Relational Web Search

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

Michele Banko
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195 U.S.A
banko@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

Facts are naturally organized in terms of entities, classes, and their relationships as in an entity-relationship diagram or a semantic network. Search engines have eschewed such structures because, in the past, their creation and processing have not been practical at Web scale.

This paper introduces the *extraction graph*, a textual approximation to an entity-relationship graph, which is automatically extracted from Web pages. The extraction graph is an intermediate representation that is more informative than a mere page-hyperlink graph but far easier to construct than a semantic network. The paper also introduces TextRunner, a search engine that utilizes this representation to answer complex relational queries that are difficult to answer using today's search engines or Web Information Extraction (IE) systems.

The paper compares TextRunner to a state-of-the-art IE system on list searches, and finds that TextRunner is 40% more precise, with 11% better recall than the IE system. Our experiments, computed over a 90-million page corpus and a 227-million node extraction graph, show how TextRunner will scale to billions of pages.

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, corpus, information extraction, structure, relation

1. INTRODUCTION

Even though modern search engines analyze hyperlinks and anchor text, they remain keyword based. A page is retrieved only when it (or the corresponding anchor text) contains the keywords in the user's query. This paper introduces *relational Web search*—search through a textual approximation to an entity-relationship graph that is automatically extracted from Web pages. This *extraction graph* is a novel intermediate representation that is more informative than a mere page-hyperlink graph but far easier to construct than a true entity-relationship diagram or a semantic network [25].

Our extraction graph explicitly represents information that is not directly available from a standard inverted index. First, entities that are closely related tend to be close in the graph. For example, the path between *"Einstein"* and *"the Theory of Relativity"* is likely to be of length one. Second, similar entities tend to have similar relationships as their edges. For example, *"Einstein"* and *"Newton"* share the relationships *"discovered"* and *"is a physicist"*. These two properties make the extraction graph invaluable for relational Web search as explained in Section 3.2.2.

Relational Web search facilitates answering several different types of queries that are laborious using today's engines:

- **qualified-list queries:** retrieve a list of objects that share multiple properties (*e.g.*, west coast liberal arts college).
- **unnamed-item queries:** qualified-list queries that aim to locate a single object whose name the user does not know or cannot recall (*e.g.*, the tallest inactive volcano in Africa).
- **relationship queries:** find the relationship(s) between two objects (*e.g.*, the relationship between Bill Clinton and Justice Ginsberg).
- **tabular queries:** find a set of objects annotated by their salient properties (*e.g.*, inventions annotated by their inventor and year of announcement).

Today, answering such queries often requires substantial effort on the user's part. In unnamed-item queries, for example, the user has a target object in mind, but cannot name it directly. Instead, the user has to describe the target indirectly, but her search will fail if she does not choose the appropriate descriptive phrases. Furthermore, the different properties of the object may be scattered over several pages and the search engine will come up empty. Consider a set of three documents, one of which lists *"Oppenheimer"* as a *"physicist,"* another as an *"American scientist."*, and another that describes him having something to do with *"Berkeley."* A standard search engine cannot integrate the information in all three; the user must piece together the answer by locating and reading the relevant web pages herself.

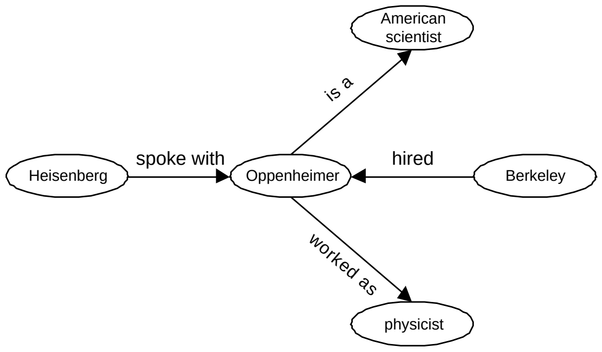

Figure 1: A portion of an extraction graph. Not all edges for these nodes are shown. An edge points from the first node in a triple toward the second.

Our search engine, called TextRunner, utilizes a standard crawler to retrieve Web pages, but applies an information extraction mechanism (see Section 3.1) to each sentence on each page. On most sentences, the mechanism yields at least one *triple* that consists of two "object" strings and a "predicate" string that links them. The triples are automatically inserted into a massive extraction graph whose nodes are object strings and whose edges are predicate strings.[1]

Given the sentence "Berkeley hired Oppenheimer soon after Oppenheimer became a physicist.", for example, TextRunner extracts the following triples:

- ("Berkeley", "hired", "Oppenheimer")
- ("Oppenheimer", "became a", "physicist")

Figure 1 shows part of an extraction graph that contains these triples as well as several other triples extracted from distinct sentences on separate pages, illustrating how the graph naturally synthesizes information that is scattered across multiple Web pages.[2] The figure does not show occurrence counts for each triple, which the extraction graph also maintains.

Of course, all the problems associated with integrating information from disparate sources surface in the extraction graph. The same entities and predicates are referred to by multiple names (*e.g.*, *"Einstein"* and *"Albert Einstein"*) and combining information from different sentences yields numerous inconsistencies and contradictions. While there are straight forward heuristics for mitigating these problems, they cannot be fully solved. Instead, TextRunner shows how to utilize the graph, despite its inherent noise, to carry out relational search effectively.

Once it has crawled the Web and produced an extraction graph, our engine is ready to process keyword queries from users. For example, the user enters a query such as *"physicist at Berkeley"* and receives a ranked list of objects as output. This novel approach raises a host of questions including:

- How is the extraction graph constructed?
- How does TextRunner search the extraction graph to answer queries?
- How effective is TextRunner at answering different types of queries, and how does it compare with state-of-the-art technology?
- How efficiently can TextRunner represent and search the extraction graph, and does its architecture scale to the billions of pages on the Web?

[1]For brevity, we will refer to these strings as objects and predicates throughout this paper.

[2]The graph can also utilize triples generated by structured data sources. For example, there is an immediate mapping from the RDF triples in N3 [1] into ours.

This paper addresses the above questions, and makes the following contributions:

- We introduce and analyze the extraction graph, a novel representation that supports relational Web search, and report on its implementation in TextRunner. TextRunner is the first relational search engine deployed on the Web.
- We measure TextRunner's performance on qualified-list queries, and compare its performance with both a state-of-the-art information extraction system.
- We analyze TextRunner's architecture and algorithm to show that it can scale to billions of pages with reasonable hardware requirements.

The remainder of this paper is organized as follows. Section 2 discusses previous work, followed by a detailed description of TextRunner in Section 3. Section 4 reports on our experiments. The paper concludes with directions for future work in Section 5.

2. PREVIOUS WORK

This section contrasts relational Web search, as embodied in TextRunner, with three areas of related work: "associative retrieval" and spreading activation, question-answering systems, and information extraction systems.

Relational Web search is a type of *associative retrieval* [27] where the associated items refer to entities and their relationships as expressed on Web pages. Relational Web search goes beyond previous attempts at associative retrieval for two key reasons. First, its extraction graph is computed automatically from data. As a result, the graph can be computed at scale, and updated automatically as new documents appear on the web, without manual knowledge engineering or human feedback. Second, its spreading activation algorithm is designed to achieve Web scale as well.

In contrast, previous work on associative retrieval was often limited to small document sets [24] or limited domains [18]. Crestani [7] provides an overview of several systems that apply spreading activation over semantic networks to the task of information retrieval. Hendler [17] also used a marker-passing algorithm which made it possible to perform inference in parallel over semantic networks. However, the creation of semantic networks requires large amounts of domain-specific, manual knowledge engineering, which severely curtails their scope (as noted by Crestani in [7]).

Some recent systems have utilized hypertext links between documents [8] and anchor text [9] as the basis for associative retrieval. In both cases, this information was observed to improve the precision of search results. However, activation was spread over a document graph as opposed to over a relational one as in TextRunner. Furthermore, these systems did not attempt to achieve Web scale.

The qualified-list and unnamed-item queries enabled by relational search are similar to those handled by Question Answering (QA) and Information Extraction (IE) systems, which also return exact answers, or sets of answers, in response to queries. Below, we consider both QA and IE systems in turn.

Several years ago, the TREC conference instituted the List track, in which systems are asked to return multiple instances of a class (e.g. "List the names of chewing gums."). TREC systems operate over relatively small document collections, and typically rely on heavy linguistic or knowledge-based machinery, which limits their scope.

Systems like Mulder [20], AskMSR [11], and commercial systems such as `ask.com` perform question-answering on the Web, but are limited to performing simple linguistic transformations on a question and then extracting the answer from a single matching sentence. These heuristics are surprisingly effective, but they are limited in scope, and fail to match TextRunner's ability to synthesize information from multiple documents.

KnowItAll [12] was the first published system to carry out unsupervised, domain-independent, large-scale extraction from Web pages [13]. KnowItAll utilizes a domain-independent set of extraction patterns. For a given relation, these generic patterns are then used to automatically instantiate class-specific extraction rules. From this basic set of extraction patterns, KnowItAll goes on to learn domain-specific extraction rules. The rules are applied to Web pages, identified via search-engine queries, and the resulting extractions are assigned a probability using mutual-information measures derived from search engine hit counts.

KnowItAll does well at extracting instances of simple object classes such as *"physicists"* (16.7 million hits [3]) but is likely to have difficulty with qualified-list queries, where objects are constrained by multiple attributes such as *"theoretical physicists"* (169,000 hits) or *"American theoretical physicists"* (212 hits). As data becomes sparse, even over billions of documents, it becomes necessary to look beyond a single sentence, or even a single document, for information. This is where TextRunner's ability to synthesize information collected from multiple documents via its extraction graph becomes essential.

Another limitation of KnowItAll is its reliance on data indexed by commercial search engines. KnowItAll issues millions of queries to search engines, thereby limiting its speed and forcing the extraction process to extend over a period of days or even weeks. While these scalability issues are addressed in the KnowItNow system [5], the need to overcome data sparsity remains a lingering issue for both systems.

3. THE TEXTRUNNER SYSTEM

TextRunner is our implementation of a relational Web search engine. Input to TextRunner consists of a set of query terms, just as with a traditional search engine. TextRunner's output depends on the type of search that is invoked. For qualified-list and unnamed-item queries, TextRunner returns one or more objects that satisfy the properties in the query. For tabular queries, TextRunner returns a table as shown in Table **??**. Finally, for relationship queries, TextRunner returns the predicate(s) that relate the objects in the query.

[3]According to http://www.google.com

TextRunner consists of a mechanism to extract and index an extraction graph, a method for scalable associative retrieval, and a clustering algorithm for post-processing retrieved nodes to improve their ranking. In the following sections, we describe each component in more detail.

3.1 The Extraction Graph

TextRunner uses a structure we call the extraction graph. The extraction graph consists of labeled nodes that are connected by labeled directed edges. A node n_i corresponds to an object, and a directed edge $e_{i,j}$ represents the relationship between nodes n_i and n_j. Edges can be of a particular *type* which indicates a special kind of relationship, for example, *Is-A* to reflect a hypernym relationship.

Each triple $T = (n_i, e_{i,j}, n_j)$ represents a fact-like statement extracted from the document corpus, and is annotated with the following information:

- The type of edge $e_{i,j}$.
- The number of times T is extracted from the corpus.
- The set of sentences and URLs from which we extracted occurrences of T.

The extraction graph bears some resemblance to semantic network models, but there are several important differences:

- The extraction graph is automatically generated from text.
- Nodes and edges in the extraction graph are strings, and not assumed to be authoritative by themselves. For example, many real-world objects might appear under several different labels, e.g. *Einstein* and *Albert Einstein*. Relationships may be represented by more than one string, e.g. *"invented"*, *", who invented"*, rather than by a canonical form, *invented(X, Y)*.
- We make no strong claims about the semantics of the extraction graph. The extractor can make errors, which lead spurious nodes and misleading edges to appear in the graph.
- We do not attempt to resolve inconsistencies in the graph (*e.g.*, *Einstein* is listed as having died in several different years).

Section 4 shows that despite noise in the extraction graph, it contains enough good information to produce useful answers to queries, which are difficult to answer using today's search engines. Moreover, as discussed in Section 5, we anticipate reducing the noise in the extraction graph in future work.

3.1.1 *Implementation and Design*

The TextRunner extraction graph currently contains two types of edges. An *Is-A* edge type indicates the existence of a hypernym relation between two objects. These edges are constructed automatically using an adaptation of Hearst's lexical patterns for locating hypernyms [15]. For instance, we might learn that (*Oppenheimer, Is-A, scientist*) after seeing the text *"... scientists such as Oppenheimer ..."*. (These are the same rules used by the KnowItAll and KnowItNow systems. TextRunner's *Is-A* edges contain all the relationships found by KnowItNow on the same corpus.)

A *predicate* edge type indicates other relations of interest between two nodes in the extraction graph. One way to obtain such edges is to take the entire string between the two entities of interest. Not surprisingly, this permissive approach captures an excess of extraneous and incoherent information. At the other extreme, a strict approach that simply looks for a verb between two nouns causes us to lose other links of importance, such as those that specify noun or attribute-centric properties, e.g. (*Oppenheimer, professor of, theoretical physics*) and (*technical schools, similar to, colleges*). A purely verb-centric method may also extract incomplete or incorrect relationships, for example, (*Berkeley, located, Bay Area*) instead of (*Berkeley, located in, Bay Area*). Our approach, which we now describe, falls in somewhere in between.

To locate predicate edges within a given sentence, we automatically tag each word in the sentence with its most probable part-of-speech. Using these tags, nodes are found by identifying noun phrases using a lightweight noun phrase chunker. For any two noun phrases that are less than a certain number of words apart, we assess the amount of *content* between them, using part-of-speech, lexical and proximity features. Strings having low-content are those with a high amount of punctuation and/or stop words, and will generally be longer in length. Strings with high-content generally contain verbs and prepositions; they tend to be shorter in length, as they express more general relationships. In particular contexts, a pattern centered around a common noun phrase may express the kind of relationship we would like to have in the extraction graph such as in the triple (*Oppenheimer, professor of, theoretical physics*). Therefore, when examining a link between two objects, we sometimes expand the link to include the second object, thus drawing the relationship to now exist between the first object, and the next entity in line. If after processing, we judge a link to have sufficiently high content, it becomes an edge in the extraction graph.

We aim to process on the order of billions of documents on the web, therefore mechanisms for entity-relation extraction must be efficient and independent of both domain and language. Compared to other systems which extract relationships between entities [14, 21], the machinery underlying the TextRunner extraction graph is quite simple. Other approaches to entity-relationship extraction use dependency parsers to label edges with descriptions corresponding to surface syntax relationships such as *subject* or *modifier*. When we want to perform extraction from billions of web pages, the use of such heavy linguistic machinery becomes problematic for several reasons.

Even if state-of-the-art parsers run fast enough at an average of 3 seconds per sentence [19], the performance of the most accurate systems, which are trained from labeled data, degrades when extended across a variety of genres [16]. Trainable parsers have been repeatedly shown to achieve high levels of accuracy within a restricted domain such as financial newswire data [2, 19], yet not much attention has been focused on extending such models to be able to handle noisy language such as that contained on web pages. The use of dependency parsers becomes even more problematic when we wish to process documents in languages other than English and Chinese, the two languages which have received the most attention in the natural language parsing community.

Our system uses only part-of-speech tag and noun-phrase information, both of which can be modeled with high accuracy across domains and languages via a small amount of training data or a human rule-writer [3, 22]. Our system uses such components contained within the OpenNLP toolkit, a freely available suite of natural language tools built upon the application of trainable maximum-entropy models of sentence-boundary detection, part-of-speech tagging and noun-phrase finding [26].

3.1.2 Analysis

In order to assess the performance of the TextRunner entity-relation extractor, we measured the rate and speed of extraction, as well as triple precision.

We estimated the extraction rate of TextRunner and found that by using a set of eight simple extraction patterns, TextRunner extracts an *Is-A* link approximately once every 40 sentences. For every 10 sentences TextRunner reads, it extracts about 7.7 predicate edges.

At a processing rate of 1 document every 0.125 seconds, TextRunner is able to process 1 million documents per machine every 35 hours. In order to limit the amount of processing time we spend running part-of-speech and noun-phrase taggers, we do not process sentences longer than 35 words.

We measured the precision of our triple extractor from a sample of approximately 1000 sentences that yielded 43 *Is-A* edges and 610 predicate edges. A human assessor was asked to look at a triple in the context of the original sentence, and judge it according to the following criteria:

- The well-formedness of the entity nodes. (*New York and, located in, U.S.*), and (*burner, is DVD-R /, RW compatible*) are examples of "bad" extractions.
- The well-formedness of the predicate link. In the triple, (*scientists, talked about how, solutions*), the predicate string is incorrect, rendering it a "bad" extraction.
- The extent to which the link is anchored by the correct nodes. (*desk, contains, object*) would be considered a "bad" triple if extracted from the text, *"The box next to the desk contains an object from Norway."*

We note that extracted triples reflect the content of Web sentences text; we make no claims as to their actual veracity. Thus, a triple is considered to be "good" if someone has written down a sentence from which TextRunner can successfully extract a triple, even if that triple reflects an opinion or a patently false statement. The assessor considered 45.8% of triples extracted to be "good" extractions. By edge type, we found that 71.7% of *Is-A* edges and 44.0% of predicate edges were judged correct.

The next section shows how spreading activation and redundancy help TextRunner to cope with noise in the extraction graph.

3.2 Spreading Activation Search

TextRunner accepts a standard search query as input. As output, it emits a ranked list R of nodes in the extraction graph. This section explains how TextRunner utilizes *spreading activation* to answer queries. Spreading activation

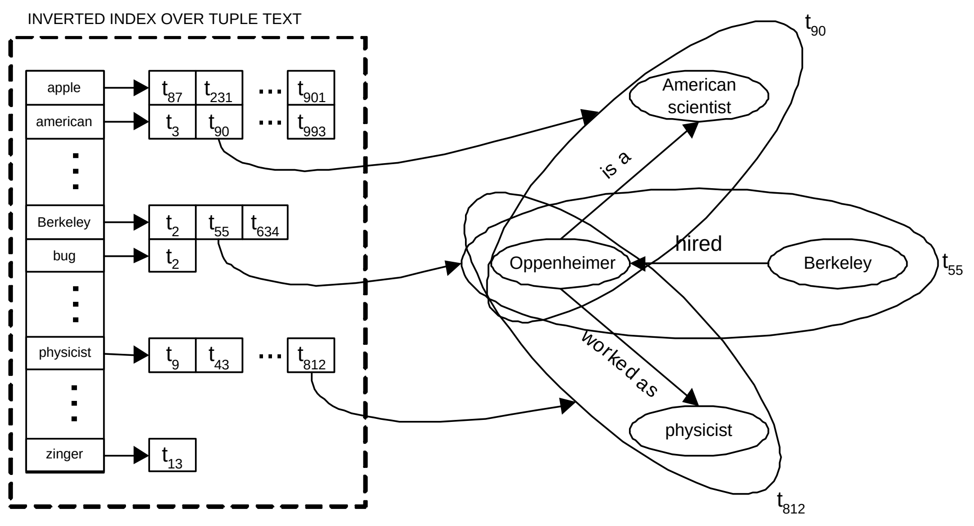

Figure 2: Query Processing. An inverted index over graph text allows the spreading activation algorithm to instantly find all the triples that contain search terms. The example query shown here consists of the terms, "American," "physicist," and "Berkeley," which all have entries in the inverted index. Retrieving graph regions that contain search terms allows spreading activation to reach other nodes within the regions quickly. In this example, the node "Oppenheimer" will score very highly as it receives activation from all three query terms.

is a technique that has been used to perform associative retrieval of nodes in a graph [7, 8, 6]. While spreading activation has been used for retrieval in hypertext, web documents, and semantic networks, TEXTRUNNER is the first system to use it for analyzing entity-relationship information automatically-extracted from Web pages.

Spreading activation systems work by first injecting "activation" into an initial target set of graph nodes. The activation then spreads from each node to its connected neighbors, then to neighbors of the neighbors, and so on. Each edge is associated with a *decay factor*, which diminishes the activation strength when it traverses the edge. The decay factors are critical: without them, activation from a single initial target node would eventually spread equally to all parts of the graph making activation level useless for discriminating between nodes.

Systems decide when to stop spreading according to a wide set of possible criteria: activation may be stopped after it travels a certain distance, or when its strength falls below a certain threshold. (Crestani discusses other criteria in [8].) Once the process is complete, nodes with high amounts of activation should be closely related to the nodes in the initial target set.

In this section, we discuss our spreading activation object scoring algorithm in detail. We also describe how TEXTRUNNER uses it to efficiently process an incoming user query, in spite of an extraction graph that contains hundreds of millions of nodes and edges.

3.2.1 Scoring

The scoring algorithm assigns a score S to nodes so they can be sorted by relevancy. Score spreads through the network in a spreading-activation-like manner. Like activation, score is first given to an initial target set of nodes. The initial set includes all nodes that contain a search query term $q_0, q_1, ...q_{n-1}$.

If a node n_i contains a query term q_j, then $TextHit(n_i, q_j) = 1$. Otherwise, $TextHit(n_i, q_j) = 0$. The algorithm also tests whether an edge e contains a query term q_j. In that case, $TextHit(e, q_j) = 1$.

The score for a single node n_i on a single search term q_j is the sum of three factors: the $TextHit()$ score for n_i itself, the $TextHit()$ scores for edges connected to n_i, and any score that was spread from neighboring nodes. The score S for a node n_i on query Q is the sum of scores earned by all of the query component terms. So, $S(n_i, Q) = \sum_{q_i \in Q} S(n_i, q_i)$. Thus, we can write:

$$\begin{aligned} S(n_i, Q) &= C_1 * \sum_{q_i \in Q} TextHit(n_i, q_i) + \\ &\quad C_2 * \sum_{q_i \in Q} \sum_{e' \in edges(n_i)} TextHit(e', q_i) + \\ &\quad C_3 * \sum_{q_i \in Q} \sum_{n' \in neighbors(n_i)} [\alpha(edge(n_i, n')) * S(n', q_i)] \end{aligned}$$

where $\alpha(edge(n_i, n'))$ is a *decay factor*, and falls between 0 and 1. The C constants are weights for the three sources.

The value $\alpha(edge(n_i, n'))$ is dependent on the type of the edge that happens to link nodes n_i and n'. We choose a value of close to 1 for edges that are of the "Is-a" type, or for an edge e where $TextHit(e, Q) = 1$. (Choosing large α values for e when $TextHit(e, Q) = 1$ allows score to flow more easily via edges that represent query-relevant predicates.) Otherwise, we choose a value closer to 0. Edges with high redundancy counts are considered more likely to be accurate, and so will be given higher α values. (In the future, we may also adjust α values based on the edge label itself, perhaps allowing easier flow along edges with rare, interesting labels.)

There are a few further points that are specific to the

TextRunner scoring mechanism. First, the system tracks the original $TextHit()$ source for all score flows. Score is not allowed to "flow backwards" to nodes it has already traversed. Second, TextRunner currently prevents score from flowing beyond a source's immediate neighbors. As the graph is extremely large, computing scores becomes more burdensome as activation flows further from its source; this issue is discussed in more depth in Section 3.4.

3.2.2 *Query Processing*

The extraction graph can be very large. Our 90 million page test corpus results in over 652 million extracted triples. After assembling these triples into the extraction graph the system must process queries over a graph of approximately 544 million edges and 227 million nodes. How can TextRunner process such queries efficiently?

Prior to processing any queries, TextRunner computes an inverted index over the text of the triples that make up the extraction graph G. This index is analogous to a standard inverted index computed over corpus documents for document retrieval. In a standard inverted index, each corpus term points to a list of all the documents in which that term appears. For the inverted index in TextRunner each term found in the triple text points to a list of all the triples in which the term appears.

Unfortunately, the extraction graph may be too large to fit on any single machine. So, TextRunner divides the triple set over a set of index machines, and maintains multiple inverted indices. Each machine computes a separate inverted index for the locally-stored triples.

TextRunner processes queries as follows:

1. A query Q arrives at a single query processor machine. The query processor breaks Q into its component terms $q_0, q_1, ..., q_{n-1}$.

2. The query processor then asks the set of index machines to use spreading activation scoring to return the most relevant objects for query terms $q_0, q_1, ..., q_{n-1}$. Each index machine has a local inverted index for looking up nodes and edges, as shown in Figure 2.

3. Each of the index machines returns the best locally-stored objects to the query processor

4. The query processor then sorts the total set of objects by the spreading activation score and emits the ranked list.

3.3 Result Display

Returning a ranked list of graph nodes can be a good way to satisfy a user query. However, because of the structure embedded in the extraction graph there are a set of methods TextRunner can use to post-process the ranked list, and make it even more useful.

One such method is *predicate clustering*. A traditional search engine searches and returns only documents, so a top-k list is fairly easy to scan and understand. In contrast, TextRunner will return objects of widely varying types. For example, in response to the query, *"American physicist"*, TextRunner will retrieve nodes that are labeled with names, specialties, organizations, and important dates. Of course, all nodes are labeled by strings, so there is no information about a node's "true type."

While the user may not know the exact name of her search target, it seems reasonable to think she will know the rough type. In that case, a heterogeneous list could be very distracting. It would be better if the ranked output could be as type-homogeneous as possible. The resulting output groups together relevant objects that might have previously been separated in the pure score ranking. Testing against a set of ten queries shows that clustering raises precision from 0.045 to 0.2690, while dropping the number of correct extractions from 45 to 39.

The clustering algorithm takes an ordered list of nodes (objects) R, and returns a set of clusters C, which partition R. Each object is assigned to a single cluster. Within a cluster, the objects retain the same ordering as in R. We hope that all the objects within any given cluster are of roughly the same "type".

All objects are simple strings, without any additional explicit type information. Although entity tagging could be used to identify a string's type, the tags are generally limited to a small handful of hard-coded values: person, organization, date, etc. Since user queries can cover arbitrary topic areas, such a limitation is substantial.

Instead of examining the contents of the object strings themselves, TextRunner clusters objects based on similarity among the predicates they use. The more predicates two objects share, the closer they will be considered. The motivation is that objects that appear with similar predicates will tend to be of similar type. For example, scientists are often on the left-side of the predicates *"discovered"* and *"described."* In contrast, dates are often on the right-side off the predicate *"died in."*

Having defined the similarity metric, TextRunner can use any clustering algorithm. The current version uses agglomerative clustering, repeatedly merging the most-similar objects. When comparing two clusters, the distance is simply the average distance computed when an object is drawn from each cluster. Merging is repeated until the two closest clusters have similarity less than a minimum similarity constant.

3.4 Scalability

A major challenge for TextRunner is to respond to queries at interactive speeds even as its corpus grows to tens of billions of pages and the extracted extraction graph grows as large as a hundred billion edges. In our experiments, conducted over a 90 million page corpus and 20 machines, TextRunner's average query response time was 41 seconds.[4] This section explains how TextRunner's design ensures that the number of disks and CPUs needed to maintain this response time grows linearly with the corpus size. First, we discuss the construction of the extraction graph. Second, we discuss the scalability of query processing.

The extraction graph is simply the composition of a large set of extracted triples. Importantly, all triples are generated with *a single pass through the corpus*. Moreover, the extraction task is naturally decomposable so different Web pages can be processed in parallel on different machines.

After triple generation, TextRunner builds a "node-centric" distributed inverted index. Each node that results from triple generation is "assigned" to a single machine in the pool. Each triple is then copied to each of the

[4]We expect that with further system optimization, this number can be cut by an order of magnitude.

machines that contain either of its two nodes. Finally, each machine computes an inverted index over the text of the locally-stored triples. Each machine is thus guaranteed to store all the triples that contain a reference to any node assigned to that machine.

Storing each triple twice (one for each of its nodes) roughly doubles the necessary index storage space to about 356 gigabytes. Duplicating triples imposes a storage cost, but collecting all of a node's single-hop neighbors on a single machine is critical to scalable query processing.

The query processing stage begins by spreading activation across the extraction graph. Most search engines that use flow through a network to compute score information (*e.g.*, PageRank) do so at index time, but TextRunner's spreading activation scores are query-dependent and thus have to be computed speedily at query time. Yet spreading activation is difficult to compute at large scale because the number of nodes affected by activation emitted by a single source will be roughly b^d, where b is the branching factor, and d is the depth of activation.

To make spreading activation both efficient and parallelizable, TextRunner's score computation for a single node is local to a single machine. Since any two objects adjacent in the extraction graph might be placed on different machines, TextRunner can only guarantee single-machine processing by duplicating regions of the triple set.

As each machine currently only stores the triples immediately connected to a locally-assigned node, in our experiments d was set to 1. Even this tightly-constrained system is still quite powerful. Most importantly, as illustrated in Figure 1, spreading activation of depth 1 enables the system to synthesize triples extracted from distinct pages.

TextRunner can support spreading activation for greater depths simply by increasing the amount of duplicated triple information. When $d = 2$, each machine must store all triples containing nodes within two hops of a locally-assigned node. If there is an average number of neighbors b in the graph, the storage penalty should be about b^2. Increasing the value d increases the amount of information available when scoring the nodes at a given machine, and so may result in a small amount of extra computation.

Since spreading activation generally requires a maximum-spread limit (else it flood the entire graph), a small d is not unreasonable [7]. Since each node appears on only one machine, the central query processor can ask each machine to return only its local top k-scoring results, and be guaranteed to retrieve the global top k. As query-processing burdens on the TextRunner system scale linearly with the corpus size, we can process roughly 4.5 million documents on each machine. So processing a ten-billion page corpus, and retaining current performance, should require fewer than 2,500 machines.

4. EXPERIMENTAL RESULTS

To assess TextRunner, we focused our evaluation on qualified list queries and used a combination of automatic processing and manual tagging to compare TextRunner with a Web Information Extraction (IE) system (KnowItNow[4]).

TextRunner runs on a cluster of 20 dual-Xeon machines, each with a single local disk of 250GB. We restricted TextRunner to extracting information from a 90-million page corpus crawled and indexed by Nutch, an open-source search engine [23]. On this collection, where average document length is 30 sentences, TextRunner extracts approximately 67.5 million individual *Is-A* edges and 1.9 billion predicate edges, each of which is inserted into its extraction graph. The total number of nodes in the graph is 227 million due to the fact that many triples are extracted more than once. All of the experiments described below were carried out on this extraction graph.

common surgical procedures
animated cartoon characters
British spy novelists
classic French fare
Italian motorcycle companies
female deities
NATO country leader
Commonwealth nations
plasma physicists

Table 1: A sample of the qualified-list queries test set.

4.1 Qualified-List Queries

Qualified-list queries enable the user to search for sets of objects described by multiple properties even though each property may be mentioned on a distinct Web page. Given the success of IE systems such as KnowItAll [13] in locating instances of simple classes (*e.g.*, cities or movies) on the Web, it is natural to ask whether qualified-list queries could be answered in the same manner. Thus, when a user submits the query *"American physicist"*, KnowItAll utilizes Hearst-style extraction patterns (*e.g.*, "American physicists such as ...") to answer the query. KnowItAll typically utilizes large numbers of Google queries, giving it an unfair advantage relative to TextRunner's 90-million page index.[5] To better control the experiment, we utilized KnowItNow [4] which invokes KnowItAll's extraction algorithms but queries its own indexed corpus. Thus, we had both TextRunner and KnowItNow extracting information from the same set of Web pages.

To create a set of test qualified-list queries, we identified a set of common noun phrases contained in KnowItNow's hypernym ontology. We restricted these phrases to a set of items that were described by at least two attributes, for instance, *"classic French fare"*, *"OPEC country"*, and *"animated cartoon character."* This enabled us to directly evaluate the value of TextRunner's predicate edges and spreading activation search which are added on top of the *Is-A* edges obtained through KnowItNow's set of hypernym patterns. More examples of qualified-list queries can be seen in Table 4.

We further restricted the test queries to ones where querying the phrase using the Google search engine returned a hit count lower than a certain threshold. In such cases, there may be no need for a system such as TextRunner. We limited evaluation to a set of twenty test queries because we had to manually tag extractions to determine precision.

4.1.1 Results

[5]Note that KnowItAll's reliance on numerous Google queries, for every qualified-list query, makes it slow and thus impractical as a search engine.

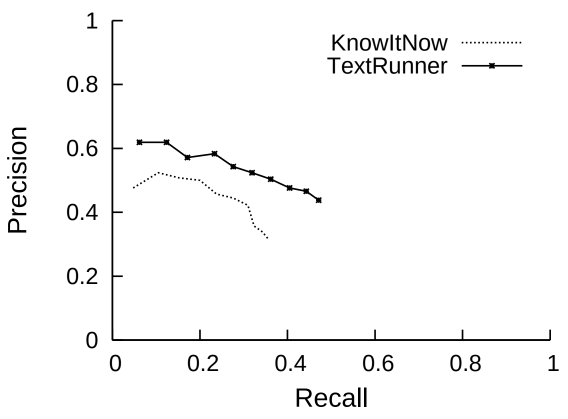

Figure 3: Precision and Recall of TextRunner **and** KnowItNow **on the qualified-list queries test set.** TextRunner **has an average recall that is 11.4% better than** KnowItNow **and is 40% more precise.**

We evaluated the precision and recall of TextRunner and KnowItNow over our query test set. To enable the evaluation of recall, given that the full set of answers to each query is not known, we asked each system to output up to a fixed number of objects L_q for each query q. This enables us to define recall R at each list position p as

$$R_p = \sum_{q \in Q} \frac{Correct(q,p)}{L_q}$$

where Q is the set of queries, and $Correct(q,p)$ represents the number of correct extractions for a query q through position p. For example, if we asked a system to output 20 objects per query, it would achieve perfect recall if it was able to return a correct instance in every position from 1 to 20, for every query. We set L, the size of the list output, to be 10. For our set of queries, this setting for L ensures that we do not ask for more answers than are known to exist.

Figure 3 shows the average precision and recall over the queries in our test set for both the TextRunner and KnowItNow systems, when asking each system to return up to ten results per query. On 80% of the test queries, KnowItNow failed to find the maximum allowable number of answers. This happened only 25% of the time for TextRunner. If we compare answers in the top-ranked position, TextRunner achieves a precision of 0.62 compared to 0.47 obtained by KnowItNow.

At a maximum of ten answers per query, TextRunner's recall is 11.4% better than KnowItNow. The answers found by TextRunner were found to be 40% more precise. Thus, when compared on the same corpus, relational search consistently finds more correct instances compared to an information extraction system that is built upon hypernym extraction patterns alone.

5. CONCLUSIONS AND FUTURE WORK

This paper introduced TextRunner, which extracts information from the Web to answer challenging qualified-list queries. TextRunner relies on the extraction graph, a novel approximation to a semantic network that is automatically extracted from Web pages. In a series over experiments over 90 million Web page corpus, and the resulting 227 million node extraction graph, we provided measurements that seek to quantify the benefit of TextRunner when compared to state-of-the-art search and information extraction systems.

We enumerate some of the most prominent directions for future work below. First, TextRunner needs to be evaluated on larger query sets, larger document collections, in a variety of different domains, and via exposure to live users. It would also be instructive to test TextRunner on additional corpora such as LexisNexis or Medline. Second, TextRunner would benefit from merging of both objects (*i.e.*, entity merging) and predicates (*i.e.*, finding synonyms). Typical algorithms for these tasks (*e.g.*, [10]) use domain specific information, but it is easy to envision a general set of heuristics as follows. Entities that share enough triples are merged, then predicates that share enough objects are merged as well. This mutually recursive process can be repeated until quiescence. Third, TextRunner should intelligently handle references to multiple real-world objects that share a single name (*e.g.* "Jaguar" is a car and an animal). This problem can lead TextRunner to mistakenly mix relationships from distinct, but similarly-named, objects. Finally, we plan to investigate TextRunner's ability to integrate information seamlessly from non-textual sources including RDF, databases, and HTML tables with information extracted from the Web.

Much of the innovation in search-engine research focuses on the Web as a set (or graph) of documents. Semantic Web research analyzes the other extreme where information is expressed in machine-processable languages such as RDF and OWL. Relational Web search is a step in the research program that seeks to unify these disparate bodies of research. TextRunner takes as input today's HTML-based Web, and commits to a scalable design, but attempts to yield better query responses by extracting structure and unearthing valuable semantic information.

6. ACKNOWLEDGEMENTS

This research was supported in part by NSF grant IIS-0312988, DARPA contract NBCHD030010, ONR grant N00014-02-1-0324 as well as gifts from Google and the Turing Center.

7. REFERENCES

[1] T. Berners-Lee. Notation3. In *http://www.w3.org/DesignIssues/Notation3*.

[2] D. M. Bikel. Intricacies of collins' parsing model. *Computational Linguistics*, 30(4):479–511, 2004.

[3] E. Brill and G. Ngai. Man (and woman) vs. machine: a case study in base noun phrase learning. In *Proceedings of the Association for Computational Lingustics*, pages 65–72, 1999.

[4] M. Cafarella and O. Etzioni. A Search Engine for Natural Language Applications. In *Procs. of the 14th International World Wide Web Conference (WWW 2005)*, Tokyo, Japan, 2005.

[5] M. J. Cafarella, D. Downey, S. Soderland, and O. Etzioni. Knowitnow: Fast, scalable information extraction from the web. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2005.

[6] P. Cohen and R. Kjeldsen. Information retrieval by constrained spreading activation on semantic networks. *Information Processing and Management*, 23(4):255–268, 1987.

[7] F. Crestani. Application of spreading activation techniques in information retrieval. *Artificial Intelligence Review*, 11(6):453–482, 1997.

[8] F. Crestani and P. Lee. Searching the web by constrained spreading activation. *Information Processing and Management*, 36(4):585–605, 2000.

[9] M. Cutler, Y. Shih, and W. Meng. Using the structure of html documents to improve retrieval. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 241–251, 1997.

[10] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, 2005.

[11] S. Dumais, M. Banko, E. Brill, J. Lin, and A. Ng. Web question answering: Is more always better? In *Proceedings of SIGIR 2002*, pages 291–298, 2002.

[12] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A. Popescu, T. Shaked, S. Soderland, D. Weld, and A. Yates. Web-Scale Information Extraction in KnowItAll. In *WWW*, pages 100–110, New York City, New York, 2004.

[13] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A. Popescu, T. Shaked, S. Soderland, D. Weld, and A. Yates. Unsupervised named-entity extraction from the web: An experimental study. *Artificial Intelligence*, 165(1):91–134, 2005.

[14] A. D. Haghighi, A. Ng, and C. D. Manning. Robust textual inference via graph matching. In *Proceeings of EMNLP*, 2005.

[15] M. Hearst. Automatic acquisitiion of hyponyms from large text corpora. In *Proceedings of the 14th International Conference on Computational Lingustics*, pages 539–545, 1999.

[16] C. F. Hempelmann, V. Rus, A. C. Graesser, and D. S. McNamara. Evaluating state-of-the-art treebank-style parsers for coh-metrix and other learning technology environments. In *Proceedings of the Second Workshop on Building Educational Applications Using Natural Language Processing and Computational Linguistics at the ACL conference*, 2005.

[17] J. Hendler. Massively-parallel marker passing in semantic networks. In *Semantic Networks in Artificial Intelligence*, pages 277–292, 1992.

[18] K. Hiramatsu, J. Akahani, and T. Satoh. Two-phase query modification using semantic relations based on ontologies. In *Proceedings of IJCAI-03 Workshop on Information Integration on the Web*, pages 155–158, 2003.

[19] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Meeting of the Association for Computational Linguistics*, 2003.

[20] C. Kwok, O. Etzioni, and D. Weld. Scaling question answering to the web. In *Proceedings of the 10th International World Wide Web Conference (WWW 2001)*, 2001.

[21] D. Lin and P. Pantel. Discovery of inference rules from text. In *Proceedings of the Seventh International Conference on Knowledge Discovery and Data Mining*, pages 323–328, 2001.

[22] G. Ngai and R. Florian. Transformation-based learning in the fast lane. In *Proceedings of the North American Association for Computational Linguistics*, pages 40–47, 2001.

[23] Nutch. http://lucene.apache.org/nutch/.

[24] A. Pisharody and H. E. Michel. A search engine technique using relation based keywords. In *Proceedings of the 2005 International Conference on Artificial Intelligence, IC-AI 05*, 2005.

[25] M. Quillan. *Semantic Information Processing*, chapter Semantic memory. MIT Press, 1968.

[26] A. Ratnaparkhi. *Maximum Entropy Models for Natural Language Ambiguity Resolution*. PhD thesis, University of Pennsylvania, 1998.

[27] G. Salton. *Automatic information organization and retrieval*. McGraw-Hill, 1968.