A Search Engine for Natural Language Applications

Michael J. Cafarella
Department of Computer Science and
Engineering
University of Washington
Seattle, WA 98195 U.S.A.
mjc@cs.washington.edu

Oren Etzioni
Department of Computer Science and
Engineering
University of Washington
Seattle, WA 98195 U.S.A.
etzioni@cs.washington.edu

ABSTRACT

Many modern natural language-processing applications utilize search engines to locate large numbers of Web documents or to compute statistics over the Web corpus. Yet Web search engines are designed and optimized for simple human queries—they are not well suited to support such applications. As a result, these applications are forced to issue millions of successive queries resulting in unnecessary search engine load and in slow applications with limited scalability.

In response, this paper introduces the Bindings Engine (BE), which supports queries containing *typed variables* and *string-processing functions*. For example, in response to the query *"powerful ⟨noun⟩"* BE will return all the nouns in its index that immediately follow the word "powerful", sorted by frequency. In response to the query *"Cities such as ProperNoun(Head(⟨NounPhrase⟩))"*, BE will return a list of proper nouns likely to be city names.

BE's novel *neighborhood index* enables it to do so with $O(k)$ random disk seeks and $O(k)$ serial disk reads, where k is the number of non-variable terms in its query. As a result, BE can yield several orders of magnitude speedup for large-scale language-processing applications. The main cost is a modest increase in space to store the index. We report on experiments validating these claims, and analyze how BE's space-time tradeoff scales with the size of its index and the number of variable types. Finally, we describe how a BE-based application extracts thousands of facts from the Web at interactive speeds in response to simple user queries.

Categories and Subject Descriptors

H.3.1 [**Information Systems**]: Content Analysis and Indexing; H.3.3 [**Information Systems**]: Information Search and Retrieval; E.2 [**Data**]: Data Storage Representations

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Search engine, indexing, variables, query, corpus, language, information extraction

1. INTRODUCTION AND MOTIVATION

Modern Natural Language Processing (NLP) applications perform computations over large corpora. With increasing frequency, NLP applications use the Web as their corpus and rely on queries to commercial search engines to support these computations [21, 10, 11, 8]. But search engines are designed and optimized to answer people's queries, not as building blocks for NLP applications. As a result, the applications are forced to issue literally millions of queries to search engines, which can overload search engines, and limit both the speed and scalability of the applications.

In response, Google has created the "Google API" to shunt programmatic queries away from Google.com and has placed hard quotas on the number of daily queries a program can issue to the API. Other search engines have also introduced mechanisms to block programmatic queries, forcing applications to introduce "courtesy waits" between queries and to limit the number of queries they issue.

Having a "private" search engine would enable an NLP application to issue a much larger number of queries quickly, but efficiency is still a problem. To support the application, each document that matches a query has to be retrieved from a random location on a disk. Thus, the number of random disk seeks scales linearly with the number of documents retrieved.[1] As index sizes grow, the number of matching documents would tend to increase as well. Moreover, many NLP applications require the extraction of strings matching particular syntactic or semantic types from each page. The lack of NLP type data in the search engine's index means that many pages are fetched and processed at query time only to be discarded as irrelevant.

We now consider two specific NLP applications to illustrate the sorts of computations they perform. Consider, first, Turney's widely used PMI-IR algorithm [21]. PMI-IR computes Pointwise Mutual Information (PMI) between terms by estimating their co-occurrence frequency based on hit counts returned by a search engine. Turney used PMI-IR to classify words as positive or negative by computing their PMI with positive words (*e.g.*, 'excellent') subtracted from their PMI with negative words (*e.g.*, 'poor') [22]. Turney then applied this word classification technique to a large number of adjectives, verbs, and adverbs drawn from product reviews in order to classify the reviews as positive or negative. In this approach, the number of search engine queries scales linearly with the number of words clas-

---

[1] Of course, it may be possible to distribute this load across a large number of machines, but BE embodies a much more efficient solution.

sified, which limits the speed and scale of PMI-IR applications.

As a second example, consider the KnowItAll information extraction system [10]. Inspired by Hearst's early work [13], KnowItAll relies on a set of generic extraction patterns such as "<Class> including <ProperNoun>" to extract facts from text. KnowItAll instantiates the patterns with the names of the predicates of interest (*e.g.*, <Class> is instantiated to 'films'), and sends the instantiated portion of the pattern as a search engine query (*e.g.*, the phrase query "films including") in order to discover pages containing sentences that match its patterns. KnowItAll is designed to quickly extract large number of facts from the Web but, like PMI-IR, is limited by the number and rate of search-engine queries it can issue.

In general, statistical NLP systems perform a wide range of corpora-based computations including training parsers, building n-gram models, identifying collocations, and more [16]. Recently, database researchers have also begun to make use of corpus statistics in order to better understand data and schema semantics (*e.g.*, [12, 15]). We have "boiled down" the requirements of this large and diverse body of applications to a concise set of desiderata for search engines that support NLP applications.

1.1 Desiderata

To satisfy the broad set of computations that NLP applications perform on corpora, we need a search engine that satisfies the following desiderata:

- Support queries that contain one or more typed variables (*e.g.*, "powerful $\langle noun \rangle$").

- Provide a facility for defining variable types (*e.g.*, syntactic types such as *verb* or semantic ones such as *address*), and for efficiently assigning types to strings at index-creation time.

- Support queries that contain simple string-processing functions over variable bindings (*e.g.*, *"Books such as ProperNoun(Head($\langle$NounPhrase$\rangle$)))"*).[2]

- Require at most $O(k)$ random disk seeks to correctly answer queries containing variables, where k is the number of *concrete* terms (*i.e.*, not variables) in the query.

- Process queries that contain only concrete terms just as efficiently as a standard search engine.

- Minimize the impact on index-construction time and space.

The main contribution of this paper is the introduction of the fully-implemented Bindings Engine (BE), which satisfies each of the above desiderata by introducing a variabilized query language, an augmented inverted index called the *neighbor index*, and an efficient algorithm for processing variabilized queries.

Our second contribution is an asymptotic analysis of BE comparing it with a standard search engine (see Table 1 in

---

[2]The function *Head* extracts the Head noun in the noun phrase, and the function *ProperNoun* is a Boolean-valued function that determines whether its argument is a proper noun.

Section 3.3). The analysis shows that the number of random disk seeks for a standard search engine is $O(B + k)$, where k is the number of variables and B is the number of possible bindings. But the number of random seeks is only $O(k)$ for BE. Thus, when B is large, which we expect for any sizable corpus, BE is *much* faster than a standard engine. Like a standard engine, the space to store BE's index increases linearly with the number of documents indexed.

Our third contribution is a set of experiments aimed at measuring the performance of BE in practice. We found that on a broad range of queries, BE was more than two orders of magnitude faster than a standard search engine. Its query-time efficiency was paid for by a factor of four increase in index size, and a corresponding increase in index-construction time.

Our final contribution is that BE enables *interactive information extraction*, whereby users can employ simple queries to extract large amounts of information from the Web at interactive speeds. For example, a person could query a BE-based application with the word "insects", and receive the results as shown in Figure 6.

The reminder of this paper is organized as follows. Section 2 introduces BE's query language, followed by a description of BE's *neighbor index* and its query-processing algorithm in Section 3. Section 4 presents our experimental results, and Section 5 sketches different applications of BE's capabilities. We conclude with related work and directions for future work.

2. QUERY LANGUAGE

This section introduces BE's query language, focusing on how query *variables*, *types* and *functions* are handled.
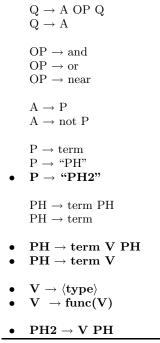
A standard search engine query consists of one or more words (or *terms*) with optional logical operators and quotation marks (which indicate a phrase query). BE extends the query language by adding *variables*, each of which has a *type*. BE processes a variable by returning every possible string in the corpus that can be substituted for the variable and still satisfy the user's query, and which has a matching type. We call a string that meets these requirements a *binding* for the variable in question.

For example, the query "President $\langle Name \rangle$ Clinton" will return as many bindings as there are distinct strings of type *Name* in the corpus, appearing between occurrences of "President" and "Clinton." See Figure 1 for the full query language.

For each type, a BE system must be provided with a type name and a *type recognizer* to find all instances of appropriate strings in the corpus. Reasonable types might include syntactic categories (*e.g.*, noun phrases, adjectives, adverbs), or semantic categories (*e.g.*, names, addresses, phone numbers). Of course, we also allow untyped variables, which simply match the adjacent term. (For example, the query "strong $\langle term \rangle$" will return all the indexed strings to the right of the word 'strong' in the index). BE accepts an arbitrary set of type recognizers, so the set of types can be tailored to the intended applications at index construction time. BE's types are fixed once the index has been computed.

A query can also include *functions*, which apply to a binding string and return a processed version of the string. As with type recognizers, a set of functions is supplied to the BE system before it can run. BE can apply functions at query time to bindings found in the index. For example, in-

$$Q \rightarrow A \text{ OP } Q$$
$$Q \rightarrow A$$

$$OP \rightarrow and$$
$$OP \rightarrow or$$
$$OP \rightarrow near$$

$$A \rightarrow P$$
$$A \rightarrow not P$$

$$P \rightarrow term$$
$$P \rightarrow \text{``PH''}$$
- **P → "PH2"**

$$PH \rightarrow term \; PH$$
$$PH \rightarrow term$$

- **PH → term V PH**
- **PH → term V**

- **V → ⟨type⟩**
- **V → func(V)**

- **PH2 → V PH**

Examples:
"President Bush ⟨Verb⟩"
"cities such as *ProperNoun(Head(⟨NounPhrase⟩))*"
"<NounPhrase> is the capital of <NounPhrase>"

Figure 1: A grammar for the BE query language. The grammar specifies that a phrase must consist of at least one term, and all variables V are non-consecutive. Non-Terminal symbols are in CAPS, and novel operations appear with a • and in bold-face. 'Term' is a whitespace-delimited string found in the corpus. 'Type' is a member of the set of string types determined at index time. The search engine binds items within angled brackets to specific strings in corpus documents. 'func()' is a binding-processor function that accepts a single string and returns a modified version of that string.

stead of creating an indexed type ⟨*Name*⟩ as above, we might instead create the more general-purpose ⟨*NounPhrase*⟩; we can constrain it to be a name with a function "fullName()", which returns any human names found inside the binding for ⟨*NounPhrase*⟩. Functions are a convenience for query processing.

2.1 Discussion

We formulated the query language in Figure 1 so that all variables are non-consecutive, and that all variables have a neighboring concrete term. This constrains the number of positions in a document where a successful variable assignment can be found, and is important for efficient query processing (see Section 3). While it would be possible to process variables that do not neighbor concrete terms, the sheer number of bindings means that such a query is unlikely to be useful or efficiently processable. Thus, we have chosen to exclude it from the language.

One common question is how variabilized queries differ from the NEAR operator. NEAR takes two terms as arguments. If the two terms appear within w words of each other in a document, then the document is returned by the search engine. Thus, NEAR can find matching documents while allowing certain positions in the text to remain unspecified. We might think of a word in the document text that occurs between NEAR terms as a kind of "wildcard match".

BE's variabilized queries are more powerful than NEAR for several reasons. First, variabilized queries enforce ordering constraints between the terms in the query, while NEAR only enforces proximity.[3] Second, a NEAR query cannot recover the actual values of its "wildcards". It only determines whether two terms are proximate. In contrast, a key aspect of variabilized queries is to return the *bindings* matching the variables in the query. Finally, BE's variabilized queries can constrain matching variable bindings by type whereas the NEAR operator has no notion of type.

We now describe how variabilized queries are implemented to minimize the number of disk accesses per query.

3. INDEX AND QUERY PROCESSING

The inverted index allows standard queries to be processed very efficiently, even with billions of indexed documents [3]. For every term in the corpus, an inverted index builds a list of every document and position where that term can be found. This enables very fast document-finding at query time.

In this section, we describe why the standard inverted index is insufficient for executing the variabilized queries introduced earlier. We also introduce *neighbor indexing*, a novel addition to the inverted index that can efficiently execute these new queries and still retain the advantages of the inverted index.

3.1 Standard Implementation

Many language-based applications have been forced into a very inefficient implementation of BE's variabilized queries. Such systems operate roughly as follows on the query ("cities such as ⟨*NounPhrase*⟩"):

1. Perform a traditional search engine query to find all URLs containing the non-variable terms (*e.g.*, "cities such as")

2. For each such URL:

   (a) obtain the document contents,
   (b) find the searched-for terms ("cities such as") in the document text,
   (c) run the noun phrase recognizer to determine whether text following "cities such as" satisfies the type requirement,
   (d) and if so, return the string

We can divide the algorithm into two stages: obtaining the list of URLs, and then processing them to find the ⟨*NounPhrase*⟩ bindings.

The first stage is a lookup using a standard inverted index. As described in [3], processing a query consists of retrieving a sorted document list for each query term, and then stepping through them in parallel to find the intersection set.

---

[3]Of course, if only proximity is desired, then the NEAR operator can be added to the BE query language.
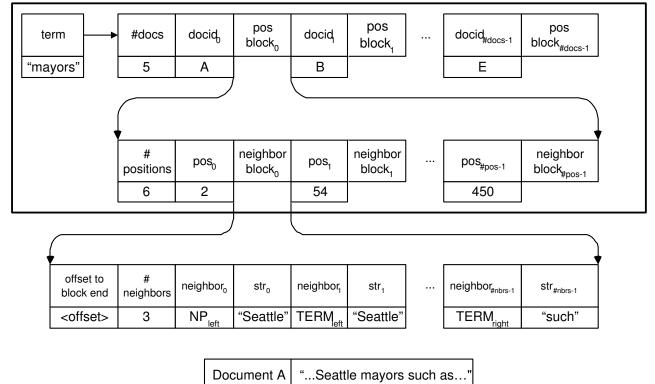
STANDARD INVERTED INDEX

term		#docs	docid$_0$	pos block$_0$	docid$_1$	pos block$_1$	...	docid$_{\#docs-1}$	pos block$_{\#docs-1}$
"mayors"		5	A		B			E	

# positions	pos$_0$	neighbor block$_0$	pos$_1$	neighbor block$_1$	...	pos$_{\#pos-1}$	neighbor block$_{\#pos-1}$
6	2		54			450	

offset to block end	# neighbors	neighbor$_0$	str$_0$	neighbor$_1$	str$_1$	...	neighbor$_{\#nbrs-1}$	str$_{\#nbrs-1}$
<offset>	3	NP$_{left}$	"Seattle"	TERM$_{left}$	"Seattle"		TERM$_{right}$	"such"

Document A	"...Seattle mayors such as…"

Figure 2: Detailed structure for a single term's list in the BE index. The top two levels, enclosed within the bold line, consist of document information present in a standard inverted index. The BE index adds information for every (document, position) pair. This additional structure holds all the neighbors for the (document, position) in question. The neighbor set consists of the typed strings immediately to the left and right of that position. Reading from left to right, the neighbor index structure adds: 1) an offset to the end of the block, so irrelevant instances can be easily skipped over; 2) the number of neighbors at that location; 3) a series of "neighbor/string" pairs. The "neighbor" value identifies the type and whether it's to the left or right. The "string" is the available binding at that location.

For phrase queries we also examine positions within each document, to ensure the words appear sequentially.

Because the system reads each document list straight from start to finish, each list can be arranged on disk as a single stream. Thus the system will require no time-consuming random disk seeks to step through a single term's list. Disk prefetching will also be more helpful. It might be possible for large search installations to keep a substantial portion of the index in memory, in which case the system can avoid even sequential disk reads.

The second stage of the standard algorithm is very slow because fetching each document is likely to result in a random disk seek to read the text. Naturally, this disk access is slow regardless of whether it happens on a locally-cached copy or on a remote document server.

3.2   Neighbor Indexing

In this section we introduce the *neighbor index*, an augmented inverted index structure that retains the advantages of the standard inverted index while allowing access to relevant parts of the corpus text. It is depicted in Figure 2.

The neighbor index retains the structure of an inverted index. For each term in the corpus, the index keeps a list of documents in which the term appears. For each of those documents, the index keeps a list of positions where the term occurs. However, the neighbor index maintains additional data at every position. Each position keeps a list of each adjacent document text string that satisfies one of the target *types*. Each one of these strings we call a *neighbor*. Thus, at each document position there is a left-hand and a right-hand neighbor for each type. As mentioned above, the set of types is determined by a set of type recognizers, applied to the corpus during index construction. Certain types, such as $\langle term \rangle$, may be present at every position in the corpus. Other types, such as $\langle NounPhrase \rangle$, only start or end at certain places in the corpus. A given position's neighbors may include all, some, or none of the types available.

Here is the algorithm used for processing a BE query q: First, break the query q into clauses, separated by logical operators. Each clause c now consists of a set of elements $e_0, e_1, ...e_E$ which are either concrete terms or variables.

The heart of the algorithm is the evaluation of each clause, which proceeds as follows:

1. For each e_i that is a concrete term, create a pointer to the corresponding term lists l_i, initialized to the first document in each list. We refer to the current head document of list l_i as $headdoc(l_i)$.

2. Increment the l_i pointer where $headdoc(l_i)$ is lowest, until $headdoc(l_0) = headdoc(l_1) = ...headdoc(l_q)$ or until one pointer advances to the end of the list. We thus advance all term list pointers to a document in which all non-variable elements e_i appear, or there are no such remaining documents. If one of the lists is exhausted, processing of this clause is complete.

   (a) We can now refer to the head position of list l_i as $headpos(l_i)$. For all concrete terms, advance term list pointer l_i with lowest $headpos(l_i)$ until $headpos(l_0) < headpos(l_1) < ...headpos(l_q)$. If some term list pointer reaches the end of positions, then exit loop and continue to next document.

   (b) There may be some elements e_i that are variables, not concrete terms. For each of these, at least one of e_{i-1} or e_{i+1} is guaranteed to be a concrete term.

   If e_{i-1} is concrete, note that $headpos(l_{i-1})$ is at the start of a neighbor block for e_{i-1} that will contain information about indexed strings to the right of $headpos(l_{i-1})$. Examine the *right-hand* neighbor for the desired $typeof(e_i)$.

   If e_{i+1} is concrete, then $headpos(l_{i+1})$ is at the start of a neighbor block for e_{i+1} that contains information about indexed strings to the left of $headpos(l_{i+1})$. Examine the *left-hand* neighbor for the desired $typeof(e_i)$.

   Find bindings for all variables e_i in this way.

   (c) In an above step, we checked that $headpos(l_0) < headpos(l_1) < ...headpos(l_q)$ for elements with concrete terms. We now check adjacency as well. For any two adjacent concrete terms with indices i and j, check that $headpos(l_i) + 1 = headpos(l_j)$. For adjacent element indices i, j, and k, where i and k are concrete terms and j is a variable, check that $headpos(l_i) + lengthof(e_j) = headpos(l_k)$.

   For a variable element e_i that does not fall between two concrete variables, simply check that $lengthof(e_i)$ is non-zero.

   (d) If the above adjacency test succeeds, then record all query variable bindings, and continue. If any of the adjacency tests fail, then simply continue.

As with the inverted index, a term's list is processed from start to finish, and can be kept on disk as a contiguous piece. The relevant string for a variable binding is included directly in the index. So, there is no need for the disk to seek to fetch the source document.

A neighbor index avoids the need to return to the original corpus, but it can consume a large amount of disk space. Depending on the variable types available, corpus text may be folded into the index several times. To conserve space, we perform simple dictionary-lookup compression of strings in the index.

	Query Time	Index Space
BE	$O(k)$	$O(N * T)$
Standard engine	$O(k + B)$	$O(N)$

Table 1: Query Time and Index Space for the two methods of implementing the BE query language. Query Time is expressed as a function of the number of disk seeks. k is the number of concrete terms in the query, which we expect will never grow beyond a small number. B is the number of bindings found when processing a query, which will grow with the size of the corpus. T is the number of indexed types, and N is the number of documents indexed. Since typical values for B are in the thousands and typical values for k are smaller than 4, BE is much faster than a standard engine. Since typical values for T are small, the space cost is manageable.

The neighbor index reads variable bindings off disk sorted first by the source document ID, and secondarily by position within that document. This ordering is critical for processing intersections between separate term lists. However, document ID ordering is probably unhelpful for most applications. So after BE finds all the available bindings, it sorts them before returning the query results.

BE has a general facility for defining sorting functions over bindings. Our BE implementation can sort bindings in ascending alphanumeric order or by frequency of appearance (*e.g.*, so Los Angeles will be sorted higher than Annandale, VA). However, there are many other reasonable sort orders. For example, bindings could be sorted according to a weight that indicates how "trustworthy" the source document is. BE allows sorting by any arbitrary criterion.

Finally, to support statistical NLP applications such as PMI-IR, BE can associate a "hit count" with each binding it returns. The hit count records the number of times that the particular binding appeared in BE's index. This is an important capability as discussed in Section 5.

3.3 Asymptotic Analysis

This section provides an asymptotic analysis of BE's behavior as compared to the "Standard Implementation" that is in use today. Query Time is expressed as a function of the number of random disk seeks, as these dominate all other processing times. Index Space is simply the number of bytes needed to store the index (not including the corpus).

Table 1 shows that BE requires only $O(k)$ random disk seeks to process queries with an arbitrary number of variables whereas a standard engine takes $O(k + B)$. Thus, BE's performance is the same as that of a standard search engine for queries containing only concrete terms. For variabilized queries, where we expect B to be large, BE is much faster. BE embodies a time-space tradeoff. The size of its index is $O(N * T)$ where N is the number of documents in the index and T is the number of variable types. In contrast, the size of the standard inverted index is $O(N)$.

In typical Web applications, we expect N to be in the billions and T to be smaller than 10. Moreover, we expect the index size to increase sub-linearly in T because elements of each type only occur for a fraction of the terms in the index. Note that atomic parts of speech such as noun and verb are mutually exclusive, so tagging terms with any number of

such types can at most double the index. Finally, semantic types such as *zip code* are rare and will only add a small space overhead.

The BE neighbor index shows its strength in the query time analysis. The only seeks needed are to find the term lists of the k concrete query terms. In contrast, the Standard Implementation first seeks k times to perform its inverted index lookup, and then fetches a document from disk for each of B bindings.[4]

BE does incur higher storage costs than the standard method. Both the standard method's inverted index and the neighbor index will grow linearly in size with the number of indexed documents. However, BE will also grow with the number of indexed types; each additional type increases the space to index a single document.

3.4 Implementation

The BE search engine draws heavily upon code from the Lucene and Nutch open source projects. Lucene is a program that produces inverted search indices over documents. Nutch is a search engine (including page database, crawler, scorer, and other components) that uses Lucene as its indexer. Like BE, Lucene and Nutch are written in Java.

Our type recognizer uses an optimized version of the Brill tagger [6] to assign part-of-speech tags, and identifies noun phrases using regular expressions based on those tags.

4. EXPERIMENTAL RESULTS

This section experimentally evaluates both the benefits and costs of the BE search engine. Much of BE's source code comes from the Nutch project, but a separate, unchanged Nutch instance also serves as a benchmark standard search engine for our experiments. The "Nutch" experiments described below refer to the Standard Implementation from Section 3.1 using this traditional Nutch index.

All of our Nutch and BE experiments were carried out on a corpus of 50 million web pages downloaded in late August of 2004. We ran all query processing and indexing on a cluster of 20 dual-Xeon machines, each with two local 140 Gb disks and 4 Gb of RAM. We used the corpus to compute both a BE index and a regular Nutch index.

Using a local Nutch instance instead of a commercial Web search engine allows us to control for network latency, machine configuration, and the corpus size. We set all configuration values to be exactly the same for both Nutch and BE.

Finally, we ran a test of a full-fledged information extraction system that uses BEstyle queries, comparing the Standard Implementation using the Google API versus BE.

4.1 Benefit at Query Time

We recorded the query processing time for 150 different queries using both BE and Nutch. We generated these queries by taking various patterns (*e.g.*, "X such as $\langle NounPhrase \rangle$," "$\langle NounPhrase \rangle$ is a X") and instantiating each with a set of classes (*e.g.*, "cities", "countries", and "films"). For each query, we measured the time necessary to find all bindings in the corpus. Both BE and Nutch queries
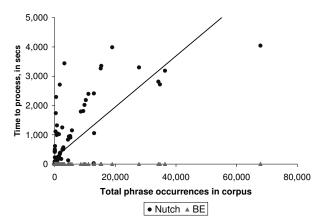
---

[4]In fact, there can be multiple bindings per document, so it is sometimes possible to use a single seek for multiple B. But we assume that bindings are evenly distributed over the corpus, so in general the number of seeks grows with B.

Figure 3: **Processing times for 150 queries, using the** BE **search engine and the Standard Implementation on Nutch. Both use the same corpus size and are hosted locally. While Nutch processing times range from 10.7 to 4044.4 seconds,** BE **times range from only 0.03 to 20.1 seconds. The straight line is the linear trend for all Nutch extraction times.** BE**'s speedup ranges from a factor of 202 to a factor of 369.**

were distributed evenly over all 20 machines in our cluster. We waited for each system to return all answers to a query before submitting the next.

Figure 3 plots the number of times the query phrase appears in the corpus versus the time required for processing the query. It shows a very large improvement for BE. A single query took between 0.3 and 20.1 seconds with BE, while Nutch took between 10.7 and 4044.4 seconds.[5] Processing time is a function of the number of times the query appears in the corpus. BE's speedup ranges from a factor of 202 to a factor of 369 for the queries in our experiments. The speedup would be correspondingly greater for queries that returned additional matches due to a larger index. Thus, for a billion-page index, we would expect speedups of three to almost four orders of magnitude.

In the Nutch case, we did not include any time spent in post-retrieval processing to recognize particular types. Yet one of the benefits of BE is moving type recognition to indexing time. Thus, the measurements in Figure 3 understate the benefit of BE. The inclusion of type recognition time would increase the Nutch query processing time by an average of 16.2%, making the BE speedup even greater.

In addition to testing Nutch and BE with the queries above, we ran a full-fledged information extraction test using the KnowItAll system introduced in [10]. KnowItAll is a natural "consumer" of BE's power because it is designed to be a high throughput extraction system, but it routinely exhausts the 100,000 daily queries allotted to it by the Google API. We created two versions of KnowItAll — one that uses the Standard Implementation on the Google API, and one that uses BE.

---

[5]All our measurements are in seconds of real time.

Num. Extractions	Google	BE
10,000	5,976 secs	95 secs (63x speedup)
50,000	29,880 secs	95 secs (314x speedup)
150,000	89,641 secs	N/A

Table 2: Time needed to find KnowItAll fact extractions (a mixture of city, actor, and film titles) using the Standard Implementation on the Google API versus BE. The BE column is constant at 95 seconds of real time because BE always returns every single binding that matches its query.
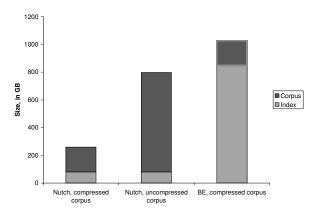

Figure 4: Comparison of storage requirements for a 50M page index. The Nutch index is a standard inverted index with document and position information. The BE index includes types ⟨*term*⟩ for every word in the corpus, and ⟨*NounPhrase*⟩ whenever such structures are present in the text. The traditional index plus uncompressed text is presented as a point of reference.

BE's impact on KnowItAll's speed is shown in Table 2. We see that when relying on Google, KnowItAll's processing time is dominated by the large number of queries required, and the "courtesy waits" in between queries. Furthermore, KnowItAll's processing time increases (roughly) linearly with the number of extractions it finds. In contrast, when KnowItAll uses BE it is from 63 to 314 times faster, and that speedup increases as the number of extractions grows. BE was not able to return 150,000 extractions due to the limited BE index size in our experiments (50 million pages). However, as the analysis in Section 1 shows, we expect that the speedup will increase linearly with larger BE index sizes.

4.2 Costs

The BE engine trades massive speedup at query time for an increase in the time and space costs incurred at indexing time. This section measures these costs, and argues that they are manageable.

Figure 4 shows that roughly 80 Gb are necessary to hold the Nutch index for the 50 million Web pages in our corpus,
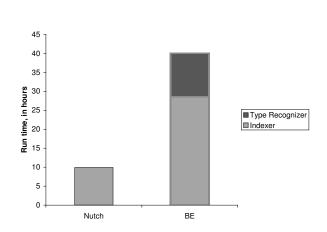

Figure 5: Time necessary to compute the Nutch index vs. time to compute the BE index. In both cases, we distributed the index computation over the same cluster of 20 machines.

and another 180 Gb are necessary to store the compressed Nutch corpus. Both are necessary for Nutch, because we need to find the documents relevant to a query, and then examine other non-query text from those documents. Storing the corpus locally obviates re-downloading pages. Thus, the total storage necessary to run Nutch is 260 Gb. The space necessary to hold the corresponding BE index is 847 Gb — roughly a factor of three more. BE does not require a copy of the corpus because any document text that might be useful as a binding has already been incorporated into the neighbor index. However, the compressed corpus can still be useful for traditional search tasks, such as generating query-sensitive document summaries, so we include it in the BE column in Figure 4.

The measurements reported in Figure 4 are for the types ⟨*term*⟩ and ⟨*NounPhrase*⟩. Adding two additional types would, in the worst case, double the amount of space required by BE. In fact, the expected increase in space is smaller for two reasons. First, a substantial fraction of the space cost for BE is the conventional inverted index component, which is fixed as we add new types. Second, a new type such as ⟨*verb*⟩ or ⟨*adjective*⟩ would cause BE to store smaller objects than ⟨*Noun Phrase*⟩ and rarer objects than ⟨*term*⟩. Thus, we believe the addition of these types would result in less than a 30% increase in storage requirements in practice.

For comparison's sake, we also present the uncompressed corpus size. The predicted size of a BE index depends on many factors, such as how many types are indexed, how frequently each type appears, and the effectiveness of the dictionary-lookup compression scheme. However, it should scale roughly with the amount of corpus text. (*e.g.*, our example includes the type *term*, which adds at least one entry to the BE index for every term in the corpus.) In essence, we think of the neighbor index as a method for rearranging corpus text so that it is amenable to the extraction of bind-

Figure 6: Most-frequently-seen extractions for query "insects". The score for each extraction is the the number of times it was retrieved over several BE extraction phrases.

ings for query variables. Thus, it is not surprising that its size is roughly that of the corpus.

Figure 5 shows the time needed to compute the index. The time is broken down into two components: the time to run type recognizers, and the time to build the index. Again, we included the types $\langle term \rangle$ and $\langle NounPhrase \rangle$. Recognizer time includes the time to run Brill's tagger and check for regular expressions over those tags. Different type recognizers may take varying amounts of time to execute, but since each recognizer makes a single pass over the corpus, total recognizer overhead at index time should be linear in the number of documents.

Overall, our measurements provide evidence for the following conclusion: if designing a search engine to support information extraction, BE offers the potential for substantial speedup at query time in exchange for a modest overhead in space and index-construction time. Moreover, as argued in Section 5, we believe that this conclusion holds for a broad set of NLP applications.

5. APPLICATIONS

The previous section showed how an information extraction system such as KnowItAll can leverage BE. This section sketches additional applications to illustrate the broad applicability of BE's capabilities.

5.1 Interactive Information Extraction

We have configured a BE application to support interactive information extraction in response to simple user queries. For example, in response to the user query "insects", the application returns the results shown in Figure 6. The application generates this list by using the query term to instantiate a set of generic extraction phrase queries such as "insects such as $\langle NounPhrase \rangle$". In effect, the application is doing a kind of query expansion to enable naive users to

extract information. In an effort to find high-quality extractions, we sort the list by the hit count for each binding, summed over all the queries.

This kind of querying is not limited to single terms. For example, the binary-relation query "cities, capital" yields the extractions shown in Figure 7. The application generates the list as follows: first, "cities" is used to generate a comprehensive list of cities, as we did with "insects". Second, the query term "capital" is used to instantiate a set of generic extraction patterns, *e.g.*, "$\langle NounPhrase \rangle$ is the capital of $\langle NounPhrase \rangle$". Third, BE is queried with the instantiated patterns. Finally, the application receives the set of binding pairs, removing any pairs where the the first $\langle NounPhrase \rangle$ is not a highly-scored member of the "cities" list. We then choose the most-frequently-seen "capital" binding for each city, and sort the overall list by the number of times that binding was found.

This kind of interactive extraction is similar to Web-based Question-answering systems such as Mulder [14] and Ask-MSR [7]. The key difference is the much larger volume of information that BE returns in response to simple keyword queries. Large-scale information extraction system already exist on the Web (*e.g.*, Froogle) but they are domain specific, and the extraction occurs off-line which limits the set of queries that such systems can support.

The key difference between this BE application and domain-independent information extraction systems such as Know-ItAll is that BE enables extraction at interactive speeds — the average time to expand and respond to a user query is between 1 and 45 seconds. With additional optimization, we believe we can reduce that time to 5 seconds or less.

5.2 PMI-IR

Turney's PMI-IR scores [21] are widely used for such tasks as finding words' semantic orientation, synonym-finding, and

Figure 7: Most-frequently-seen extractions for the query "cities, capital". We show each city only once, picking the most-frequently-seen binding for its "capital" slot. The score for each extraction is the number of times the capital relation was extracted. The query does not require the second extracted object to be a country. Hence, the Chinese provincial capital "Nanjing" can appear in result 3.

antonym-finding. KnowItAll also uses them for assessing the quality of extracted information [10].

It is often useful to compute a very large number of PMI-IR scores. For example, we may want to assess a PMI-IR score for every possible extraction from a given corpus.

To compute a PMI-IR score for the "cities such as" extraction phrase, we need to solve

$$\frac{numHits(\text{``cities such as } X\text{''})}{numHits(\text{``}X\text{''})}$$

for each of the n values for X. For a single extraction phrase, this will require $2n$ hit count queries from a traditional search engine. Yet, with a small amount of additional work, BE can compute the same values with just a single query.

We can compute the numerators for every X in the corpus by issuing a BE query, then counting how many times each unique value is returned. Since BE has pre-defined the string types of interest at index-construction time, it can also compute a *denominator list* for every type. This is simply a list of every unique typed string (say, $\langle Noun\ Phrase \rangle$) found in the neighbor index, followed by the number of times the string appears. The denominator list may be large, and like the neighbor index grows with both the number of corpus documents and the number of types. However, it can never be larger than a fraction of the neighborhood index itself, which includes left- and right-hand copies of each typed string. Also, the denominator list is likely to be more amenable to compression methods such as front-coding.

During PMI-IR query processing, the BE engine takes its standard results to compile an alphanumerically-sorted list of all bindings, along with a hit count for each. It then intersects this list with the denominator list, generating a new PMI-IR score every time a string can be found in both lists. This can execute as quickly as a single linear pass through the denominator list.

Most of the work in constructing the denominator list is a side-effect of constructing the neighborhood index. Whenever a type recognizer finds a string, we add it to a special denominator list file as well as the neighborhood index. After the neighborhood index is complete, the denominator list file is sorted alphanumerically. We then count adjacent identical items, merging them and adding the count value.

6. FUTURE WORK

Our plans for BE center around making it more suitable for an interactive-speed information extraction system. We plan to study extraction ranking schemes in more depth, see whether we can use extractions to improve a traditional search engine, and finally to improve query execution speed.

Ranking a simple list of documents is a well-studied problem for search engines. But ranking the results of an extraction query is a new and unexamined problem. Consider the list of city and state pairs from Figure 7. Competing criteria include confidence in the extraction, confidence in the extraction's source web pages, and content-specific sorting demands (*e.g.*, by population or geography).

In addition, we might use the BE system to improve standard search engine result ranking. Bindings that are not requested by the query but are present in the index can provide clues as to a document's content. For example, bindings of type $\langle NounPhrase \rangle$ might be useful for clustering search results into subsets.

Since we expect our system to perform more phrase queries

than the average search engine, we will want to optimize these queries whenever possible. We might index pairs of search terms instead of single terms, in an effort to cut down the average list length. Another possibility is some use of the *nextword index*, which is described in further detail in Section 7 [5].

7. RELATED WORK

There has been substantial work in query languages for structured data, but none are wholly appropriate for a search engine. WebSQL and Squeal are database systems that create special schema for querying the set of web objects [17, 20]. Unlike traditional databases, both consider the web as a fast-changing object that cannot be stored entirely locally (possibly requiring the database system to go to the web to service a query). Like traditional databases, they can process arbitrary SQL-like queries that are defined using their schema. Such queries are not limited to just the text, so they are generally more expressive than BE's queries. (Though since text is not treated in a special way, there is certainly no idea of a text type.) However, even if these database-driven systems offer general text indexing support, they will still suffer from the same poor performance as a general-purpose search engine.

The LAPIS system contains a sophisticated algebra for defining text regions in a document [18]. Users define text regions according to their physical relationship to other tokens or regions in the text stream. In addition, there are named "patterns" which function very loosely like our arbitrary text types. (Though BE text types can be classified by arbitrary code.) The LAPIS and BE query languages are not directly comparable, but there is a wide set of text-adjacency queries that LAPIS can process, while BE cannot.

While LAPIS' query language is powerful, its runtime performance is likely to be quite poor at scale. The LAPIS system was designed for and tested on small sets of documents, on the scale of a few hundred. It has no inverted index structure, and so must investigate every document in order to process a query. The query performance is therefore likely to be even worse than the Standard Implementation, which efficiently finds the relevant document set.

Agrawal and Srikant conducted an interesting study of how to make documents with numerical data more amenable to search engine-style queries [2]. The central idea is that a search query that contains a numerical quantity should elicit documents containing quantities that are numerically similar (but textually distant). The system requires some preprocessing of the corpus beyond the standard inverted index, along with a few extensions to the query algorithm. However, the task is still one of basic document finding. Unlike BE, the engine does not return text regions from the documents, so executing BE-style queries would involve fetching each individual document's text.

The Linguist's Search Engine (http://lse.umiacs.umd.edu/) from Resnik is a tool for searching large corpuses of parse trees [1]. Like BE, it computes an index in advance to allow for fast query processing. Unfortunately, there is not yet much published detail on its precise query syntax or indexing mechanism.

The most related work is in the area of index design. Cho and Rajagopalan build a *multigram index* over a corpus to support fast regular expression matching [9]. The multi-gram index is an inverted index that includes postings for certain non-English character sequences. The query processor finds the relevant sequences from a regular expression query, and uses the inverted index to find documents where they appear. The resulting, much smaller, document set is then examined with a full-power regular expression parser.

Regular expressions can express a number of strings that the BE language cannot, but BE types can be generated from type recognizers that can be far more complex than regular expressions. For the queries we expect to execute, the multigram index seems likely to do well in finding just the set of relevant documents. However, with a standard inverted index, the original documents again still need to be fetched, so performance will be similar to that of the Standard Implementation.

The GuruQA System is a search engine for answering natural language questions [19]. GuruQA first annotates its corpus with extra "words" called QA-Tokens. Each QA-Token indicates the location of a phrase that might be useful for answering a certain kind of question. For example, QA-Tokens might indicate places in the corpus where years, times, person names, *etc.*, appear. GuruQA then computes an inverted index over this annotated corpus, running over the original text as well as the set of QA-Tokens.

When processing a user query, GuruQA examines the question to find what kind of answer the user is probably looking for. By searching for a QA-Token of a certain type, it can then quickly find all occurrences of, say, dates.

The GuruQA "query language" is just natural language, so it is not directly comparable to that of BE. And unlike the neighbor index when treating linguistic types, the GuruQA index does not retain the actual text value for QA-Tokens. GuruQA's query processor must still fetch the original texts, and incur the performance hit that entails. (Of course, GuruQA is not designed to find all relevant documents, like BE does.)

A series of articles describes the *nextword index* [5, 23, 4], a structure designed to speed up phrase queries and to enable some amount of "phrase browsing." It is an inverted index where each term list contains a list of the successor words found in the corpus. Each successor word is followed by position information.

However, the nextword index lacks both expressive power and performance when compared to BE's neighbor index. Given a query phrase, a nextword index can find just the right-hand single-word string. In contrast, a neighbor index can find strings of multiple words at positions to the left, right, and within the query phrase boundaries; further, those strings can be typed. A nextword index processes multi-word query phrases in two serialized stages of index lookups; BE's neighbor index can process multi-word queries without any such serialization, and is thus fully parallelizable. Assuming all index lookups run at equal speed, query time for BE would thus be a factor two smaller on multi-word queries as compared with an engine that utilizes the nextword index.

8. CONCLUSIONS

The Bindings Engine (BE) consists of a generalized query language (containing typed variables and functions), the neighbor index, and an efficient query processing algorithm. Utilizing BE, we reported on a set of experiments that provide evidence for the following conclusions. First, BE yields

two to three orders of magnitude speedup when supporting information extraction. Second, this speedup comes at the cost of only a modest increase in space and in index-construction time. Moreover, Section 3.3 analyzed how BE's performance scales with each of the relevant parameters, and showed that it has the potential for enormous speedups on billion-page indices with only a constant factor space increase for a fixed set of variable types.

Finally, BE can support a broad range of novel language-processing applications. As one example, we have sketched a BE application that extracts large amounts of information from the Web in response to simple user queries, and does so at interactive speeds.

Acknowledgments

9. REFERENCES

[1] Corpus Colossal. *The Economist*, Jan. 2005.

[2] R. Agrawal and R. Srikant. Searching with numbers. In *WWW*, pages 420–431. ACM Press, 2002.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[4] D. Bahle, H. E. Williams, and J. Zobel. Optimised Phrase Querying and Browsing in Text Databases. In M. Oudshoorn, editor, *Proceedings of the Australasian Computer Science Conference*, pages 11–19, Gold Coast, Australia, Jan. 2001.

[5] D. Bahle, H. E. Williams, and J. Zobel. Efficient Phrase Querying with an Auxiliary Index. In *Proceedings of the ACM-SIGIR Conference on Research and Development in Information Retrieval.*, pages 215–221, 2002.

[6] E. Brill. Some Advances in Rule-Based Part of Speech Tagging. In *AAAI*, pages 722–727, 1994.

[7] E. Brill, S. Dumais, and M. Banko. An Analysis of the AskMSR Question-Answering System. In *EMNLP*, 2002.

[8] E. Brill, J. Lin, M. Banko, S. T. Dumais, and A. Y. Ng. Data-Intensive Question Answering. In *TREC 2001 Proceedings*, 2001.

[9] J. Cho and S. Rajagopalan. A Fast Regular Expression Indexing Engine. In *Proceedings of the 18th International Conference on Data Engineering*, 2002.

[10] O. Etzioni, M. Cafarella, D. Downey, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Web-scale Information Extraction in KnowItAll. In *Proceedings of the 13th International World-Wide Web Conference*, 2004.

[11] O. Etzioni, M. Cafarella, D. Downey, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Unsupervised Named-Entity Extraction from the Web: An Experimental Study. *Artificial Intelligence*, 2005.

[12] A. Y. Halevy and J. Madhavan. Corpus-Based Knowledge Representation. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1567–1572, 2003.

[13] M. Hearst. Automatic Acquisition of Hyponyms from Large Text Corpora. In *Proceedings of the 14th International Conference on Computational Linguistics*, pages 539–545, 1992.

[14] C. C. T. Kwok, O. Etzioni, and D. S. Weld. Scaling Question Answering to the Web. In *Proceedings of the 10th International World-Wide Web Conference*, pages 150–161, 2001.

[15] J. Madhavan, P. Bernstein, A. Doan, and A. Halevy. Corpus-based Schema Matching. In *Proceedings of the International Conference on Data Engineering, Tokyo, Japan*, 2005.

[16] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[17] A. O. Mendelzon, G. A. Mihalia, and T. Milo. Querying the World Wide Web. *International Journal on Digital Libraries*, 1996.

[18] R. C. Miller and B. C. Myers. Lightweight Structured Text Processing. In *Proceedings of 1999 USENIX Annual Technical Conference*, pages 131–144, Monterey, CA, 1999.

[19] J. Prager, E. Brown, A. Coden, and D. Radev. Question-Answering by Predictive Annotation. In *23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 184–191, 2000.

[20] E. Spertus and L. A. Stein. Squeal: A Structured Query Language for the Web. In *Proceedings of the 9th International World Wide Web Conference (WWW9)*, pages 95–103, 2000.

[21] P. D. Turney. Mining the Web for Synonyms: PMI-IR versus LSA on TOEFL. In *Proceedings of the Twelfth European Conference on Machine Learning*, 2001.

[22] P. D. Turney. Thumbs Up or Thumbs Down? Semantic Orientation Applied to Unsupervised Classification of Reviews. In *ACL*, pages 417–424, 2002.

[23] H. E. Williams, J. Zobel, and P. Anderson. What's Next? Index Structures for Efficient Phrase Querying. In J. Roddick, editor, *Proceedings on the Australasian Database Conference*, pages 141–152, Auckland, New Zealand, 1999.