Optimizing Scoring Functions and Indexes for Proximity Search in Type-annotated Corpora

Soumen Chakrabarti
IIT Bombay
soumen@cse.iitb.ac.in

Kriti Puniyani
IIT Bombay
kriti@cse.iitb.ac.in

Sujatha Das
IIT Bombay
gsdas@cse.iitb.ac.in

ABSTRACT

We introduce a new, powerful class of text proximity queries: find an instance of a given "answer type" (person, place, distance) near "selector" tokens matching given literals or satisfying given ground predicates. An example query is `type=distance NEAR Hamburg Munich`. Nearness is defined as a flexible, trainable parameterized aggregation function of the selectors, their frequency in the corpus, and their distance from the candidate answer. Such queries provide a key data reduction step for information extraction, data integration, question answering, and other text-processing applications. We describe the architecture of a next-generation information retrieval engine for such applications, and investigate two key technical problems faced in building it. First, we propose a new algorithm that estimates a scoring function from past logs of queries and answer spans. Plugging the scoring function into the query processor gives high accuracy: typically, an answer is found at rank 2–4. Second, we exploit the skew in the distribution over types seen in query logs to optimize the space required by the new index structures required by our system. Extensive performance studies with a 10GB, 2-million document TREC corpus and several hundred TREC queries show both the accuracy and the efficiency of our system. From an initial 4.3GB index using 18,000 types from WordNet, we can discard 88% of the space, while inflating query times by a factor of only 1.9. Our final index overhead is only 20% of the total index space needed.

Categories and subject descriptors: [H.3.1 Content Analysis and Indexing; Linguistic processing] [H.3.3 Information Search and Retrieval; Query formulation, Retrieval models].
General terms: Algorithms, Experimentation.
Keywords: Indexing annotated text.

1. INTRODUCTION

Information Retrieval (IR) and XML query systems are headed for significant unification. Starting with simple fielded search [2], IR engines are getting more sophisticated at handling more complex annotation tags inlined with source text [25]. Meanwhile, XML engines are adding text search features [13, 5, 30, 14, 1]. The Unstructured Information Management Architecture (UIMA, `http://www.alphaworks.ibm.com/tech/uima`) is moving to standardize annotation and indexing modules and define a query paradigm based on XML fragments.

WWW 2006, May 23–26, 2006, Edinburgh, Scotland.
ACM 1-59593-323-9/06/0005.

Ontologies (e.g. `http://www.ontoweb.org/`), linguistic knowledge bases [26] and processors [8], semantic taggers [15, 10], and named entity recognizers are the major sources of annotations added to a corpus. Consider the noun is-a (hypernym) hierarchy in WordNet, and suppose every noun (phrase) in the corpus has been annotated with all its hypernyms. E.g., if the token *Einstein* appears at some position, we recognize that a special case of types *Physicist* and *person* (among others) appears at this position.

Suitable positional indexes of such annotations, integrated with text, are key enablers of the Semantic Web vision, because they let us ask powerful queries that combine syntactic match with type constraints.

Such queries can be used to guide contextual information extraction and aggregation where scanning the whole corpus (e.g. the Web) would be prohibitive. E.g., we may be interested in extracting numeric tokens adjacent to *hours* and close to the words *laptop* and *battery*. We may even use instances of specified types to activate instances of other types, e.g., we may want to instantiate type *person* near type *money amount* and words *UN* and *oil*.

Apart from soft aggregation, we can also use such queries as initial approximations to natural language questions. E.g., to answer the question "Who discovered the theory of relativity", we can seek instances of *person* in close proximity to stems *discover*, *theory* and *relativity*, using a suitable function to weight and aggregate the evidence from occurrences of the three keywords. Recent NLP work [21] has enabled mapping questions to target entity types with high accuracy. Here we consider the task of retrieving answer candidates using those types.

In this paper we will work with the is-a relation, but our system works with any partial order on entities, such as is-part-of, is-contained-in, etc. While small type systems may suffice for specific applications, open-domain query systems need very large type systems to be useful. Bootstrapping large type systems and semantic relations from the Web [10, 28, 11, 6] is an active research area, but efficient index and query support are needed to fully realize their potential.

1.1 Our contributions

We identify a broad class of proximity queries that combine text and annotation. Despite its simplicity, our framework is surprisingly general and useful for a number of text mining applications. We describe a system we are building around Lucene [2] and UIMA (`http://www.alphaworks.ibm.com/tech/uima`) that enables efficient annotation indexing and search. We intend to make the system publicly available. We identify two key technical problems that must be solved: learning proximity scoring functions and workload-adaptive index optimization.

Scoring functions are fairly standardized in traditional IR

but largely secret in Web search. Early XML query semantics were set- and path-oriented; later, scoring schemes were adopted from IR and Pagerank [13, 5, 30, 16, 14, 1], but these are not yet grounded in evaluation as extensive as standard IR. Therefore, we must learn good proximity scoring functions automatically before we can consider index and query optimization.

We design a simple and efficient learning algorithm to fit a proximity-based scoring function to logs of queries and known answers. When it is plugged into the query processor, we get high accuracy: recall at rank 300 increases from 0.8 (standard IR) to 0.9 and typically, a correct answer token appears at rank 2–4. Surprisingly, we find the best-fit scoring function to be non-monotonic wrt to the distance between selectors and candidate tokens.

Large and deep type hierarchies are essential to support open-domain search. Consequently, the index space required for the type annotations becomes very large compared to the standard inverted index, which can be compressed significantly [31]. We devise new algorithms that exploit the skew in the distribution of answer types (atypes) in query logs (i.e., historical workload statistics) and cut down the additional index storage requirement by 85–90%, while sacrificing query processing speed by a modest factor between 1 and 3. Our indexes occupy about the same total space as the corpus in gzipped form and an optimized IR index.

We present large-scale experiments with two 5 GB corpora having a million documents each, some 400 million distinct token positions, an atype hierarchy with 80,000 types (18,000 non-leaf types), and several hundred real-life queries from the TREC (`http://trec.nist.gov/data/qa.html`) competition. We use TREC data mainly because of the "truthed" collection of questions and answers, but our conclusions appear general. We present data on both the quality of answers and the performance of our system.

1.2 Related work

IR engines can support basic proximity clauses with user-supplied windows. The popular IR package Lucene [2] provides basic fielded search with typically a limited number of fields, and does not natively support proximity clauses across fields. INDRI [25] has a rich and powerful query language that can support hard proximity clauses across tagged fields, but a hard proximity window must be specified by the user; also, we do not know of any workload-sensitive index optimization in INDRI.

The Bindings Engine [4] (BE) is a critical breakthrough toward semantic pattern searches, and is specifically optimized for text, but it depends strongly on immediate juxtaposition of ground constants with patterns to instantiate, as in "*cruel* `ProperNoun` *said*" or "*cities such as* `NounPhraseList`". BE is reported as using a handful of types (parts of speech and named entities like person, place, time) which still leads to an index 10 times larger than an IR index. However, thanks to the more restricted query class, BE uses more sequential disk access than our system.

The database and XML communities are steadily adding IR support [20]. COMPASS [14] and XXL [30] are notable systems integrating text and structure search, but to our knowledge do not learn textual proximity. COMPASS supports "concept" searches of the form "concept=value" where the concept may have soft aliases, but not "concept *near* value" in our sense.

Index space is a significant concern in adding IR support to XML systems. Florescu et al. [12] survey work in the database and XML communities and propose a text extension to XML-QL that allow element-level word matches. They report an index that is 12 times larger than the uncompressed XML source, and do not consider proximity-based scoring.

In principle, we can turn our corpus into a giant XML graph with one token per node and try to apply activation-based query systems such as ObjectRank [3], XRank [16] or TeXQuery [1]. However, given an average English token is 4–5 bytes, and compresses down to 0.5–1 byte in the IR index, even a 32-bit ID per such node would be prohibitive. Another problem is that (as we shall see) good scoring functions in the linear token space do not necessarily decay monotonically with distance.

Elegant data structures have been proposed to deal with "hard" proximity constraints [23, 27], but they do not support learnable proximity scores, type containment or self-tuning indexes that exploit the query distribution. Earlier IR systems have exploited query skew in index-pruning [9] and caching [29] to achieve impressive reduction of IR index sizes and query times, but do not consider queries involving a type hierarchy.

2. SYSTEM ARCHITECTURE

Figure 1 shows our system architecture. The system input consists of a text corpus, an atype taxonomy with associated annotators, and pairs of sample queries with truthed response tokens embedded in their document contexts. From the query-response pairs, we learn a scoring function that rewards and combines proximity between candidate tokens and matched selectors. This module is described in Section 3. The scoring function is plugged into the query processor. We also optimize our indexes using the query logs; this is described in Section 4.

2.1 Atype taxonomy, corpus and annotators

The **atype taxonomy** is a DAG where nodes are atypes and edges represent the `is-a` relation. A number of sources can provide atype taxonomy information. Here we use WordNet [26], but we are also integrating data from `http://en.wikipedia.org/` to increase recall. Is-a instances can also be bootstrapped from the Web effectively [11]. In this paper, we will refer to nodes using WordNet synset notation, e.g., `Einstein#n#2` is the second noun sense of *Einstein*, meaning *genius*.

The **corpus** is a set of documents. Each document is a sequence of tokens. Tokens can be compound, such as `New_York`. An annotator module (Figure 1) connects some tokens to nodes in the atype taxonomy. E.g. the string token *Einstein* might be connected to both senses `Einstein#n#1` (the specific Physicist) `Einstein#n#2` (genius). Now, through is-a connections in WordNet, we know that *Einstein* is-a `person#n#1`. Disambiguation can be integrated into the annotator module, but is an extensive research area in NLP [24] and is outside our current scope.

2.2 Query and candidate responses

A **query** consists of two parts. The first part is an atype from the taxonomy. The second part is a set of indexable predicates on token strings. The simplest predicate is string or stem equality, but we can also use a fixed library of regular

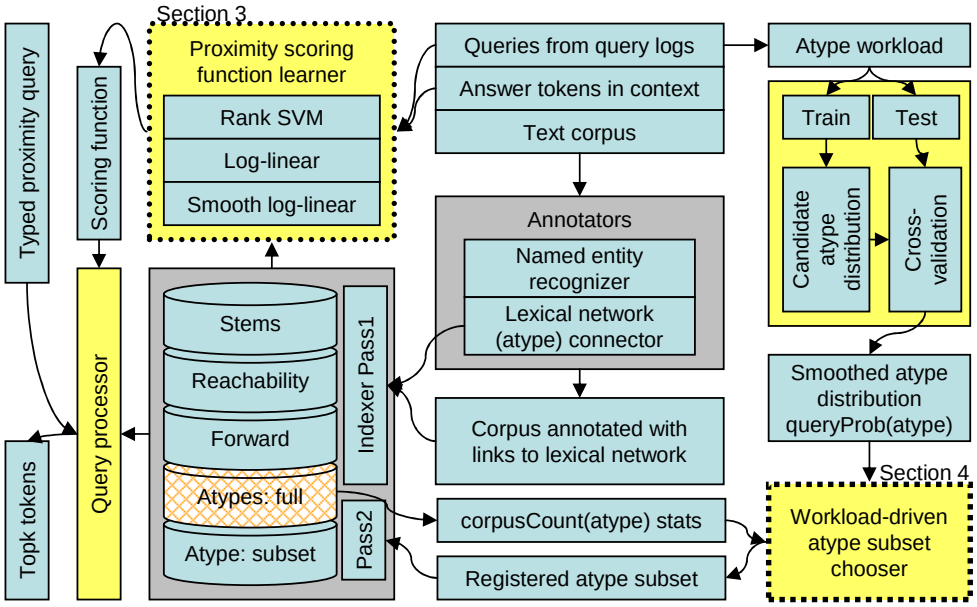

Figure 1: System architecture. In this paper we are concerned with the blocks highlighted with dotted lines: learning a scoring function, and workload-driven index optimization.

expressions, also called *surface patterns*. (Multiple atypes can be allowed easily, but we will not discuss this here.)

To find the distance between Hamburg and Munich, we might use the atype `linear_measure#n#1` and literal matches for *Hamburg* and *Munich*. The system and experiments in this paper used only equality predicates on stems; in such cases, we call string literals **selectors**. The atype may also be specified by a surface pattern, in this example, say, by `[:digit:]+`, a sequence of digits.

Any token in the corpus that is connected to a descendant of `linear_measure#n#1` is a candidate answer token. We admit a candidate only if at least one selector appears within W (typically, 50) tokens of the candidate. Many IR systems use similar pruning policies.

The score of a candidate depends on the selectors that appear nearby, as described in Section 3. For some applications, we simply need to report k candidate tokens in order of decreasing score. In other applications such as question answering, a short context around the candidate is submitted to a more sophisticated NLP module.

2.3 Indexes

The query processor opens a cursor on the posting list for each atype and selector, and does a BE-style [4] multi-way merge while pushing candidate tokens with scores into a scoring heap. In Section 4 we will build smaller indexes where most atypes will be "missing"—in that case, the query atype must be generalized to a coarser atype, and tentative result tokens checked using a *forward index* and a *reachability index* (described later in this section).

2.3.1 The stem and full atype indexes

As shown in Figure 1, in the first pass we build, apart from the forward and reachability indexes, two ordinary inverted indexes [31]. One maps from stems to *posting lists*. Each posting is a record containing a document ID where the stem appears, the number of times it appears in the document, and a list of token offsets where it appears. For English corpora, the size of the positional index is typically between 15–40% of size of the original (uncompressed) corpus, depending on the index compression techniques used. An IR index provides efficient sequential access through a posting list, so that multiple postings can be merged efficiently.

While indexing a stem, we also look for any atype attached to the token, and then follow all is-a links in the atype DAG up to the roots. For example, from the token *Einstein* in some document d, we can reach synsets `physicist#n#1`, `intellectual#n#1`, `scientist#n#1`, `person#n#1`, `organism#n#1`, `living_thing#n#1`, `object#n#1`, `causal_agent#n#1`, `entity#n#1`. We index all these atypes as if they all occurred at the same token offset in d as *Einstein*, in a second inverted index called the (full) atype index, i.e., in the full atype index, the posting list of each of the above atypes will include d.

A realistic and useful atype taxonomy can be quite deep—many prominent noun classes in WordNet are 6–12 links down from the noun roots. Consequently, as Figure 2 shows, the full atype index is almost the size of the uncompressed original corpus, over thrice the size of the gzipped corpus, and almost five times the size of the stem index. This would be unacceptable in most search applications, definitely large-scale services that need to cache substantial parts of the index in RAM. Luckily the query workload is very skewed, so we can choose only a subset of atypes to index; this is the topic of Section 4.

2.3.2 Search using stem and full atype indexes

The posting list for a term (an atype or a selector) helps us scan through (*docid*, *tokenOffset*) pairs in increasing lexicographic order. We use a bounded max-heap (of size k where k is the number of results sought) to keep track of the highest k scores. The memory required is bounded by the sum of the size of the score heap and the space required for holding all occurrences of candidates and selectors within a single document. We scan the postings lock-step, completely processing one document and inserting all its candidate lo-

Corpus/index	Size (GB)
Original corpus	5.72
Gzipped corpus	1.33
Stem index	0.91
Full atype index	4.30
Reachability index	0.005
Forward index	1.16

Figure 2: Relative sizes of the corpus and various indexes for TREC 2000.

cations into the score heap (evicting low-score locations if needed) and then move on to the next document.

2.3.3 Reachability index

In Section 4 we will also need a **reachability index**: given two atypes a_1 and a_2, or an atype a_1 and a token w, it can quickly (in $O(1)$ time) tell us if a_2 or w is-a a_1 i.e. if a_1 is an ancestor or generalization of a_2 or w in the atype taxonomy. Because a type hierarchy is "largely" a tree, a simple Dewey coding [16] of the atype taxonomy leads to a reachability index small enough (5 MB from 80,000 atypes of which 18,000 are non-leaf, see Figure 2) to fit easily in RAM. Given a token, we can verify if it is a candidate, but the reachability index does not give us a direct index into candidate positions in the corpus. Obtaining compact reachability indices for arbitrary graphs is more complicated [7].

2.3.4 Forward index

Another useful index is the **forward index** which basically stores the original corpus as close to the gzipped size as possible, while allowing fast (*docid*, *tokenOffset*) queries, returning the actual token (which can now be tested using the reachability index). Most search systems need to keep around a forward index to generate query-specific snippets along with the top responses. We will use the reachability and forward indexes in conjunction in Section 4.

The forward index[1] is built in two passes over the corpus. In the first pass, we count the frequency of each token in the corpus. Then we sort tokens by frequency and assign them variable-length integer IDs; frequent tokens get shorter IDs. In the second pass, we rescan the corpus and store packed bit-vectors for each document that can be unpacked and mapped back to tokens or token IDs.

Observe in Figure 2 that the forward index is even smaller than the gzipped corpus. Also, the forward index plus the reachability index add up to only about 25% of the full atype index.

Access to the forward index does require one random disk access per probe (a disk block generally holds several complete documents), but, as we shall see in Section 4, these accesses happen only for tokens that are top contenders for answering the query.

2.4 Experimental setup and evaluation

We used the TREC collection because it comes with a set of questions that can be mapped to precise atypes [21], and "truthed" answer passages. We used the TIPSTER and AQUAINT corpora adding up to over two million documents, which led to over 400 million term positions. Approximately 500 TREC questions were annotated with an atype.

[1]Implemented partly by Kuldeep Gharat, IIT Bombay.

Query time is measured as usual using real time and CPU time as appropriate. Our experiments were run on otherwise-unloaded P3/Xeon computers with 2–4GB RAM and Ultra320 SCSI disk. Index size is measured in bytes on disk.

The quality of a ranking of candidate tokens is measured in two standard ways. First, we measure the "recall-at-k": if our system returns k top-scoring tokens, in what fraction of questions do we nail an answer token (in the correct document and context specified by TREC)? Second, we measure the "mean reciprocal rank" or MRR used in the TREC community: if the first answer to query q is at rank r_q, award a score of $1/r_q$, and average over the query set Q to get the MRR $1/|Q| \sum_{q \in Q}(1/r_q)$.

3. LEARNING TO SCORE CANDIDATES

Our goal here is to learn how to score and rank candidate tokens. The score of a candidate token (i.e., one that is a subtype or instance of the query atype) depends on three sets of quantities: statistical properties of matched selectors in the vicinity, the distance between those selectors and the candidate, and the manner in which contributions from different selectors are aggregated at the candidate token.

Selector energy. Each matched selector s has an associated positive number called its **energy**, denoted $energy(s)$. A common notion of energy is the inverse document frequency or IDF standard in IR: the number N of documents in the corpus divided by the number N_s of documents containing the selector token s. This is a linear form of IDF, the logarithmic form $\log(1+N/N_s)$ is more commonly used. We use the term *selector energy* in place of the more familiar *term weight* to highlight that our scoring function has a spreading-activation form.

Gap and Decay. The gap between a candidate token w and a matched selector s, called $gap(w, s)$, is one plus the number of intervening tokens. We consider each selector as spreading activation or energy to the candidate. We define the energy transmitted by the selector to the candidate to be $energy(s)\ decay(gap(w, s))$, where $decay(g)$ is some function of the gap.

In many graph-based scoring systems such as ObjectRank [3], XRank [16] or TeXQuery [1] it is common to exploit a monotone decreasing $decay(g) = \delta^g$, where $0 < \delta < 1$ is a magic decay factor, for fast query execution. However, as we shall see, this form may not perfectly match data behavior.

Aggregation. A selector s can appear multiple times near a candidate, we call this set $\{s_i\}$. If a is the candidate, our generic scoring function looks like

$$score(a) \quad = \quad \bigoplus_s \bigoslash_i energy(s_i)\ decay(gap(s_i, a)), \qquad (1)$$

where $\oslash$ aggregates over multiple occurrences of s and $\oplus$ aggregates over different selectors. Sum or max can be used for $\oslash$ and $\oplus$, although sum behaves poorly as $\oslash$ because even a low-IDF selector can make the score less reliable by appearing often near a candidate.

Our query processor does not require any assumptions on the aggregation function, but we can think of other execution strategies that can take advantage of $\oslash$ = max and $\oplus = \Sigma$.

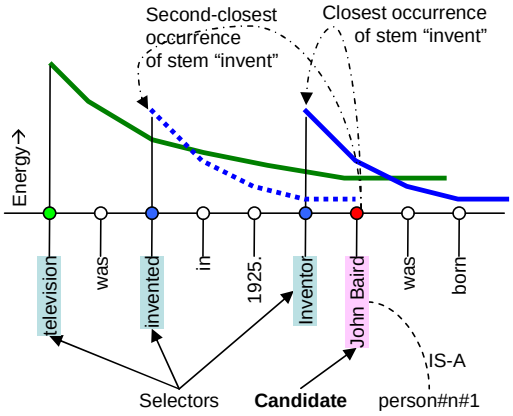

Figure 3: Selector activation example. Stems *television* and *invent* are selectors, `person#n#1` is the atype, *John Baird* is an atype candidate.

3.1 Data collection and representation

A few thousand TREC questions are available annotated with atypes [21]. We collected questions for which the answer tokens prescribed by TREC included at least one instance or subtype of the atype of the question. We report results on 261 usable questions from TREC 2000. For each question, we need positive (answer) and negative (candidate but not answer) tokens, and, to learn their distinction well, we should collect negative tokens that are "closest" to the positive ones, i.e., strongly activated by selectors.

To achieve this, we used the full atype index to locate all candidate tokens, and made a generous estimate of the activation from (the nearest occurrence of) each selector. This generous estimate used the log IDF as *energy* and no *decay*, i.e., *energy* was accrued unattenuated at the candidate position. For each query, we retained all positive answer tokens and the 300 negative tokens with top scores. Overall, we finished with 169662 positive and negative *contexts*. 5-fold cross-validation (i.e. 80% training, 20% testing in each fold) was used.

The next job was to turn contexts into feature vectors. Recall that there must be at least one selector match within W tokens of the candidate a. We set up this window with $2W+1$ tokens centered at a, and retain only one instance of each selector, the one closest to a[2].

As a first-cut, we can represent a context by a W-dimensional vector $\mathbf{f}$, where f_j is the energy sourced at position $\pm j$ wrt the candidate. If we are not sure which *energy*$(\cdot)$ function to use (log or linear form), we can always stick both values in a $2W$-dimensional feature vector. Clearly, we can extend to more than two forms of the energy function, and can use separate parameters for the left and right context. In experiments, the log form by itself with symmetric decay gave better results, so, for simplicity, we will assume we have W-dimensional feature vectors.

3.2 Learning *decay* with hinge loss

For every gap value $1 \le j \le W$, suppose the decay is β_j. Then the score of a feature vector $\mathbf{f}$ is $\sum_{j=1}^{W} f_j \beta_j$, or $\beta \cdot \mathbf{f}$. If $\mathbf{f}^+$ ($\mathbf{f}^-$) is a feature vector for a positive (negative) context, β should ensure that $\beta \cdot \mathbf{f}^+ > \beta \cdot \mathbf{f}^-$, or $\beta \cdot (\mathbf{f}^+ - \mathbf{f}^-) > 0$. From the answer contexts collected as above, we construct pairs $(\mathbf{f}_i^+, \mathbf{f}_i^-)$ of positive and negative answer contexts, indexed by i, so that the constraints on β look like

$$\beta \cdot (\mathbf{f}_i^+ - \mathbf{f}_i^-) \quad \ge \quad 0 \qquad \text{for all } i. \tag{2}$$

Herbrich *et al.* [17] and Joachims [19] suggested elegant max-margin solutions to ordering problems that we generically call **RankSVM**:

$$\min_{\beta, \mathbf{s} \ge \mathbf{0}} \tfrac{1}{2}\beta'\beta + C \textstyle\sum_i s_i \text{ s.t. for all } i,\ \beta \cdot \mathbf{x}_i + s_i \ge 1 \tag{3}$$

where $\mathbf{x}_i$ is shorthand for the difference vector $\mathbf{f}_i^+ - \mathbf{f}_i^-$ and no offset parameter β_0 is needed. As with support vector classifiers, C is a tuned parameter that trades off the model complexity $\|\beta\|$ against violations of the ordering requirements.

From our application perspective, 169662 passage contexts and millions of difference vectors $\mathbf{x}$ is just a beginning; the data is still sparse and any additional data helps accuracy. However, RankSVM executed millions of iterations with hundreds of millions of kernel evaluations, and, for some folds, failed to terminate in a day on a 3GHz CPU.

3.3 Learning *decay* with exponential loss

In equation (3), $\mathbf{x}_i$ is assigned penalty $C \max\{0, 1 - \beta \cdot \mathbf{x}_i\}$, which can be bounded above by $C \exp(-\beta \cdot \mathbf{x}_i)$, giving us the unconstrained optimization that we call **RankExp**:

$$\min_{\beta} \tfrac{1}{2}\beta'\beta + C \textstyle\sum_i \exp(-\beta \cdot \mathbf{x}_i) \tag{4}$$

which may be potentially less accurate than a hinge-loss formulation, but allows us to use simpler optimizers such as L-BFGS [22]. This lets RankExp scale much better with increasing training set size, as shown in Figure 4. On identical data sets, for $C \in [0.01, 0.3]$ the fraction of orderings satisfied by RankSVM and RankExp, as well as the MRRs were typically within 3% of each other, while RankExp took 14–40 iterations or 10–20 minutes to train and RankSVM took between 2 and 24 hours.

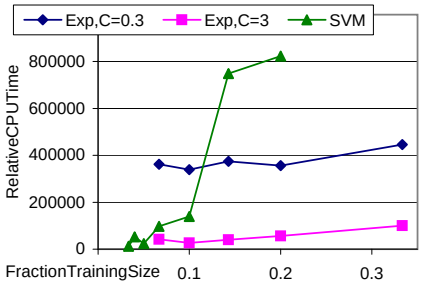

Figure 4: Relative CPU times needed by RankSVM and RankExp as a function of the number of ordering constraints.

3.4 Typical parameters and roughness

Figure 5 shows a typical β vector, where β_j gives the relative importance of a selector match at gap j (β_js can be shifted or scaled without changing the ranking). We did not expect the clearly non-monotonic behavior near $j = 0$, and only in hindsight found that this is a property of language (perhaps already appreciated by linguists): selectors are often named entities, and are often connected to the answer token via prepositions and articles that creates a gap. This

[2]Ties were broken arbitrarily. Obviously, we can also aggregate over multiple occurrences of a selector if ⊘ warrants.

goes against conventional wisdom that spreading activation should monotonically decay with distance. We believe that a more detailed study of proximity functions in IR, taking linguistic effects into account, is overdue.

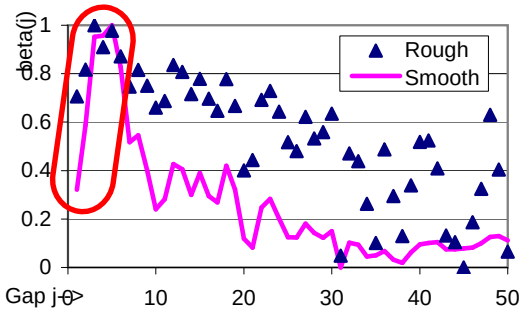

Figure 5: β_j shows a noisy unimodal pattern.

In spite of the prominent non-monotonicity near $j = 1 \ldots 5$, there is much noise at larger gap. To bias the learner to greater smoothness, we can pin a phantom $\beta_{W+1} = 0$, and penalize deviation of β_j from β_{j+1}:

$$\min_{\beta} \sum_{j=1}^{W} (\beta_j - \beta_{j+1})^2 + C \sum_i \exp(-\beta \cdot \mathbf{x}_i), \qquad (5)$$

where C is set by cross-validation. Figure 5 shows the smooth β, which also improves cross-validation accuracy slightly. It also gives us confidence in the non-monotonic nature of the best *decay* function.

3.5 Accuracy of learnt score function

In a standard IR system [31], the score of a snippet would be decided by a vector space model using selectors alone. We gave the standard score the additional benefit of considering only those snippets centered at an atype candidate, and considering each matched selector only once (use only IDF and not TF). Even so, a basic IR scoring approach was significantly worse than the result of plugging in the scoring function learnt by RankExp, as shown in Figure 6. Both recall and MRR (see Section 2.4) over held-out test data improve substantially.

β from	Train	Test	R300	MRR
IR-IDF	-	2000	211	0.16
RankExp	1999	2000	**231**	**0.27**
RankExp	2000	2000	235	0.31
RankExp	2001	2000	**235**	**0.29**

Figure 6: End-to-end accuracy using RankExp β is significantly better than IR-style ranking. Train and test years are from 1999, 2000, 2001. R300 is recall at $k = 300$ out of 261 test questions. $C = 0.1$, $C = 1$ and $C = 10$ gave almost identical results.

4. WORKLOAD-TUNED ATYPE INDEX

As we saw in Figure 2, for a 5.72 GB corpus, the stem index occupied only 0.91 GB while the full atype index took 4.30 GB. In this section we explore how to exploit the skew in the distribution of query atypes to achieve a graceful trade-off between index space and query performance.

4.1 Characterizing a skewed workload

Figure 7 shows a sample from TREC query logs. There is much skew, but, as is generally appreciated in the Web search community, the distribution is heavy-tailed and the skew may be somewhat misleading.

100	integer#n#1	5	president#n#2
78	location#n#1	5	inventor#n#1
77	person#n#1	4	astronaut#n#1
20	city#n#1	4	creator#n#2
10	name#n#1	4	food#n#1
7	author#n#1	4	mountain#n#1
7	company#n#1	4	musical_instrument#n#1
6	actor#n#1	4	newspaper#n#1
6	date#n#1	4	sweetener#n#1
6	number#n#1	4	time_period#n#1
6	state#n#2	4	word#n#1
5	monarch#n#1	3	state#n#1
5	movie#n#1	3	university#n#1

Figure 7: Highly skewed atype frequencies in TREC query logs.

The atype subset selection algorithm we propose uses an estimate of the probability of seeing an atype a in a new query, $queryProb(a)$. For WordNet alone, a can have over 18,000 (non-leaf) values, and the skew makes it difficult to estimate the probabilities of atypes never seen in past data—even thousands of questions will mostly hit a few atypes, but new data will always surprise us.

This is a standard issue in language modeling [24], and a variety of smoothing schemes are known. We use the well-known Lidstone smoother:

$$queryProb(a) = \frac{queryCount(a) + \ell}{\sum_{a'} queryCount(a') + \ell}, \qquad (6)$$

where $0 < \ell \leq 1$ is a parameter to be set via cross-validation. Several times, we randomly split the workload into halves W_1 and W_2, estimate $queryProb(a)$ using W_1, and estimate the probability of W_2 as

$$\sum_{a \in W_2} queryCount_{W_2}(a) \log \left(queryProb_{W_1}(a) \right).$$

Results are shown in Figure 8; it is fairly easy to pick off a prominently best ℓ for a given data set.

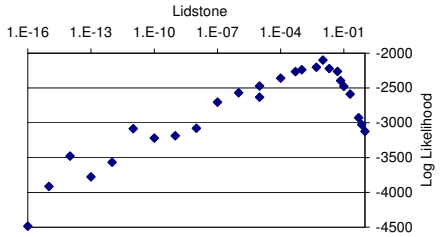

Figure 8: Log likelihood of validation data against the Lidstone smoothing parameter ℓ.

4.2 Pre-generalize and post-filter

Let the full set of atypes be A and imagine that some subset $R \subseteq A$ are **registered**, meaning that, when tokens are attached to the taxonomy during indexing (Section 2.3) and we walk up is-a links, only registered atypes are included in the index. Given such a registered atype index R and a query with atype a, we

1. find the best (defined later) registered generalization g in the taxonomy (see commments below)

2. perform a proximity search using g and the selectors in the query, which ensures recall, but generally lowers precision (therefore we must inflate k in the top-k search to some $k' > k$, more about this later)

3. use a forward index to get the actual instance token i of g in each high-scoring response

4. retain response i if a reachability index probe certifies that i is-a a (this consumes some more time and eliminates a fraction of responses)

5. in case fewer than k results survive, repeat with a larger k'; this is very expensive

The central issue is how to choose the registered subset R. Another issue is the choice of k'.

(While selecting R, we pretend all roots of A are included in R as sentinels, but we can avoid actually indexing these. While processing a query, in case no g can be found, we can pretend every word is a potential candidate, a situation that will essentially never arise given a reasonable algorithm for selecting R.)

4.3 Index space and query bloat models

With $|A|$ in tens of thousands, actually computing the atype subset index for many different candidate subsets R and evaluating query times would be prohibitively expensive. (One pass of atype indexing using Lucene on a 5 GB corpus takes a few hours.) Therefore we need estimates of the index storage required if R were indexed instead of A, as well as the impact on query times. We need to be able to compute these quantities quickly from bulk statistics collected from the corpus and workload.

An exact estimate of inverted index size is difficult in the face of index compression techniques [31]. The posting list for an atype a (or a token in general) has $corpusCount(a)$ entries in it, so as a first approximation, it takes space proportional to $corpusCount(a)$. Therefore, if subset R is indexed, the space needed can be approximated as

$$\sum_{a \in R} corpusCount(a). \tag{7}$$

Figure 9 shows that this crude approximation is surprisingly accurate.

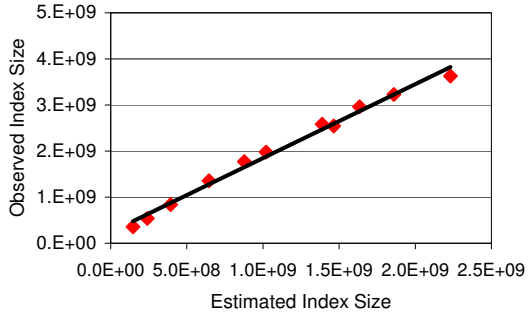

Figure 9: $\sum_{a \in R}$ *corpusCount*(a) is a very good predictor of the size of the atype subset index. (Root atypes are not indexed.)

Next we consider the bloat in query processing time owing to our incomplete atype index. In general, this depends on co-occurrence statistics between all possible atypes and all possible words. Even with a small number of tables and attributes, estimating multidimensional "selectivity" of select and join predicates for query optimization in relational databases is a challenging problem [18]. With over a million distinct tokens ("attributes") and $O(10000)$ atypes in our setting, we must necessarily make simplifying assumptions.

Query bloat happens in two stages: first, scanning the IR index posting lists takes longer because the posting list of the more general atype $g \in R$ is longer than the posting list of the query atype a; and second, because we are now obliged to screen the results using expensive forward index accesses.

For the first part, we assume that the time spent scanning posting of the atype a and intersecting them with selector postings takes time proportional to $corpusCount(a)$.

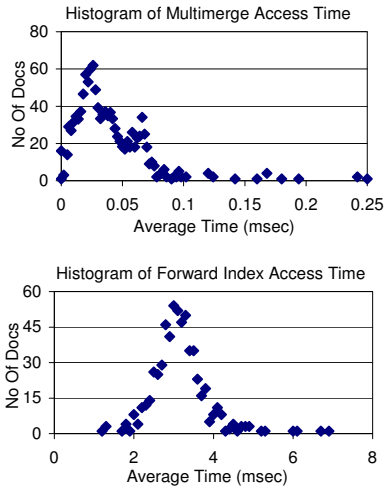

Figure 10: Distributions of t_{scan} and t_{forward}.

The second part depends on k', the number of results sought from the pre-generalized query. It also depends on the average time t_{scan} it takes to process one document during the merge of postings (Section 2.3.2), and the average time t_{forward} it takes to probe the forward index for one document and do the reachability test (Section 2.3.3 and Section 2.3.4). Figure 10 shows sample distributions of t_{scan} and t_{forward}; luckily, they are sufficiently peaked and centered to use point estimates. The overall query bloat factor is therefore

$$\frac{t_{\text{scan}}\, corpusCount(g) + k' t_{\text{forward}}}{t_{\text{scan}}\, corpusCount(a)}$$

Now we come to the question of what k' should be. If we make the crude assumption that the selectors occur independently of the candidates, we see

$$k' = k \frac{corpusCount(g)}{corpusCount(a)} \tag{8}$$

as a natural and simple choice (but see Section 4.4), using which we can write the query bloat factor as

$$\frac{corpusCount(g)}{corpusCount(a)} + k \frac{t_{\text{forward}}}{t_{\text{scan}}} \frac{corpusCount(g)}{corpusCount(a)^2}.$$

We call this $queryBloat(a, g)$, the bloat because a had to be generalized to $g \in R$, and for a given R, write the estimated

$$queryBloat(a, R) = \begin{cases} 1 & \text{if } a \in R \\ \min\limits_{g \in R, a \text{ IsA } g} queryBloat(a, g) & \text{else} \end{cases} \tag{9}$$

Note that at query execution time the choice of g from a given R is simple, but choosing a good R ahead of time is nontrivial.

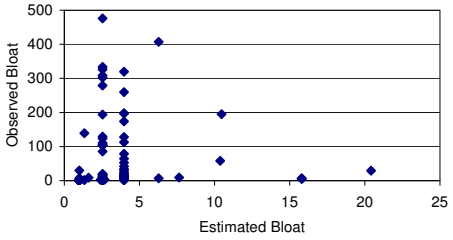

Figure 11: Scatter of observed against estimated query bloat.

Figure 11 shows a study of estimated bloat compared to observed bloat. The fit is not nearly as good as with the other half of our model in Figure 9, because 1. IO wait times are highly nondeterministic because of file-system buffering and RAID, and 2. To remain practical, our model ignores the effect of selectors. Similar variability is seen in the Bindings Engine [4, Figure 3, page 447] as well. In the relational query optimizer literature, join size estimates (and therefore CPU/IO cost estimates) are often relatively crude [18] but nevertheless lead to reasonable query plans.

For a specific R picked by **AtypeSubsetChooser** (Section 4.5) and 138 sample queries where $g \neq a$ given R, Figure 12 shows the cumulative distribution of the ratio of the observed to estimated bloat. As can be seen, 68% of the queries have observed bloats less than five times the estimated bloats, and 75% are within 10×. The fit of observed to estimated bloats is reasonable for most queries, with only a few queries exhibiting a large difference between the two.

Ratio ≤	Count	%	Ratio ≤	Count	%
.5–1	16	11.6	10–20	110	79.7
1–2	78	56.5	20–50	123	89.1
2–5	93	67.3	50–100	128	92.8
6–10	104	75.3	100–200	138	100

Figure 12: Histogram of observed-to-estimated bloat ratio for individual queries with a specific R occupying an estimated 145 MB of atype index.

4.4 Choice of k'

When the value of $\frac{corpusCount(g)}{corpusCount(a)}$ is small, equation (8) is reasonable, but when it is large, k' is too conservative, wasting resources. In practice, we found that clipping at a constant multiple x of k works well, i.e.,

$$k' \quad = \quad k \min\left\{x, \frac{corpusCount(g)}{corpusCount(a)}\right\} \tag{10}$$

Figure 13 shows that clipping at $x \approx 100$ is a good choice. Note that this is not a correctness issue because in case the query "runs dry" we will always restart with a larger k'. (The quality of our choice of R was largely unaffected by x.)

4.5 Choosing an atype subset

We thus have a bi-criteria optimization problem: given the corpus, query workload W, and atype set A, choose $R \subseteq A$ so as to minimize $\sum_{r \in R} corpusCount(r)$ and also minimize the expected query bloat

$$\sum_{a \in A} \underset{W}{queryProb}(a)\, queryBloat(a, R). \tag{11}$$

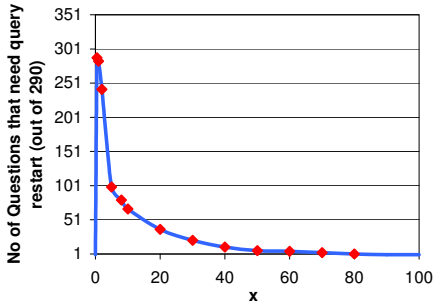

Figure 13: The effect of the choice of x on the ability of our system to produce top-k results without a query restart.

This optimization problem can be shown to be NP hard, even when the type hierarchy is a tree. Therefore we look for practical heuristics. We adopt a greedy approach of starting R with only the roots of A and progressively adding the locally "most profitable" atype c. Here "profit" depends inversely on the additional space δS that will be required by the posting list of c, and directly on the reduction δB of expected bloat that will result from including c in R. We use the ratio $\delta B/\delta S$ to pick the best c at every step. Once c is included, each descendant h might see a reduction in bloat. If h's bloat decreases, all ancestors u of h must update their $\delta B/\delta S$ scores.

The pseudocode is shown in Figure 14. There are a few false starts and subtleties about the algorithm that we cannot elaborate owing to lack of space. E.g., we cannot run it

AtypeSubsetChooser(A, W)
1: $R \leftarrow \{$roots of $A\}$, candidates $C \leftarrow A \setminus R$
2: initial estimated space $S \leftarrow \sum_{r \in R} corpusCount(r)$
3: using equations (6) and (9), expected bloat $B \leftarrow \sum_{a \in R \cup C} \underset{W}{queryProb}(a)\, queryBloat(a, R)$
4: **UpdateBloatsAndScores**$(\forall c \in C,$ commit=false$)$
5: **while** R is small and/or B is large **do**
6: choose $c \in C$ with the largest $score(c)$
7: **UpdateBloatsAndScores**$(c,$ commit=true$)$
8: **end while**

UpdateBloatsAndScores$(a,$ commitFlag$)$
1: $B' \leftarrow B$, $S' \leftarrow S + corpusCount(a)$
2: "cousins" of a to be patched $U \leftarrow \varnothing$
3: **for** each $h \notin R, h \in C, h$ IsA a **do**
4: $b = queryBloat(h, R)$, $b' = queryBloat(h, R \cup a)$
5: **if** $b' < b$ (bloat reduces) **then**
6: $B' \leftarrow (b' - b)\, \underset{W}{queryProb}(h)$
7: **if** commitFlag **then**
8: $U \leftarrow U \cup \{g : g \in C, g \neq a, h$ IsA $g\}$
9: **end if**
10: **end if**
11: **end for**
12: $score(a) \leftarrow (B - B')/(S' - S)$
13: **if** commitFlag **then**
14: move a from C to R
15: $S \leftarrow S'$, $B \leftarrow B'$
16: **UpdateBloatsAndScores**$(\forall u \in U,$ commit=false$)$
17: **end if**

Figure 14: The inputs are atype set A and workload W. The output is a series of trade-offs between index size of R and average query bloat over W.

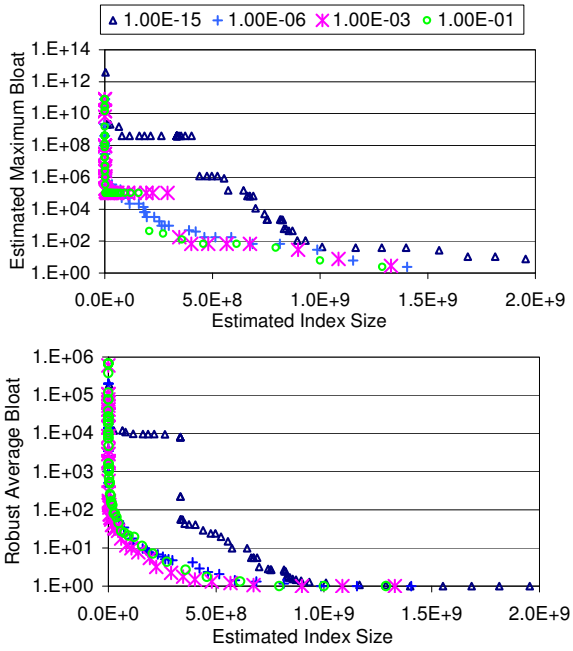

Figure 15: Estimated space-time tradeoffs produced by AtypeSubsetChooser. The y-axis uses a log scale. Note that the curve for $\ell = 10^{-3}$ (suggested by Figure 8) has the lowest average bloat.

"in reverse", starting with $R = A$ and discarding unworthy atypes.

Our experimental testbed has been described in Section 2.4. For picking ℓ we used a workload of 1200 atypes obtained from TREC 1999, 2000 and 2001, with a 60/40 train/test split; then the smoothed workload was used to pick R. For bloat and timing experiments we separated out 259 queries (with 304 atypes) from TREC 2000 (these were also used in Section 3).

4.5.1 Estimated space-time tradeoff

Figure 15 (upper chart) shows the reduction in estimated maximum bloat over all queries as **AtypeSubsetChooser** grows R. Each curve is for a different Lidstone parameter ℓ. The estimated *average* bloat over all queries would be overly influenced by a few outliers (see Figure 12). Therefore we discard the lowest and highest 2% of bloats and show a robust average over the rest (lower chart).

The curves in Figure 15 show a prominent knee: by the time the (estimated) index size is allowed to grow to 145 MB, the robust average bloat is 7, and it drops to 2 with an estimated index size of only 300 MB ($\ell = 10^{-3}$).

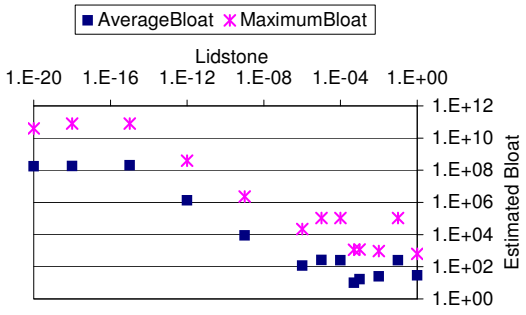

Figure 16: Estimated bloat for various values of ℓ for a specific estimated index size of 145 MB. The y-axis uses a log scale.

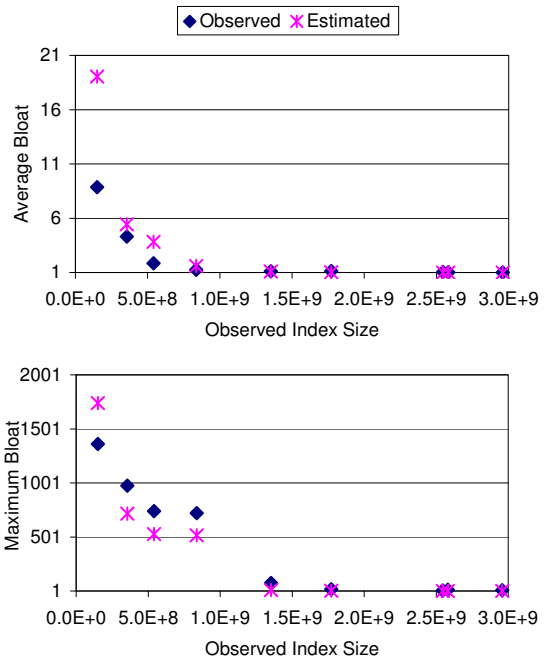

Figure 17: Estimated and observed space-time tradeoffs produced by AtypeSubsetChooser.

Very low ℓ results in low *queryProb* for atypes not seen in the training set, leading to an excessively aggressive discarding of atypes and consequently high test-set bloats. As ℓ is increased, *queryProb* increases, forcing **AtypeSubsetChooser** to conservatively include more atypes not seen in the training set.

It is comforting to see in Figure 16 that the best trade-off happens for roughly the same value of ℓ that provided the largest cross-validated log-likelihood in Figure 8. This need not have happened: maximizing workload likelihood is not the same as reducing query bloat.

4.5.2 Observed space-time trade-off

Next we ran multiple queries with various Rs having different index sizes to find actual running times and hence, actual bloats (Figure 17). The average observed bloat curve follows the estimated bloat curve in Figure 15 quite closely. In fact, averaged over many queries, our simple bloat prediction model does even better than at a per-query level (see Figure 11). With a modest 515 MB atype subset index, the average bloat is brought down to only 1.85.

4.5.3 Query execution dynamics

Figure 18 shows the average time taken per query, for various Rs with increasing index sizes, broken down into Lucene scan+merge time taken if $R = A$ ("FineTime"), Lucene scan+merge time using a generalized g if $R \subset A$ ("PreTime"), and the post-filtering time ("PostTime"). As can be seen, there are regimes where scan time dominates and others where filtering time dominates. This highlights why the choice of a good R is a tricky operation: we cannot assume cost estimates that are any simpler.

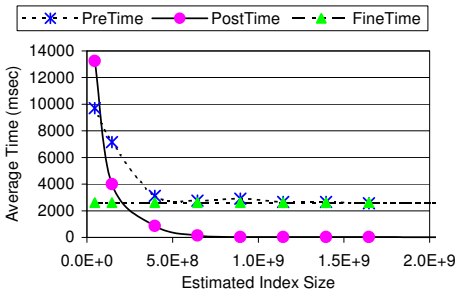

Figure 18: Average time per query (with and without generalization) for various estimated index sizes.

5. SUMMARY AND CONCLUSION

We have identified an important class of proximity queries in annotated text, and are building a system around Lucene and UIMA to answer such queries. This system can be used as a foundation for information extraction and shallow language-processing tasks. In this paper we described our solutions to two important technical problems: design of scoring function and workload-driven optimization of indexes. Even though the worst-case index reduction problem is intractable, we exploited skew in real-life workload to give sound engineering solutions.

Several related issues and extensions suggest themselves. Will our approach successfully extend to relatively "wild" Web text, other important semantic relations and general entity-relationship graphs? How to deal with dynamic corpus updates and support distributed indices? Can we evolve query planners that choose more intelligently between sequential posting access and random forward index access [14, 30]? Can we give provable guarantees for structured tasks such as information extraction?

6. REFERENCES

[1] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. TeXQuery: A full-text search extension to XQuery. In *WWW Conference*, pages 583–594, New York, 2004.

[2] Apache Software Group. Jakarta Lucene text search engine. GPL Library, 2002.

[3] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Authority-based keyword queries in databases using ObjectRank. In *VLDB*, Toronto, 2004.

[4] M. J. Cafarella and O. Etzioni. A search engine for natural language applications. In *WWW Conference*, pages 442–452, 2005.

[5] T. T. Chinenyanga and N. Kushmerick. An expressive and efficient language for XML information retrieval. *JASIST*, 53(6):438–453, 2002.

[6] P. Cimiano, G. Ladwig, and S. Staab. Gimme' the context: Context-driven automatic semantic annotation with C-PANKOW. In *WWW Conference*, pages 332–341. ACM Press, 2005.

[7] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal of Computing*, 32(5):1338–1355, 2003.

[8] H. Cunningham, K. Humphreys, R. Gaizauskas, and Y. Wilks. GATE: A TIPSTER-based general architecture for text engineering. In *Proceedings of the TIPSTER Text Program (Phase III) 6 Month Workshop*. Morgan-Kaufmann, 1997.

[9] E. S. de Moura et al. Improving web search efficiency via a locality-based static pruning method. In *WWW Conference*, pages 235–244, Chiba, Japan, 2005.

[10] S. Dill et al. SemTag and Seeker: Bootstrapping the semantic Web via automated semantic annotation. In *WWW Conference*, 2003.

[11] O. Etzioni, M. Cafarella, et al. Web-scale information extraction in KnowItAll. In *WWW Conference*, New York, 2004. ACM.

[12] D. Florescu, D. Kossman, and I. Manolescu. Integrating keyword searches into XML query processing. In *WWW Conference*, pages 119–135, Amsterdam, 2000.

[13] N. Fuhr and K. Grosjohann. XIRQL: A query language for information retrieval in XML documents. In *Research and Development in Information Retrieval*, pages 172–180, 2001.

[14] J. Graupmann, M. Biwer, C. Zimmer, P. Zimmer, M. Bender, M. Theobald, and G. Weikum. COMPASS: A concept-based Web search engine for HTML, XML, and deep Web data. In *VLDB*, pages 1313–1316, 2004.

[15] R. V. Guha and R. McCool. Tap: A semantic web test-bed. *J. Web Sem.*, 1(1):81–87, 2003.

[16] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD Conference*, pages 16–27, 2003.

[17] R. Herbrich, T. Graepel, and K. Obermayer. Support vector learning for ordinal regression. In *International Conference on Artificial Neural Networks*, pages 97–102, 1999.

[18] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD Conference*, pages 268–277, 1991.

[19] T. Joachims. Optimizing search engines using clickthrough data. In *SIGKDD Conference*. ACM, 2002.

[20] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *ICDE*, page 829, 2004.

[21] V. Krishnan, S. Das, and S. Chakrabarti. Enhanced answer type inference from questions using sequential models. In *EMNLP/HLT*, 2005.

[22] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Math. Programming*, 45(3, (Ser. B)):503–528, 1989.

[23] U. Manber and R. A. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Information Processing Letters*, 37(3):133–136, 1991.

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

[25] D. Metzler and W. B. Croft. Combining the language model and inference network approaches to retrieval. *Inf. Process. Manage.*, 40(5):735–750, 2004.

[26] G. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. Miller. Introduction to WordNet: An online lexical database. *International Journal of Lexicography*, 1993.

[27] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666, 2002.

[28] S. D. Richardson, W. B. Dolan, and L. Vanderwende. MindNet: Acquiring and structuring semantic information from text. In *COLING*, pages 1098–1102. ACL, 1998.

[29] T. Suel and X. Long. Three-level caching for efficient query processing in large Web search engines. In *WWW Conference*, Chiba, Japan, 2005.

[30] A. Theobald and G. Weikum. The XXL search engine: ranked retrieval of XML data using indexes and ontologies. In *SIGMOD*, page 615, 2002.

[31] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan-Kaufmann, May 1999.