Q2P: Discovering Query Templates via Autocompletion

WENSHENG WU, University of Southern California, USA
WEIYI MENG, State University of New York at Binghamton, USA
WEIFENG SU, BNU-HKBU United International College, China
GUANGYOU ZHOU, Central China Normal University, China
YAO-YI CHIANG, University of Southern California, USA

We present Q2P, a system that discovers query templates from search engines via their query autocompletion services. Q2P is distinct from the existing works in that it does not rely on query logs of search engines that are typically not readily available. Q2P is also unique in that it uses a trie to economically store queries sampled from a search engine and employs a beam-search strategy that focuses the expansion of the trie on its most promising nodes. Furthermore, Q2P leverages the trie-based storage of query sample to discover query templates using only two passes over the trie. Q2P is a key part of our ongoing project Deep2Q on a template-driven data integration on the Deep Web, where the templates learned by Q2P are used to guide the integration process in Deep2Q. Experimental results on four major search engines indicate that (1) Q2P sends only a moderate number of queries (ranging from 597 to 1135) to the engines, while obtaining a significant number of completions per query (ranging from 4.2 to 8.5 on the average); (2) a significant number of templates (ranging from 8 to 32 when the minimum support for frequent templates is set to 1%) may be discovered from the samples.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Design, Languages, Algorithms, Experimentation

Additional Key Words and Phrases: Query templates; search engines; autocompletion; trie; pattern discovery

ACM Reference Format:
Wensheng Wu, Weiyi Meng, Weifeng Su, Guangyou Zhou, and Yao-Yi Chiang, 2015. Q2P: Discovering Query Templates via Autocompletion *ACM Trans. Web* V, N, Article A (January YYYY), 29 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

We consider the problem of discovering templates for the queries that users pose on search engines. Here we use search engines to refer to any databases or data sources on the Web that accept keyword queries, e.g., Google and Amazon. Query templates are parameterized queries that capture the common patterns of user queries. For example, jobs in [location] is a template with a parameter location. The template represents a set

Authors' addresses: Wensheng Wu, Computer Science Department, University of Southern California, Los Angeles, CA 90089, USA (wenshenw@usc.edu). Weiyi Meng, Computer Science Department, State University of New York at Binghamton, Binghamton, NY 13902, USA (meng@cs.binghamton.edu). Weifeng Su, Computer Science and Technology Program, BNU-HKBU United International College, ZhuHai, 519085, China (wfsu@uic.edu.hk). Guangyou Zhou, School of Computer Science, Central China Normal University, 152 Luoyu Road, WuHan, 430079, China (gyzhou@mail.ccnu.edu.cn). Yao-Yi Chiang, Spatial Sciences Institute, University of Southern California, Los Angeles, CA 90089, USA (yaoyic@dornsife.usc.edu).

DOI:http://dx.doi.org/10.1145/0000000.0000000

of user queries that start with “jobs in” followed by a desired location for the jobs. Example instances of this template are “jobs in Chicago” and “jobs in New York”.

Knowing query patterns and templates not only helps search engine understand user query interests and improve its search algorithms to better answer common queries, but also becomes increasingly critical to many middleware applications. These applications often require the knowledge of queries and patterns of queries that users frequently pose on the underlying search engines, in order to better serve the users of the applications.

An important class of such middleware applications are Web data integration systems [Wu 2013b]. A data integration system provides uniform access to a set of data sources in a domain of interest, thereby making the access to individual data sources transparent to the users. In this article, we focus on the integration systems that accept keyword queries. There are now many such systems available on the Web. For examples, AddAll[1] allows users to search and compare books in over 40 bookstores by posing queries on a global query interface; NCBI (National Center for Biotechnology Information)[2] permits a unified keyword search over about 50 databases on DNA sequences, biomedical literature, and epigenomics; and Dogpile[3] is a metasearch engine that simultaneously finds search results from multiple search engines, including Google, Yahoo!, and Bing.

As we will further discuss in Section 4, a traditional data integration system often suffers from slow query response since the system needs to determine on the fly which sources can answer the query, and fetch and merge the results from different sources. The problem is exacerbated when there are a large number of data sources to be integrated. To address this challenge, recently we proposed a novel template-driven data integration system Deep2Q [Wu 2013b] that uses templates to represent common query interests and plans ahead offline on how to answer the queries in the templates. For popular queries, the system may also cache their results for fast query response.

Thus, a key problem in building Deep2Q is to discover the templates that capture the queries commonly asked by users. An attractive solution to this problem is to learn such templates from the underlying data sources or search engines. Towards this goal, there have been some works on discovering query patterns from query logs of search engines, e.g., [Pandey and Punera 2012; Agarwal et al. 2010].

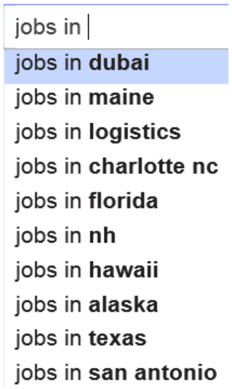
These works are typically limited to analyzing query logs of major search engines such as Google, Yahoo!, and Bing. They are either conducted by search engines themselves or through the collaboration with the search engines. However, in general, search engines, in particular those for e-commerce web sites, e.g., Amazon and Barnes & Noble, tend to treat their query logs as valuable business assets that contain sensitive user information, and thus are not willing to disclose them to the public.

In this article, we consider the problem of discovering query templates without relying on query logs of search engines. Instead, we leverage the query autocompletion features, which have become prevalent among search engines, to discover the patterns of the queries posed by the users. Specifically, query autocompletion is a service provided by a search engine that automatically suggests completions to the (partial) queries while users are typing. Search engines typically recommend the completions based on their popularity, i.e., how many times the completed queries have been entered by other users before. Query autocompletion has been very effective in saving user efforts in typing queries and reducing mistakes in query entries.

[1] http://www.addall.com/

[2] http://www.ncbi.nlm.nih.gov/

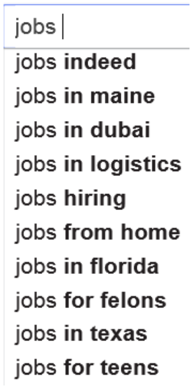
[3] http://www.dogpile.com/



jobs in |

jobs in **dubai**
jobs in **maine**
jobs in **logistics**
jobs in **charlotte nc**
jobs in **florida**
jobs in **nh**
jobs in **hawaii**
jobs in **alaska**
jobs in **texas**
jobs in **san antonio**

Fig. 1. Yahoo! completions for “jobs in”



jobs |

jobs **indeed**
jobs **in maine**
jobs **in dubai**
jobs **in logistics**
jobs **hiring**
jobs **from home**
jobs **in florida**
jobs **for felons**
jobs **in texas**
jobs **for teens**

Fig. 2. Yahoo! completions for “jobs”

To further illustrate query autocompletion, Figure 1 shows the query autocompletion service from Yahoo!. It displays a list of possible completions such as “jobs in Dubai” for the partial query “jobs in”. Each completion typically corresponds to a query that (other) users have entered before.

As the example shows, the queries suggested by search engines are good resources for learning query templates. For example, if the template (e.g., jobs in [location]) starts with an entity name (e.g., “jobs”) followed by a preposition (“in”), then the completions given by search engines will very likely contain the instances of the parameter after the preposition (e.g., locations such as “Dubai” and “Maine”). Furthermore, if there are a large number of queries that follow the template, then it is very likely that the preposition (e.g., “in”) will be among the top completions given by search engines for the partial query that contains only the entity name (e.g., “jobs”). For example, five of the 10 completions given by Yahoo! for “jobs” (Figure 2) start with “in”.

There are two *key* challenges to be addressed when leveraging the above observations to discover query templates and instances.

- How to obtain a good sample of queries that can reveal query patterns, while not overwhelming search engines with sampling queries?
- How to efficiently discover query patterns from the sample that may contain a *large* number of queries?

In this article, we present Q2P, a system that discovers query templates of search engines via query autocompletion that addresses the above challenges. To address the *first* challenge, Q2P incorporates a novel query sampler that incrementally grows the length of sampled queries and uses a *beam-search* strategy to focus the growth and reduce the cost of sampling. The sampler utilizes a *trie* (i.e., prefix tree) to store sampled queries efficiently and help identify queries for further expansion.

To address the *second* challenge, Q2P implements a novel algorithm that leverages the structure of query trie to efficiently discover query patterns. The algorithm only requires *two* passes over the trie. In the first pass, it discovers frequent nodes in the trie. These nodes are then used in the second pass to induce query patterns.

The problem of discovering search engine queries via their autocompletion services has started to receive active attentions in the past few years. The work closest to us is [Bar-Yossef and Gurevich 2008]. However, the goal of [Bar-Yossef and Gurevich 2008] is to obtain a *random* sample of queries from a search engine, which can then be posed to the search engine and obtain a sample of its content. In contrast, we aim to discover a *focused* set of queries that center around an entity, so that we can learn query templates about the entity.

To the best of our knowledge, this is the *first* work on a focused sampling of search engine queries using query autocompletion. Our solution is also unique in its use of trie-based representations of queries for pattern discovery.

The rest of the article is organized as follows. Section 2 defines the template discovery problem. Section 3 presents the architecture of Q2P and describes in detail its beam-search query sampler and trie-based template miner. Section 4 describes the template-driven data integration system Deep2Q and discusses how to integrate Q2P into Deep2Q. Section 5 presents our experimental results. Section 6 discusses related work. The article is concluded in Section 7.

2. PROBLEM DEFINITION

In this section, we first define the template discovery problem and then give an overview of the Q2P's solution. We will describe the internals of Q2P in Section 3. We will focus on discovering noun phrase query templates as defined below.

Definition 2.1 (*Noun phrase query templates*). A noun phrase query consists of three parts: <M, N, O>, where N is a head noun (e.g., "books") that represents an entity of interest; M and O are a set of pre-modifiers (e.g., "new") and post-modifiers (e.g., "for teenagers") of N respectively. A query template or pattern is then a *parameterized* noun phrase query whose parameters represent attributes of entities in a domain of interest. We sometimes call the words or phrases that "introduce" the parameters in the template "cue words" or "cue phrases".

For example, books about [subject] is a noun phrase template with a parameter subject, and a noun phrase query "books about history" is an instance of this template. Furthermore, "about" is a prepositional cue word that introduces the value of the parameter, i.e., "history".

We use RESTful web service of search engine to automatically obtain query completions (see also Section 5). Through the web service, a partial query may be specified as the value for a designated parameter in the request URL for the service and search engine will return more complete queries that expand the partial query, e.g., by adding more words at the *end* of the partial query. For example, a possible completion to "books in" may be "books in spanish".

Note that through its query interface, search engine may provide more complex completions than those available through the web service. For example, it may detect that user has moved the cursor to the beginning or middle of partial query and suggest completions that expand the partial query at the cursor location. For example, suppose that user first enters "books" and then adds a letter 'n' before "books", search engine may suggest completions such as "new books". However, it is difficult to obtain such completions through web services.

In this article, we will focus on discovering *single-parameter* noun phrase templates that do not have pre-modifiers, i.e., template that starts with a head noun (entity name) followed by a list of post-modifiers, one of which represents the parameter in the template. Examples of such templates are: jobs in [location] and books for kids under [age]. We pose partial queries that start with the head nouns, e.g., "jobs" and "books" through web service of search engine, obtain query completions, and discover patterns and templates based on the completions.

Note that it is possible that a search engine may sometimes return completions with additional words that do not appear in the original partial queries. For example, a search engine may complete "jobs in" into "IT jobs in Chicago". While such completions may also be useful for discovering templates, they are discarded in the experiments

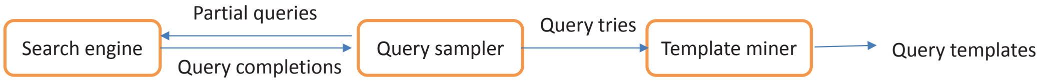


Fig. 3. The Q2P architecture

reported in this article, since our goal is to find sample queries that all start with the same head noun, e.g., “jobs”, as discussed above.

Since our goal is to discover query templates that can capture common patterns of user queries, we introduce a concept of *frequent template*, similar to frequent itemset in association rule mining [Agrawal and Srikant 1995], to quantify the usefulness of templates.

Definition 2.2 (*Template support and frequent templates*). Given a set of queries T and a template p, the *support count* (or support) of p with respect to T is the number (or fraction) of queries in T that follow the template p. A query q follows a template p if q can be obtained by instantiating the parameter(s) in p. A template p is *frequent* if its support (or support count) $\geq \sigma$, a threshold for the minimum support (or support count).

Based on the above definitions, we now define our problem as follows.

Template discovery problem: Given a set of queries T from a search engine and a threshold σ for the *minimum* support count, discover from T a set of frequent noun phrase templates such that the instances of each template start with the same head noun.

3. THE Q2P SOLUTION

Figure 3 shows the architecture of Q2P that solves the template discovery problem as defined in Section 2. Q2P consists of three major components: search engine, query sampler, and template miner. As described in Section 2, we assume that search engine provides a RESTful web service for query autocompletion, which is now commonly available among search engines. Using the service, the *query sampler* repeatedly sends partial queries to the engine and obtains completions suggested by the engine. As motivated in Section 1, the key challenge to query sampler is how to obtain a good sample of queries that may reveal user query patterns without sending too many queries to the engine. In Q2P, this is accomplished by using a trie to economically store the sampled queries and employing a beam-search strategy to limit the amount of expansion on the queries in the trie.

After the query sampler obtains a sample of queries, the queries are then given to the *template miner*, stored in a trie. The template miner takes the advantage of the trie-based representation of queries and efficiently discovers query templates and instances via only two traversals over the trie.

In the rest of this section, we describe in great detail the beam-search query sampler (Section 3.1) and the trie-based template miner (Section 3.2).

3.1. The beam-search query sampler

Algorithm 1 shows the algorithm BEAMSAMPLER which implements the beam-search query sampler. As described earlier, BEAMSAMPLER uses a *trie* to economically store the sampled queries and direct the sampler to perform a focused search controlled by a *beam*. It takes as the input: (1) a search engine S that provides query autocompletion service; (2) $seed$, a noun or noun phrase in the plural form that represents the type of entities of interest, e.g., to discover templates of queries searching for books, we may set $seed$ = “books”; (3) k, the desired beam size; (4) τ, a threshold that determines if

ALGORITHM 1: BEAMSAMPLER(S, $seed$, k, τ, γ): The beam-search query sampler

```
Input: $S$, a search engine with query autocompletion;
       $seed$, noun (phrase) representing an entity type;
       $k$, beam size;
       $\tau$, probing threshold;
       $\gamma$, yield-ratio threshold
Output: $T$, a trie of queries sampled from $S$ using $seed$
1. /* Initialization */
  (1) Create a stub trie $T$ with $seed$ as the root;
  (2) Current level $l \leftarrow 1$;
  (3) Current beam $B \leftarrow \{\text{root}\}$;
repeat
    2. /* Expansion */
    foreach node $n$ in $B$ do
        (1) Ask $S$ to complete $n$'s prefix query, denoted as PrefixQuery($n$);
        (2) if # of completions $< \tau$ then
                Add completions to $T$;
                Mark $n$ as unexpandable;
        (3) else
                (a) if there are completions starting with a letter then
                        Ask $S$ to complete "PrefixQuery($n$) [a-z]";
                (b) if there are completions starting with a digit then
                        Ask $S$ to complete "PrefixQuery($n$) [0-9]";
                (c) if there are completions starting with a special character $c$ then
                        Ask $S$ to complete "PrefixQuery($n$) $c$";
                (d) Add completions to $T$;
                (e) if yield ratio of $n < \gamma$ then
                        Mark $n$ as unexpandable;
    end
    3. /* Beam construction */
      (1) $l \leftarrow l + 1$;
      (2) Evaluate the utility $f(n)$ for each expandable node $n$ at level $l$;
      (3) Sort the nodes by their utility into an array $N$;
      (4) $B \leftarrow N[0 : k - 1]$, i.e., the top-$k$ nodes with the largest utilities;
until no expandable nodes found at the level $l$;
return $T$
```

it is necessary to expand a node in the trie; and (5) γ, a threshold on the yield-ratio (defined below) that determines the expandability of a node (defined below too). The algorithm outputs a trie with queries sampled from S.

Definition 3.1 (*Yield ratio*). Consider a search engine S and a node n in a trie T. Suppose the sampler obtains x number of completions from S by issuing y number of queries while it tries to expand n. Then the *yield ratio* of this expansion effort is x/y. Note that a node with a large yield ratio likely represents a cue word in a query template.

Definition 3.2 (*Expandable node*). A node n in a trie T is not *expandable* if (1) none of its ancestors is expandable or (2) its yield ratio is less than a threshold γ. Otherwise, n is expandable.

Algorithm: The algorithm consists of three major steps: (1) initialization; (2) expansion; and (3) beam construction.

(1) Initialization: This step sets up an initial trie T that contains only a root node storing the given $seed$. It then sets the current level l to 1, and the current beam B to

contain only the root of T. Note that B always stores the nodes selected for (possible) expansion at the current level l.

After the initialization, the algorithm then repeats step 2 (expand the nodes in the beam) and step 3 (form a new beam for the next level) until no more expandable nodes can be found. We now describe these two steps.

(2) Expansion: This step considers each node in the beam in turn. For each node n, the algorithm first determines if n is expandable. For this, it asks the engine S to complete the prefix query (formally defined below) for n, denoted as PrefixQuery(n) or simply as p (step 2.1). For example, suppose the root contains "books" and it has a child "for", then the prefix query for "for" will be "books for".

If the number of completions for p is less than a threshold τ, BEAMSAMPLER simply adds the completions to the trie and marks n as unexpandable (step 2.2). For example, a good value for τ is 10, since a search engine typically returns no more than 10 suggestions for a query. In other words, when the search engine S returns fewer than 10 completions, it is very likely that there are not many queries in S that can complete the partial query, and so further expanding n will not be productive.

Definition 3.3 (*Prefix path and query*). Given a trie T, a prefix path of a node n, denoted as PrefixPath(n), is the path from the root of T to n. The query formed by concatenating the words on the prefix path of n is called the prefix query of n, denoted as PrefixQuery(n). Prefix paths and queries in a subtree of T rooted at a node m of T are defined similarly, but with m acting as the root.

If S returns a sufficient number of completions, BEAMSAMPLER will then analyze the completions to see if there are completions that start with (a) a letter, (b) a digit, or (c) some special characters such as '$'. Note that it is possible that the completions contain queries for all the three cases.

Case a: If BEAMSAMPLER finds a completion that falls into the first case, it will construct 26 enumeration queries, each in the form of "PrefixQuery(n) [a-z]" (step 2.3.a). These enumeration queries are to discover completions to the prefix query at the node n that start with a particular letter (in [a-z]). For example, if the prefix query for the node n is "books for". Then the enumeration queries are "books for a", "books for b", etc. A possible completion to "books for a" is "books for adult".

Case b: If there is a completion that falls into the second case, the algorithm will build 10 partial queries, each consisting of PrefixQuery(n) followed by a digit, e.g., "books published in 2" (step 2.3.b). These queries will be useful for finding query templates (e.g., books published in [year]) whose parameter values are numbers (e.g., "books published in 2014").

Case c: Lastly, if there is a completion in the third case, the algorithm will form a query for each special character c, e.g., "books under $" (step 2.3.c). These queries will be useful to discover instances of parameters that have special data types, e.g., monetary values as in "books under $5".

All the queries obtained above are sent to S to obtain completions which are then added to T (step 2.3.d). The algorithm next computes the yield ratio for n (see Definition 3.1), based on the obtained completions. If the yield ratio is less than the threshold γ, BEAMSAMPLER will then mark n as unexpandable, or to be more exact, not further expandable (step 2.3.e).

(3) Beam construction: In this step, the algorithm moves onto the next level by incrementing l (step 3.1). Each expandable node n at this level is evaluated for its utility for expansion, using a utility function $f(n)$ as defined below. In particular, Q2P takes $f(n)$

to be the number of children of n. This captures the intuition that the more children n has, the more likely it will be part of a query template (e.g., a cue word).

The nodes are then sorted by their utility scores (step 3.3) and top-k nodes with the largest utilities are then selected as the candidates for the expansion (step 3.4). They effectively form the beam for the next iteration.

Definition 3.4 (*Utility of node*). The *utility* of a node n in a trie T, denoted by $f(n)$, measures the *usefulness* of n to the sampler for discovering queries that may reveal query templates.

It is important to note that a node can have children before it is expanded. These children are typically obtained when the ancestors of the node were expanded. Note also that the number of children of a node will typically grow after its expansion. The value of $f(n)$ above refers to the number of children of node n *before* n is expanded. A node with a large utility before the expansion is likely to obtain more children after the expansion.

For example, consider a trie with only the root "books". After expanding the root, we may obtain completions "books for kids" and "books for men". In this case, the node "for" (a child of root) already has two children: "kids" and "men", before it is even expanded. In other words, its $f(n) = 2$.

Example: To illustrate the working of the BEAMSAMPLER algorithm, consider sampling queries for the entity "books" with the beam size k set to 2, the probing threshold τ set to 2, and the yield-ratio threshold γ set to 5.

The initial trie and beam will then both contain a single node "books". Suppose that there are more than two completions obtained for the prefix query "books" (thereby exceeding the probing threshold). The "books" node is then eligible for further expansion. Suppose that the completions to the prefix query "books" all start with a letter. Then the algorithm will only send partial queries in the form of "books [a-z]", e.g., "books a", to the engine, when it expands "books".

Suppose that after expanding "books", we obtain a set of completions as shown in Figure 4.a. Since there are 7 completions, the yield ratio for "books" is 7, which exceeds the yield-ratio threshold. Hence, "books" will not be marked as unexpandable. As a result, all its children are now eligible for further expansion.

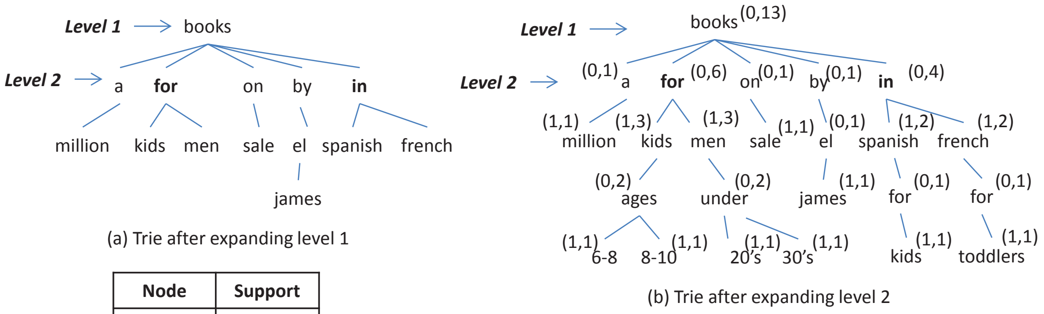
To decide on the expansion candidates for the nodes at the second level, the algorithm evaluates the utility of all the nodes at this level. Since nodes "for" and "in" (shown in bold font) have the largest numbers of children (thus the largest utility scores), they are added into the beam. Since the beam size is 2, no other nodes may be added.

The algorithm then goes on to the next iteration, i.e., expanding the nodes in the beam for the level 2. Figure 4.b shows the expansion result.

The similar process is then repeated on level 3 and so on, until no more expandable nodes can be found at some level.

Discussions: (1) Suppose that the maximum number of completions given by a search engine is b, e.g., typically $b = 10$. The maximum utility of a node may be greater than b. For example, consider expanding the root "books". Suppose that one of the completions to "books" (step 2.1 in Algorithm 1) contains "for", e.g., "books for kids". Suppose further that after completing "books f" (step 2.3.a), we obtain 10 completions such as "books for teens" and "books for boys" that all start with "books for". Further assume that "books for kids" is not among the completions. Then the "for" node (a child of "books") will have 11 children.

(2) The situation where all the completions to an enumeration query (e.g., "books f" in step 2.3.a) start with the same word (e.g., "for") is rare, unless there is an ex-



(a) Trie after expanding level 1

(b) Trie after expanding level 2

Node	Support
books	13
for	6
In	4
kids	3
under	2
men	3
ages	2
spanish	2
french	2

(c) Frequent nodes when min_sup = 2

Pattern	Support	Instances
books for	6	kids, men
books in	4	spanish, french
books for men under	2	20's, 30's
books for kids ages	2	6-8, 8-10

(d) Frequent noun phrase patterns and instances when min_sup = 2

Fig. 4. Examples illustrating the beam sampler and template discovery algorithms

tremely common phrase (especially a prepositional phrase, e.g., "books for ") used in the queries. More commonly, the completions to an enumeration query may start with a variety of words. For example, the completions to "books i" may contain popular phrases, e.g., "books in" as in "books in Spanish", but may also contain less popular phrases such as "books I should read". This in a sense reflects the diversity of queries users pose on the search engines. As a result, it is rare that the top-k nodes chosen to be expanded (step 3.4) would have exactly the same utility value.

(3) BEAMSAMPLER uses a beam of the same size, i.e., k (step 3.4), to restrict the expansion of the nodes at all levels of the trie. However, since short queries tend to have more completions than long ones, search engine may provide more suggestions for short queries. Thus, an alternative is to reduce the beam size when the expansion moves down to the lower levels of the trie.

Complexity analysis: First, we consider the maximum number of queries that BEAMSAMPLER sends to the search engine for a given seed, e.g., "books". To simplify the discussions, we assume that the completions to a prefix query (step 2.1) all start with a letter, which is a common case. Hence only 26 enumeration queries are sent in step 2.3 for each of the top-k nodes chosen to be expanded at level l. We further assume that the last level of the trie where nodes are expanded is h. Recall that the root is located at level one (e.g., see Figure 4.a). Then, except for level one, up to k nodes may be expanded at a level. For every expanded node, BEAMSAMPLER poses a prefix query and 26 enumeration queries to the search engine. Thus, the maximum number of queries is $27 + (h - 1) * k * 27$.

Next, we estimate the maximum size of the trie, i.e., the largest number of nodes that the trie can have. Here we make a simplifying assumption that there are no more than w words in a completion, e.g., w = 10. Then, for an expanded node at level l, each

ALGORITHM 2: PatMiner(T, σ): The frequent template miner

```
Input: $T$, a trie of queries obtained from a search engine;
       $\sigma$, a threshold on the minimum support count
Output: $\mathcal{P}$, a set of frequent templates
1. $\mathcal{N} \leftarrow$ DiscFreqNodes($T, \sigma$) /* Discover frequent nodes with support count $\geq \sigma$ */
2. $\mathcal{P} \leftarrow$ DiscFreqPats($T, \mathcal{N}$, $\sigma$) /* Discover frequent prefix templates with support count $\geq \sigma$ */
```

of its completions may add $w - l$ new nodes to the trie, one for each of the new words in the completion.

Suppose that the maximum number of completions to a query given by a search engine is b. Then an expanded node n at level l can have up to $27 * b$ completions and thus up to $27 * b * (w - l)$ new nodes may be added to the trie. Note that the completions to the prefix query of n (step 2.1) may not be among these obtained by the enumeration queries for n (step 2.3). Thus, the maximum number of nodes in the trie is $\sum_{l=1}^{w}(k' * 27 * b * (w - l))$, where k' = 1 when $l = 1$; and k otherwise.

3.2. Trie-based template discovery

As described in Section 3, the job of the template miner in Q2P (Figure 3) is to mine the templates hidden in the set of queries discovered by BeamSampler from a search engine. One way to implement the template miner is to represent each query as a sequence of words and apply existing sequential pattern mining algorithms, e.g., [Agrawal and Srikant 1995], to discover query templates. However, existing algorithms often require many passes through the data. Furthermore, they do not exploit the uniqueness of our problem setting: (1) Each discovered template captures the common prefix shared by a number of queries (see Definition 3.5 below). (2) Queries are already stored in a prefix-tree structure that reflects the common structure among different templates. Existing algorithms often need to spend extra efforts to construct such a structure from raw data, e.g., see PrefixScan [Pei et al. 2001], in order to reduce the computational costs in generating candidate subsequences.

We now describe PatMiner, a template mining algorithm that takes the advantage of our unique problem setting and mines query templates directly from the queries stored in a trie. We now first define several concepts used in the algorithm.

Definition 3.5 (*k-sequence, subsequence, and frequent k-sequence*). A query q with k words is a k-sequence: $<w_1, w_2, ..., w_k>$. A sequence q: $<w_1, w_2, ..., w_k>$ is a prefix subsequence of a sequence p if p contains q as its prefix. That is, p can be represented as $<w_1, w_2, ..., w_k, w_{k+1}, ..., w_n>$. We may refer to prefix subsequence simply as subsequence. Given a set of queries $\mathcal{Q}$, the support (or support count) of a sequence q is the fraction (or number) of queries in $\mathcal{Q}$ that contain q as a subsequence. A k-sequence is *frequent* if its support (or support count) is $\geq$ a threshold σ. σ may also be denoted as min_sup, i.e., the minimum support or support count.

Algorithm: Algorithm 2 shows the PatMiner algorithm. It takes as the input (1) a trie T that contains the queries obtained by BeamSampler from a search engine; and (2) σ, a threshold on the minimum support count for frequent templates. It outputs a set of frequent prefix templates in T that have at least the minimum support σ. We assume that the query trie T provided by BeamSampler also records, for each node n in T, the number of queries that end at n. These numbers will be used in PatMiner to compute the support counts of nodes in T.

PatMiner performs the discovery in two steps.

(1) Discovering frequent nodes: In the first step, the algorithm finds all frequent nodes $\mathcal{N}$ in the trie T by calling the function DISCFREQNODES. A node n is frequent if the prefix query of n, i.e., PrefixQuery(n) (see Definition 3.3), has at least the minimum support. In other words, there are at least σ number of queries in T that contain PrefixQuery(n) as a subsequence.

DISCFREQNODES uses a post-order traversal over T to find the frequent nodes. During the traversal, frequent leaf nodes are first found. These nodes are the ones which have at least σ number of queries ending at the nodes.

Next, DISCFREQNODES finds frequent internal nodes. It computes the support count of an internal node n as the sum of the support counts of all n's children plus the number of queries that end at n. It marks n as frequent if its support count is $\geq \sigma$.

(2) Discovering frequent patterns: In the second step, the function DISCFREQPATS discovers a set of frequent templates $\mathcal{P}$ based on the frequent nodes discovered at the first step. It traverses the trie T in pre-order, and checks if the current node n is frequent and also satisfies three conditions: (1) n is not the root of T; (2) n has at least two children; and (3) there are no queries that end at n. Note that if n has only one child, say m, then m is regarded as a node that expands the (incomplete) template of n. For example, if the template for n is "flights from lax" and m is "to", then the template expanded with m is "flights from lax to". Note also that condition 3 is to ensure that n will be followed by a parameter (see below).

If n satisfies the above conditions, DISCFREQPATS will output a template in the form of "E C P", where E represents the entity name at the root, C is a cue word or phrase formed by concatenating the words on the prefix path to n except for the root, and P is a parameter that represents the content following the cue phrase. The values of P are the (immediate) completions to C in the trie.

Example: Consider the trie produced by the BEAMSAMPLER and shown in Figure 4.b. Note that each node n in the trie is annotated with two numbers (inside ()). The first number is the number of queries that ends at n. As described above, these numbers are provided by BEAMSAMPLER. For example, no queries end at the node "for" at the second level; but there is one query "books for kids" that ends at the node "kids" at the third level.

The second number is generated by the DISCFREQNODES function, which represents the support count of the node n. As described above, the support counts of nodes are computed by DISCFREQNODES via a post-order traversal of the trie. For example, the support count of the node "kids" at the third level of Figure 4.b is 3, since the support count of its only child "ages" is 2, and there is only one query "books for kids" that ends at the node.

With the computed support counts, we know which nodes in the trie are frequent. For example, suppose min_sup (i.e., the minimum support count or σ in Algorithm 2) is 2. Figure 4.c lists all the frequent nodes found from the trie in Figure 4.b.

Finally, based on the frequent nodes, the algorithm discovers frequent patterns and their instances using the DISCFREQPATS function. The results are shown in Figure 4.d. For example, the first template is books for [P], where "books" is the entity, "for" is the cue word, and P is a parameter representing the targeted readers of books. The algorithm also discovers instances of P such as "kids" and "men", which are the immediate completions to the "for" node in the trie.

Discussions: In Section 5, we show that the majority of frequent instances, i.e., those appearing in a large number of queries, are semantically valid. However, it is possible that the same entity and cue phrase (e.g., "books for") may introduce instances of multiple semantic types (e.g., reader as in "books for kids" and holiday as in "books for Halloween") or may be followed by non-instances (e.g., "books for free").

One approach to determining the semantic types of instances is to learn, e.g., from a domain model, a classifier that predicts which attribute the instance in the sample query belongs to. Such a domain model may consist of a set of attributes and some values for the attributes, e.g., similar to the one in Deep2Q (Section 4).

Furthermore, if an instance is not recognized as having an existing type, it may be presented to the system developer for the possible addition of a new attribute to the domain model for representing the instance. A similar approach is often used in data integration systems to recognize the semantic types of attributes in a new data source, e.g., [Ives et al. 2009]. We leave this as one of the future works.

Properties: It can be shown that PATMINER has the following key properties on the efficiency of the algorithm and the characteristics of discovered templates.

PROPOSITION 3.6. PATMINER *makes two passes over the trie.*

Recall that PATMINER (Algorithm 2) makes the first pass through the trie (in a post-order traversal) to discover frequent nodes; it then makes another pass (in a pre-order traversal) to discover frequent patterns and instances.

PROPOSITION 3.7. *All templates discovered by* PATMINER *start with an entity name which corresponds to the root of the trie.*

As described above, PATMINER outputs templates in the form of "$E\ C\ P$", where E represents the entity at the root, P the parameter, and C the sequence of words on the path between the root and the parameter value in the trie.

4. INTEGRATING Q2P INTO DEEP2Q

We now first describe Deep2Q, a template-driven hybrid integration system for the Deep Web. We then discuss how Deep2Q can leverage Q2P to discover query templates for the integration system.

The Deep Web consists of over 25 million data sources whose contents are only accessible through form-based query interfaces [Madhavan et al. 2007]. These sources contain a huge amount of high-quality data and are becoming indispensable resources in almost every aspect of our lives, from online shopping, booking flights, to job hunting. A *key* challenge to finding the information on the Deep Web is that the desired information is often dispersed over a large number of heterogeneous and autonomous data sources. As a result, users need to locate the sources, understand their query forms, formulate separate queries, and assemble results from different sources—an extremely time-consuming and labor-intensive process.

To lift this burden from users, past research has proposed two very different approaches: virtual and surfacing, to integrating the Deep Web sources. Both approaches have their advantages and disadvantages. A virtual approach [Dragut et al. 2012; Khare et al. 2010; Dong and Naumann 2009] answers queries using fresh data from the sources, but suffers from slow query response. On the other hand, a surfacing approach [Madhavan et al. 2008; Raghavan and Garcia-Molina 2001; Sheng et al. 2012; Jin et al. 2011b; Jin et al. 2011a; Wu et al. 2006] can process queries more efficiently using the data prefetched from the sources. But it requires a hefty storage space and places an excessive load on the sources for crawling and refreshing the data. These reasons, plus the difficulty in formulating crawling queries, limit the current surfacing approaches to only being able to fetch a small portion of the Deep Web.

To address these limitations, we are developing Deep2Q, a novel template-driven *hybrid* integration system that aims to strike a balance between the two (extreme) approaches [Wu 2013b; Wu and Zhong 2013; Wu 2013a]. The *key* idea is that the system organizes the queries that it can answer into templates or parameterized queries

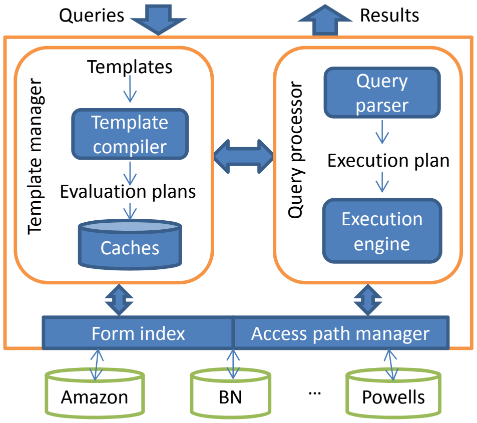


Fig. 5. The Deep2Q architecture

(e.g., jobs in [location] where location is a parameter in the template) and proactively determines how to answer the queries in the templates. This in effect constructs evaluation plans for the templates. For popular (i.e., frequently asked) queries (e.g., “jobs in Chicago”) in the templates, the system may also obtain answers from the sources ahead of query time and cache the results for fast query response. This *selective* caching strategy avoids the high computational and storage costs as in the surfacing approach. For the queries that the system does not have cached results, their answers may be obtained on the fly from the sources by leveraging the precompiled evaluation plans for their corresponding templates. This avoids the need to perform expensive query planning and optimization at runtime [Ives et al. 2000] as in the virtual approach.

Deep2Q **architecture:** Figure 5 shows the architecture of Deep2Q for a domain $\mathcal{D}$ of interest (e.g., book search). Deep2Q maintains a model M for $\mathcal{D}$ which consists of a set of domain attributes (e.g., author and language) and a set of templates which are parameterized queries defined using the domain attributes in $\mathcal{D}$.

Deep2Q grows the domain model M in a bootstrapping fashion. Initially, M may be empty or may contain a few popular attributes and instances extracted from a domain ontology or provided by domain experts. Next, Deep2Q starts to discover query templates and their instances from its underlying search engines. As described in Section 3.2, when a new query template T is found, the parameter P in T will be checked, e.g., via a classifier trained using the current version of M, to see if it is semantically similar to some attribute in M. If yes, new instances of P found in the queries will be added into M. Otherwise, a new attribute (representing P) and its instances will be added to M, after the possible verification by the system developer or domain expert. This in effect grows M into a new version M'.

Deep2Q operates in two modes: offline and online. A *template manager* (rounded rectangle on the left in Figure 5) and a *query processor* (on the right in Figure 5) are responsible for the offline and online operations respectively. When offline, the template manager takes a set of templates as the input and invokes a *template compiler* to construct evaluation plans for the templates. For popular queries, it also instantiates the plans (by replacing parameters with values from the queries), executes the plans, and store results in the *caches* repository. When online, the *query parser* takes a user query Q, matches Q with the templates, and instantiates the matched templates. If Q turns out to be a popular query and has already been processed, it simply fetches its result from the caches. If not, it will execute the instantiated plan for Q and fetch the result from the sources.

Query language: A *key* issue that needs to be addressed when developing a data integration system is the choice of *query language*. Existing data integration systems typically expect users to write queries in some sorts of structured query languages such as SQL and XQuery. We clearly cannot expect average web users to have the knowledge of these languages. In fact, as described above, even the forms could be overly complex for web users. On the other hand, keyword queries, even though they have become extremely popular especially with web search engines, are notoriously known for their ambiguities. To address this issue, we propose to use *parameterized phrase queries* and their instances to represent templates and queries in Deep2Q respectively. In particular, we focus on queries expressed in noun phrases (see Definition 2.1). A noun phrase query language L then consists of all noun phrases in a domain of interest.

Research has found that a large fraction of web queries (i.e., queries posed on search engines) are noun phrases [Li 2010]. Furthermore, we believe that noun phrase queries are much more precise than the keyword queries. They are also more natural and less complex to formulate than the form queries for the average web users. Readers are referred to [Wu 2013b] for the challenges and our solutions to query parsing.

Template discovery: In addition, there is an important issue of *how templates are obtained*. Intuitively, Deep2Q would want to have templates that can capture most (if not all) of user queries. It would also want to know what queries in each template are most likely to be asked so that it can make informed decision on caching their results. One solution is to have Deep2Q continuously monitor user queries, discover frequent queries, and then build query templates. The downside is that Deep2Q may perform very poorly initially and thus might not attract enough users to learn query patterns.

Another solution is to learn popular queries and query patterns from *query logs* of web search engines [Pandey and Punera 2012; Agarwal et al. 2010]. However, as discussed earlier, while search engine logs are invaluable resources for discovering query templates, they are often proprietary and not readily available to the general public.

To address this challenge, Deep2Q incorporates Q2P into its template manager (Figure 3) as a new module for template discovery. This module periodically samples the queries from the underlying search engines, using the beam-search query sampler as described in Section 3.1, and discovers new query patterns from the sampled queries, using the trie-based pattern miner, as described in Section 3.2.

5. PERFORMANCE EVALUATION

We have conducted an extensive set of experiments to evaluate the performance of Q2P. We used four search engines in these experiments and obtained query completions via their RESTful web services: Amazon (product search) [Amazon 2014], Bing [Bing 2014], Google [Google 2014], and Yahoo! [Yahoo! 2014]. We considered a variety of entities including book, flight, job, movie, car, and apartment.

The goal of our experiments is to measure the efficiency of BEAMSAMPLER in sampling queries from search engines and the effectiveness of PATMINER in discovering meaningful templates. In particular, a key metric in measuring the efficiency of BEAMSAMPLER is its *yield ratio*, which is given by the ratio of the number of unique completions (i.e., completed queries) obtained from a search engine to the number of queries BEAMSAMPLER poses to the engine. In other words,

Table I. The performance of BeamSampler over different entities and engines

Entity	Engine	# of levels	# of queries	# of completions	Yield ratio
flight	Bing	8	1,122	9,993	8.9
	Google	7	799	5,284	6.6
	Yahoo!	11	1,600	8,562	5.3
job	Bing	10	1,473	12,650	8.6
	Google	9	1,290	6,429	5
	Yahoo!	10	1,398	7,042	5
movie	Bing	8	967	10,897	11.3
	Google	7	815	4,581	5.6
	Yahoo!	8	925	5,393	5.8
car	Bing	9	1,223	8,267	6.7
	Google	9	1,231	5,123	4.2
	Yahoo!	7	824	4,458	5.4
apartment	Bing	9	1,439	8,674	6
	Google	8	965	4,680	4.8
	Yahoo!	8	1,173	5,952	5.1

$$\mathrm{YieldRatio}(\mathrm{seed}, \mathrm{engine}) = \frac{\#\mathrm{completions}(\mathrm{seed}, \mathrm{engine})}{\#\mathrm{queries}(\mathrm{seed}, \mathrm{engine})} \quad (1)$$

Note that a search engine may suggest up to k (e.g., k = 10) completions to a partial query. If there are a large number of past queries that start with the partial query (e.g., “jobs in”), then the engine will typically return the k most popular completions. Otherwise, the number of completions may be much smaller than k (e.g., there may be fewer completions to “jobs xerox” than “jobs in”). Since our goal is to learn query templates, we would want our sampler to be able to sample popular queries (e.g., those completing “jobs in” such as “jobs in Chicago” and “jobs in Houston”) that reveal common interests of users (e.g., as captured in the query template jobs in [location]). If a partial query does not have many completions, then it is unlikely to be a part of popular queries, and hence less useful for template discovery. To achieve this, BeamSampler uses the beam-search strategy to avoid the expansion of nodes in the trie that correspond to such partial queries. Thus, if the strategy works, BeamSampler should have a large yield ratio as defined in Formula 1.

Furthermore, since it is prohibitive to send a large number of queries to a search engine, we would also want our sampler to be able to obtain as many completions as possible with a minimum number of partial queries. Thus, if a sampler is productive or efficient, then it should also have a large yield ratio.

5.1. Evaluating BeamSampler

We conducted *three* sets of experiments to evaluate BeamSampler. In the first set of experiments, we ran BeamSampler over different entities and engines, and analyzed its effectiveness. In the second set of experiments, we examined the contribution of different components in BeamSampler to its efficiency. In the third set of experiments, we studied the effect of beam size on the performance.

(1) Overall performance: Table I shows the results of the *first* set of experiments. For each of the five entity types in the first column, it shows the results of running BeamSampler on three engines: Bing, Google, and Yahoo!. In each case, the beam size (i.e., k in Algorithm 1) was set to 10; the threshold on the minimum number of completions for expansion (i.e., τ) was set to 10; and the threshold on the minimum yield ratio of an expandable node (i.e., γ) was set to 5. Starting from the third column, the table shows

Table II. The contribution of components in BEAMSAMPLER to its performance (entity = "book", beam size = 10)

Engine	Component	# of queries	# of completions	Yield ratio
Amazon	All	597	2,516	**4.2**
	No beam	4,727	7,825	1.6
	No min yield	645	2,381	3.7
	No probing	2,257	2,361	1
Bing	All	1,091	9,250	**8.5**
	No beam	stopped at level 3		
	No min yield	1,117	9,143	8.2
	No probing	2,997	8,908	3
Google	All	1,135	5,470	**4.8**
	No beam	stopped at level 5		
	No min yield	1,097	5,180	4.7
	No probing	2,997	4,909	1.6
Yahoo!	All	930	5,327	**5.7**
	No beam	stopped at level 2		
	No min yield	1,787	9,802	5.5
	No probing	7,437	21,461	2.9

the number of levels in the trie constructed by BEAMSAMPLER, the number of queries sent to the engine, the number of completions obtained, and the overall yield ratio.

We can observe that the number of levels ranges from 7 (flight at Google) to 11 (flight at Yahoo!), the number of queries ranges from 799 (flight at Google) to 1,600 (flight at Yahoo!), and the number of completions ranges from 4,458 (car at Yahoo!) to 12,640 (job at Bing). The yield ratio ranges from 4.2 (car at Google) to 11.3 (movie at Bing).

These indicate that: (1) the engines may produce substantially long completions, e.g., with 11 words; (2) the number of queries posed to the engine is modest, e.g., no more than 1,600; and (3) the algorithm is quite efficient, e.g., only two cases where the overall yield ratio falls below 5.

Note that the yield ratio may be greater than 10 (e.g., 11.3 in Table I for movie at Bing). The reason is that search engine, e.g., Bing, may return more than 10 completions to a partial query, and all completions are retained by the algorithm.

(2) Contribution of components: Table II shows the results of the *second* set of experiments. In this set of experiments, we ran BEAMSAMPLER on four engines to discover queries for the "book" entity. The beam size was set to 10 in all these experiments. For each engine, we collected the performance data in four cases. In the first case (*All* in the table), all key components in BEAMSAMPLER are in effect. These components are (a) *beam*, i.e., the method for reducing the search space using beam; (b) *min yield*, i.e., the technique of using a threshold γ on the minimum yield-ratio for stopping the expansion of unproductive nodes; and (c) *probing*, i.e., the method of probing the engine using a prefix query before sending expensive enumeration queries.

The next three cases each corresponds to the BEAMSAMPLER with one of the key components removed. For example, *no beam* means that BEAMSAMPLER does not use the beam to restrict the nodes for further completion, but the minimum yield threshold γ and the probing are still in effect.

From the table, we can observe that the full-fledged BEAMSAMPLER (i.e., the *All* case with no component removed) has the highest yield ratio at all four engines, compared to the version of BEAMSAMPLER with one of the key components removed. These ratios (e.g., 4.2 for Amazon) were highlighted using bold font in the last column of the table.

When beam was not used to restrict the expansion, the yield ratio dropped to 1.6 (from 4.2) at Amazon. Recall that the beam is used to pick the top-k most promising nodes (i.e., likely to generate many completions). So when beam is not used, the al-

Table III. The effect of beam size in BeamSampler (entity = book)

Engine	Beam size	# of queries	# of completions	Yield ratio
Amazon	5	319	1,350	**4.23**
	10	597	2,516	4.21
	15	713	2,580	3.6
	20	966	3,224	3.3
Bing	5	569	6,296	**11.1**
	10	1,091	9,250	8.5
	15	1,721	13,298	7.7
	20	2,641	18,481	7
Google	5	503	3,211	**6.4**
	10	1,135	5,470	4.8
	15	1,510	6,878	4.5
	20	1,974	8,940	4.5
Yahoo!	5	658	3,640	5.5
	10	930	5,327	**5.7**
	15	1,629	8,764	5.4
	20	2,088	10,765	5.1

gorithm will send a query to search engines for every node in the trie. Since many of these nodes might not bring back many completions, the yield ratio, which measures the average yield of nodes, may drop significantly.

To make it worse, sending too many queries to the engines may cause the algorithm to be blocked from the access to the engines. Indeed, this occurred to all engines except for Amazon. For example, the sampling process was forced to terminate at Bing, when the algorithm was expanding the nodes on the third level of the trie. These indicate the importance of the beam-based sampling.

Using the minimum yield-ratio threshold γ further prunes away some of the less productive nodes among the top-k nodes selected by the beam. This results in a notable increase in the yield ratio, ranging from .1 at Google to .5 at Amazon.

The results also show that probing was very effective in improving the yield rate. For example, significant increase can be observed in all four engines, with the increase ranging from 2.8 (Yahoo) to as high as 5.5 (Bing).

(3) Effect of beam size: Table III compares, for each of the four engines, the performance of BeamSampler for the entity “book”, when the beam size is set to 5, 10, 15, and 20. We can observe that the yield ratio tends to drop when the size of beam increases. For example, three engines had the best ratio with the beam size set to 5; and one engine (Yahoo!) had the best with the beam size of 10. The drop of the yield ratio from the beam size of 5 to 20 ranges from .4 (Yahoo!) to 4.1 (Bing).

So the question is what is the best beam size? Could it be 5 or 10 as the above results suggested? To better answer this question, we further examined the performance of BeamSampler when the beam size ranges from 1 to 10. Figure 6 plots the results.

Note that as described in Section 3.1, when the beam size is k, BeamSampler only select top-k nodes with the largest utility at each level of the trie for expansion. So if the beam size is one, then first, the root is selected for expansion; next, only one of its children will be selected for the next level of expansion; and so on.

From the figure, we can observe that the beam size that gives the best yield ratio may differ from engine to engine. For example, the best beam size is 2 for Bing, 4 for Google, and 1 for Yahoo! and Amazon.

We can also observe the overall trend that the yield ratio tends to drop when the beam size is increased, although some fluctuations may occur, e.g., there is a large jump from the beam size of 1 to 2 for Bing. This suggests that the beam size can not be

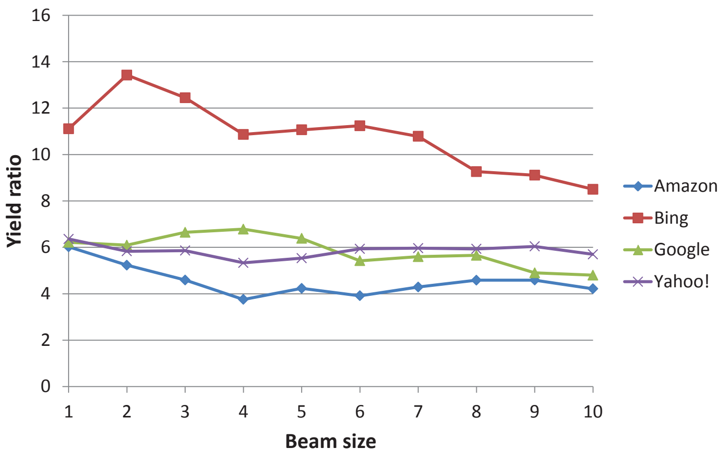


Fig. 6. Yield ratios when the beam size increases from 1 to 10

Table IV. Number of templates discovered by PATMINER when the minimun support varies

Entity	min_sup = .001			min_sup = .01			min_sup = .05		
	Bing	Google	Yahoo!	Bing	Google	Yahoo!	Bing	Google	Yahoo!
flight	57	56	55	28	25	32	8	9	7
job	98	103	92	29	32	32	10	7	7
movie	45	78	49	14	17	20	9	7	8
car	73	92	58	8	22	16	6	8	9
apartment	33	64	55	17	22	26	9	8	9

too large. On the other hand, smaller beam size tends to produce fewer completions, as can be observed from Table III. This suggests that the beam size can not be too small.

Thus, if we set the minimum beam size to be 5, we can estimate from Figure 6 that the best beam sizes for Amazon, Bing, Google, and Yahoo! are 8, 6, 8, and 9 respectively.

5.2. Evaluating PATMINER

In evaluating PATMINER, we performed three sets of experiments. First, we evaluated the quantity and quality of the templates discovered by PATMINER. Second, we collected and examined the instances discovered by the algorithm. Third, we studied the effects of the beam size in BEAMSAMPLER and the minimum support threshold for the frequent templates in PATMINER on the quality of discovered templates.

(1) Template discovery: In this set of experiments, we used the the completion queries obtained by BEAMSAMPLER from three engines for five different types of entities as shown in Table I. The beam size of BEAMSAMPLER was set to 10 for these experiments.

Recall from Section 3.2 that PATMINER traverses twice over each trie to discover frequent templates that have at least the minimum support. It also extracts instances of parameters from the completions that follow the templates (examined next).

Table IV shows, for each entity and engine, the number of templates discovered by PATMINER when the minimum support threshold, denoted as min_sup, varies from .001, .01, to .05.

We can observe that the number of discovered templates ranges from 33 (apartment at Bing) to 103 (job at Google) when min_sup is .001, from 8 (car at Bing) to 32 (e.g.,

Table V. Top-10 templates discovered by PATMINER (min_sup = .001)

Entity	Bing	Google	Yahoo!
flight	flights from (4361)	flights from (2152)	flights new (5226)
	flights to (667)	flights in (302)	flights new xbox 360 jasper youth (2956)
	flights out of (561)	flights from london to (294)	flights new xbox 360 jasper youth nike (2278)
	flights into (541)	flights with (276)	flights of (614)
	flights from lax to (531)	flights to (272)	flights new xbox 360 jasper youth nike zoom (614)
	flights from chicago to (525)	flights out of (267)	flights new xbox 360 jasper youth nike golf (408)
	flights from houston to (505)	flights london to (260)	flights with (403)
	flights from atlanta to (478)	flights from denver to (256)	flights in (347)
	flights from tampa to (431)	flights from paris to (250)	flights new xbox 360 jasper youth knee (331)
	flights from new (421)	flights from charlotte (245)	flights new xbox 360 jasper youth nike vapor (276)
job	jobs that (4312)	jobs that (3042)	jobs that (3609)
	jobs with (1467)	jobs with (859)	jobs that require (895)
	jobs for (1385)	jobs that pay (650)	jobs you can (588)
	jobs that pay (1098)	jobs for (641)	jobs you can get with a (584)
	jobs on (999)	jobs that require (633)	jobs that make (434)
	jobs that require (972)	jobs with a (604)	jobs for (355)
	jobs near (882)	jobs that make (444)	jobs that pay over (332)
	jobs with a (734)	jobs for 15 year olds in (294)	jobs that require a (324)
	jobs in (671)	jobs that involve (292)	jobs that need (292)
	jobs at (664)	jobs in (270)	jobs in (269)
movie	movies for (1337)	movies on (1112)	movies coming (1744)
	movies like (1248)	movies like (700)	movies coming out (1151)
	movies in (928)	movies from (519)	movies of (688)
	movies playing at (909)	movies new (321)	movies coming out on (398)
	movies on (790)	movies like the (286)	movies in (367)
	movies playing in (645)	movies in (269)	movies for (360)
	movies with (642)	movies at (268)	movies of the (334)
	movies at (627)	movies that (169)	movies coming out in (316)
	movies like the (523)	movies that came out in (167)	movies coming to (213)
	movies playing at the (335)	movies on netflix about (139)	movies 8 in (210)
car	cars for (1854)	cars that (1643)	cars with (904)
	cars in (1046)	cars in (390)	cars 4 (661)
	cars with (838)	cars for sale by owner in (386)	cars for (590)
	cars for sale in (715)	cars that are (357)	cars to (462)
	cars under (541)	cars that start with (320)	cars with the (402)
	cars in the (450)	cars with (311)	cars with the best (400)
	cars for under (336)	cars for sale in (286)	cars 1 (309)
	cars under 5000 in (206)	cars that have (260)	cars in (267)
	cars under 4000 in (66)	cars that run (190)	cars and (254)
	cars in the movie (63)	cars that run on (189)	cars xbox (167)
apartment	apartments for (2141)	apartments with (1015)	apartments for (2392)
	apartments for rent in (2138)	apartments on (734)	apartments for rent in (1386)
	apartments near (1542)	apartments near (631)	apartments on (580)
	apartments on (1167)	apartments at (476)	apartments for rent in new (567)
	apartments in (660)	apartments on the (402)	apartments columbus (555)
	apartments for rent in new (494)	apartments for rent in (314)	apartments near (527)
	apartments west (473)	apartments in (297)	apartments for rent in san (488)
	apartments for rent in san (445)	apartments downtown (277)	apartments on the (331)
	apartments san (429)	apartments south (215)	apartments for sale in (325)
	apartments for rent in fort (234)	apartments at the (211)	apartments in (270)

flight at Yahoo!) when min_sup is .01, and from 6 (car at Bing) to 10 (job at Bing) when min_sup is .05.

We further examined the templates discovered by PATMINER. Table V shows the top-10 most frequent templates and their support counts (in parenthesis), when the min_sup is set to .001. For each template, it shows its "E C" part where E is the entity name (e.g., "flights") and C is the cue phrase (e.g., "from", see Section 3.2). For example, the most frequent template for flight discovered from the completion queries from Bing is "flights from", which appeared as a prefix in 4,361 (out of 9,993, see Table I) completions.

From Table V, we can observe that almost all the templates start with the entity name as the head noun. A few exceptions occur in "cars and" (e.g., "cars and trucks") and "cars 1" at Yahoo! (e.g., "cars 1 game"). We can also see that the head noun is typically followed by either a phrase or a clause. There are two types of phrases: (1) prepositional, e.g., "flights from", "jobs for", and "movies on"; and (2) participial, e.g., "movies playing at" and "movies coming out". The clauses are mostly introduced by "that", e.g., "jobs that require", "jobs that need", and "cars that run".

Furthermore, we can see that prepositions are commonly used to introduce the parameters (and thus their values). For example, "flights from", "jobs in", "movies at", "cars under", and "apartments near". Nevertheless, other types of introductory words may also occur in the templates. For example, verbs as in "jobs that require", determiners as in "jobs with a", and nouns as in "flights from new" (where "new" starts a place name, e.g., "New York" and "New Orleans").

An interesting situation occurs when the entity name has multiple senses. For example, flights could be airline flights or flight games as in the templates for "xbox 360 flight games" found at Yahoo!. Furthermore, sometimes the cue phrases for the parameters may start with a number, e.g., "cars 4" as in "cars 4 less" where 4 stands for "for".

(2) Instance discovery: In the second set of experiments, we examined the instances discovered by PATMINER. Table VI shows the top-10 instances that contain a single word for some of the top templates shown in Table V. These instances are ranked by the number of occurrences in the queries that follow the templates.

We can observe that for the majority of the templates, the discovered instances have similar semantic meanings. For example, the instances that follow "flights from" are mostly city, state, or country names. These instances can thus be captured in the same parameter, e.g., [location], to form a complete template, e.g., flights from [location].

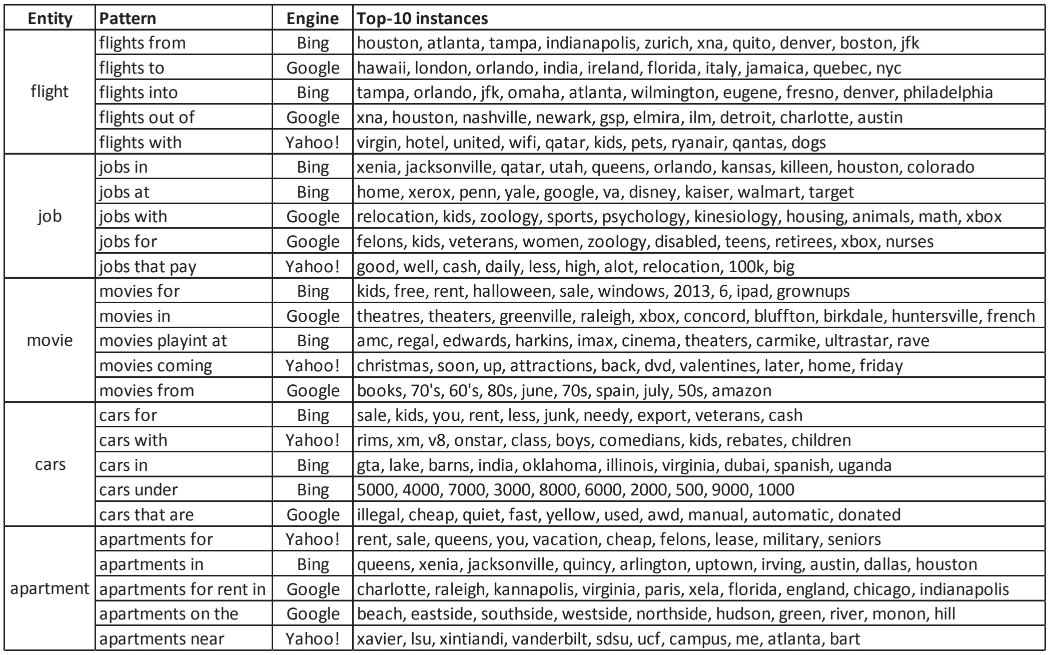
There are also several templates whose instances are of mixed types. For example, the instances for "movies for" can be people (e.g., "kids" and "grownups"), holidays (e.g., "Halloween"), and years (e.g., "2014"). In these cases, it is best to construct a (refined) template for each different type of instances. For example, movies for [person], movies for [holiday], and movies for [year].

As discussed in Section 3.2, to determine the semantic type of the instances, we may employ a classifier trained using the attributes and instances in a domain model (e.g., the domain model maintained by Deep2Q).

We also examined the instances discovered by different engines for the same template. Table VII shows the results for five templates at three engines: Bing, Google, and Yahoo!.

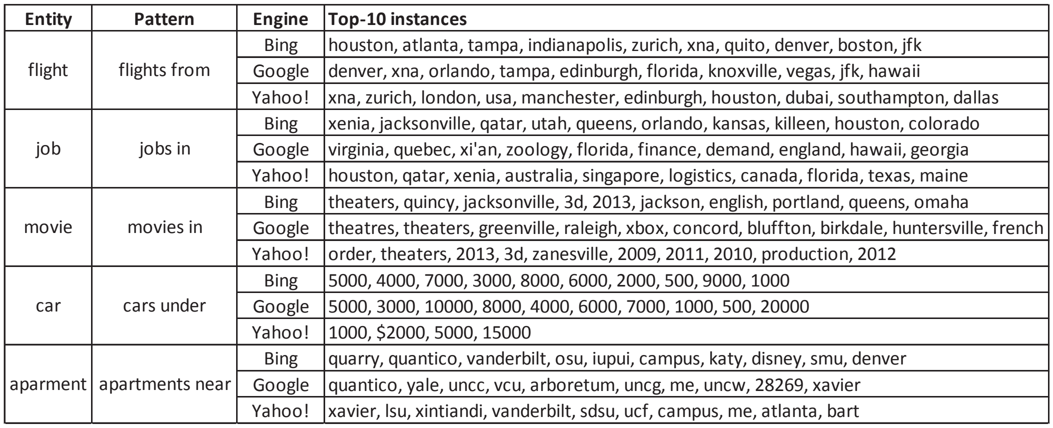
We can observe that the top instances from different engines may be very different. For example, there are only three ("Denver", "xna", and "jfk") in common between the top-10 instances for "flights from" discovered at Bing and Google. Moreover, the top templates from different engines may also vary a lot as Table V shows. Hence, mining templates and instances simultaneously from multiple engines can not only increase

Table VI. Top-10 one-word instances for some of the top templates



Entity	Pattern	Engine	Top-10 instances
flight	flights from	Bing	houston, atlanta, tampa, indianapolis, zurich, xna, quito, denver, boston, jfk
	flights to	Google	hawaii, london, orlando, india, ireland, florida, italy, jamaica, quebec, nyc
	flights into	Bing	tampa, orlando, jfk, omaha, atlanta, wilmington, eugene, fresno, denver, philadelphia
	flights out of	Google	xna, houston, nashville, newark, gsp, elmira, ilm, detroit, charlotte, austin
	flights with	Yahoo!	virgin, hotel, united, wifi, qatar, kids, pets, ryanair, qantas, dogs
job	jobs in	Bing	xenia, jacksonville, qatar, utah, queens, orlando, kansas, killeen, houston, colorado
	jobs at	Bing	home, xerox, penn, yale, google, va, disney, kaiser, walmart, target
	jobs with	Google	relocation, kids, zoology, sports, psychology, kinesiology, housing, animals, math, xbox
	jobs for	Google	felons, kids, veterans, women, zoology, disabled, teens, retirees, xbox, nurses
	jobs that pay	Yahoo!	good, well, cash, daily, less, high, alot, relocation, 100k, big
movie	movies for	Bing	kids, free, rent, halloween, sale, windows, 2013, 6, ipad, grownups
	movies in	Google	theatres, theaters, greenville, raleigh, xbox, concord, bluffton, birkdale, huntersville, french
	movies playint at	Bing	amc, regal, edwards, harkins, imax, cinema, theaters, carmike, ultrastar, rave
	movies coming	Yahoo!	christmas, soon, up, attractions, back, dvd, valentines, later, home, friday
	movies from	Google	books, 70's, 60's, 80s, june, 70s, spain, july, 50s, amazon
cars	cars for	Bing	sale, kids, you, rent, less, junk, needy, export, veterans, cash
	cars with	Yahoo!	rims, xm, v8, onstar, class, boys, comedians, kids, rebates, children
	cars in	Bing	gta, lake, barns, india, oklahoma, illinois, virginia, dubai, spanish, uganda
	cars under	Bing	5000, 4000, 7000, 3000, 8000, 6000, 2000, 500, 9000, 1000
	cars that are	Google	illegal, cheap, quiet, fast, yellow, used, awd, manual, automatic, donated
apartment	apartments for	Yahoo!	rent, sale, queens, you, vacation, cheap, felons, lease, military, seniors
	apartments in	Bing	queens, xenia, jacksonville, quincy, arlington, uptown, irving, austin, dallas, houston
	apartments for rent in	Google	charlotte, raleigh, kannapolis, virginia, paris, xela, florida, england, chicago, indianapolis
	apartments on the	Google	beach, eastside, southside, westside, northside, hudson, green, river, monon, hill
	apartments near	Yahoo!	xavier, lsu, xintiandi, vanderbilt, sdsu, ucf, campus, me, atlanta, bart

Table VII. Top-10 instances discovered by different engines for the same template

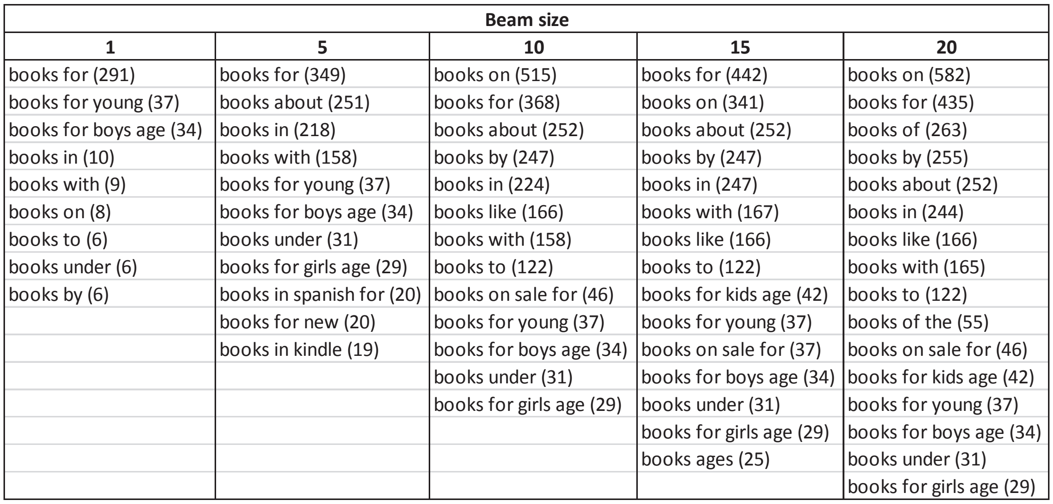


Entity	Pattern	Engine	Top-10 instances
flight	flights from	Bing	houston, atlanta, tampa, indianapolis, zurich, xna, quito, denver, boston, jfk
		Google	denver, xna, orlando, tampa, edinburgh, florida, knoxville, vegas, jfk, hawaii
		Yahoo!	xna, zurich, london, usa, manchester, edinburgh, houston, dubai, southampton, dallas
job	jobs in	Bing	xenia, jacksonville, qatar, utah, queens, orlando, kansas, killeen, houston, colorado
		Google	virginia, quebec, xi'an, zoology, florida, finance, demand, england, hawaii, georgia
		Yahoo!	houston, qatar, xenia, australia, singapore, logistics, canada, florida, texas, maine
movie	movies in	Bing	theaters, quincy, jacksonville, 3d, 2013, jackson, english, portland, queens, omaha
		Google	theatres, theaters, greenville, raleigh, xbox, concord, bluffton, birkdale, huntersville, french
		Yahoo!	order, theaters, 2013, 3d, zanesville, 2009, 2011, 2010, production, 2012
car	cars under	Bing	5000, 4000, 7000, 3000, 8000, 6000, 2000, 500, 9000, 1000
		Google	5000, 3000, 10000, 8000, 4000, 6000, 7000, 1000, 500, 20000
		Yahoo!	1000, $2000, 5000, 15000
aparment	apartments near	Bing	quarry, quantico, vanderbilt, osu, iupui, campus, katy, disney, smu, denver
		Google	quantico, yale, uncc, vcu, arboretum, uncg, me, uncw, 28269, xavier
		Yahoo!	xavier, lsu, xintiandi, vanderbilt, sdsu, ucf, campus, me, atlanta, bart

the productivity but also improve the diversity of discovery results. It may also reduce the burden (i.e., the need to complete a large number of sampling queries) on individual engines since the mining task is in a sense distributed among multiple engines.

Discovering from multiple sources may even be necessary when the sources have different coverage on the subject. For example, consider two bookstores: one that sells mostly textbooks and the other novels. In this case, the instances (i.e., authors) discovered from the two sources for the template books written by [author] could be very

Table VIII. Frequent templates when beam size varies (min_sup = .01, entity = Amazon, entity = book)



Beam size				
1	5	10	15	20
books for (291)	books for (349)	books on (515)	books for (442)	books on (582)
books for young (37)	books about (251)	books for (368)	books on (341)	books for (435)
books for boys age (34)	books in (218)	books about (252)	books about (252)	books of (263)
books in (10)	books with (158)	books by (247)	books by (247)	books by (255)
books with (9)	books for young (37)	books in (224)	books in (247)	books about (252)
books on (8)	books for boys age (34)	books like (166)	books with (167)	books in (244)
books to (6)	books under (31)	books with (158)	books like (166)	books like (166)
books under (6)	books for girls age (29)	books to (122)	books to (122)	books with (165)
books by (6)	books in spanish for (20)	books on sale for (46)	books for kids age (42)	books to (122)
	books for new (20)	books for young (37)	books for young (37)	books of the (55)
	books in kindle (19)	books for boys age (34)	books on sale for (37)	books on sale for (46)
		books under (31)	books for boys age (34)	books for kids age (42)
		books for girls age (29)	books under (31)	books for young (37)
			books for girls age (29)	books for boys age (34)
			books ages (25)	books under (31)
				books for girls age (29)

different, since there are not many authors who have written both textbook and novel. Note that knowing that different sources contain different instances for the same template is extremely valuable to Deep2Q (Section 4), which can direct queries asking for specific instances (e.g., books by certain authors) to the appropriate sources (e.g.,, bookstores that carry the books by the authors), thereby greatly reducing the query overhead.

(3) Effects of beam size and min_sup threshold: In the third set of experiments, we further studied the effect of two key parameters in the algorithms on the discovered templates. First, we examined if the beam size in BEAMSAMPLER affects the quantity and diversity of templates. Intuitively, a larger beam size should lead to the discovery of more templates which may also be more diverse. Second, we examined the quantity and quality of templates when the minimum support threshold, min_sup, for frequent templates in PATMINER varies. Here, we wanted to understand if all the templates discovered using a certain min_sup are meaningful and how to choose a good min_sup. This expands the first set of experiments where we only examined the templates when min_sup was set to .001 (Table V).

Effect of beam size: Table VIII shows the frequent templates discovered by PATMINER for the entity "book", using the queries obtained by BEAMSAMPLER from Amazon with varied beam sizes: 1, 5, 10, 15, and 20. The minimum support threshold for the frequent templates, min_sup, was set to .01 (i.e., 1%) in this experiment.

From the table, we can make the following observations.

- The algorithm tends to discover more frequent templates when the beam size increases. For example, the number of templates increases from 9 to 11, when the beam size is increased from 5 to 10. Furthermore, the percentage of increase becomes smaller with larger beam sizes. For example, there is 22% increase (9 increased to 11) from the beam size of 1 to 5, compared to 6% increase (15 to 16) from the beam size of 15 to 20.
- Although the min_sup (1%) stays the same, the number of queries that support the templates, i.e., the support count, may increase with a larger beam size. The increase is most significant on smaller beam sizes, e.g., the support for "books in" is 10 when

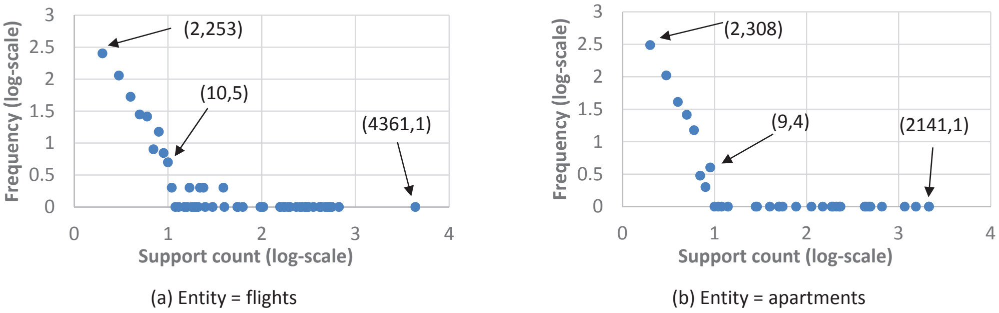


Fig. 7. Distribution of support counts of templates (engine = Bing)

the beam size is 1, and 218 when the beam size is increased to 5. However, the increase is becoming less significant for larger beam sizes. For example, the support count “books by” stays the same (247) when the beam size is increased from 10 to 15, and increases only by 8 (to 255) when the beam size is 20.

— Suppose that a template T' is considered to be a *refinement* of another template T, if T is a prefix of T'; and T is considered to be a *major* template, if it is not a refinement of any other templates. For example, “books for” is a major template in the beam size of 1, while “books for young” is a refinement to “books for”. We see that the number of major templates is 7, 5, 9, 10, and 10 respectively for the beam size of 1, 5, 10, 15, and 20. The most significant increase in the number of major templates occurs when the beam size is increased from 5 to 10, where *four* more major templates were discovered, e.g., “books on”. Interestingly, the algorithm discovered all the major templates that were discovered with the beam size of 1 and 5, when the beam size was set to 10. Finally, we can see that the number of major templates tends to change less when the beam size is sufficiently large, e.g., greater than 5.

— There are a significant number of refinement templates for almost every different beam size. Except for the beam size of 1 (two refinement templates), all other beam sizes generate either 4 (beam size = 10 and 15) or 6 refinement templates (beam size = 5 and 20). The percentage of refinement templates ranges from 22% (beam size = 1) to as high as 54.5% (beam size = 5). For example, there are four refinement templates for “books for” and two for “books in”, when the beam size is 5.

Similar observations as above can be made for other entities at other search engines. In summary, if we measure the diversity of a set of templates by the number of major templates among them, then we can see that when beam size is sufficiently large (e.g., greater than 5), further increasing beam size does not change the diversity of templates dramatically. On the other hand, larger beam size (e.g., greater than 10) tends to have lower yield-ratio, hence higher cost of sampling. These suggest that the best beam size as suggested by yield-ratio (see Figure 6 and discussions in Section 5.1) typically also leads to the discovery of much diverse templates.

Effect of min_sup: Figure 7 shows the distribution of support counts of templates discovered for the entities flights (Figure 7.a) and apartments (Figure 7.b) at the engine Bing. x-axis represents the *distinct* support count of templates and y-axis the number of templates having the support count (i.e., the frequency). Both numbers are in log-scale (base 10). For example, point (4361, 1) in Figure 7.a corresponds to the template “flights from” (first one in Table IX). There is only one template with the same support, hence the frequency 1. Similarly, point (2, 253) indicates that there are

253 templates with the support count of 2. Note that the minimum support count of a template is 2 (see Section 3.2, condition 2 in step "discovering frequent patterns").

We can clearly observe the elbows shapes of point distribution in both figures (similar observations can also be made for other entities and engines). The left part of the elbow consists of low-support templates, while the right part high-support ones. The "joint" of the elbow contains the bordering points.

Thus, one way to determine the min_sup threshold for frequent templates is to choose a point at the joint of the elbow and use its support count as the cut-off value. For example, we may choose a point at the end of the left part of the elbow whose support count occurs a sufficiently large number of times, e.g., about 5, and use the support count as the threshold. For example, this method may pick (10, 5) for the flights templates and (9, 4) for the apartments template (as shown in the figures). In other words, the support count threshold for frequent templates is 10 and 9 respectively for the two entities. Since the total number of queries is 9,086 and 7,647 respectively for flights and apartments, the above support counts correspond to the support thresholds (i.e., in fraction) of .11% for flights and .12% for apartments. Similar observations can be made for other entities and engines. This suggests that .1% may be a reasonable threshold for our template discovery task.

Furthermore, this threshold (.1%) may prune away a large number of infrequent templates. For example, it removes about 90% of flight templates (with a support count of less than 10) and about 95% of apartment templates (with a support count of less than 9) from further considerations.

From Table IV, we can see that the threshold of .1% will produce frequent templates in the number of 33 to 103. In some applications, e.g., Deep2Q, it may be too costly to support a large number of templates, due to the overhead for planning the queries in the templates. For these applications, larger thresholds may be more desirable. For example, Table IV shows that the threshold of 1% may reduce the number of templates to somewhere between 8 to 32; and the threshold of 5% will only produce a very small number (e.g., no more than 10) of strong templates with very large support.

Finally, to provide an idea of the templates discovered using varied thresholds, Table IX shows all the templates for flights discovered from Bing with the support threshold of .1%. The templates are ranked (columns 1 and 5) by their support counts (columns 3 and 7) whose corresponding fractional support values are given in columns 4 and 8. To ease the comparison, we highlight the last templates with the support of .1%, 1%, and 5% in bold font.

We can see that the templates capture a great variety of questions users may ask when searching for flights such as origin, destination, and airfare. They also capture variations and refinements of questions. The questions, variations, and refinements may vary greatly in their popularity (i.e., how frequently they are asked); hence their corresponding templates may have different levels of support.

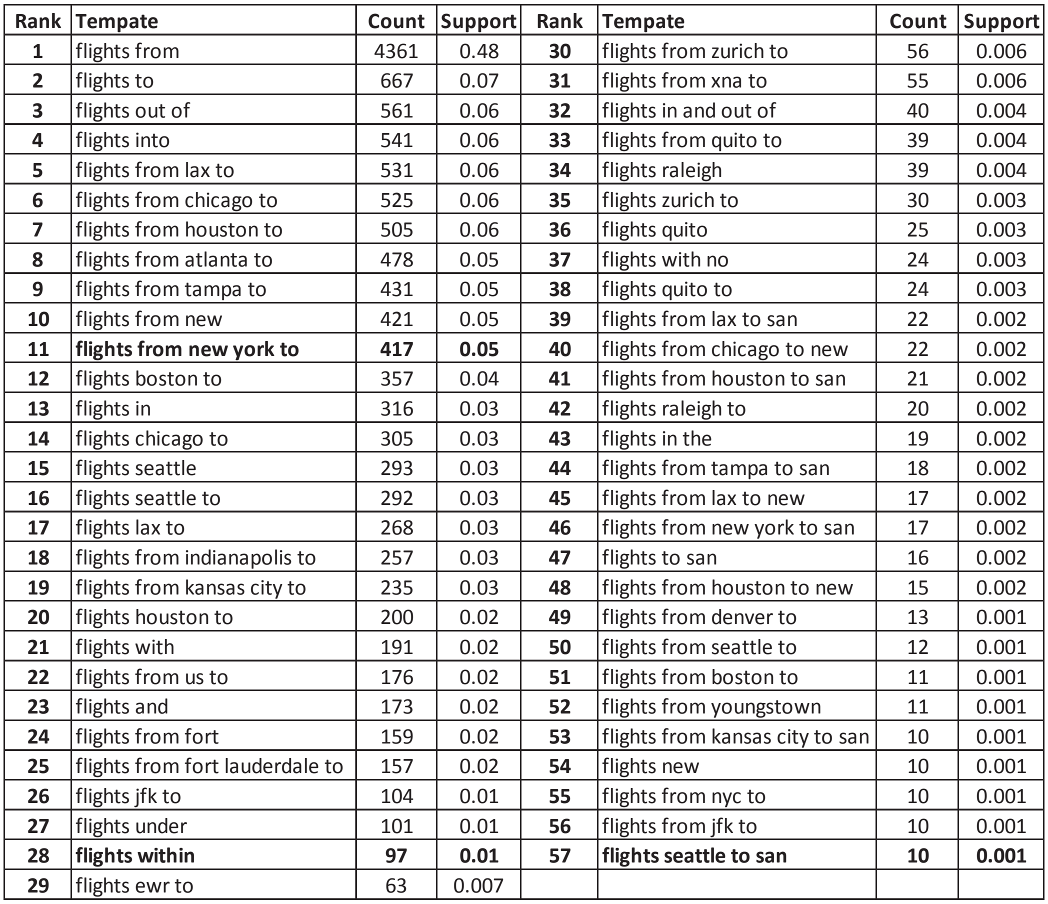
For example, "flights from" (rank 1, support = 48%) is much more popular than "flights under" (rank 27, support = 1%). This shows that people search for flights with specific origin more often than airfare. "flights from" is also much more popular than its variant "flights out of" (rank 3, support = 6%).

As another example, "flights from lax to" (rank 5, support = 6%) is much more popular than "flights from seattle to" (rank 50, support = .1%). Presumably more people fly out of LAX than Seattle. Note that both templates refine "flights from" by specifying flight origins (i.e., LAX and Seattle).

6. RELATED WORK

We discuss related works from the following perspectives.

Table IX. Top templates ranked by their support counts (engine = Bing, entity = flights)



Rank	Tempate	Count	Support	Rank	Tempate	Count	Support
1	flights from	4361	0.48	**30**	flights from zurich to	56	0.006
2	flights to	667	0.07	**31**	flights from xna to	55	0.006
3	flights out of	561	0.06	**32**	flights in and out of	40	0.004
4	flights into	541	0.06	**33**	flights from quito to	39	0.004
5	flights from lax to	531	0.06	**34**	flights raleigh	39	0.004
6	flights from chicago to	525	0.06	**35**	flights zurich to	30	0.003
7	flights from houston to	505	0.06	**36**	flights quito	25	0.003
8	flights from atlanta to	478	0.05	**37**	flights with no	24	0.003
9	flights from tampa to	431	0.05	**38**	flights quito to	24	0.003
10	flights from new	421	0.05	**39**	flights from lax to san	22	0.002
11	**flights from new york to**	**417**	**0.05**	**40**	flights from chicago to new	22	0.002
12	flights boston to	357	0.04	**41**	flights from houston to san	21	0.002
13	flights in	316	0.03	**42**	flights raleigh to	20	0.002
14	flights chicago to	305	0.03	**43**	flights in the	19	0.002
15	flights seattle	293	0.03	**44**	flights from tampa to san	18	0.002
16	flights seattle to	292	0.03	**45**	flights from lax to new	17	0.002
17	flights lax to	268	0.03	**46**	flights from new york to san	17	0.002
18	flights from indianapolis to	257	0.03	**47**	flights to san	16	0.002
19	flights from kansas city to	235	0.03	**48**	flights from houston to new	15	0.002
20	flights houston to	200	0.02	**49**	flights from denver to	13	0.001
21	flights with	191	0.02	**50**	flights from seattle to	12	0.001
22	flights from us to	176	0.02	**51**	flights from boston to	11	0.001
23	flights and	173	0.02	**52**	flights from youngstown	11	0.001
24	flights from fort	159	0.02	**53**	flights from kansas city to san	10	0.001
25	flights from fort lauderdale to	157	0.02	**54**	flights new	10	0.001
26	flights jfk to	104	0.01	**55**	flights from nyc to	10	0.001
27	flights under	101	0.01	**56**	flights from jfk to	10	0.001
28	**flights within**	**97**	**0.01**	**57**	**flights seattle to san**	**10**	**0.001**
29	flights ewr to	63	0.007				

Query template discovery: As mentioned in Section 1, existing works [Agarwal et al. 2010; Pandey and Punera 2012] require that search engines provide query logs for discovering query templates. For example, [Agarwal et al. 2010] generates candidate query templates using domain schemas and instances, and discovers popular templates by matching candidates with queries in a search engine query log. [Pandey and Punera 2012] develops a generative modeling approach to discovering query templates from a query log.

Besides search engines, autocompletion features may also be found in command shells, word processors, and query forms [Nandi and Jagadish 2007]. Tries are commonly used to implement query autocompletion. However, we are not aware of any prior works that utilize tries to store and mine queries sampled from search engines.

[Bar-Yossef and Gurevich 2008] also considers the problem of sampling queries in search engines via autocompletion. It considers two scenarios: one where the completions follow a uniform distribution, i.e., all completions have the same importance; and the other where some suggestions may be more important than others, e.g., judged by their popularity (i.e., the number of times people have asked the queries).

It presents a random-walk sampler that obtains random query samples through autocompletion interfaces of engines. It uses AOL query log [Pass et al. 2006] to estimate the importance of query completions for the popularity-based scenario above, to direct the process of random walk.

As discussed earlier (Section 1), a key distinction between our work and [Bar-Yossef and Gurevich 2008] is that, we focus on obtaining popular queries, rather than a random sample of queries that include both popular and less popular ones. In other words, our sampling is biased vs. the random sampling in [Bar-Yossef and Gurevich 2008].

Note that, in addition to the past user queries found in the query logs, search engines may also use content analysis and dictionaries to help suggest query completions [Bar-Yossef and Gurevich 2008]. Note also that query autocompletion services typically do not provide frequency counts on the completed queries. Nevertheless, the fact that a query is being suggested (indirectly) indicates its popularity or significance.

Beam search is commonly used to reduce the search space in a search problem [Russell and Norvig 2010]. For example, [Dhamankar et al. 2004] employs a similar beam-search strategy to help quickly identify complex matches among elements from different schemas.

The problem of sequential template mining has been extensively studied in data mining [Agrawal and Srikant 1995]. A key difference between these works and ours is that we focus on discovering special templates that have a common first item (i.e., the entity name). Furthermore, we leverage the fact that the queries are already stored in a trie and develop a novel trie-based mining algorithm for discovering query templates.

Template-driven data integration: Data integration has been extensively studied in the database community for over 30 years [Halevy et al. 2006b; Doan et al. 2001; Madhavan et al. 2005; Doan and Halevy 2005; Doan et al. 2012]. Despite these efforts, building a data integration system over a large number of sources remains to be an extremely challenging task that requires a huge amount of upfront efforts and maintenance costs. The flurry of sources now available on the Web further presses the need for a scalable solution. To address this challenge, recently *dataspace* [Halevy et al. 2006a; Salles et al. 2007; Jeffery et al. 2008; Sarma et al. 2008; Chai et al. 2009; Talukdar et al. 2010] has been proposed as a new paradigm for managing a large number of diverse data sources. It advocates a best-effort pay-as-you-go approach to bootstrapping and evolving a complex system. An open question then is *where should the initial efforts be focused on and how can the system evolve with continuous efforts?* Our research on Deep2Q contributes to this quest by suggesting a principled template-driven approach to incrementally constructing and evolving a complex data integration system.

Query templates in Deep2Q: [Nandi and Jagadish 2009] proposes to create views in a database to capture user interests and permit keyword search over the views instead of the entire database to improve the search efficiency. Although these views bear some similarity with the templates in Deep2Q, they are *distinct* in several key aspects. First, views in [Nandi and Jagadish 2009] may include all attributes of an entity and thus are more similar to query forms. In contrast, the templates in Deep2Q rarely have more than 3 attributes. Second, these views are defined on a single database, while answers to the queries in the templates come from multiple data sources. Third, users are expected to search the content of these views using keywords. In other words, views are not user queries. In contrast, a template in Deep2Q directly captures a set of user queries.

7. CONCLUSIONS AND FUTURE WORK

We have presented Q2P, a system for discovering query templates from search engines via their query autocompletion services. Q2P is part of our Deep2Q project that aims to develop a hybrid template-driven data integration system for integrating a set of data sources or search engines on the Deep Web in a domain of interest. The key to the success of Deep2Q is its ability to capture common user query interests using a set of query templates. However, discovering such templates from search engines is challenging since search engines typically do not want to disclose their query logs.

To address this challenge, Q2P leverages query autocompletion services of search engines to discover query templates. Q2P has two distinct aspects. (1) It uses a trie to store the completed queries and monitor the progress of continuous query sampling, and employs a beam-search strategy to focus the expansion of the trie on the most promising nodes. (2) It takes the advantage of the fact that sample queries are stored in a trie to efficiently discover query templates using only two traversals of the nodes in the trie.

Experiments using four search engines (beam size = 10) indicate that Q2P sends only a moderate number of queries (ranging from 597 to 1135) to the engines, while being quite productive in obtaining query samples (yielding 4.2 to 8.5 completions per query). Moreover, a significant number of templates (ranging from 8 to 32 with the minimum support = 1%) may be discovered from the samples. We also observe that prepositions are commonly used to introduce the parameters (e.g., "jobs in Chicago" and "flights from LA") in the templates.

Besides further evaluation of Q2P on additional engines and domains, we are extending it to learn query templates with multiple parameters. A possible solution is to combine and generalize the discovered single-parameter templates. For example, from "flights from Chicago", "flights from Houston", and many other similar queries, Q2P may learn a template T_1: flights from [city]. Furthermore, assuming that "Chicago" is a popular origin, there may be many queries like "flights from Chicago to Hawaii" and "flights from Chicago to New York". From this, we may discover another template T_2: "flights from Chicago to [city]". Then, by combining T_1 and T_2, we may infer a new two-parameter template: flights from [city] to [city].

We are also integrating Q2P into Deep2Q to evaluate the effectiveness of the discovered templates in capturing user query interests, e.g., how frequent each template is used by Deep2Q users, and how many user queries are not captured by the templates. An interesting direction is to start Deep2Q with the templates learned by Q2P, monitor user interaction with Deep2Q, record user queries, and then gradually add new templates into Deep2Q to support queries that are not captured by the existing templates.

Acknowledgments: This work was supported in part by the Guangdong Natural Science Foundation (Grant No. S2013010016852), BNU-HKBU United International College internal grant, and the National Natural Science Foundation of China (Grant No. 61303180 and 61573163).

REFERENCES

Ganesh Agarwal, Govind Kabra, and Kevin Chen-Chuan Chang. 2010. Towards rich query interpretation: walking back and forth for mining query templates. In *Proc. of WWW*. 1–10.

Rakesh Agrawal and Ramakrishnan Srikant. 1995. Mining Sequential Patterns. In *Proc. of ICDE*. 3–14.

Amazon. 2014. Amazon Autocompletion API. http://completion.amazon.com/search/complete?method=completion&search-alias=aps&client=amazon-search-ui&mkt=1&x=updateISSCompletion&sc=1&noCacheIE=1294493634389&q=[query]. (2014).

Ziv Bar-Yossef and Maxim Gurevich. 2008. Mining search engine query logs via suggestion sampling. *PVLDB* 1, 1 (2008), 54–65.

Bing. 2014. Bing Autocompletion API. http://api.search.live.com/osjson.aspx?query=[query]. (2014).

Xiaoyong Chai, Ba-Quy Vuong, AnHai Doan, and Jeffrey F. Naughton. 2009. Efficiently incorporating user feedback into information extraction and integration programs. In *Proc. of SIGMOD*. 87–100.

Robin Dhamankar, Yoonkyong Lee, AnHai Doan, Alon Y. Halevy, and Pedro Domingos. 2004. iMAP: Discovering Complex Mappings between Database Schemas. In *Proc. of SIGMOD*. 383–394.

AnHai Doan, Pedro Domingos, and Alon Halevy. 2001. Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. In *Proc. of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*. 509–520.

AnHai Doan, Alon Halevy, and Zazhary Ives. 2012. *Principles of Data Integration*. Morgan Kaufmann.

AnHai Doan and Alon Y. Halevy. 2005. Semantic Integration Research in the Database Community: A Brief Survey. *AI Magazine* 26, 1 (2005), 83–94.

Xin Luna Dong and Felix Naumann. 2009. Data fusion - Resolving Data Conflicts for Integration. *PVLDB* 2, 2 (2009), 1654–1655.

Eduard C. Dragut, Weiyi Meng, and Clement T. Yu. 2012. *Deep Web Query Interface Understanding and Integration*. Morgan & Claypool Publishers.

Google. 2014. Google Autocompletion API. http://google.com/complete/search?output=firefox&q=[query]. (2014).

Alon Y. Halevy, Michael J. Franklin, and David Maier. 2006a. Dataspaces: A New Abstraction for Information Management. In *Proc. of DASFAA*. 1–2.

Alon Y. Halevy, Anand Rajaraman, and Joann J. Ordille. 2006b. Data Integration: The Teenage Years. In *Proc. of VLDB*. 9–16.

Zachary G. Ives, Craig A. Knoblock, Steven Minton, Marie Jacob, Partha Pratim Talukdar, Rattapoom Tuchinda, José Luis Ambite, Maria Muslea, and Cenk Gazen. 2009. Interactive Data Integration through Smart Copy & Paste. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*. http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_71.pdf

Zachary G. Ives, Alon Y. Levy, Daniel S. Weld, Daniela Florescu, and Marc Friedman. 2000. Adaptive Query Processing for Internet Applications. *IEEE Data Eng. Bull.* 23, 2 (2000), 19–26.

Shawn R. Jeffery, Michael J. Franklin, and Alon Y. Halevy. 2008. Pay-as-you-go user feedback for dataspace systems. In *Proc. of SIGMOD*. 847–860.

Xin Jin, Nan Zhang, and Gautam Das. 2011a. Attribute domain discovery for hidden web databases. In *Proc. of SIGMOD*. 553–564.

Xin Jin, Nan Zhang, Aditya Mone, and Gautam Das. 2011b. Randomized Generalization for Aggregate Suppression Over Hidden Web Databases. *PVLDB* 4, 11 (2011), 1099–1110.

Ritu Khare, Yuan An, and Il-Yeol Song. 2010. Understanding deep web search interfaces: a survey. *SIGMOD Record* 39, 1 (2010), 33–40.

Xiao Li. 2010. Understanding the Semantic Structure of Noun Phrase Queries. In *Proc. of ACL*. 1337–1345.

J. Madhavan, P. Bernstein, A. Doan, and A. Halevy. 2005. Corpus-based Schema Matching. In *Proc. of ICDE*. 57–68.

Jayant Madhavan, Shirley Cohen, Xin Luna Dong, Alon Y. Halevy, Shawn R. Jeffery, David Ko, and Cong Yu. 2007. Web-Scale Data Integration: You can afford to Pay as You Go. In *Proc. of CIDR*. 342–350.

Jayant Madhavan, David Ko, Lucja Kot, Vignesh Ganapathy, Alex Rasmussen, and Alon Y. Halevy. 2008. Google's Deep Web crawl. *PVLDB* 1, 2 (2008), 1241–1252.

Arnab Nandi and H. V. Jagadish. 2007. Effective Phrase Prediction. In *Proc. of VLDB*. 219–230.

Arnab Nandi and H. V. Jagadish. 2009. Qunits: queried units in database search. In *Proc. of CIDR*.

Sandeep Pandey and Kunal Punera. 2012. Unsupervised extraction of template structure in web search queries. In *Proc. of WWW*. 409–418.

Greg Pass, Abdur Chowdhury, and Cayley Torgeson. 2006. A picture of search. In *Proceedings of the 1st International Conference on Scalable Information Systems, Infoscale 2006, Hong Kong, May 30-June 1, 2006*. 1.

Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. 2001. PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth. In *Proc. of ICDE*. 215–224.

Sriram Raghavan and Hector Garcia-Molina. 2001. Crawling the Hidden Web. In *Proc. of VLDB*. 129–138.

Stuart J. Russell and Peter Norvig. 2010. *Artificial Intelligence - A Modern Approach*. Pearson Education. I–XVIII, 1–1132 pages.

Marcos Antonio Vaz Salles, Jens-Peter Dittrich, Shant Kirakos Karakashian, Olivier René Girard, and Lukas Blunschi. 2007. iTrails: Pay-as-you-go Information Integration in Dataspaces. In *Proc. of VLDB*. 663–674.

Anish Das Sarma, Xin Dong, and Alon Y. Halevy. 2008. Bootstrapping pay-as-you-go data integration systems. In *Proc. of SIGMOD*. 861–874.

Cheng Sheng, Nan Zhang, Yufei Tao, and Xin Jin. 2012. Optimal Algorithms for Crawling a Hidden Database in the Web. *PVLDB* 5, 11 (2012), 1112–1123.

Partha Pratim Talukdar, Zachary G. Ives, and Fernando Pereira. 2010. Automatically incorporating new sources in keyword search-based data integration. In *Proc. of SIGMOD*. 387–398.

Ping Wu, Ji-Rong Wen, Huan Liu, and Wei-Ying Ma. 2006. Query Selection Techniques for Efficient Crawling of Structured Web Sources. In *Proc. of ICDE*. 47.

Wensheng Wu. 2013a. The Deep Web: Woven to Catch the Middle Ground. In *Proc. of ACM CIKM Workshop on Web-scale Knowledge Representation, Retrieval, and Reasoning (Web-KR)*. 5–8.

Wensheng Wu. 2013b. Proactive Natural Language Search Engine: Tapping into Structured Data on the Web. In *Proc. of EDBT*. 61–67.

Wensheng Wu and Tingting Zhong. 2013. Searching the deep web using proactive phrase queries. In *Prof. of WWW*. 137–138.

Yahoo! 2014. Yahoo! Autocompletion API. http://ff.search.yahoo.com/gossip?output=fxjson&command=[query]. (2014).