Relevance-based Retrieval on Hidden-Web Text Databases without Ranking Support

Vagelis Hristidis, Yuheng Hu, and Panagiotis G. Ipeirotis

Abstract—Many online or local data sources provide powerful querying mechanisms but limited ranking capabilities. For instance, PubMed allows users to submit highly expressive Boolean keyword queries, but ranks the query results by date only. However, a user would typically prefer a ranking by relevance, measured by an information retrieval (IR) ranking function. A naive approach would be to submit a disjunctive query with all query keywords, retrieve all the returned matching documents, and then re-rank them. Unfortunately, such an operation would be very expensive due to the large number of results returned by disjunctive queries. In this paper we present algorithms that return the top results for a query, ranked according to an IR-style ranking function, while operating on top of a source with a Boolean query interface with no ranking capabilities (or a ranking capability of no interest to the end user). The algorithms generate a series of conjunctive queries that return only documents that are candidates for being highly ranked according to a relevance metric. Our approach can also be applied to other settings where the ranking is monotonic on a set of factors (query keywords in IR) and the source query interface is a Boolean expression of these factors. Our comprehensive experimental evaluation on the PubMed database and a TREC dataset show that we achieve order of magnitude improvement compared to the current baseline approaches.

Index Terms—Hidden-web databases, Keyword Search, Top-k ranking

1 Introduction

Many online or local data sources provide powerful querying mechanisms but limited ranking capabilities. For instance, PubMed[1] allows users to submit Boolean keyword queries on the biomedical publications database, but ranks the query results by publication date *only*. Similarly, the US Patent and Trademark Office (USPTO)[2] allows Boolean keyword queries or searching patents but only ranks by patent date. Furthermore, job search databases, such as the job search of LinkedIn,[3] allow users to sort job listings by date or title (alphabetically), but not by IR relevance of the job posting to the submitted query. As a more recent example, the micro-blogging service Twitter[4] offers a highly expressive Boolean search interface but ranks the results by date only. In most cases, these sources do not allow downloading and indexing of data or the size of the underlying database makes any comprehensive download [1], [2] an expensive operation.

Often, the user prefers a ranking other than the default sorting (e.g., by date) provided by the source. For instance, a user of the PubMed or USPTO Web sites may prefer a ranking by relevance [3], [4], measured by an Information Retrieval (IR) ranking function, as opposed to a date-based retrieval. Given that traditional IR ranking functions [5] like Okapi [6] and BM25 [7] implicitly assume disjunctive (OR) semantics, the naive approach would be to submit to the database a disjunctive query with all query keywords, retrieve all the returned documents, and then rank them according to the relevance metric of choice. However, this would be very expensive due to the large number of results returned by disjunctive queries. For example, consider the query "immunodeficiency virus structure," an example query used to teach information specialists how to search the PubMed database [8]. Executing the corresponding disjunctive query "immunodeficiency OR virus OR structure" on PubMed returns 1,451,446 publication results. Downloading and ranking them is infeasible for an interactive query system, even if the source is on the local network. The problem becomes even more critical if we use the public web services provided by PubMed for programmatic (API) access over the web. Given the large overhead incurred when retrieving publications, PubMed imposes quotas on the amount of data an application can retrieve per minute, rendering infeasible any attempt to download large number of documents.

To overcome such problems, in this paper, we present algorithms to compute the top results for an IR ranked query, over a source with a Boolean query interface but without any ranking capabilities (or with a ranking function that is generally uncorrelated to the user's ranking: e.g., by date). A key idea behind our technique is to use a probabilistic modeling approach, and estimate the distribution of document scores that are expected to be returned by the database. Hence, we can estimate what are the minimum cutoff scores for

• *Vagelis Hristidis and Yuheng Hu are with the School of Computing and Information Sciences, Florida International University Miami, FL.*
E-mail: {vagelis,yhu002}@cis.fiu.edu
• *Panagiotis G. Ipeirotis is with the Department of Information, Operations, and Management Sciences, New York University New York, NY. E-mail: panos@stern.nyu.edu*

1. http://www.ncbi.nlm.nih.gov/pubmed/
2. http://patft.uspto.gov/
3. http://www.linkedin.com/
4. http://www.twitter.com/

including a document in the list of highly ranked documents. To achieve this result over a database that allows only query-based access of documents, we generate a querying strategy that submits a minimal sequence of conjunctive queries to the source. (Note that conjunctive queries are cheaper since they return significantly fewer results than disjunctive ones.) After every submitted conjunctive query we update the estimated probability distributions of the query keywords in the database and decide whether the algorithm should terminate given the user's results confidence requirement or whether further querying is necessary; in the latter case, our algorithm also decides which is the best query to submit next. For instance, for the above query "immunodeficiency virus structure", the algorithm may first execute "immunodeficiency AND virus AND structure", then "immunodeficiency AND structure" and then terminate, after estimating that the returned documents contain all the documents that would be highly ranked under an IR-style ranking mechanism. As we will see, our work fits into the "exploration vs. exploitation" paradigm [9], [10], [11], since we iteratively explore the source by submitting conjunctive queries to learn the probability distributions of the keywords, and at the same time we exploit the returned "document samples" to retrieve results for the user query.

Our approach can also be extended and applied to other settings where the ranking is monotonic on a set of factors (query keywords in IR) and the source query interface is a Boolean expression of these factors. For instance, consider a database of products with Boolean attributes, like cars for sale that have attributes such as "used," "new," "two-doors," "four-doors," "convertible," and so on. Suppose that the query interface only allows specifying attribute values (e.g., "used AND convertible AND two-doors"). Suppose the source ranks cars always by price. If the user wants to rank by a weighted sum of the attribute values (e.g., $0.2 \cdot used + 0.4 \cdot convertible + 0.4 \cdot two - doors$), then we can apply an adaptation of our approach.

Our work has the following contributions:

1) We define the novel problem of applying ranking on top of sources with no ranking capabilities by exploiting their query interface.
2) We describe sampling strategies for estimating the relevance of the documents retrieved by different keyword queries. We present a static sampling approach and a dynamic sampling approach that simultaneously executes the query, estimates the parameters required for efficient query execution, and compensates for the biases in the sampling process.
3) We present algorithms that, given a user confidence input, retrieve a minimal number of results from the source through submitting high-selectivity (conjunctive) queries, so that the user's confidence requirement is satisfied.
4) We experimentally evaluate our algorithms using the PubMed database and examine two settings: (i) the remote setting, where we use web services to query the database, and (ii) the local setting where we query a locally installed subset of PubMed. Our results show an order of magnitude improvement compared to the naive query evaluation approach.

The rest of the paper is organized as follows. In Section 2 we describe related work and place our work in the context of the existing literature. In Section 3 we give the framework, problem definition, and notation, while in Section 4 we outline the basic ideas of our approach. Then, in Section 5 we describe in detail our algorithms, and in Section 6 we present the results of our experiments. Finally, Section 7 concludes the paper.

2 Related Work

A preliminary version of this work has been published as a short paper in [12].

Top-k queries: A significant amount of work has been devoted to the evaluation of top-k queries in databases. Ilyas et al. [13] provide a survey of the research on top-k queries on relational databases. This line of work typically handles the aggregation of attribute values of objects in the case where the attribute values lie in different sources [14], [15] or in a single source [16]. For example, Bruno et al. [15] consider the problem of ordering a set of restaurants by distance and price. They present an optimal sequence of random or sequential accesses on the sources (e.g., Zagat for price and Mapquest for distance) in order to compute the top-k restaurants. Since the concept of top-k is typically a heuristic itself for locating the most interesting items in the database, Theobald et al. [17] describe a framework for generating an approximate top-k answer, with some probabilistic guarantees. In our work, we use the same idea; the main and crucial difference is that we only have "random access" to the underlying database (i.e., through querying), and no "sorted access." Theobald et al. assumed that at least one source provides "sorted access" to the underlying content.

Exploration vs. exploitation: The idea of the exploitation/exploration tradeoff [9], [10], [11] (also called the "multi-armed bandit problem") is to determine a strategy of sequential execution of actions, each of which has a stochastic payoff. While executing an action we get back some (uncertain) payoff, and at the same time we get some information that allows us to decrease the uncertainty of the payoff of future actions. The problem has been first posed in the 1930's and has been used to model problems in a wide variety of areas, ranging from medicine and economics to ad placement in web pages. In our work, we are trying to maximize the payoff/exploitation of each query (which is the number of new, relevant top-k documents that the query retrieves) while minimizing the expense/exploration (number of queries sent, and documents retrieved).

Deep Web: Our work bears some similarities to the problem of searching and extracting data from the Deep Web [18] databases. Meng et al. [19], [20] examine the problem of estimating the number of useful documents in the database, assuming that the statistics about the frequency and the *tf.idf* weights of each word in the database is given. In our work, we estimate such statistics on-the-fly, as part of the explorative sampling process. Ntoulas et al. [2] attempt to download the contents of a Deep Web database by issuing queries through a web form interface. The goal of Ntoulas et al. is to download and index the contents of databases with limited query capabilities, whereas in our case the focus is on achieving on-the-fly ranking of query results, on top of sources with no (or non-useful) ranking capabilities. An alternative approach is to characterize databases by extracting a small sample of documents that is then used to describe the contents of the database. For example, it is possible to use query-based sampling [21], [22] to extract such a document sample, generate estimates for the distribution of each term, and then use the estimates to guide the choice of queries that should be submitted to the database. In the experimental section, we compare against this "static sampling" alternative and demonstrate the superiority of the dynamic sampling technique, which dynamically generates estimates tailored to the query at hand.

3 Problem Definition

Query Model Consider a text database D with documents $d, \ldots, d_m$. The user submits a keyword query $Q = \{t_1 ... t_n\}$ containing the terms $t_1 ... t_n$. The answer to the query is a list of the top k documents; the documents are ranked according to a relevance score $score(Q, d)$, which estimates the relevance of a document d to the query Q.

The score of a document can be computed using any of the the well studied *tf.idf* scoring functions like BM25 and Okapi [5], [6], [7]. The key arguments of a *tf.idf* function are the term frequency (tf), the document frequency (df) and the document length (dl). The *term frequency* $tf(t, d)$ is the number of times that the word t appears in document d. The document frequency $df(t, D)$ is the number of documents in D that contain t. Hence, $score(Q, d) = F(tf, df, dl)$. At its basic form, the *tf.idf* ranking function is:

$$score(Q, d) = \sum_{t \in Q, d} tf(t, d) \cdot \ln \frac{|D| + 1}{df(t, D)} \tag{1}$$

where $|D| = m$ is the size of the database D. In our experiments, we use the Okapi scoring function, although any other *tf.idf* function could be used. For simplicity though we use the basic *tf.idf* scoring function as the running example.

Data Source Model We assume that database D is only accessible through a Boolean query interface and we do not have direct access to the underlying documents. The query interface evaluates the Boolean query Q and returns the documents ranked using a non-desirable ranking function, e.g., by date (as is the case for PubMed and USPTO).

For instance, if the user query is Q=[*anemia, diabetes, sclerosis*], then we can submit to the data source queries q_1 = [*anemia AND diabetes AND sclerosis*], q_2 = [*anemia AND diabetes AND NOT sclerosis*], q_3 = [*diabetes OR sclerosis*], and so on. The returned results are guaranteed to match the Boolean conditions but the documents are not expected to be ranked in any useful manner.

Objective We want to devise a scheme for retrieving from D the top-k documents, ranked according to $F(tf, df, dl)$. The trivial solution is to send an extremely broad disjunctive query, returning all documents that have a non-zero $F(tf, df, dl)$ score. Then, we can retrieve the documents, examine their contents, and rerank them locally before presenting the results to the user. Unfortunately, this is a very time-consuming solution. Therefore, our objective is to construct a query sequence $q_1, q_2, \cdots, q_v$ of Boolean queries, that can be submitted to the database, retrieve as few documents as possible, and still contain all the documents that would be in the top-k results.

4 Overview of Approach

As mentioned above, our approach is based on choosing the best sequence $q_1, q_2, \cdots, q_v$ of Boolean queries to submit to the data source, such that we retrieve the top-k ranked documents for Q. Of course, to select the best sequence of queries, we need to know some statistics about the type of documents retrieved by each query q_i. To get these statistics we need to sample the database through query-based sampling. So, through querying we are both retrieving documents to generate the necessary statistics and at the same time aim to retrieve documents that are in the top-k relevant documents. So, we can consider our approach as a case of "exploration vs. exploitation."

Even though we can use any Boolean query in our strategy, we only consider conjunctive Boolean queries as candidates, given that a disjunctive query can be split to a set of conjunctive queries. Conjunctive queries provide a good query granularity and simplify the analysis below. Note that in practice we add negation conditions to the issued conjunctive queries in order to avoid retrieving the same results multiple times. For instance, if $Q = \{a, b\}$, after submitting $q_1 = a\ AND\ b$, we submit $q_2 = a\ AND\ NOT\ b$ instead of $q_2 = a$.

So, what are the goals of our querying strategy? Following Equation 1, we need to know the *tf* and *df* values for the terms in the database, to estimate the similarity score of a query to a document. Using these values, we can then estimate the overall *similarity score distribution* for all the documents in the database. Given the score distribution, we can compute how many documents in the database have score higher than the documents that we have seen so far.

The relatively easy part is the estimation of the *df* values. We can estimate these values in two ways: (a) We can send n queries to the database, one for each query term t_i, and compute the *df* value for each term. Note that the PubMed eUtils, which we use in our experiments, have a method to directly return the number of results (*df*) for a query. (b) We can use estimates of the *idf* (inverse *df*) values by using some other database with similar content (for example, using the Google Web 1T 5-gram collection[5]).

The more challenging part is the estimation of the *tf* values. We need to estimate the value of *tf* for each query term and for each document, that is, a total of $n \times |D|$ values. This is rather unrealistic without having direct access to the underlying database. So, we adopt a *query-based* probabilistic approach and we use the fact that term frequencies (*tf*) tend to follow a Poisson distribution within the documents of a database [17]. The more accurately we know the parameters of the distribution, the better we can estimate the document score distribution, and the better we can estimate how many documents should be in the top-k results but are still not retrieved.

One strategy for estimating the distribution parameter values is to generate a static document sample from the database and use this sample for our estimations. As we will see, this strategy suffers from some shortcomings. So, we present an alternative strategy as well, which relies on the exploitation-exploration framework, and combines sampling with actual query execution. We provide further details on our sampling strategy in Section 5 and compare the performance of the two approaches in Section 6.

Now, assuming that we know the score distribution for Q of the documents, we can estimate the benefit that each issued query will generate: we can estimate the distribution of document scores (with respect to Q) for the documents retrieved by a conjunctive query q. Therefore, we can estimate the *benefit* of a query q, defined as the probability that a randomly selected document from the answer of q will have score higher than the k-th ranked score for Q among the documents retrieved so far.

To achieve that, we create a priority queue with all candidate queries q, ordered by expected benefit. We select the query at the top of the priority queue, retrieve documents, and based on the results we update the expected benefits of the other queries. Then, we pick the query with the next-highest expected benefit and so on. The algorithm terminates when the benefit (i.e., probability of retrieving a top-k document) drops below a user-specified probability constant P. That is, the algorithm terminates when every unseen result has probability less than P to be in the top-k answer. Note that P is provided by a domain expert to balance response time and accuracy, and hence users do not have to worry about it in practice. In the next sections we describe in detail our approach.

5. http://www.ldc.upenn.edu/Catalog/docs/LDC2006T13/readme.txt

TABLE 1: Key Notation

Notation	Description
λ_t	tf parameter of Poisson for word t
λ_q^s	score (tf.idf) parameter of Poisson for query q
d	document
t	term in query Q
q	conjunctive query, subset of Q
$\mathcal{P}$	user-specified benefit threshold
PQ	priority queue
Z_q	set of fetched results by query q
S_q	size of Z_q
S	Total number of documents retrieved so far
$pr(q)$	benefit of query q

5 Exploration and Exploitation of the Database Contents through Querying

In this section, we describe the core of our approach. We show how we can use selective querying to:

(a) **Explore the database**: get the necessary statistics to estimate the parameters that our algorithms need; and
(b) **Exploit the database**: retrieve documents that are candidates for the top-k results of user query Q.

We will see how our scheme achieves both (a) and (b) in parallel.

5.1 Initial Probabilistic Modeling of the Source

Our overall goal is to figure out during our querying process, how many of the top-k relevant documents we have retrieved and how many are still unretrieved in the database. Unfortunately, we cannot be absolutely certain about these numbers unless we retrieve and score all documents: an expensive operation. Alternatively, we can build a probabilistic model of score distributions and examine, probabilistically, how many good documents are still not retrieved. We describe our approach here.

It is generally accepted that the term frequencies of the terms in a database tend to follow a Poisson distribution. In other words, for a word t, the probability that a randomly chosen document d from database D has term frequency r is:

$$Pr\{tf(t,d) = r\} = \frac{\exp(\lambda_t)}{r!} \cdot (\lambda_t)^r \tag{2}$$

where λ_t is a word-specific parameter. Now, instead of knowing the $n \times |D|$ *tf* values in the database, we only need to estimate n values: the λ_t values for each of the n words in the query Q.

Following this, the estimation of the score distribution is reduced to the problem of estimating the distribution of a sum (see Equation 1) of Poisson distributed variables. We know that if X and Y are two independent random variables following a Poisson distribution with parameters λ_x and λ_y respectively, then the sum $X + Y$ follows a Poisson distribution with parameter $\lambda_x + \lambda_y$. Therefore, in our case, the score distribution will also be, approximately,[6] a Poisson distribution with parameter:

$$\lambda_Q^s = \sum_{t \in Q} (\lambda_t \cdot idf(t)) \tag{3}$$

where we note that $idf(t) = ln\frac{|D|+1}{df(t,D)}$ according to Equation 1. We use the s superscript to denote the $tf.idf$ parameter as opposed to just the tf parameter.

This model implicitly assumes independence across terms. The assumption of independence across terms is admittedly not realistic, but used by many algorithms that deal with text, including many IR relevance models e.g., tf/idf and LM models, and many text classification algorithms. The independence assumption also tends to work in practice (see the work of Domingos and Pazzani [23] for a theoretical justification) and contributes to the tractability of our algorithms.

Now, given that we have a functional form for the score distribution, we can estimate the number of documents in the database that are expected to have scores higher than the currently retrieved documents. Suppose that our currently retrieved top-k documents have a cutoff score τ. (We can estimate the exact similarity value for these documents since we have retrieved them locally.) We are trying to estimate how many documents in the database have score higher than τ:

$$\begin{aligned} |Docs(score > \tau)| &= |D| \cdot Pr\{score > \tau\}) && (4) \\ &= |D| \cdot Pr\{Poi(\lambda_Q^s) > \tau\} && (5) \\ &= |D| \cdot \left(1 - \frac{\Gamma(\lfloor \tau + 1 \rfloor, \lambda_Q^s)}{\lfloor \tau \rfloor !}\right) && (6) \end{aligned}$$

where $\Gamma(a, x) = (a-1)! e^{-x} \sum_{r=0}^{a-1} \frac{x^r}{r!}$ is the incomplete Gamma function.

We now have an estimate for the number of documents that have similarity above a given threshold τ. Of course, before proceeding further, we need to estimate the λ_t parameters required to compute the λ_Q^s parameter for the score distribution. Next, in Section 5.2, we describe an approach that relies on "static" query-based sampling [21]. Then, in Sections 5.3 and 5.4, we describe our approach that simultaneously explores the database *and* retrieves as many good documents as possible at the same time.

6. Strictly speaking, the *weighted* sum of two Poisson random variables is a quasi-discrete distribution, which technically cannot be called Poisson. However, in practice, it behaves like a continuous version of the discrete Poisson distribution.

5.2 Summary-based Estimation of Poisson Parameters

One strategy for generating estimates for the λ values is to generate a static document sample from the database, and then use the retrieved documents to generate the estimates. For example, Callan et al. [21] generate a summary from the database by sending random keyword queries and retrieving 300 documents. (By random, we mean queries with *any* word, not only queries with words from the issued user query Q.)

Given such a document sample, we can measure the $tf(t, d)$ values for each term t and document d, and use the *maximum likelihood estimate* (MLE) to compute estimates for the λ_t values. To avoid zero estimates for terms that do not appear in the sample, we use *Laplace smoothing*:[7]

$$\widehat{\lambda}_t^{\mathrm{MLE}} = \frac{1 + \sum_{i=1}^{S} tf(t, d_i)}{S + 1} \tag{7}$$

This strategy tends to have a few shortcomings. First, the estimates assume that query-based sampling is equivalent to random sampling, an assumption that does not necessarily hold [24]: there is a bias to retrieve more often longer documents, or documents with higher priority in the underlying ranking function (e.g., more recent documents in date-based ranking). Second, the estimates for the terms that were sent to the database as query probes are significant overestimates of the real values as by definition the retrieved documents contain *only* documents with the submitted terms. Third, and more importantly, there is a very significant data sparseness issue: many terms do not appear in the retrieved document sample and their estimates are simply the Laplacean-corrected values.

Next, we describe an alternative approach that compensates for the data sparseness by retrieving document samples through a sampling process customized to the issued query Q (Section 5.3). Then, we show how to compensate for the overestimates introduced by the very nature of the query-based sampling (Section 5.4). As we will see, this exploitation-biased strategy tends to be slightly more expensive than the summary-based strategy (as it generates customized document samples on the fly, instead of having a static summary shared by all queries) but generates results of superior quality.

5.3 Exploitation-biased Query-based Estimation of Poisson Parameters

In the previous section, we have described a query strategy in which we were sending random queries for sampling. Now, we describe an approach in which the sampling queries involve only the actual query words, biasing the retrieval of documents towards *beneficial* documents. In parallel, this query strategy avoids the issue

7. We can also use the Bayesian estimator instead of the MLE one; for brevity, we do not present this variant here, since the differences are small and restricted at the very first stages of the estimation.

of data sparseness by generating estimates specifically for the query at hand. (We describe later the exact query formulation strategies.) However, such a query strategy generates biases in the sampling, which affect the basic MLE estimation. So, we show now how to compensate for these biases.

Suppose that we submit as query the word t that appears in the user query Q. We need to estimate properly the parameter λ_t. In this case, the results that we received back are *not* a random sample, so we cannot use directly the method described above. Instead, now all the returned documents are guaranteed to contain the word t. That is, the returned document results are a "conditional" random sample, with the condition that $tf(t,d) > 0$. So, our retrieved sample misses all the documents that do not contain t. Therefore, our calculations need to account for this fact. Hence, we estimate, given the retrieved documents for query t, how many empty documents we would have seen if we were performing random sampling.

Suppose that after submitting the query t, we retrieve and process a set of S documents. For the word t we also know its document frequency $df(t)$ in D. So there are $df(t)$ documents in the database that contain t and $|D| - df(t)$ that do not contain t. Therefore, for every document with t that we retrieve from D, we expect to have $\frac{|D|-df(t)}{df(t)}$ documents without t (i.e., with $tf(t,d) = 0$). So, we modify the estimator from the previous section to account for the unseen documents that do not contain t and the sample size from S becomes $S' = S + \frac{|D|-df(t)}{df(t)} \cdot S = \frac{|D|}{df(t)} \cdot S$. So, by changing the normalizing factor $1/S$ to $1/S'$ in Equation 7, we have:

$$\widehat{\lambda}_t^{\mathrm{MLE}} = \frac{1}{S} \sum_{i=1}^{S} \frac{df(t)}{|D|} \cdot tf(t,d) \tag{8}$$

The key observation in Equation 8 is that when we update the MLE estimates for the term t using results from a query that contains t, we should scale the estimates using the $\frac{df(t)}{|D|}$ factor.

Below, in Procedure *updateLambdaByMLE*, we present the algorithm to update the current λ_t estimations after a conjunctive query q is submitted, which produces a set of S_q results. S is the number of results retrieved so far by all submitted conjunctive queries. Note that for simplicity we use the notation λ_t instead of λ_t^{MLE} in the rest of the analysis.

5.4 Query-based Document Score Distribution

In the previous section, we described how to adjust the λ_t estimates to compensate for the bias introduced through query-based sampling. Given these estimates, we can generate the distribution of similarity scores for the user-submitted query Q *in the database*. However, we are not going to retrieve documents randomly from the database. Instead, we submit a sequence of conjunctive queries trying to retrieve the most highly similar documents. So, to identify which queries will retrieve the most similar documents, we need to estimate the score distribution for the *query results* for a given query q, which is not necessarily the same as the original user query Q. We now describe how to compute the *score distribution of the query results* for any conjunctive query q.

Procedure `updateLambdaByMLE(q)`

```
1  Let $Z_q$ be the results set for $q$. It is $S_q = |Z_q|$.
2  foreach $t \in Q$ do
3      if $t \in q$ then
4          $\lambda_t^{prev} \leftarrow \lambda_t^{prev} + \frac{df(t)}{|D|} \cdot tf(t, Z_q)$
5          // $\lambda_t^{prev}$ is an extra variable we need to keep
           for each $t$,    given that $\lambda_t$ stores the
           averaged value.
6          //$tf(t, Z_q) = \sum_{d \in Z_q} tf(t,d)$
7      end
8      else if $t \in Q \setminus q$ then
9          $\lambda_t^{prev} \leftarrow \lambda_t^{prev} + tf(t, Z_q)$
10     end
11     $\lambda_t \leftarrow \lambda_t^{prev}/S$   //$S$ is the total number of results
       retrieved so far by all queries
12 end
```

In Section 5.1, we gave a functional form for the score distribution, assuming that we get a randomized sample from the database. However, when the documents that we examine are retrieved by using a query q that contains some of the terms in the original user-issued query Q, then the retrieved document sample is biased: a conjunctive query guarantees that the returned documents have $tf(t,d) > 0$ when $t \in q$. In this case we have:

$$\begin{aligned} Pr\{tf(t,d) = r | tf(t,d) > 0\} &= \frac{Pr\{tf(t,d) = r, r > 0\}}{Pr\{tf(t,d) > 0\}} \\ &= \frac{\frac{\exp(\lambda_t)}{r!} \cdot (\lambda_t)^r}{1 - Pr\{tf(t,d) = 0\}} \\ &= \frac{\frac{\exp(\lambda_t)}{r!} \cdot (\lambda_t)^r}{1 - \exp(\lambda_t)} \end{aligned} \tag{9}$$

In other words, the new *tf* distribution is a Poisson distribution with a normalizing factor $\frac{1}{1-\exp(\lambda_t)}$. Therefore, when we send a query q to the database, the document score distribution (the score is always defined with respect to the user query Q) follows the Poisson distribution with a configuring parameter λ_q^s:

$$\lambda_q^s = \sum_{t \in q} \left(\frac{\lambda_t}{1 - \exp(\lambda_t)} \cdot idf(t) \right) + \sum_{t \in Q \setminus q} (\lambda_t \cdot idf(t)) \tag{10}$$

which is different that the functional form depicted in Equation 3. Following the analysis from Section 5.1, the number of documents in the results of a *conjunctive* query q, with score above a threshold τ are:

$$\begin{aligned} |Docs(score > \tau)| &= S_q \cdot Pr\{score > \tau\}) \\ &= S_q \cdot Pr\{Poi(\lambda_q^s) > \tau\} \\ &= S_q \cdot \left(1 - \frac{\Gamma(\lfloor \tau + 1 \rfloor, \lambda_q^s)}{\lfloor \tau \rfloor !}\right) \quad (11) \end{aligned}$$

This analysis gives us the basis for formulating our querying strategy, which we describe next.

5.5 Top-k Querying Algorithm

Above we presented the estimation for a general query-based approach, without specifying how to select queries to send. However, we know now the expected score distribution for each conjunctive query, and how these estimates are updated every time that we retrieve a new document (Procedure *updateLambdaByMLE* above).

Our querying strategy is as follows. We start by sending all the terms of the query as a conjunctive query to the database. This query is expected to retrieve the documents with the highest scores. Obviously, if the query matches less than k documents, we need to submit relaxed versions of the query (e.g., remove one keyword – we describe below how to select which query to submit). If we have more than k results, we still need to compute the confidence that the retrieved documents contain the correct top-k results. (Note that a document with fewer query keywords may achieve higher ranking according to an IR function.)

Since we do not have access to the complete database, we cannot be absolutely certain that we retrieved all the "real" top-k documents. Instead, we adopt a probabilistic approach, and we use an input parameter $\mathcal{P}$ which is the probability that any unseen document belongs on the top-k results is less than $\mathcal{P}$. That is for all the (unseen) documents d with relevance $score(d, Q)$, we have:

$$Pr\{score(d, Q) > \tau\} < P \qquad (12)$$

where as τ we set the relevance score of the k-th highest scoring document retrieved so far. (See Equations 11 and 10 to see how to compute the value $Pr\{score(d, Q) > \tau\}$. Notice that we are trying to estimate the distribution of scores for the user-submitted query Q, and we retrieve the documents by sending a set of conjunctive queries q_i that contain only a subset of the terms from Q.)

Hence, at every step we compute the *benefit* of every candidate query, which is $Pr\{score > \tau\}$, and is computed as shown in Section 5.4. If for all candidate queries the *benefit* is less than $\mathcal{P}$, the algorithm terminates. Else, the query q with the maximum *benefit* is submitted. Then the λ_t parameters estimations are updated and the process repeats.

We maintain a priority queue with the expected benefit of each query so we can select which query to issue next. The main algorithm is shown in Procedure *QueryExecution*.

Procedure `QueryExecution`(k, Q, $\mathcal{P}$)

```
1  Initialize:
2  - Add to priority queue $PQ$ all combinations $q$ of terms
   of $Q$, that is, $PQ$ has all candidate conjunctive
   queries; $PQ$ is ordered by the benefit $pr(q)$ of $q$
3  - Default $\lambda_t$ parameters are assigned to each $t \in Q$,
   and accordingly initial benefits $pr(q)$ for each $q \in PQ$
   are computed;
4  - Create results array $R$ with size k, where results are
   ordered by score;
5  while $PQ \neq \emptyset$ do
6      $q \leftarrow PQ.pop()$
7      if $pr(q) < \mathcal{P}$ and $R$ contains $k$ results then
8          break
9      end
10     else
11         $Z_q \leftarrow$ Fetch($q$)
12         $S \leftarrow S + S_q$   // S is the number of
               documents retrieved so far.
               $S_q = |Z_q|$
13         Insert $Z_q$ into $R$   // $Z_q$ and $R$ are
               merged into $R$
14         UpdateLambdaByMLE($q$)
15         UpdatePQ ($q$)
16     end
17 end
18 return $R$
```

As mentioned in Section 4, a practical issue that we face is that we may retrieve the same documents many times as we issue the queries. From an estimation point of view, we should always include such documents in the updating of the estimates. Practically though we do not want to retrieve the same documents multiple times so that we can save the retrieval and processing cost. We can achieve this by either adding negations of the previously issued queries in the submitted query or by simply not retrieving documents with document ids identical to previously retrieved ones. In our querying technique we use the negation trick: The process is as follows. We save the set of documents Z_q for each past query q, and then for a new query q_{new}, we do the following:

- Let $L = q_1, ..., q_l$ be the set of past queries for which $q_{new} \subset q_i$.
- Submit $q'_{new} = q_{new} - q_1, ..., q_l$, i.e., q_{new} augmented by the negation of all previously submitted queries (to avoid retrieving documents retrieved in the past).
- The result of q_{new} for our probabilistic analysis purposes is $Z_{q_{new}} = Z_{q'_{new}} \cup Z_{q_1} \cup ... \cup Z_{q_l}$

Candidate Queries Lattice The sequence $q_1, q_2, \cdots, q_v$ of queries that the algorithm submits can be viewed as a prefix of the queries lattice, shown in Figure 1. Except for the most selective, vanilla conjunctive query that returns the AND of all terms (the top of the lattice), it is not clear which query is going to be the "next most beneficial" (See Figure 1). Identifying which are the "most beneficial" (or "most selective") queries is part of

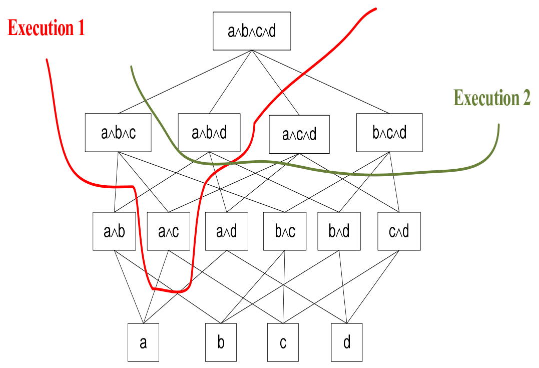


Fig. 1: Lattice of candidate conjunctive queries.

Procedure `UpdatePQ(q)`

```
1 foreach $(pr, q)$ in $PQ$ do
2     $\lambda_q^s = \sum_{t \in q} F\left(\frac{\lambda_t}{1-\exp(\lambda_t)}, df(t), avdl\right) + \sum_{t \in Q \setminus q} F(\lambda_t, df(t), avdl)$
      // according to Equation 10
3     $pr(q) \leftarrow 1 - \mathsf{IGF}(\tau, \lambda_q^s)$ // IGF stands for
          incomplete gamma function
4 end
```

our algorithm, as described above. Our algorithm always executes a prefix of the lattice (the upper queries of a cut are executed) because the parent always has higher benefit $pr(q)$ than a child: the benefit increases monotonically with the number of keywords. Two possible executions of a user query are shown in Figure 1. The execution of the algorithm can be viewed as a cut going down the lattice. Hence, in the priority queue we only need to keep the candidate queries that are just below the current cut.

Total vs. Block Variants of the Algorithm In the above description, the algorithm submits a query q at a time and retrieves all its results. That is, the granularity is a whole query. We refer to this variant as *Total*. Alternatively, we could refine the granularity by only retrieving B results at a time from the results of a query q if $B < |Z_q|$ (like line 11 in Procedure `QueryExecution`). Then, the probabilities are updated as usual and the rest of the query is placed back to the priority queue. We refer to this variant as *Block*. These variants are compared experimentally in Section 6 for various block sizes B.

6 Experiments

We experimentally evaluate the performance and quality of the retrieval algorithms. We compare the Query-based probability estimation strategy described in Section 5.4 to the Summary-based estimation strategy of Section 5.2, and also consider the Total vs. Block variants of the top-k querying algorithm of Section 5.5. For that, we compare the following algorithm variants:

- *Baseline*: This algorithm submits the disjunction of all query keywords to the database and retrieves *all* matching results. Documents that do *not* match this disjunctive query, and hence are not returned, are guaranteed to have zero *tf.idf* score. Then this algorithm computes the IR score for each document, and returns the *true* top-k to the user. Therefore, this algorithm is guaranteed to generate a perfect ranking, at the expense of a significant cost of downloading all documents before ranking them.
- *Blind*: This algorithm is a simplified version of the Query-based algorithm. The Blind algorithm does not use the accumulated statistics about the *tf* frequency of the terms in the database. Insteadm Blind submits a "static" sequence of conjunctive queries, based only on the global document frequencies of the terms. Blind initially submits the conjunction of all n terms. Next, the queries with $n-1$ terms are submitted, sending first the queries that do not include the term with the highest document frequency (i.e., do not include the term with the low idf), and so on.
- *Summary-based*: Our "Total" algorithm with summary-based probability estimation of the λ's.
- *Query-based*: Our "Total" algorithm with query-based probability estimation of the λ's.
- *Block-based*: Our "Block" algorithm with query-based probability estimation of the λ's.

Note that we do not show results for a Block variant with summary-based estimation, because our experiments show that the Block variant is worse than the Total variant, and also that Summary-based is worse than Query-based estimation.

6.1 Experimental Setup

Configuration: All experiments were run on a PC with a 2.5G Intel quad-core processor with 4G RAM running Windows XP SP2. The algorithms were implemented in Java.

Datasets: We ran our algorithm on three real datasets shown in Table 2, two "Local" and one "Remote."

The first Local dataset is the "PMC Open Access Subset" (*LocalPubMed*)[8] dataset, which is a subset of PubMed which comprises of 117,860 open-access articles, with the full text available for download. All of the documents are XML files. The second Local dataset is the TREC Disk 1-5 dataset (*LocalTREC*)[9], which comprises of over three million articles from newspapers and government agencies. We used Lucene[10] to index every article in the two Local datasets. Note that Lucene allows IR ranking of the documents, but we assumed this feature is not

8. http://www.pubmedcentral.nih.gov/about/openftlist.html
9. http://trec.nist.gov/data/test_coll.html
10. http://lucene.apache.org/

TABLE 2: Dataset detail

Dataset	Type	Total # of doc.
PMC Open Access	Local	117,860
TREC text collections	Local	3,000,000
PubMed Portal	Remote	17,000,000

TABLE 3: Sample queries from the *LocalPubMed* dataset

#Query	*df*
cancer promoter	6280
cystic fibrosis	5285
immunodeficiency virus structure	8341
flanking SNPs gene	13276
alveolar carcinoma cell TDs	10051
PCR DMD BMD exon	2546
climate Control Prevention change impacts	7001
DNA mRNA overlapping isolated cDNA	19787

TABLE 4: Sample queries from the *Remote* dataset

#Query	*df*
Herniotomies orchidopexy	645
NTBC Fah	363
NTBC fumarylacetoacetate Fah	435
tetrahydropyridines thiazolidinones carboxaldehydes	242
FAH NTBC tyrosinemia FAA	2032
MAAI DCA dichloroacetate NTBC	2364
MPI MAAI polyaromatic Ralstonia gentisate	5447
pEGFP SSBs CCCs fuma Sakashita	1953

TABLE 5: System parameters

Parameter	Range
Probability threshold ($\mathcal{P}$)	0.01, 0.1, 0.2, $\cdots$, 0.5
Result cardinality (k)	1, 10, 50, 100
Keyword cardinality (#keywords)	2, 3, 4, 5
Block-based size (infinity for Query-based)	100, 500, 1000, 2000

available in this experiment. Instead, we set Lucene to return the documents ordered by date.

The Remote dataset, which is more appropriate for this paper's motivation, is the whole PubMed, which can only be remotely accessed through PubMed Web access utility services (*RemotePubMed*).[11] We only retrieve the abstracts of the articles since the body of many articles is missing from PubMed. Note that PubMed does not offer any form of relevance-based ranking. All results are ranked by date.

For LocalPubMed, we picked 60 queries that have been used as exercises to train bioinformatics information specialists [8]. Then, we separated the queries into two sets of 30 queries each: "frequent" and "infrequent" based on the number of results that they generated when evaluated on the web interface of PubMed. Due to restrictions imposed by the web interface of PubMed, we could only use the "infrequent" queries with the Remote dataset. This is the result of the restrictions imposed by PubMed, which does not allow massive downloads of documents over the web service interface. Therefore, we could not fully evaluate and retrieve *all* the returned documents for the "frequent" queries and, hence, we could not generate the baseline against which to evaluate the quality of the results. (We *could* use our algorithms that retrieve significantly less documents, but we would not be able to evaluate the results in terms of quality.) For our LocalPubmed dataset we use *both* frequent and infrequent queries. For LocalTREC, we used 60 english test questions from the TREC website[12] as our queries. For these queries our baseline is the relevance ranking provided by the TREC relevance judgments. Tables 3 and 4 show a sample of the queries submitted to the Local and Remote datasets, respectively.

Quality Measure: We measure the quality of the algorithms as follows: we first execute the Baseline algorithm to compute the optimal top-k results. Then, we measure the quality of Query-based and Block-based algorithms by comparing their top-k search results to this optimal list generated by the Baseline algorithm. We compare two top-k lists using the normalized top-k Spearman's Footrule metric [25].

Table 5 summarizes the parameters varied in our experiments, along with their ranges.

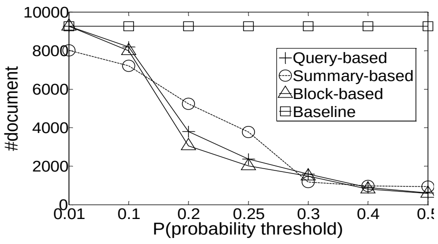
6.2 Experiments on Local Datasets

Varying $\mathcal{P}$: First, we examine the effect of $\mathcal{P}$ in the performance of our algorithms. $\mathcal{P}$ is the parameter that defines the confidence that the returned results are close to the optimal. Smaller values of $\mathcal{P}$ mean that the algorithms tries harder to approximate the optimal list, while large values of $\mathcal{P}$ mean that the algorithm can stop earlier, returning more rough approximations of the optimal list.

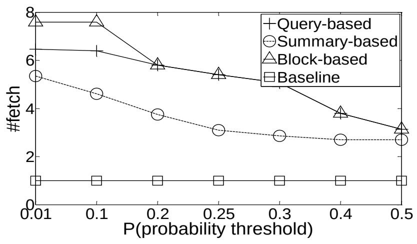
In Figures 2 and 3 we set the number of keywords to 3 and fix $k = 50$. For Block-based algorithm, we set the Block size to 2000. We vary $\mathcal{P}$ from 0.01 to 0.5. Figures 2(a) and 3(a) show that Summary-based, Query-based and Block-based fetch fewer documents as $\mathcal{P}$ grows. We observe that Block-based retrieves slightly fewer documents but submits more conjunctive queries compared with Query-based (called fetches in Figures 2(b) and 3(b)). As expected, Summary-based retrieves the least documents in most cases. (As discussed in Section 5.2, the summary-based algorithm retrieves 300 documents for the initial document summary to generate the estimates but we do not include this one-time cost in the reported results.) Moveover, in Fig 2(b) we see that for $\mathcal{P} \geq 0.2$, Query-based and Block-based coincide, because the number of the documents Block-based fetches is less than Block size B. The same phenomenon also happens in Fig 3(b) for $\mathcal{P} \geq 0.25$.

Overall, in terms of efficiency, all algorithms perform better than Baseline because Baseline fetches all the documents in the results before reranking them. For $\mathcal{P} = 0.01$, Query-based is slightly faster than Block-based because they both retrieve the same number of documents (see Figure 2(a)) but Block-based needs more

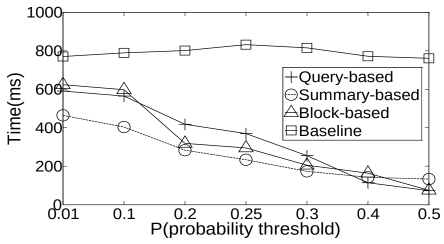
11. http://www.ncbi.nlm.nih.gov/entrez/query/static/eutils_help.html
12. http://trec.nist.gov/data/testq_eng.html



(a) number of doc. vs. $\mathcal{P}$

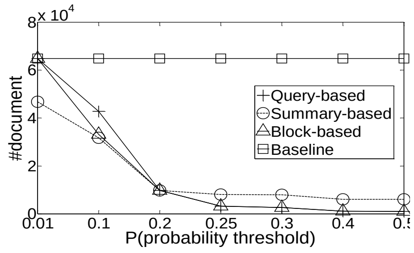


(b) number of fetches vs $\mathcal{P}$

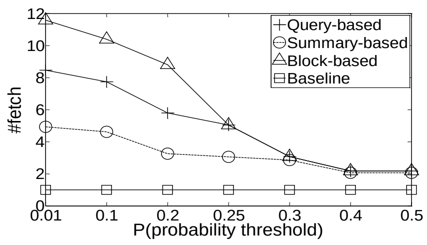


(c) time vs. $\mathcal{P}$

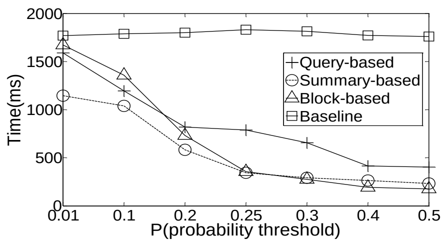
Fig. 2: LocalPubMed: Varying $\mathcal{P}$



(a) number of doc. vs. $\mathcal{P}$

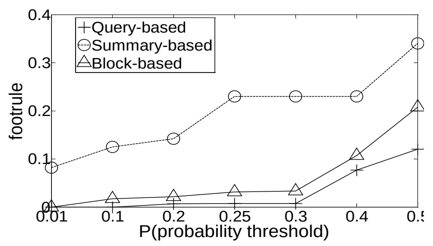


(b) number of fetches vs $\mathcal{P}$

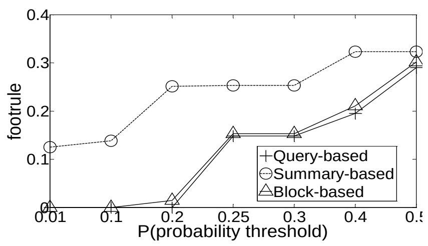


(c) time vs. $\mathcal{P}$

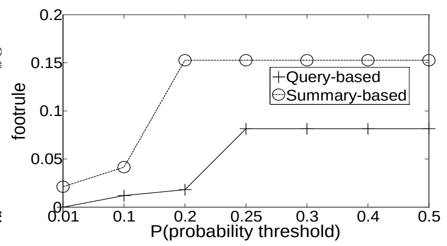
Fig. 3: LocalTREC: Varying $\mathcal{P}$



(a) LocalPubMed



(b) LocalTREC



(c) Remote

Fig. 4: Footrule vs. $\mathcal{P}$

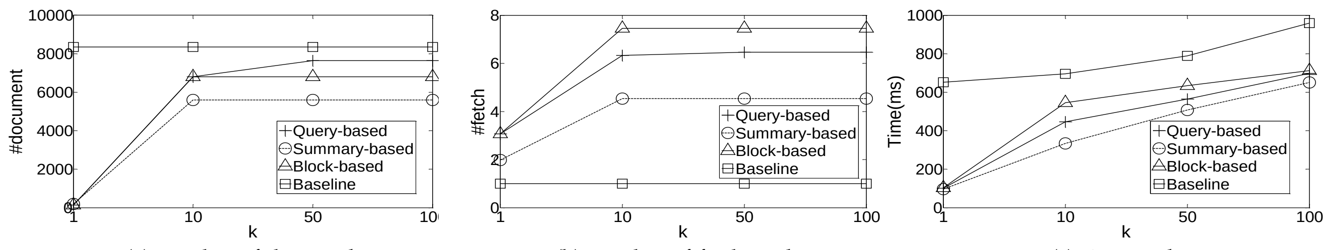
fetches which incur an overhead. Summary-based is the fastest because it performs the fewest fetches (queries) and also the lambda estimation is performed off-line.

Although the Summary-based algorithm is the most efficient, we observed that the speed comes at the expense of the quality of the results. In terms of quality, Figures 4(a) and 4(b) show that both Query-based and Block-based achieve excellent Footrule values for $\mathcal{P}$ up to 0.3 (for LocalPubMed) or 0.2 (for LocalTREC) while Summary-based is the worst in all cases as expected: this is the result of the rough probability estimates.

In the rest of this section, due to space constraints, we only report the results for LocalPubMed, given that the results of LocalTREC follow similar trends.

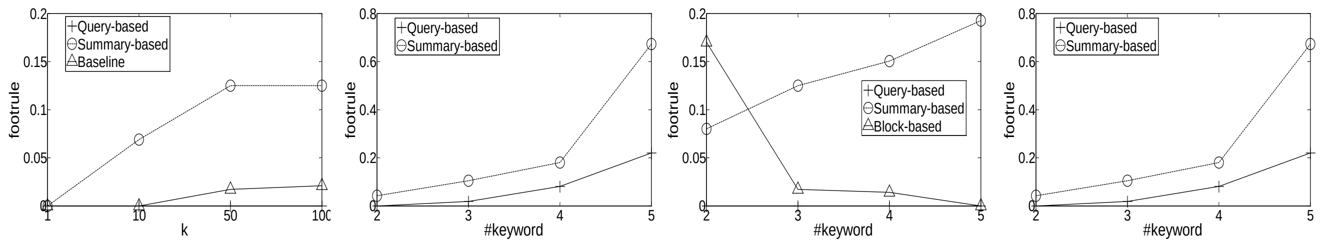
Varying k: Next, we set the number of keywords to 3, $\mathcal{P} = 0.1$, and vary k from 1 to 100, as shown in Figure 5. As displayed in Figure 5(a), the number of fetched documents increases with k for Summary-based, Block-based and Query-based, as expected: with small k we can easily retrieve "a few good documents" but when k increases the task of locating all similar documents becomes increasingly harder. Furthermore, observe that the number of documents grows slowly from $k = 10$ to $k = 100$ but fast from $k = 1$ to $k = 10$. The reason is that very few documents have very high relevance score, as expected from the Poisson distribution of the similarity scores, but after that the similarity threshold does not change as drastically with k. The observations for the number of documents naturally carry for number of fetches in Figure 5(b). As shown in Figure 5(c), the execution time of the four algorithms increases with k. For Baseline, this is because it has to compute the top-k results from all retrieved results. Query-based is slightly faster than Block-based when $k = 10$, because both algorithms fetch the same number of documents when $k = 10$ (Figure 5(a)) but Query-based needs fewer fetches. Summary-based is faster than other three algorithms, because it performs fewer fetches and retrieves fewer documents, as we explained above in the "varying $\mathcal{P}$" paragraph. The quality results are also similar: As shown in Figure 6(a), Query-based has perfect accuracy, whereas Block-based's accuracy decreases slightly as k increases. Summary-based is the most efficient but again has the worst accuracy as measured by the footrule distance.

Varying the number of keywords: Figure 7 depicts



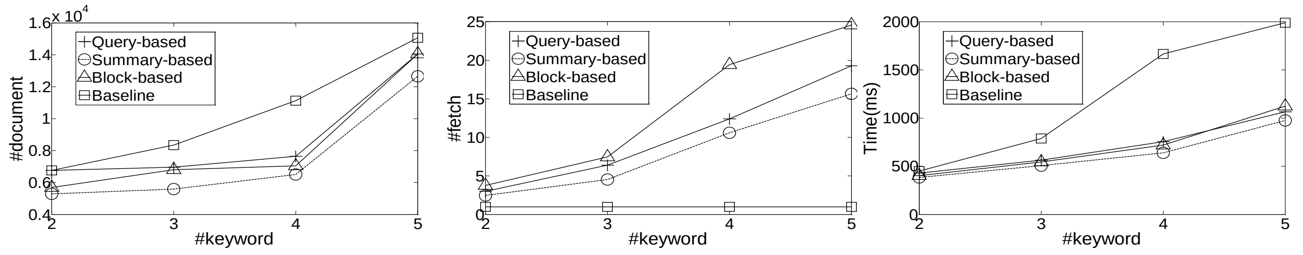
(a) number of doc. vs. k (b) number of fetch vs. k (c) time vs. k

Fig. 5: LocalPubMed: Varying k



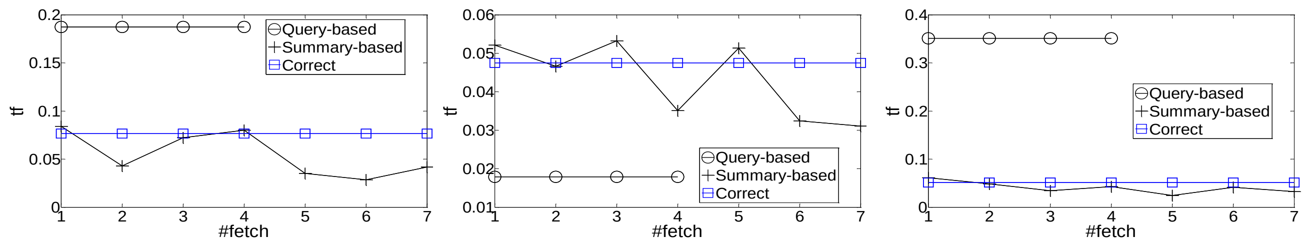
(a) LocalPubMed (fr vs. k) (b) Remote (fr vs. k) (c) LocalPubMed (fr vs. #kwd) (d) Remote (fr vs. #kwd)

Fig. 6: Footrule vs. k and #kwd



(a) #document vs. #keyword (b) #fetch vs. #keyword (c) time vs. #keyword

Fig. 7: LocaLPubMed: Varying number of keywords.

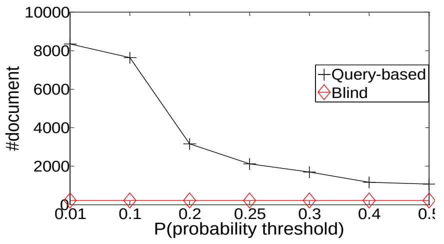


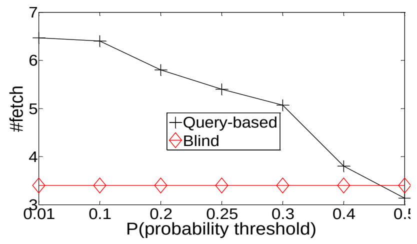
(a) #fetch vs. tf: disease (b) #fetch vs. tf: genetic (c) #fetch vs. tf: treatment

Fig. 8: Estimation of tf vs. #fetch (LocalPubMed)

the results for different number of keywords for two local datasets. In this experiment we fix $k = 50$ and $\mathcal{P} = 0.1$. As shown in Figure 7(a), Query-based fetches slightly more documents in most cases. An exception is $\#keywords = 5$ where both Query-based and Block-based retrieve about the same number of documents because each executed conjunctive query has fewer results than the Block-based size. (Note that conjunctive queries with more keywords return fewer results.) As shown in Figure 7(b), the number of fetches for all methods increases fast because the number of keyword combinations grows exponentially with the number of keywords.

In terms of quality, interestingly, as we see in Figure 6(c), the performance of Summary-based degrades as the number of keywords increases. The reason is that for more keywords, the number of candidate conjunctive queries explodes and hence the inaccurate parameter estimation of Summary-based leads often to bad query choices. In contrast, we see that the performance of the Block-based algorithm increases with number of keywords, because the submitted conjunctive queries become more selective and the correct top results often appear in these very focused queries.


(a) number of doc. vs. $\mathcal{P}$


(b) number of fetch vs. $\mathcal{P}$

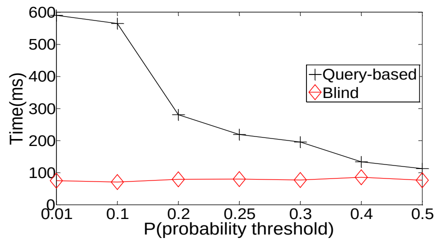
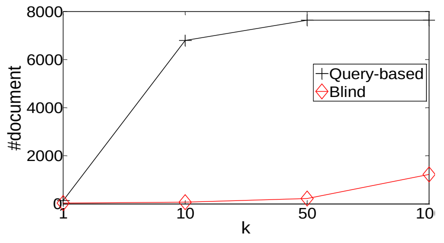
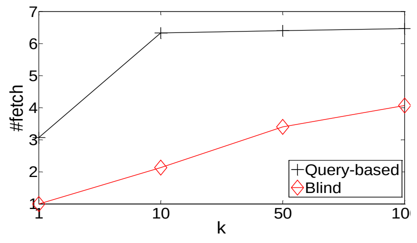

(c) time vs. $\mathcal{P}$

Fig. 9: Blind Algorithm on LocalPubMed Dataset: Varying $\mathcal{P}$


(a) number of doc. vs. k


(b) number of fetch vs. k

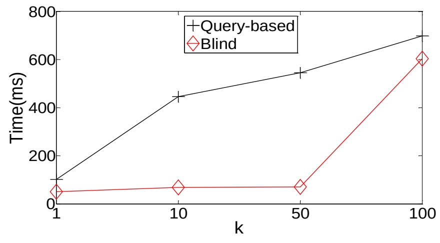
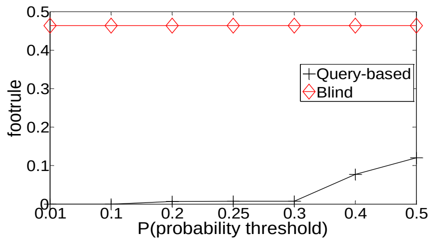

(c) time vs. k

Fig. 10: Blind Algorithm on LocalPubMed Dataset: Varying k


(a) Footrule vs. $\mathcal{P}$

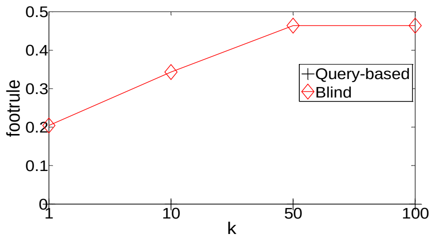
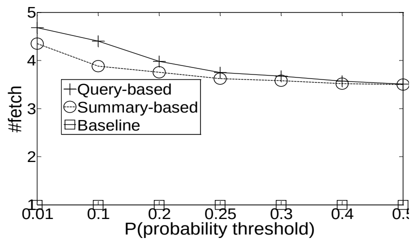
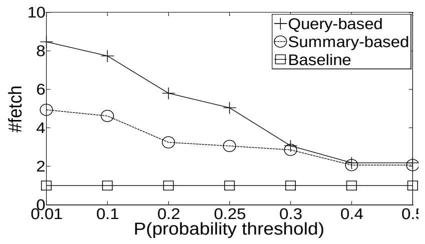

(b) Footrule vs. k

Fig. 11: Quality Evaluation of Blind Algorithm: Footrule vs. $\mathcal{P}$ and k


(a) number of doc. vs. $\mathcal{P}$


(b) number of fetch vs. $\mathcal{P}$

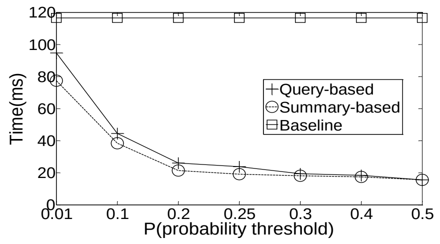
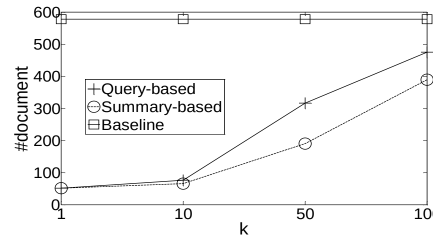

(c) time vs. $\mathcal{P}$

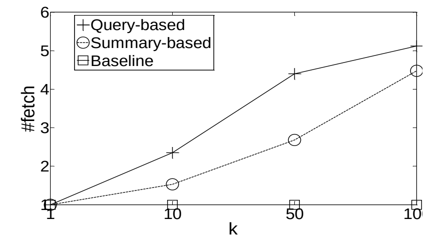
Fig. 12: Remote Dataset: Varying $\mathcal{P}$

Varying Block Size: In Table 6, we set the number of keywords to 3, $k = 50$, $\mathcal{P} = 0.1$, and we measure the performance of Block-based algorithm by varying Block size $B = 100$ to $B = 2000$. Note that Query-based algorithm is equivalent to Block-based when Block size is infinity. The number of fetched documents increases with Block size, since Block-based algorithm can stop earlier if Block-based size is smaller. As expected, the number of fetches decreases as Block size increases. The time of Block-based algorithm decreases with increasing Block size, even though the number of retrieved documents slightly increases. This is because of the overhead incurred by each fetch, which includes the query overhead and the additional tasks for each fetch, like updating the estimated frequencies. In terms of quality, as Block-based size increases the Footrule of Block-based drops, because more results are retrieved, as expected.

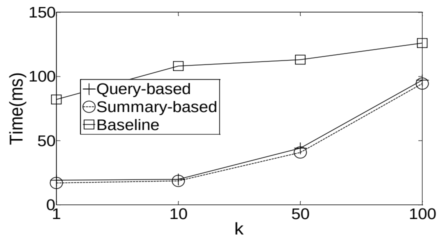
Compare *tf* estimations of Summary-based vs. Query-based: In the above experiments we showed that the quality of the Summary-based variant is consistently worse than the Query-based variants. The main reason is that Summary-based does not estimates as accurately the λ parameters, which intuitively means that it does not estimates as accurately the expected tfs of the query



(a) number of documents vs. k

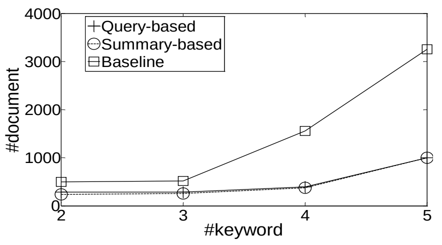


(b) number of fetch vs. k

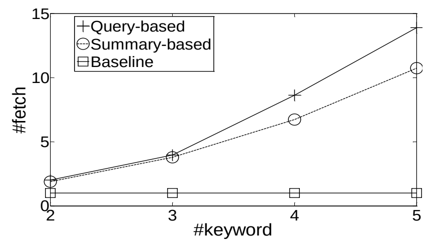


(c) time vs. k

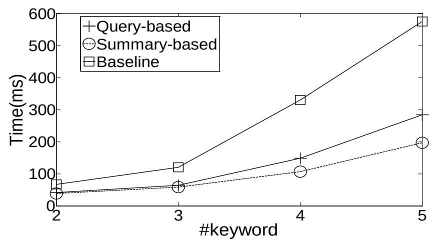
Fig. 13: Remote Dataset: Varying k



(a) #doc vs. #keywords



(b) #fetch vs.#keywords



(c) time vs. #keywords

Fig. 14: Remote Dataset: Varying #keywords

TABLE 6: Varying Block size

	Block size				
	100	**500**	**1000**	**2000**	∞
#doc	8340	8440	8500	8870	9540
#fetch	89	20	18	15	12
time(ms)	4352	1512	1351	899	878
Footrule	0.034	0.031	0.029	0.026	0.00

words. Given the fact that LocalPubMed dataset shares many aspects with LocalTREC dataset, here we just use LocalPubMed as our testbed to verify this fact.

Figure 8 shows a qualitative depiction for $k = 50$ and $\mathcal{P} = 0.1$, for the 3-keyword queries: `genetic`, `disease` and `treatment` on the local dataset. We compare the *tf* estimations of the two methods to the correct values, which are calculated off-line by scanning the complete dataset and using the Maximum Likehood Estimation (MLE) formula (Eq. 7). We see that Query-based is consistently better than Summary-based for the reasons explained in Section 5.2.

Evaluate Blind algorithm: We compare the performance of Blind and Query-based algorithms on LocalPubmed dataset by varying P and k. The results are shown in Figures 9, 10 and 11. We note that for P=0.5, the times of Blind and Query-based are very close (See Figure 9(c)), but Query-based has about 4 times better quality, according to the Footrule (Figure 11(a)). This shows that Blind is clearly inferior to Query-based; this is why it was not included in the previous graphs.

6.3 Remote Dataset

Given the graphs of Section 6.2, we conclude that the Query-based algorithm is generally better than Block-based because for a slightly higher execution time, it leads to considerable quality improvement. Hence, we only consider Query-based and Summary-based in this section.

In Figures 12, 13 and 14 we repeat the above experiments on the Remote database for varying $\mathcal{P}$, k and number of keywords, respectively. Due to the characteristics of the remote dataset, which are the much larger size and the slow query response times, we observe the following key differences from the results on the Local dataset.

As shown in Figure 4(c), the Footrule has much less variation than in Figure 4(a) because the number of retrieved documents has much smaller variation with $\mathcal{P}$. The reason for the latter is that the queries we used in the Remote dataset have more infrequent keywords as we explain in Section 6.1.

Also, in Figure 6(b) we see that the Footrule decreases with k, in contrast to Figure 6(a) where it was 0 for Query-based and increasing for Block-based. The reason is that, as we see in Figure 13(a), the number of retrieved documents increases dramatically with k, which was not the case for the Local dataset (Figure 5(a)). The reason for the latter fact is that the Remote dataset has much more documents. When increasing the #keywords, in Figures 6(d) and 6(c), we observe that all algorithms are stable or degrade, since the search space increases. The only exception is the Block-based for LocalPubMed, which improves because it reads a too small number of documents for small #keywords (Figure 7(a)).

6.4 Discussion

Generally, as we have seen in previously reported experiments, the Summary-based variant is slightly faster than the Query-based variant. On the other hand, Query-based is more accurate since its estimation strategy is

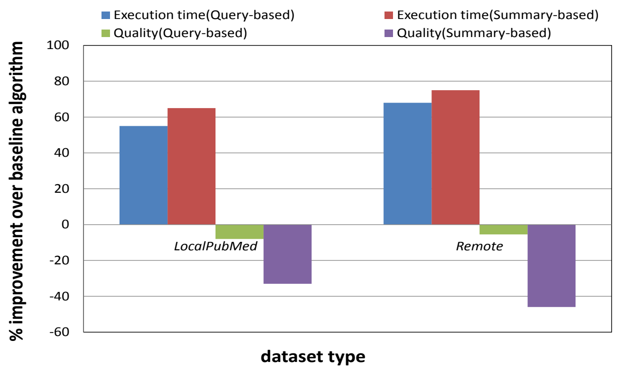


Fig. 15: Query cost vs. Quality

better. In this section, we combine previous results to give a general picture of two methods and show the time vs. quality tradeoffs. We use the results of 3-keyword queries for this analysis.

Figure 15 illustrates the time and quality improvement of the Query-based and Summary-based algorithms for the LocalPubMed and Remote datasets. We see that Summary-based has a very slight advantage in terms of execution time at the expense of a considerable disadvantage in terms of quality.

7 Conclusions

We presented a framework and efficient algorithms to build a ranking wrapper on top of a documents data source that only serves Boolean keyword queries. This setting is common in various major databases today, including PubMed and USPTO. Our algorithm submits a minimal sequence of conjunctive queries instead of a very expensive disjunctive one. The query score distributions of the candidate conjunctive queries are learned as documents are retrieved from the source. Our comprehensive experimental evaluation on the PubMed database shows that we achieve order of magnitude improvement compared to the baseline approach. We found that applying tf probabilistic estimation techniques and processing a whole conjunctive query at a time (without splitting it to blocks) lead to better performance.

8 Acknowledgement

Vagelis Hristidis was partially supported by NSF grant IIS-0811922 and DHS grant 2009-ST-062-000016. Panagiotis G. Ipeirotis was supported by the National Science Foundation under Grant No. IIS-0643846.

References

[1] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Y. Halevy, "Google's deep web crawl," *PVLDB*, vol. 1, no. 2, pp. 1241–1252, 2008.

[2] A. Ntoulas, P. Zerfos, and J. Cho, "Downloading textual hidden web content by keyword queries," in *Proceedings of the Fifth ACM+IEEE Joint Conference on Digital Libraries (JCDL 2005)*, 2005.

[3] J. R. Herskovic and E. V. Bernstam, "Using incomplete citation data for medline results ranking," in *AMIA Annual Symposium proceedings*, 2005, pp. 316–20.

[4] Z. Lu, W. Kim, and W. J. Wilbur, "Evaluating relevance ranking strategies for medline retrieval," *Journal of American Medical Informatics Association*, vol. 16, no. 1, pp. 32–36, 2009.

[5] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*. New York, NY, USA: McGraw-Hill, Inc., 1986.

[6] A. Singhal, "Modern information retrieval: A brief overview," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 24, no. 4, pp. 35–42, 2001. [Online]. Available: http://singhal.info/ieee2001.pdf

[7] S. E. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford, "Okapi at trec-3," in *TREC*, 1994.

[8] R. C. Geer, D. J. Messersmith, K. Alpi, M. Bhagwat, A. Chattopadhyay, N. Gaedeke, J. Lyon, M. E. Minie, R. C. Morris, J. A. Ohles, D. L. Osterbur, and M. R. Tennant, "Ncbi advanced workshop for bioinformatics information specialists: Sample user questions and answers," Accessible at http://www.ncbi.nlm.nih.gov/Class/NAWBIS/index.html, 2002, last revised on August 6th 2007.

[9] D. A. Berry and B. Fristedt, *Bandit Problems: Sequential Allocation of Experiments*. Springer, 1985.

[10] J. Lee, J. Lee, and H. Lee, "Exploration and exploitation in the presence of network externalities," *Management Science*, vol. 49, no. 4, pp. 553–570, Apr. 2003.

[11] W. G. Macready and D. H. Wolpert, "Bandit problems and the exploration/exploitation tradeoff," *IEEE Transactions on Evolutionary Computation*, vol. 2, no. 1, pp. 2–22, Apr. 1998.

[12] V. Hristidis, Y. Hu, and P. G. Ipeirotis, "Ranked queries over sources with boolean query interfaces without ranking support," in *26th IEEE International Conference on Data Engineering (ICDE 2010)*, 2010.

[13] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Comput. Surv.*, vol. 40, no. 4, pp. 1–58, 2008.

[14] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM, 2001, pp. 102–113.

[15] N. Bruno, L. Gravano, and A. Marian., "Evaluating top-k queries over web-accessible databases," in *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2002, p. 369.

[16] V. Hristidis and Y. Papakonstantinou, "Algorithms and applications for answering ranked queries using ranked views," *The VLDB Journal*, vol. 13, no. 1, pp. 49–70, 2004.

[17] M. Theobald, G. Weikum, and R. Schenkel, "Top-k query evaluation with probabilistic guarantees," in *VLDB*, 2004.

[18] M. K. Bergman, "The Deep Web: Surfacing hidden value," *Journal of Electronic Publishing*, vol. 7, no. 1, Aug. 2001.

[19] W. Meng, K.-L. Liu, C. T. Yu, X. Wang, Y. Chang, and N. Rishe, "Determining text databases to search in the internet," in *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases*, 1998, pp. 14–25.

[20] W. Meng, K.-L. Liu, C. T. Yu, W. Wu, and N. Rishe, "Estimating the usefulness of search engines," in *Proceedings of the 15th International Conference on Data Engineering (ICDE 1999)*, 1999, pp. 146–153.

[21] J. P. Callan and M. Connell, "Query-based sampling of text databases," *ACM Transactions on Information Systems*, vol. 19, no. 2, pp. 97–130, 2001.

[22] P. G. Ipeirotis and L. Gravano, "Distributed search over the hidden web: Hierarchical database sampling and selection," in *Proceedings of the 28th International Conference on Very Large Databases (VLDB 2002)*, 2002, pp. 394–405.

[23] P. Domingos and M. J. Pazzani, "On the optimality of the simple bayesian classifier under zero-one loss," *Machine Learning*, vol. 29, no. 2-3, pp. 103–130, 1997.

[24] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano, "To search or to crawl? towards a query optimizer for text-centric tasks," in *SIGMOD Conference*, 2006, pp. 265–276.

[25] R. Fagin, R. Kumar, and D. Sivakumar, "Comparing top k lists," in *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003, pp. 28–36.