots, \tilde{e}_m] \in \alpha(x)} \frac{1}{D([\tilde{e}_1, \ldots, \tilde{e}_m])}$, where $D([\tilde{e}_1, \ldots, \tilde{e}_m]) = \frac{\sqrt{\sum_1^{m-1} (\tilde{e}_i.pos - \tilde{e}_{i+1}.pos)^2}}{m-1}$
mi	$\log \frac{f(e_1, \ldots, e_m)}{\prod_{i=1}^{m} f(e_i)}$
tscore	$\frac{f(e_1, \ldots, e_m) - E}{\sqrt{f(e_1, \ldots, e_m)}}$, where with D as corpus size: $E = f(e_1) \prod_{i=2}^{m} \frac{windowSize \cdot f(e_i)}{D} \prod_{j=1}^{l} \frac{windowSize \cdot f(k_j)}{D}$
cprod	$\prod_{i=1}^{m} conf(e_i)$

Figure 4: Scoring measures.

The β Measures: Once instances of entities are matched, they form tuples. A β measure *quantifies* how promising a tuple $\langle e_1, \ldots, e_m \rangle$ is. A tuple $\langle e_1, \ldots, e_m \rangle$ may appear in the corpus many times– let $[\tilde{e}_1, \ldots, \tilde{e}_m]$ be one such occurrence, *i.e.*, $[\tilde{e}_1, \ldots, \tilde{e}_m] \in \alpha(x)$. We note that a scoring function β for determining the tuple score, *i.e.*, $\beta(e_1, \ldots, e_m)$, can build upon the following quantities:

1. *Frequency* of the tuple: How many times has $\langle e_1, \ldots, e_m \rangle$ occurred? That is, how many $[\tilde{e}_1, \ldots, \tilde{e}_m]$ are matched?
2. *Strength* of each occurrence: How well does each $[\tilde{e}_1, \ldots, \tilde{e}_m]$ match the pattern?
3. *Frequency* of individual entity instance: How many times has e_i appeared in the corpus? A tuple may be frequent simply because its entity instances are common.
4. *Uncertainty* of entity instance: What is the probability of instance e_i being of the entity type E_i.

Our initial implementation thus tried, for experimentation, to use all four in various ways, as Figure 4 summarizes a few representative ones: *tf*: tuple frequency (using 1); *dtf*: distance weighted tuple frequency (using 1, 3); *mi*: mutual information (using 1, 2); *t-score*(using 1, 3); *conf* (using 4).

While these measures are validated in our experiments to be useful, we believe a more systematic study of the scoring measure is still needed. Coming up with more effective scoring measures is itself an interesting research problem. We plan to work towards this direction in the future.

Next, we describe the key system components to accomplish the goal of entity search, towards agile best-effort information integration. The overall system architecture is illustrated in Figure 5. We now zoom into each specific system component.

Data Collector: Our data webpages could come from both the surface Web and the deep Web. To get webpages from the surface Web, our Data Collector works like a crawler, getting webpages related with a specific topic/application. Our current implementation obtains webpages from the Stanford WebBase Project[2]. To get webpages from the deep Web sources,

[2] http://www-diglib.stanford.edu/~testbed/doc2/WebBase/

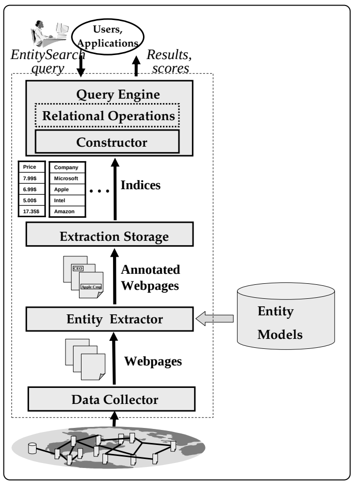


Figure 5: System Architecture

our Data Collector works by collecting result pages of specific queries on deep Web sources , *e.g.*, houses available in a certain zipcode area, books written by some specific authors, etc.

Entity Extractor: In order to discover semantics of the domain, our Entity Extractor extracts domain entities, the key components of the domain, independently. Each entities is extracted using an entity model, which describes how to identify instances of an entity in the webpages.

Query Engine This component essentially supports online querying by performing *pattern matching* and *tuple scoring*, as we abstracted in Section 3 and described in the beginning of this Section. As we produce tuples in the results, naturally many relational operations could be performed, which corresponds to the *Relational Operations* sub component in Figure 5.

Our Query Engine is morphed using the Lemur Toolkit (version 2.2), an information retrieval engine [1]. This IR engine facilitates easy storage of the extracted entities in the form of ordered lists based on document ID, much like storing inverted indices for keywords. It also enables the construction in a natural way as sort-merge-join based on document ID is one of the most common operations in answering IR queries.

5. DEMONSTRATION

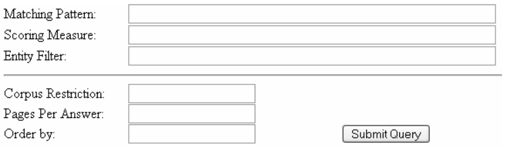
In this demonstration, we specifically show the online component of our system, which is our query engine. The current sever is running on a Pentium-4 2.6GHz PC with 1GB memory. This section will first discuss the interface of our query engine. Then two demonstration scenarios are presented to show the effectiveness of our system. Finally, the demonstration plan is described.

5.1 System Interface

We show our query interface for the query engine of *Entity-Search* in Figure 6.

The first three input boxes in Figure 6 directly refers to the operators explained in Section 4.

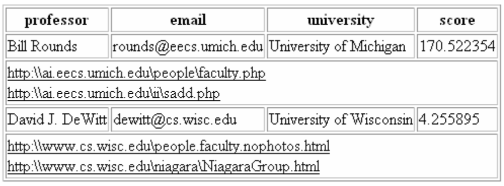
Matching Pattern specifies a pattern $\alpha(x)$, in which x is a list of terms to be joined, as either entities E_i or literal keywords K_j, and α is a *pattern measure*, shown in Figure 3, specifying how the terms are connected into a pattern.



Matching Pattern:
Scoring Measure:
Entity Filter:

Corpus Restriction:
Pages Per Answer:
Order by:

Figure 6: System Interface.



professor	email	university	score
Bill Rounds	rounds@eecs.umich.edu	University of Michigan	170.522354
http:\\ai.eecs.umich.edu\people\faculty.php http:\\ai.eecs.umich.edu\ii\sadd.php			
David J. DeWitt	dewitt@cs.wisc.edu	University of Wisconsin	4.255895
http:\\www.cs.wisc.edu\people.faculty.nophotos.html http:\\www.cs.wisc.edu\niagara\NiagaraGroup.html			

Figure 7: Sample Output of Query C1.

Scoring Measure specifies which β scoring measure, among the ones we support in Figure 4, will be used to score the matched tuples.

Entity Filter enables specifying conditions on each entity. In principle, any filter operation could be applied to an entity. Our system currently supports two operators: equalto (string value of the instance matches exactly with the specified keywords) and contains (string value of the instance contains the specified keywords).

In addition, We provide three auxiliary features to improve the effectiveness of the engine. These features could be regarded as a light-weight Relational Operation layer, which performs minimal but useful relational operations upon the output relation. We discuss these three features one by one.

Corpus Restriction: The Corpus Restriction operator is designed to facilitate application's control over the corpus. We support Corpus Restriction by URL domains as regular patterns. For instance, we could set this field to *.amzon.com, which matches the sub-corpus of pages at the domains with that pattern (*e.g.*, book.amzon.com).

Pages Per Answer: This feature requests the number of Web pages to return as support pages or "evidences" for each tuple returned by the query engine because results are integrated from multiple Web pages across the corpus.

Order By specifies the order of listing results (*e.g.*, by #research alphabetically)– much like the same clause in SQL. By default it will rank the results according to their ranking score calculated by the β Measure.

Application issues queries by filling in the interface (Matching Pattern, Scoring Measure and Entity Filter), as well as the other three auxiliary operators and then clicking the "Submit Query" button. Figure 7 shows a sample system output. As you can see from the figure, it is in the form of a relation. Each tuple has a score associated with it. The URL below each tuple shows the supporting pages where this tuple is found.

5.2 Demonstration Scenarios

In this subsection, we use two concrete scenarios on real data to demonstrate the practical usage of our system. The two scenarios we selected are very different in nature. The first scenario, regarding education domain regarding midwest CS departments, focuses on the surface Web whose pages are mostly unstructured. In contrast, the second scenario, using the book domain, focuses on the deep Web. Result pages collected from the deep Web tend to be more structured. We show that our system works well in both scenarios.

Scenario 1: Midwest Computer Science Domain

Currently, we have collected pages regarding CS department in six midwest universities (IIT, Illinois, Indiana, Michigan, Purdue and Wisconsin) from the WebBase project. The Domain Extractor extracts the following attributes: professor, research, university, email and phone.

Three example queries:
◇ C1: emails of professors across universities
◇ C2: research areas of professors across universities
◇ C3: professors conducting DB related research across universities

Above we have shown three example queries that could be asked in this domain. Detailed query operators for each query is excluded. In Query C1, the application is interested in integrating all the emails of professors across universities into one table. Figure 7 shows a snippet of the result, which is good. We manually checked the result returned by our system for Query C1, over 90% of professor's emails are included in the output relation. And if for each unique professor, we only remain the tuple with the highest score in the relation, we can achieve precision over 85%. Please refer to the results in our online demo for more details. Similar results are observed from other example queries, such as Query C2 and C3.

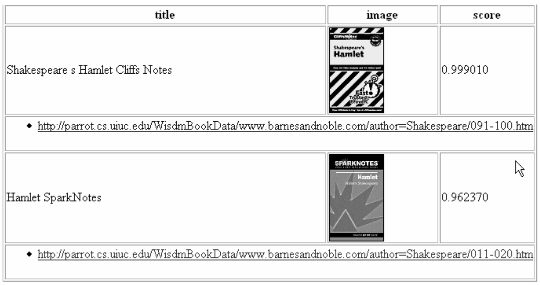
Scenario 2: Shakespeare's Book Domain

This scenario focuses on the deep Web. We issued a query by filling author attribute with "Shakespeare" to the three major online book stores: www.amazon.com, www.barnesandnoble.com, www.buy.com. We then manually collect webpages containing the top 100 results from each site. The Domain Extractor extracts the following attributes: title, author, image, price, date.

Three example queries:
◇ B1: title and price of books
◇ B2: images of books with title containing "Hamlet"
◇ B3: title of books that are in stock

Above we have shown three example queries that could be asked in this domain, excluding detailed query operators. Query B1 is a very common integration scenario where the application wants to find all the books and their prices across multiple websites. More interestingly, Query B2 asks for images of titles with keyword "Hamlet" in it. As you can see from the top results shown in Figure 8, the images indeed are all cover images for books with keyword "Hamlet" in their titles. Query B3 could be issued using the title attribute together with keywords "in stock" as application finds vendors normally put keywords "in stock" to indicate the books currently available for purchase. Similarly, we can query for books that are "out of stock", "on order", etc.

Due to the lack of space, we are only able to briefly show very limited example queries and results in our two scenarios. To experience more about the above two scenarios and have a real feeling of the results, we invite users to the following online demo site[3].



title	image	score
Shakespeare s Hamlet Cliffs Notes		0.999010
• http://parrot.cs.uiuc.edu/WisdmBookData/www.barnesandnoble.com/author=Shakespeare/091-100.htm		
Hamlet SparkNotes		0.962370
• http://parrot.cs.uiuc.edu/WisdmBookData/www.barnesandnoble.com/author=Shakespeare/011-020.htm		

Figure 8: Sample Output of Query B2.

5.3 Demo Plan

Users can follow the link from our demo site to experience our two demo scenarios. For each scenario, we have listed a set of example queries, including all the example queries shown in Section 5.2. By clicking on an example query, the query operators of the chosen query will be automatically filled into the input boxes in our query interface. Alternatively, users are welcome to modify the query operators of the example queries or come up with their own queries. Query results are normally returned within seconds.

6. CONCLUSION

In this paper, we proposed the concept of Entity Search for supporting agile best-effort information integration over the Web. While we observe entity search, as a mid-way marriage of search and integration, is meaningful and useful to access the data-rich Web, there are several open research issues towards its full realization: First, as the core, how to rank each tuple, so that the best matching surfaces to the top? Second, to provide efficient online search, can we generalize the current page-based search engine into an entity search engine? Finally, as entity search returns a ranked list of tuples (*e.g.*, Figure 2), how can we integrate the "derived relations" with relational SQL-based querying– *e.g.*, joining ⟨#company, #phone⟩ with ⟨#company, #email⟩? Or, filtering only those companies with ".com"? We are continuing to investigate these challenging and interesting problems as our future research agenda.

7. REFERENCES

[1] Lemur toolkit for language modeling and information retrieval. `http://www-2.cs.cmu.edu/~lemur`.
[2] Multiple listing service. http://en.wikipedia.org/wiki/Multiple_Listing_Service.
[3] Sabre holdings corporation. http://www.sabre.com.
[4] K. C.-C. Chang, B. He, and Z. Zhang. Toward large scale integration: Building a metaquerier over databases on the web. In *Proceedings CIDR 2005*.
[5] W. W. Cohen. Some practical observations on integration of web information. In *WebDB (Informal Proceedings)*, pages 55–60, 1999.
[6] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *VLDB*, pages 129–138, 2001.
[7] L. J. Seligman, A. Rosenthal, P. E. Lehner, and A. Smith. Data integration: Where does the time go? *IEEE Data Eng. Bull.*, 25(3):3–10, 2002.
[8] Z. Zhang, B. He, and K. C.-C. Chang. Light-weight domain-based form assistant: Querying web databases on the fly. In *Proceedings of VLDB 2005*.

[3] http://parrot.cs.uiuc.edu/entitysearch