Sampling Search-Engine Results

Aris Anagnostopoulos*
Dept. of Computer Science
Brown University
Providence, RI 02912, USA
aris@cs.brown.edu

Andrei Z. Broder
IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532, USA
abroder@us.ibm.com

David Carmel
IBM Haifa Research Lab
Haifa 31905, ISRAEL
carmel@il.ibm.com

ABSTRACT

We consider the problem of efficiently sampling Web search engine query results. In turn, using a small random sample instead of the full set of results leads to efficient approximate algorithms for several applications, such as:

- Determining the set of categories in a given taxonomy spanned by the search results;
- Finding the range of metadata values associated to the result set in order to enable "multi-faceted search;"
- Estimating the size of the result set;
- Data mining associations to the query terms.

We present and analyze an efficient algorithm for obtaining uniform random samples applicable to any search engine based on posting lists and document-at-a-time evaluation. (To our knowledge, all popular Web search engines, e.g. Google, Inktomi, AltaVista, AllTheWeb, belong to this class.)

Furthermore, our algorithm can be modified to follow the modern object-oriented approach whereby posting lists are viewed as streams equipped with a *next* method, and the *next* method for Boolean and other complex queries is built from the *next* method for primitive terms. In our case we show how to construct a basic *next(p)* method that samples term posting lists with probability p, and show how to construct *next(p)* methods for Boolean operators (**AND**, **OR**, **WAND**) from primitive methods.

Finally, we test the efficiency and quality of our approach on both synthetic and real-world data.

Categories and Subject Descriptors

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

General Terms

Algorithms

Keywords

Search Engines, Sampling, Weighted AND, WAND

*Work performed while this author was at IBM T. J. Watson Research Center.

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

1. INTRODUCTION

Web search continues its explosive growth: according to the Pew Internet & American Life Project [11], there are over 107 million Web search users in United States alone, and they did over 3.9 billion queries in the month of June 2004. At the same time, the Web corpus grows: as of February 8, 2005, `google.com` claims over 8 billion pages indexed.

Thus search algorithmic efficiency is as important as ever: although processor speeds are increasing and hardware is getting less expensive every day, the size of the corpus and the number of searches is growing at an even faster pace.

On the other hand, Web search users tend to make very short queries (less than 3 words long [19]) that result in very large result sets. Although by now search engines have become very accurate with respect to navigational queries (see [6] for definitions), for informational queries the situation is murkier: quite often the responses do not meet the user's needs, especially for ambiguous queries.

As an example, consider a user that is interested in finding out about famous opera sopranos and enters the query `sopranos` in the Google search box. It turns out that the most popular responses refer to the HBO's TV-series with the same name: in the top 100 Google results, only 7 documents *do not refer* to the HBO program. (All Google numbers, here and below, refer to experiments conducted on February 8, 2005.)

This situation has stimulated search engines to offer various "post-search" tools to help users deal with large sets of somewhat imprecise results. Such tools include query suggestions or refinements (e.g., `yahoo.com` and `teoma.com`), result clustering and the naming of clusters (e.g., `wisenut.com` and `vivisimo.com`), and mapping of results against a predetermined taxonomy, such as ODP (the Open Directory Project used by Google and many others), Yahoo, and LookSmart. All these tools are based in full or in part on the analysis of the result set.

For instance in the previous example, the search engine may present the categories "TV series," "Opera," etc. or the query extensions "HBO sopranos," "mezzo sopranos," etc. Ideally, in order to extract the most frequent categories within the results set, all the documents matching the query should be examined; for Web size corpora this is of course prohibitive, as thousands or millions of documents may match. Therefore, a common technique is to restrict attention only to the top few hundreds ranked documents and extract the categories from those. This is much faster since search engines use a combination of static (query-independent) rank factors (such as PageRank [5]) and query

dependent factors. By sorting the index in decreasing order of static rank and using a branch-and-bound approach, the top 200 (say) results can be produced much faster than the entire set of results.

The problem with this approach is that the highly-ranked documents are not necessarily representative for the entire set of documents, as they may be biased towards popular categories. In the "sopranos" example, although 93 of the top 100 documents in Google refer to the HBO series, the query for `sopranos AND HBO` matches about 265,000 pages in Google (per Google report), while the query `sopranos AND opera -HBO` matches about 320,000, a completely different picture.

Many corporate search engines, and especially e-commerce sites, implement a technique called *multi-faceted* or *multi-dimensional search*. This approach allows the refinement of full-text queries according to meta-data specifications associated to the matching items (e.g., price range, weight) in any order, but only nonempty refinements are possible. The refinement is presented as a "browsing" of those results that satisfy certain metadata conditions, very similar to narrowing results in a particular category.

As an example, consider a user who visits an online music store such as `towerrecords.com`, and performs a query, say, the string `james`. The engine (from `mercado.com`) provides a number of hits, but also numerous possible refinements, according to various "facets," for instance by "Genre" (Blues, Children's, Country, ...), by price (Under $7, Under $10, Under $15, ...), by "Format" (Cassette, CD, Maxi-Single, Compact Disc, ...), and so on. The refinements offered depend on the initial query, so that only nonempty categories are offered, and sparse subcategories are merged into an "Other" subcategory. Similar approaches are used by many other e-tailers.

Multi-faceted search is used in other contexts as well, for instance, Yee et al. [22] show the benefits of this approach as applied within the "Flamenco" project at U. C. Berkeley for searching images using metadata refinement.

Since the categories displayed for multi-faceted search depend on the result set of the query, they have to be extracted quickly, which becomes a problem when the corpus is large. It seems that some current multi-faceted search engines are limited to corpora that can be represented in memory.

1.1 Sampling the Search Results

The applications described above require significant processing time; in order to apply them to large corpora we propose to only *sample* the set of documents that match the user's query. Asymptotically, under term independence assumptions, the average running time of our sampling approach is proportional to the sample size and grows only logarithmically in the size of the full matching set. On the other hand, sampling allows us to extract information that is unbiased with respect to the search-engine's ranking, and therefore produce better coverage of all topics or all metadata values present in the full result set.

The main technical difficulty in sampling follows from the fact that we do not have the results of the query explicitly available, but instead the results are generated one after the other, by a rather expensive process, potentially involving numerous disk accesses for each query term. The straightforward implementation is to pay the price, find and store pointers to all the documents matching the original query, and build a uniform sample from these results. However, as we already mentioned, our algorithm will obtain the sample after generating and examining only a small fraction of the result set and yet the sample produced is uniform, that is, every set of matching pages of size k (the desired sample size) has an equal probability to be selected as the output sample.

Although, to the best of our knowledge, the idea of sampling query results from search engines is new, sampling has been applied in different contexts as a means to give fast approximate answers to a particular problem. The areas of randomized and approximation algorithms provide numerous examples. In the area of data streams, where the input size is very large, sampling the input and operating on it is a common technique (see e.g., [3, 13, 17]). Even databases allow the user to specify a sampling rate in a *select* operation that instead of performing the query on the full set of data it operates on a sample [15]; as a result the DB2 standard has been augmented in order to support this option.

Besides the two applications already mentioned, result categorization and multi-faceted search, a random sample of the query results has more potential uses. In Theorem 2.2 we show that after the execution of our algorithm we can obtain an unbiased estimator of the total number of documents matching the user's original query, while in Theorem 2.3 we show that the estimator can achieve any prespecified degree of accuracy and confidence. Many users seem to like such estimates, maybe to help them decide whether they should try to refine the query further. In any case, Web search engines generally provide estimates of the number of results matching a query. For instance both Google and Yahoo provide such estimates at the top of the search results page. However these estimates are notoriously unreliable, especially for disjunctions. As an example, as of February 8, 2005, Google reports about 105M results containing the term "George," about 185M pages containing the term "Washington," while its estimate for the documents satisfying the query "George **OR** Washington" (done via advanced search) is about 33M. In contrast, in our experiments (see Section 4) even a 50-result uniform sample yielded estimates within 15% of target in all cases.

Yet another application of random sampling is to identify terms or other properties associated to the query terms. For instance one might ask "Who is the person most often mentioned on the Web together with Osama bin Laden?" The approach we envisage is to sample the results of the query `"Osama bin Laden"`, fetch the sample pages, run an entity detection text analyzer that can recognize people names, extract these names, and so on. Again the advantage of this approach compared to using the top results for the query `"Osama bin Laden"` is that the top results might be biased towards a particular context.

A similar application is suggested by the paper [2] where the authors demonstrate how finding (by "hand") new terms relevant or irrelevant to a given query can be useful for building "corpus independent" performance measures for information retrieval systems. The main idea is that by providing a set of relevant and a set of irrelevant terms for a given query, we can evaluate the performance of the information retrieval system by checking whether the documents retrieved contained the specified relevant and irrelevant terms. However, discovering these sets of terms is a daunting task, that requires the time and skill of an IR specialist; a sample

of the search results for the query can help the specialist identify both relevant and irrelevant terms. Again the lack of bias is probably useful.

Yet another application is suggested by the paper [18] that proposes the use of the Web as a knowledge source for domain-independent question answering by paraphrasing natural language questions in a way that is most likely to produce a list of hits containing the answer(s) to the question. It might well be the case that the results would be better when using a random sample of matches rather than a ranked set of matches, since the ranking is based on a very different idea of "best" results.

The list of potential applications of search results sampling that we proposed above is probably far from complete. We hope that our work will stimulate search engines to implement a random sampling feature, and this in turn will lead to many more uses than we can conceive now.

1.2 Alternative Implementations

A very simple way of producing (pseudo) random samples is to keep the index in a random order. Then the first k matches of a query can be viewed as a random sample, or, if more than one sample is needed, we can take matches x to $x + k$ as our sample. In fact this is the approach used in IBM's WebFountain [14], a system for large scale Web data mining.

However, in a standard Web search engine, there are many disadvantages for such an architecture:

1. If the index is in random order, rather than in decreasing static rank order, ranking regular searches ("top-k") is very expensive since no branch-and-bound optimization can be used. Thus the random-order index has to be stored separately from the search index which doubles the storage cost. (This is not an issue in WebFountain where "top-k" searches are a small fraction of the load.)
2. Maintaining a true random order as documents are added and deleted is nontrivial. A good solution is to have a "random static score" associated to each document and keep the index sorted by this "random score." This allows having an old index and a delta index to deal with additions.
3. Creating multiple truly independent random samples for the same query is nontrivial.

Thus, for regular Web search engines, sampling is a much better alternative.

1.3 Retrieval Model and Notations

Our model is a traditional *Document-at-a-time* (DAAT) model for IR systems [20]. Every document in the database is assigned a unique document identifier (DID). (As we mentioned in the introduction, the DIDs are assigned in such a way that increasing DIDs corresponds to decreasing static scores. However this is not relevant to the rest of our discussion.) Every possible term is associated with a *posting list*. This list contains an entry for each document in the collection that contains the index term. The entry consists of the document's DID, as well as any other information required by the system's scoring model such as number of occurrences of the term in the document, offsets of occurrences, etc. Posting lists are ordered in increasing order of the document identifiers.

Posting lists are stored on secondary storage media, and we assume that we can access them through stream-reader operations. In particular, each pointer to a term's posting list, supports the following standard operations.

1. *loc*(): returns the current location of the pointer.
2. *next*(): advances the pointer to the next entry in the term's posting list.
3. *next*(r): moves the pointer to the first document with DID greater or equal to r.

For our purposes, we need a special operator

4. *jump*(r,s): moves the pointer to the s-th entry in the posting list after the document with DID greater or equal to r. (Equivalent to *next*(r) followed by s *next*() operations. However simulating *jump*(r,s) this way would cost s moves rather than one – see below.)

Operations *loc* and *next* are easily implemented with a linked-list data structure, while for *next*(r) search engines augment the linked lists with tree-like data structures in order to perform the operation efficiently. For example one can use a binary tree where the leaves are posting locations corresponding to the first posting in consecutive disk records and every inner node x contains the first location in the subtree rooted at x.

The *jump* operation is not traditionally supported but can be easily implemented using the same tree data-structures needed for *next*(r) – we simply augment the inner nodes with a count of all the postings contained within the rooted subtree.

In the modern object-oriented approach to search engines based on posting lists and DAAT evaluation, posting lists are viewed as streams equipped with the *next* method above, and the *next* method for Boolean and other complex queries is built from the *next* method for primitive terms. For instance, $next(A \,\mathbf{OR}\, B) = \min(next(A), next(B))$. We will show later how to construct a basic *next(p)* method that samples term posting lists with probability p, and show how to construct *next(p)* methods for Boolean operators (**AND**, **OR**, **WAND**) from primitive methods.

Since the posting lists are stored on secondary storage, each *next* or *jump* operation may result to one or more disk accesses. The additional search-engine data structures ensure that we have at most one disk access per operation. Our goal is to minimize the number of disk accesses, and hence we want to minimize the number of the stream-reader move operations. In the rest of the paper, we assume that these moves have unit cost, while any other calculation has a negligible cost. (This assumption is of course only a first approximation, but it is well correlated with observed wall clock times [7]. A more accurate model would have to distinguish at least between "within-a-block" moves and "block-to-block" moves.)

For easy reference, we list here the notations used in the remainder of the paper. The total number of documents is N, while the number of documents containing term T_i is N_i. For the query under consideration, we let t be the number of terms contained in the query, and $m \leq N$ be the number of documents that satisfy the query. The sample size that we require is of size k; we expect in general to have $k \ll m$. Finally, in many cases we assume that $p = k/m$. This assumption will be clear from the context.

The most general sampling technique that we propose is applicable to many search engine architectures. We describe it in Section 2. Next, in Section 3, we specialize to a particular architecture based on the **WAND** operator that was first introduced in [7] and this specialization allows us to achieve better performance. We implemented and performed various experiments, and we present the results in Section 4. In Section 5 we summarize and conclude our results.

2. A GENERAL SCHEME FOR SAMPLING

2.1 Two Motivating Examples

In order to build some intuition for the sampling problem, we present two examples: one where the query is a conjunction (**AND**) of two terms and another where the query is a disjunction (**OR**) of two terms. Later in the paper we will provide more details about the sampling mechanism, and generalize it to a broader class of queries.

For the **AND** example consider some term A that appears in $10M$ documents, a term B that appears in $100M$ documents, and assume that the number of documents containing both terms is $5M$. Assume, moreover, that we want a sample of 1000 results. Then sampling each document that satisfies the **AND** query with probability equal to $p = 1000/5M = 1/5000$ creates a random sample with the desired expected size.

We will use the notation A (resp. B, C, etc.) to mean both the term A and the set of postings associated to A. The meaning should be clear from context.

An initial problem arises from the fact that although we may know how many documents contain the term A and how many contain the term B, we do not know a priori the number of documents that contain both terms, and thus we do not know the proper sampling probability. There are ways to circumvent this issue and we discuss them later in Subsection 2.3. For now, assume that we know the correct sampling probability, and the question is how to sample efficiently.

The naive approach would be to identify every document that contains both terms and, for each document independently, add it to the sample with probability p. This means checking at least all the postings for the rarest of the terms, so we need to examine at least the $10M$ postings on A's posting list.

Instead consider the following approach: Sample the posting list of A (the rarest term) with probability p and create a virtual term A_p whose posting list contains the sampled postings of A. Then the posting list for A_p contains roughly $10M/5000 = 2000$ documents. We return the documents satisfying the query A_p **AND** B. It is easy to verify that the result is a uniform sample over all the documents containing A **AND** B. Later we will show how, given p, we can create the posting list of A_p online in time proportional to $|A_p|$; hence, this method allows us to examine only 2000 postings, a clear gain over the $10M$ postings examined by the naive approach.

Now let's look at the **OR** example that turns out to be somewhat more complicated. Consider another term C that appears also in $10M$ documents and assume that there are $15M$ documents containing A **OR** C. Again we want a sample of 1000 documents, so in this case $p = 1000/15M = 1/15000$. The naive approach is to check every document in A **OR** C and insert it into the sample with probability p, which means traversing the posting lists of both A and C, or $20M$ operations. However we can apply the same technique as before and create a term A_p in time proportional to $|A_p|$. However, a document may satisfy the query even if it does not contain A, so we create also a virtual term C_p in the same manner, and return documents in A_p **OR** C_p. Thus the total number of postings examined is $|A_p|+|C_p| = 20M/15000 \simeq 1333$, a factor of 15000 improvement. But now we need to be more careful: if a document contains only the term A then it is inserted in A_p with probability p, and similarly if it contains only the term C then it is inserted in C_p with probability p. But if a document contains both terms, the probability to be contained in either A_p or C_p is $2p - p^2$. Hence, every document containing both A and C and contained in A_p **OR** C_p must be rejected from the sample with probability $1 - p/(2p - p^2)$. This will ensure that every document in A **OR** C is included in the sample with probability exactly p.

2.2 Sampling Search Results for a General Query

We now generalize the examples of the previous section and show how to apply the same procedure for sampling query results to any search engine based on inverted indices and a *Document-at-a-time* retrieval strategy. This class includes Google [5], AltaVista [8] and IBM's Trevi [12].

Consider a query Q, that can be as simple as the prior examples, or a more complicated boolean expression (including **NOT** terms, but not exclusively **NOT** terms). It could even contain more advanced operators like phrases or proximity operators. Every such query contains a number of simple terms, say $T_1, T_2, \ldots, T_\ell$, to which the operators are applied, and each term is associated with a posting list. Although the exact details depend on the specific implementation, every search engine traverses those lists and evaluates Q over the documents in the lists and several heuristics and optimization techniques are applied to reduce the number of documents examined (so, for example, for an **AND** query the engine will ideally traverse only the most infrequent term). Recall that the total number of documents satisfying the query is m, and that we need a sample of size k, which means that every document satisfying the query should be sampled with probability $p = k/m$. Assume, moreover, for the moment that we know m, and therefore we know the sampling probability p – in Subsection 2.3 we show how to handle this.

The way to sample the results is simple in concept. For every term T_i (but not for terms **NOT** T_i) we create a *pruned posting list* of document entries, which contains every document from the posting list of T_i with probability p, independently of anything else. The naive way to create the pruned list, is to traverse the original posting list and insert every document into the pruned list with probability p. An efficient equivalent way is to skip over a random number X of documents, where X is distributed according to a geometric distribution with parameter p. We can create a geometrically distributed random variable with parameter p, in constant time, by using the formula

$$X = \left\lceil \frac{\log(U)}{\log(1-p)} \right\rceil,$$

where U is a real random variable uniformly distributed in the interval $[0, 1]$ (see [10]).

The random skip is then simulated by executing a *jump*(r, X) operation, where r is the last document considered. (Recall from the discussion of Section 1.3 that the data structure used for postings allows for efficiently skipping documents in the posting lists, thus the skip has unit cost.) We then insert the document into the pruned list and we skip another random number of documents, continuing until the posting list is completely traversed. Note that the pruned lists can be precomputed at the beginning of the query, or they can be created on the fly, as the documents are examined.

We now perform the query by considering only documents that contain at least one term in the pruned lists. This is equivalent to replacing the original query $Q(T_1, T_2, \dots)$ with the query

$$Q(T_1, T_2, \dots) \,\mathbf{AND}\big(T_{1,p} \,\mathbf{OR}\, T_{2,p} \,\mathbf{OR} \cdots\big).$$

By this construction, every document that appears in some posting list has probability at least p to be considered. There are, however, documents that originally appear in more than one posting list. Consider some document that appears in the posting lists of r terms that are also being pruned. Then this document has increased chances to appear in *some* pruned list, the probability being exactly $1-(1-p)^r$. Therefore, for every document that satisfies the query, we should also count the number r of posting lists subject to pruning, in which it originally appears. Then we insert the document into the sample with probability $p/(1-(1-p)^r)$, so that overall the probability that the document is accepted becomes exactly p.

There are several remarks to be made about this technique:

- First we want to stress its generality that allows it to be incorporated in a large class of search engines.
- Second, the method is very clean and simple, since it does not require any additional nontrivial data structures; indeed, although the pruned lists can be precomputed (and, to improve response time, even stored on disk for common search terms and fixed pruning probabilities), the pruned lists can exist only at a conceptual level. When an iterator traverses a pruned list, in the actual implementation, it may traverse the original posting list and skip the necessary documents. Our implementation that we describe in detail in Section 3, demonstrates this approach. The only addition we require is the support of the *jump* operation described in Section 1.3, which is not significantly different from the *next* operation. Therefore from a programming point of view, the needed modifications are very transparent.
- Furthermore, the modern object-oriented approach to search engines is to view posting lists as streams that have a *next* method, and to build a *next* method for Boolean and other complex queries from the basic *next* method for primitive terms. Our geometric jumps method provides a method that samples term posting lists with probability p providing the primitive *next(p)* method, and the approach described above provides a *next(p)* method for arbitrary queries: we first advance to the minimum posting in all pruned posting lists via the primitive *next(p)* method, we evaluate the query, and if we have a match, we perform the rejection method as described.
- Finally, we want to mention that the general mechanism can be appropriately modified and made more efficient for particular implementations. For example, in the **AND** example of the previous section, we saw that we need to create the pruned list of only one of the terms. In Section 3 we show how we apply the technique to the **WAND** operator used in IBM's Trevi search engine [12] and JURU search engine [9] and gain similar benefits.

2.3 Estimating the Sampling Probability

During the previous discussion we assumed that we know the total number of documents m matching the query and hence that we can compute the sampling probability $p = k/m$. In reality we do not know m, and therefore we have to adjust the probability during the execution of the algorithm. The problem of sequential sampling (sample exactly k out of m elements that arrive sequentially) when m is unknown beforehand, has been considered in the past. Vitter [21] was first to propose efficient algorithms to address that problem, using a technique called *reservoir sampling*. The main idea is that when the i-th item arrives we insert it into the sample (reservoir) with probability k/i (for $i \geq k$) replacing a random element already in the sample. This technique ensures that at every moment, the reservoir contains a random sample of the elements seen so far. Vitter and subsequent researchers proposed efficient algorithms to simulate this procedure that instead of checking every element skip over a number of them (see, for example, [21, 16]).

It seems, however, that those techniques cannot be applied directly to our problem, because the list of matching documents represents the union or intersection of several lists. If we simply skip over a number of documents, we do not know how many skipped documents matched the query and, therefore, we cannot decide what the acceptance probability of the chosen document should be.

Instead we apply the following technique, related to the method used in [13] in the context of stream processing: We maintain a buffer of size $B > k$ (e.g., B can equal $2k$), and initially we set the sampling probability equal to some upper bound for the correct sampling probability, p_0 (p_0 can be 1). In other words, we accept every document that satisfies the query with probability $p = p_0$. Whenever the buffer is full, that is, the number of documents accepted equals B (which indicates that p was probably too large) we set a new sampling probability $p' = \alpha \cdot p$, for some constant $k/B < \alpha < 1$. Then every already accepted document is retained in the sample with probability α and deleted from the sample with probability $1 - \alpha$. Thus the expected sample size becomes $B\alpha > k$ and a Chernoff bound shows that with high probability the actual size is close to $B\alpha$, if k is large enough. Subsequent documents that satisfy the query are inserted into the sample with probability $p = p'$ independently of all other documents and p is decreased again whenever the buffer becomes full.

Eventually, the algorithm goes over all the posting lists and it ends up with a final sampling probability equal to some value p^*, and with a final number of documents in the sample, K, where $K < B$ always, and $K \geq k$ with high probability. Assuming the latter holds, we can then easily sample without replacement from this set and extract a sample of exactly k documents.

Observe that the number of times the sampling probability is decreased is bounded by

$$\frac{\log(1/p^*)}{\log(1/\alpha)} \approx \frac{\log(m/k)}{\log(1/\alpha)}.$$

Every time the probability is decreased the expected number of samples removed from the buffer is $(1-\alpha)B$. Thus the total number of samples considered can be bound by

$$(1-\alpha)B\frac{\log(m/k)}{\log(1/\alpha)} + B = O\big(k\log(m/k)\big). \qquad (1)$$

It is tempting to assert that the algorithm chooses independently every document with probability p^*. Unfortunately this is not the case: for every independent sampling probability p there is some probability that the sample will be larger than B; however our algorithm never produces a sample larger than B. What holds is that conditional on its size, the sample is uniform. Furthermore we can use the final size and the final sampling probability to compute m, the size of the set we sampled from. This is captured by the following three theorems.

THEOREM 2.1. *Assume that at the end of the sampling algorithm the actual size of the sample is K. Then the produced sample set is uniform over all sets of K documents that satisfy the query.*

PROOF. We use a coupling (simulation) argument and here we give the main intuition. Assume that each of the m documents that satisfy the query has an associated real random variable X_i, chosen independently uniformly at random in the interval $(0,1]$.

We build a new algorithm that proceeds exactly as before except that whenever the buffer is full, p is reduced to p' and we keep in the buffer only those documents i that have $X_i < p'$. Every new document j is inserted in the buffer iff it has $X_j < p$ (the new sampling probability).

Let $S_p = \{i \mid X_i < p\}$. Then p^* is the largest value in the set $\{p_0, \alpha p_0, \alpha^2 p_0, \ldots\}$ such that

$$|S_{p^*}| = |\{i \mid X_i < p^*\}| < B,$$

and the final sample is S_{p^*}. Clearly the set S_{p^*} is uniform over all sets of size $K = |S_{p^*}|$. On the other hand the original algorithm and the new algorithm are in an obvious 1-1 correspondence, and thus conditional on its size, the final sample is uniform. □

Notice, that the algorithm does not know initially the number of documents that satisfy the query, a value that is usually hard to estimate. As mentioned, an additional feature of the algorithm is that we can estimate the number of documents matching the query. The following theorem summarizes the result.

THEOREM 2.2. *Assume that at the end of the algorithm the size of the sample is K, and the final sampling probability is p^*. Then the ratio K/p^* is an unbiased estimator for the number of documents m matching the query, that is, $\mathbf{E}[K/p^*] = m$.*

PROOF. View the algorithm as performing two types of steps: if the buffer is full then the algorithm reduces the sampling probability and resamples the buffer; if the buffer is not full, the algorithm considers the next candidate document and inserts it with probability p.

Assume that after t steps there were were K_t documents in the sample, the sampling probability was p_t, and we have considered m_t candidate documents. Thus if the algorithm stops after f steps $K_f = K$ and $p_f = p^*$ and $m_f = m$. We prove that at every step the ratio $\mathbf{E}[K_t/p_t] = m_t$.

To this end we define a sequence of random variables $\{X_t\}_{t\geq 0}$ as follows. We let $X_0 = 0$ and

$$X_t = \frac{K_t}{p_t} - m_t.$$

We now show that the sequence $\{X_t\}$ is a martingale (i.e., that $\mathbf{E}[X_t \,|\, X_0, X_1, \ldots, X_{t-1}] = X_{t-1}$) which implies that $\mathbf{E}[K_f/p_f] - m_f = 0$, and completes the proof. (For brevity, we gloss over some technical details; the complete proof will be included in a longer version of this work.)

Notice that if $K_{t-1} < B$ then the sampling probability does not change ($p_t = p_{t-1}$) but we will consider a new document that is inserted with probability p_t. Therefore, if we let Z be the indicator random variable of the event that at time t a document becomes accepted, we get

$$\begin{aligned}
&\mathbf{E}[X_t \mid X_0, \ldots, X_{t-1}, K_{t-1} < B] \\
&\quad = \mathbf{E}\left[\frac{K_{t-1}+Z}{p_t} - m_t \,\middle|\, X_0, \ldots X_{t-1}\right] \\
&\quad = \frac{K_{t-1}+p_t}{p_t} - m_t = \frac{K_{t-1}}{p_t} - (m_t - 1) \\
&\quad = X_{t-1}.
\end{aligned}$$

On the other hand, if $K_{t-1} = B$ then every document already in the sample is resampled with probability p_t/p_{t-1} but we are not considering any new document, that is $m_t = m_{t-1}$. Therefore

$$\begin{aligned}
&\mathbf{E}[X_t \mid X_0, \ldots, X_{t-1}, K_{t-1} = B] \\
&\quad = \mathbf{E}\left[\frac{\text{Binomial}(K_{t-1}, p_t/p_{t-1})}{p_t} - m_t \,\middle|\, X_0, \ldots, X_{t-1}\right] \\
&\quad = \frac{K_{t-1}\frac{p_t}{p_{t-1}}}{p_t} - m_{t-1} = X_{t-1}.
\end{aligned}$$

Hence, we conclude that the sequence $\{X_t\}$ is a martingale, and this implies via the Optional Sampling Theorem that $\mathbf{E}[X_f] = \mathbf{E}[X_0] = 0$. □

Besides having the correct expectation, a good estimator should be close to the correct value with high probability. In general an (ϵ,δ)-*approximation scheme* for a quantity X, is defined as a procedure that given any positive $\epsilon < 1$ and $\delta < 1$ computes an estimate $\hat{X}$ of X that is within relative error of ϵ with probability at least $1-\delta$, that is

$$\mathbf{Pr}(|\hat{X} - X| \leq \epsilon X) \geq 1 - \delta.$$

The following theorem shows that our sampling procedure using a buffer size quadratic in $1/\epsilon$ and logarithmic in $1/\delta$ is in fact an (ϵ,δ)-approximation scheme.

THEOREM 2.3. *There is a constant C such that for any positive $\epsilon < 1$ and $\delta < 1$, the algorithm above with a buffer size $B = \frac{C}{\epsilon^2}\log\frac{1}{\delta}$ is an (ϵ,δ)-approximation scheme, that is, if at the end of the algorithm the size of the sample is K and the final sampling probability is p^* we have:*

$$\mathbf{Pr}\left(\left|\frac{K}{p^*} - m\right| \leq \epsilon m\right) \geq 1 - \delta.$$

PROOF. The proof is similar to that of Theorem 3 in [13] and we omit it here for lack of space. □

3. EFFICIENT SAMPLING OF THE WAND OPERATOR

Although we described a general sampling mechanism that can be applied to diverse settings, we have also seen that when we specialize to some particular operator such as **AND** we can achieve improved performance. In this section we describe the operator **WAND**, introduced in [7], that generalizes **AND** and **OR** and we present an efficient implementation for sampling the results of **WAND**.

3.1 The WAND Operator

Here we briefly describe the **WAND** operator that was introduced in [7] as a means to optimize the speed of search queries. **WAND** stands for Weak **AND**, or Weighted **AND**. It takes as arguments a list of Boolean variables $X_1, X_2, \ldots, X_k$, a list of associated positive *weights*, $w_1, w_2, \ldots, w_k$, and a threshold θ. By definition, $\mathbf{WAND}(X_1, w_1, \ldots X_k, w_k, \theta)$ is true iff

$$\sum_{1 \le i \le k} x_i w_i \ge \theta, \tag{2}$$

where x_i is the indicator variable for X_i, that is

$$x_i = \begin{cases} 1, & \text{if } X_i \text{ is true} \\ 0, & \text{otherwise.} \end{cases}$$

Observe that **WAND** can be used to implement **AND** and **OR** via

$$\mathbf{AND}(X_1, X_2, \ldots X_k) \equiv \mathbf{WAND}(X_1, 1, X_2, 1, \ldots X_k, 1, k),$$

and

$$\mathbf{OR}(X_1, X_2, \ldots X_k) \equiv \mathbf{WAND}(X_1, 1, X_2, 1, \ldots X_k, 1, 1).$$

For the purposes of this paper we shall assume that the goal is simply to sample the set of documents that satisfies Equation (2) with X_i indicating the presence of query term T_i in document d. We note however that the situation considered in [7] is more complicated: there each term T_i is associated with an upper bound on its maximal contribution to any document score, UB_i, and each document d is subject to a preliminary filtering given by

$$\mathbf{WAND}(X_1, UB_1, X_2, UB_2, \ldots, X_k, UB_k, \theta),$$

where X_i again indicates the presence of query term T_i in document d. If **WAND** evaluates to true, then the document undergoes a full evaluation, hence a document that matches **WAND** does not necessarily match the query. We can deal with this approach by doing a full evaluation on every document that we would normally insert into the buffer. (That is, a document that won the coin toss.) The document is then inserted into the buffer only if it passes the full evaluation. This insures that p is reduced only as needed. Further refinements considered in [7], such as varying the threshold θ during the algorithm, are meant to increase the efficiency of finding the top k results and thus are beyond the scope of this paper.

3.2 Sampling WAND Results

In the **AND** example that we saw previously, we can sample only the rarest term, and hence minimize the total number of *next* operations. In contrast, in the **OR** example, we must sample the posting lists of all terms. Since the **WAND** operator varies between **OR** and **AND**, a good sampling algorithm must handle efficiently both extremes.

Let T be the set of the query terms. We divide it into two subsets, the set S that contains the terms that must be sampled, and the set S^c that contains the rest of the terms in T. In the **AND** example, the set S contains only the least frequent term, while in the **OR** example the set S contains all the terms.

The first issue is how to select the set S. We will discuss the optimal way to do it, after discussing the running time of the algorithm. For the time being, assume that we choose the set S arbitrarily such that

$$\sum_{i \in S^c} w_i < \theta.$$

Hence S is such that any document that satisfies Equation (2) must contain at least one term from S. It is easy to check that the **AND** and the **OR** examples expressed as **WAND** obey this inequality for their respective choices of S.

Following the description in Section 2.2, we create *pruned lists* for the terms in S (but not for the terms in S^c), and again as before, a document in the posting list of a term is included in the pruned list of that term with probability p, independently of other documents and other terms.

Of course the algorithm does not know p beforehand, so it initially starts accepting all the documents with some probability $p = p_0$, maybe $p = 1$, and it reduces p over time, using the process described in Subsection 2.3.

The algorithm guarantees that every document that contains at least one term in S has probability at least p to be selected. If it becomes selected and it satisfies **WAND**, we normalize the probability to be exactly p using the rejection method described in Subsection 2.2. If a document does not contain any term from S, its total weight is strictly smaller than θ and, therefore, it does not satisfy **WAND**.

We now give a high-level description of the sampling algorithm. The details appear in Figure 1; Figure 2 contains a visual example.

Every term in the set S is associated with a *producer*, which is an iterator traversing the pruned list, selecting documents for evaluation against the query. Furthermore, in order to perform the evaluation, every term in the query is also associated with a *checker* that traverses the original posting list. At one iteration of the algorithm we advance the producers that point to the document with the smallest DID, and some document is selected (with probability p) by some of them. Then the checkers will determine the terms that are contained in the document and if the sum of their weights exceeds the threshold θ, then the document becomes a candidate to be selected for the sample. Like in the general approach, the pruned list may exist only at the conceptual level, and the producers may traverse the original posting lists and jump over a random number (geometrically distributed) of documents.

Once a document (whose DID is held in the variable *global*) is selected for consideration, we use the checkers to determine if *global* satisfies the query. Some checkers point to documents with DID smaller than *global* and these are terms that, as far as we know at this point, might be contained in the document with DID=*global*. The algorithm maintains an upper bound equal to the sum of the weights of the terms whose checkers point to a document with DID not greater than *global*. As long as the upper bound exceeds the threshold θ (and therefore *global* might satisfy the query),

```
1. Function getWANDSample()
2.    /* First some initializations. */
3.    curDoc ← 0
4.    global ← 0
5.    p ← 1
6.    foreach (term i)
7.       checker[i].next(0)
8.    foreach (term i ∈ S)
9.       producer[i].nextPruned(0)
10.
11.   repeat
12.      advance global to smallest DID for which
          ∑_{i:checker[i]≤global} w_i ≥ θ
13.      if (global < min DID of producers)
14.         global ← min DID of producers
15.      /* Now at least one producer is ≤ global. */
16.      A ← {terms i ∈ S s.t. producer[i].DID < global}
17.      while (A ≠ ∅ && no producer points to global)
18.         pick i ∈ A
19.         producer[i].nextPruned(global)
20.      if (no producer points to global)
21.         global ← min DID of producers
22.      if (global = lastID)
23.         return /* Finished with all the documents */
24.      /* Now the global points to a DID that exists in some
          pruned list, and such that the accumulated weight
          behind it is at least θ. */
25.      B ← {terms i ∈ T s.t. checker[i] ≤ global}
26.      /* B contains the terms that contribute to the upper
          bound */
27.      if (global ≤ curDoc)
28.         /* document at global has already been considered */
29.         pick i ∈ B
30.         /* it is probably best to pick an i ∈ B ∩ S */
31.         checker[i].next(curDoc + 1)
32.      else /* global > curDoc */
33.         if (∑_{i∈B:checker[i].DID = global} w_i ≥ θ)
34.            /* Success, we have enough mass on global. */
35.            curDoc ← global
36.            /* We consider curDoc as a candidate. Now we
                must count exactly how many posting lists in S
                contain global in order to perform the probability
                normalization correctly. */
37.            foreach (i ∈ S ∩ B s.t. checker[i].DID < curDoc)
38.               checker[i].next(curDoc)
39.            D ← {terms i ∈ S s.t. checker[i] = global}
40.            with probability normalizedProbability(|D|)
                addToSample(curDoc)
41.         else (of line 33)
42.            /* Not enough mass yet on global, advance one of
                the preceding terms. */
43.            pick i ∈ B s.t. checker[i].DID < global
44.            /* it is probably best to pick an i ∈ B ∩ S */
45.            checker[i].next(global)
46.   end repeat


1. Function producer[i].nextPruned(r)
2.    X ← Geometric(p)
3.    producer[i].jump(r, X)


1. Function normalizedProbability(r)
2.    return p/(1 − (1 − p)^r)


1. Function addToSample(DID)
2.    Add DID to the sample
3.    /* Let B be the size of the buffer. */
4.    while (size of sample = B)
5.       /* we should take a smaller sample */
6.       p′ ← α · p
7.       foreach (i ∈ sample)
8.          keep i with probability p′/p
9.       p ← p′
```

Figure 1: Sampling WAND.

we advance some term's checker to the first document with DID $\geq$ *global*. Assume its DID is *doc*. If *doc* = *global* then the term is contained in *global*. We continue by advancing the rest of the checkers that are behind *global* until either the total sum of weights of the terms whose checkers are in positions $\leq$ *global* is less than the threshold θ, in which case the document does not satisfy the query, or until the sum of the weights of the terms that were found to be contained in *global* exceeds the threshold θ, in which case the document becomes a candidate to be selected for the sample. In the latter case, the next step is to count the exact number of terms in S that are contained in the document. Each of these terms offers a chance to the document to be inserted to the corresponding pruned list, therefore, by counting the terms in S that are contained in the document we can apply the rejection method, described in Subsection 2.2, and accept the document with the correct probability (i.e., with probability p).

Notice that the algorithmic description leaves some details unspecified. For instance, whenever some checker has to be advanced there is usually more than one choice. The goal is to select the checker that will advance the farthest possible, and a simple heuristic is to select the checker of the most infrequent term. This problem appears in the general context of query constraints satisfaction for posting list iterators and there are more advanced heuristics that try to guess the best move based on the results seen so far (see [8] and [4]). In our particular case, at some point during the execution of the algorithm, there is even more flexibility: we can either advance a checker or a producer (for example at line 31 we can advance a producer instead of a checker). Hence in principle, we can select whether it is better to advance a producer or a checker, based on our experience so far and the expected benefit of the choice and, indeed, our implementation uses this heuristic.

3.3 Running-Time Estimation and the Choice of the Set S

We now bound the running time of the algorithm, assuming that we know the correct value of the sampling probability $p = k/m$. Consider a query with t terms, and recall that N_i is the total number of documents containing the i-th term and that w_i is the weight of the i-th term in the **WAND** operator. In order to obtain an upper bound for the number of pointer advances, we note that whenever we advance a checker we advance it to at least past a producer, since during the execution of the algorithm the document under consideration (*global*) has been originally selected by

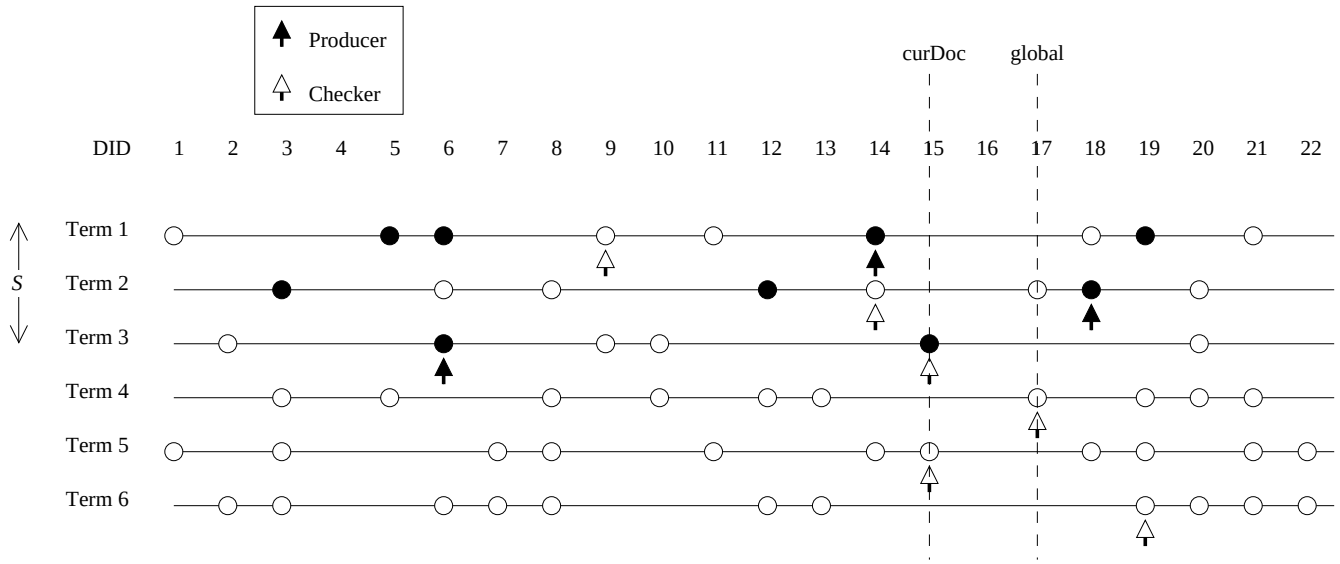

Figure 2: An example of the posting lists. A bullet indicates that the term exists in the corresponding document. A black bullet indicates that the document was sampled (or will be), hence it exists in the pruned list.

some producer. Therefore the total number of each checker's advances is bounded by the total number of producer advances which is expected to be $\frac{k}{m}\sum_{i\in S} N_i$. Therefore the running time is expected to be

$$O\Big(t\frac{k}{m}\sum_{i\in S} N_i\Big). \tag{3}$$

If the sampling probability is not known in advance, then in the worst case sampling will not help much. For instance if the standard search **WAND** spends a large amount of time getting the first B matches and then starts producing matches very fast, the sampling **WAND** will spend an equal amount of time until the first decrease of p from 1 to α. This is of course unlikely but entirely possible.

Hence for the average case we need to assume that the results are uniformly distributed with respect to DID numbers. To this end we assume the often-used probability model in IR, that is, we assume that each document contains the query terms independently with certain probabilities. In this case, conditional on a document d containing a term $t_i \in S$ there is a fixed probability π_i that d satisfies the query. Similarly there are fixed probabilities, $\pi_{i,1}, \pi_{i,2}, \ldots, \pi_{i,r}$ that d satisfies the query and contains exactly $1, 2, \ldots, r$ terms from S. Now consider the first time a document d is selected by a producer, say for the term t_i. Assume that at that time the sampling probability was p. In view of the above, the probability that d satisfies the query and also passes the rejection procedure is

$$\sum_j \frac{\pi_{i,j}p}{1-(1-p)^j} \geq \sum_j \frac{\pi_{i,j}}{j} = \rho_i.$$

On the other hand, in view of Equation (1), we know that the total number of samples ever inserted in the buffer is bounded by $O\big(k\log(m/k)\big)$. Hence the number of occurrences of the term t_i selected by its producer is bounded by

$$O\Big(\frac{k}{\rho_i}\log(m/k)\Big),$$

and therefore the total number of moves (producers and checkers) is

$$O\Big(tk\log(m/k)\sum_{i\in S}\frac{1}{\rho_i}\Big) = O\big(k\log(m/k)\big), \tag{4}$$

for any fixed query, and $k, m \to \infty$.

In order to minimize the running time of the algorithm, we want to select S so that the sum $\sum_{i\in S}\rho_i^{-1}$ is minimized. Of course ρ_i is not known in advance, but it can be estimated as the query progresses. Another approach, for $m \ll N_i$, is to make the rough estimate $\rho_i \approx m/N_i$. Then Equation (4) again suggests that a good choice for S is to try to minimize $\sum_{i\in S} N_i$.

A simple way to achieve a good selection for S in this vein is to sort the terms in increasing order of frequencies (and decreasing order of weights in case of ties), and let

$$\ell = \min_i \text{s.t.} : \quad \sum_{j=i+1}^{t} w_j < \theta.$$

Then let $S = \{1, 2, \ldots, \ell\}$. Notice that this greedy approach includes both the examples of **AND** and **OR** as special cases.

The optimal choice for the set S to minimize $\sum_{i\in S} N_i$ is obtained by solving the following integer program:

$$\begin{aligned} \min \quad & \sum_{i\in S} N_i \\ \text{s.t.} : \quad & \sum_{i\in S^c} w_i < \theta, \end{aligned}$$

or, equivalently,

$$\max \quad \sum_{i \in S^c} N_i$$
$$\text{s.t. :} \quad \sum_{i \in S^c} w_i < \theta,$$

which can be interpreted as a Knapsack problem. Since the values N_i are integral we can solve it exactly in polynomial (in t and N) time through dynamic programming, but since we have a small number of terms we can solve it much more efficiently by brute force. Sometimes we have some flexibility in assigning weights (usually we want terms with low frequency to have large weight), in which case the greedy approach will suffice to obtain an optimal solution.

The analysis above is based on minimizing the running-time upper bound, but the actual running time will usually be smaller, and will depend on the actual joint distribution of the query terms that generally changes as the algorithm iterates through the posting lists. In practice we can achieve better performance by observing the performance of each producer and dynamically changing the set S as the algorithm progresses. We want to insert terms that both produce large jumps and are well correlated with successful samples so that the sampling probability will go down quickly.

4. EXPERIMENTS

We implemented the sampling mechanism for the **WAND** operator and performed a series of experiments to test the efficiency of the approach as well as the accuracy of the results. We used the JURU search engine developed by IBM [9].

The data consisted of a set of 1.8 million Web pages, consisting of a total of 1.1 billion words (18 million total distinct words). Each document was classified according to its content to several categories. The taxonomy of the categories, as well as the classification of the documents to categories, were performed by IBM's Eureka classifier described in [1]. We used a total of 3000 categories, and each document belonged to zero, one, or more categories. Eureka's taxonomy contains additionally a number of broader super-categories that form a hierarchical structure. Although we did not make use of this structure in our experimental evaluation, we argue later in this section that it can be used to provide more meaningful results for the category-suggestion problem.

In order to estimate the gain in run-time efficiency, we count the number of times a pointer is advanced (via *next* or *jump*) over the terms' posting lists. As we argued previously, the total running time depends heavily on the number of those advances, since the posting lists are usually stored on secondary storage and accessing them is the main bottleneck in the query response time.

We experimented with nine ambiguous queries depicted in Table 1 chosen to produce results in many different categories. For each query we created different samples of sizes $k = 50$, 200, and 1000. In all the experiments the resampling probability equals $\alpha = 3/4$ and the buffer size is $B = 2k$. In Table 2 we compare the number of pointer advances for different sample sizes. Notice that even though the total number of matching documents is small (in the order of several thousands, while the motivation for our techniques is for applying them to queries with result sizes in the millions) we show a significant gain for small sample sizes. In order to further establish this point we performed additional queries using artificially created documents built from random sequences of numbers, such that the result sets would be larger. We present the results in Table 3.

From the two tables it is clear that sampling is justified if the sampling size k is at least 2 orders of magnitude smaller than the actual result size m. In this case the total time can be reduced by a factor of 10, 100, or even more, depending on the ratio k/m, as well as on the query type. On the other hand, if k is comparable to m, the overhead of the sampling (due to more than one pointer for each term) might even increase the total time.

#	Query
Q_1	Schumacher AND (Joel OR Michael)
Q_2	Olympic AND (Airline OR Games OR Gods)
Q_3	Turkey AND Customs
Q_4	Long AND Island AND Tea
Q_5	Schwarzenegger AND (California OR Terminator)
Q_6	Taxi AND Driver
Q_7	Dylan AND (Musician OR Poet)
Q_8	Football AND (Lazio OR Patriots)
Q_9	Indian AND (America OR Asia)

Table 1: The queries that we inserted to the sampling algorithm.

Query	Matches	No Sampl.	50	200	1000
Q_1	587	3275	2627	4297	4561
Q_2	5109	31121	4323	12716	31231
Q_3	3111	33849	13841	24192	35461
Q_4	1111	28604	12120	28547	40151
Q_5	407	2497	1532	3278	3314
Q_6	1028	6491	3783	6401	7475
Q_7	356	3678	3173	4967	4967
Q_8	566	8796	5060	8699	9123
Q_9	15721	96997	6437	19423	55248

Table 2: Number of pointer advances for the nine queries. The second column contains the total number of pages matching each query. The rest of the columns contain the number of pointer advances performed without sampling, and for samples of 50, 200, and 1000 pages.

4.1 Estimating the Most Frequent Categories of the Search Results

We also evaluated the accuracy of the results, that is, how well the small sample we produced represented the most fre-

Query	Matches	No Sampling	10	100
T_1 AND T_2	13011	104087	977	7161
T_3 OR T_4	57046	120102	566	4392
T_3 OR T_4 OR T_5	62890	134874	715	5351

Table 3: Comparison of pointer advances for queries performed on artificially created documents with samples of sizes 10 and 100.

Query	50	200	1000
Q_1	10	10	10
Q_2	7	10	10
Q_3	7	10	10
Q_4	4	8	10
Q_5	5	10	10
Q_6	7	9	10
Q_7	7	10	10
Q_8	8	10	10
Q_9	2	9	10

Table 4: Number of the top-10 frequent categories that appear in the samples.

Query	50	200	1000
Q_1	7	8	10
Q_2	6	5	8
Q_3	4	6	9
Q_4	3	6	10
Q_5	3	7	10
Q_6	3	4	10
Q_7	5	10	10
Q_8	7	8	10
Q_9	0	3	7

Table 5: Number of the top-10 frequent categories that appear in the 10 most frequent sample categories.

quent categories of the matched documents. For that we consider the same queries of Table 2. Each of these query results induces a set of categories from the Eureka Taxonomy. In order to determine whether the sampling succeeds in discovering the most frequent categories, we measured how many of the 10 most frequent categories are found in each of the sample size, and we present the results at Table 4.

An additional desirable property is for frequent categories in the result set to be also frequent in the sample so that we can identify them. For that, we check how many of the top-10 frequent categories for each query appear also in the top-10 frequent categories according to the sample, and we depict the results in Table 5.

There are a few facts worth noticing with respect to the results of sampling, some which are not revealed in the tables. First notice that in most cases, even small sample sizes succeed in sampling documents from the frequent categories (Table 4) but a somehow larger sample size is needed in order to ensure that the frequent categories manage to be popular in the sample as well as depicted in Table 5. It also seems that a sample of size 1000 is always successful in our examples, but this is somewhat misleading since in some of the examples the total number of documents is small, and therefore the sampling extracts all the original categories.

A final important remark, explains the poor performance in most of the cases of Table 5, compared with Table 4. Let us focus, for concreteness, on Q_9 (corresponding to the query "Indian AND (America OR Asia"). The total number of matching documents is 15721, and the sample of size 50 fails completely to identify the frequent categories, while the sample of size 200 also fails to spot out the most frequent categories in Table 5 (although, notice in Table 4 that it does manage to sample some documents related to 9 out of the 10 frequent categories). This is due to the Eureka categorization: the 3000 categories used to tag the documents are very fine, resulting in documents matching very specific categories. For query Q_9, the 15721 matching documents were found to be related to 1935 categories, from which we tried to extract the top 10. Each of these categories contains a number of documents, the most frequent one contains 125 documents, the 10th most frequent contains 54; the accumulated mass in the top 10 categories (sum of the number of documents contained within the top 10 categories) is 753, while the total mass is 9404. Therefore, each of the 50 sampled documents, has less than 1% chance to be a document contained within the top-10 categories, and negligible probability (0.57%) to be contained within the top 10th category.

Q'ry	Match Count	50		200		1000	
		Est.	Err	Est.	Err	Est.	Err
Q_1	587	562	4.3	597	1.7	587	0
Q_2	5109	5388	5.5	5088	0.4	5050	1.1
Q_3	3111	2652	14.8	3376	8.5	3150	1.3
Q_4	1111	1119	0.7	1150	3.5	1111	0
Q_5	407	433	6.4	395	2.9	407	0
Q_6	1028	1172	14.0	989	3.8	1028	0
Q_7	356	316	11.2	356	0	356	0
Q_8	566	545	3.7	596	5.3	566	0
Q_9	15721	17028	8.3	15448	1.7	15902	1.2

Table 6: Evaluation of the estimates for the sizes of the query results. The table shows the actual value, and for each sampling size the estimate and the percentage of the error.

The solution to this categorization artifact is straightforward: after obtaining the samples, we must aggregate the categories to coarser super-categories according to the taxonomy (e.g., the categories Lions, Cheetahs and Monkeys can be aggregated to Mammals, or Animals). Then the final result is a sample of a smaller number of categories each with a large mass, in which case even a small sample size can efficiently discover the popular super-categories and present them to the user. Since the emphasis of our work lies mainly on the method for sampling, we have not pursued this line of research any further.

4.2 Estimating the Size of the Result Set

Finally we evaluate the quality of the estimator for the size of the result set. Table 6 shows the estimates and the relative errors. We mention again that many commercial Web search engines fail to provide an accurate estimation of the number of results. In contrast, notice that for even the smallest sampling size the error never exceeds 15%, and usually it is negligible for a sample size greater than 200.

5. SUMMARY

We propose performing sampling on the results of search-engine queries in order to extract fast summary information from the ensemble of the results. We can use this information as a means of providing feedback to the user in order to refine his query. We develop a general scheme for performing the sampling efficiently, and we show how we can increase

the performance for particular implementations. Finally, we test the efficiency and quality of our methods on both synthetic and real-world data.

There are several issues worth further investigation. First, for general **WAND** sampling there are many choices that might improve the running time, such as the optimal selection of the set S and the selection of the checkers and producers to advance. One approach inspired by [8], is to use an adaptive mechanism that keeps track of the effect of past choices while the query is running. Second, it would be interesting to understand which classes of queries can be sampled with a more efficient method than the general procedure of Section 2.2. In particular simple but common Boolean combinations, even if expressible as a single WAND, could probably be sampled more efficiently than either the general procedure or even the general WAND mechanism. Third, a model for the average running time for sampling WAND that allows a rigorous analysis and requires fewer or no independence assumptions, remains a challenge.

6. ACKNOWLEDGEMENTS

We would like to thank Steve Gates and Wilfried Teiken for many useful discussions and suggestions, as well as for providing us with all the experimental set-up (hardware, the Eureka taxonomy, and the Web crawled data) that allowed us to perform our experiments. We are indebted to Andrew Tomkins for his observations regarding an early draft of our paper and we benefitted from comments received from Andreas Neumann, Ronny Lempel, Runping Qi, Jason Zien, and the anonymous referees.

7. REFERENCES

[1] C. C. Aggarwal, S. C. Gates, and P. S. Yu. On using partial supervision for text categorization. *IEEE Trans. Knowl. Data Eng.*, 16(2):245–255, 2004.

[2] E. Amitay, D. Carmel, R. Lempel, and A. Soffer. Scaling IR-system evaluation using term relevance sets. In *Proceedings of the 27th annual international conference on Research and development in information retrieval*, pages 10–17. ACM Press, 2004.

[3] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 633–634. Society for Industrial and Applied Mathematics, 2002.

[4] K. Beyer, A. Jhingran, B. Lyle, S. Rajagopalan, and E. Shekita. Pivot Join: A runtime operator for text search. Manuscript, 2003.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *WWW7/Computer Networks and ISDN Systems*, 30:107–117, April 1998.

[6] A. Z. Broder. A taxonomy of web search. *SIGIR Forum*, 36(2):3–10, 2002.

[7] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, pages 426–434. ACM Press, 2003.

[8] M. Burrows. Sequential searching of a database index using costraints on word-location pairs. United States Patent 5 745 890, 1998.

[9] D. Carmel, E. Amitay, M. Herscovici, Y. S. Maarek, Y. Petruschka, and A. Soffer. Juru at TREC 10 - Experiments with Index Pruning. In *Proceedings of the Tenth Text REtrieval Conference (TREC-10)*. National Institute of Standards and Technology (NIST), 2001.

[10] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986.

[11] D. Fallows, L. Rainie, and G. Mudd. The popularity and importance of search engines, August 2004. The Pew Internet & American Life Project, `http://www.pewinternet.org/pdfs/PIP_Data_Memo_Searchengines.pdf`.

[12] M. Fontoura, E. J. Shekita, J. Y. Zien, S. Rajagopalan, and A. Neumann. High performance index build algorithms for intranet search engines. In *VLDB 2004, Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 1158–1169. Morgan Kaufmann, 2004.

[13] P. B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 281–291. ACM Press, 2001.

[14] D. Gruhl, L. Chavet, D. Gibson, J. Meyer, P. Pattanayak, A. Tomkins, and J. Zien. How to build a webfountain: An architecture for very large-scale text analytics. *IBM Systems Journal*, 43(1), 2004.

[15] P. J. Haas, J. F. Naughton, and A. N. Swami. On the relative cost of sampling for join selectivity estimation. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 14–24. ACM Press, 1994.

[16] K.-H. Li. Reservoir-sampling algorithms of time complexity o(n(1 + log(n/n))). *ACM Trans. Math. Softw.*, 20(4):481–493, 1994.

[17] S. M. Muthukrishnan. Data streams: Algorithms and applications. `http://athos.rutgers.edu/~muthu/stream-1-1.ps`.

[18] D. R. Radev, H. Qi, Z. Zheng, S. Blair-Goldensohn, Z. Zhang, W. Fan, and J. Prager. Mining the web for answers to natural language questions. In *Proceedings of the tenth international conference on Information and knowledge management*, pages 143–150. ACM Press, 2001.

[19] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, 1999.

[20] H. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[21] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[22] K.-P. Yee, K. Swearingen, K. Li, and M. Hearst. Faceted metadata for image search and browsing. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 401–408. ACM Press, 2003.