Sampling Search-Engine Results*

Aris Anagnostopoulos†
Yahoo! Research
701 First Avenue
Sunnyvale, CA 94089, USA
aris@yahoo-inc.com

Andrei Z. Broder†
Yahoo! Research
701 First Avenue
Sunnyvale, CA 94089, USA
broder@yahoo-inc.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 with 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 efficient algorithms for obtaining uniform random samples applicable to any search engine that is based on posting lists and document-at-a-time evaluation. (To our knowledge, all popular Web search engines, for example, Google, Yahoo Search, MSN Search, Ask, 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 *sample-next(p)* method that samples term posting lists with probability p, and show how to construct *sample-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.

1 Introduction

Web search continues its explosive growth: according to the Pew Internet & American Life Project [12], 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: a study during the beginning of 2005 argues that the size of the indexable Web is at least 11.5 billion pages [16].

*A preliminary version of this work has appeared in [3]

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

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 [21]), which result in very large result sets. Although by now search engines have become very accurate with respect to navigational queries (see [7] 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 in early 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 `ask.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 [6]) 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 *multidimensional 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. [25] 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, a procedure that 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, the average running time of one of our sampling approaches is only proportional to the sample size. 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 meta-data 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., [4, 14, 19]). 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 operates on a sample [17]; as a result the DB2 relational database has been augmented to support this option.

1.2 Further Applications

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 2.3 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 2.4 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 (see the discussion in [5]). As an example, as of early 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`" is about 33M. We get similar inconsistencies with other search engines, such as MSN Search. In contrast, in our experiments (see Section 5) 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 Amitay et al. [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, which 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 Radev et al. [20], who propose 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.

Finally, it might be possible to use sampling on the results of search-engine queries in order to extract summary information from the ensemble of the results and then we can use this information as a means of providing feedback to the user in order to refine his query.

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.

1.3 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 [15], 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, and this 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.

1.4 Retrieval Model and Notations

Our model is a traditional *Document-at-a-time* (DAAT) model for IR systems [22]. Every document in the database is assigned a unique document identifier (DID). 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 the posting list of some term A supports the following standard operations.

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

For our purposes, we need a special operator

4. $A.jump(r, s)$: moves the pointer to the sth entry in the posting list after the document with DID greater than or equal to r, and returns this DID. (Equivalent to $A.next(r)$ followed by s $A.next()$ operations. However, simulating $A.jump(r, s)$ 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 $jump(r, s)$ 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, $(A \,\mathbf{OR}\, B).next() = \min(A.next(), B.next())$. We will show later how to construct a basic *sample-next*(p) method that samples term posting lists with probability p, and show how to construct *sample-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 in 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 pointer move operations. In the rest of the paper, we assume that these moves (i.e., *next*, *jump*, and *sample-jump*) 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 [8]. 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 $k \ll m$, and we let $p_s = k/m$ to be the ideal sampling probability.

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, which was first introduced in [8], and this specialization allows us to achieve better performance. Subsequently, in Section 4, we present an alternative scheme to sample results; this method is more efficient theoretically, but probably less efficient in practice. We implemented some of our algorithms and performed various experiments, and we present the results in Section 5. In Section 6 we summarize and discuss 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 $p_s = 1000/5M = 1/5000$ creates a random sample with the desired expected size.

We 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 Section 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_s. 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_s and create a virtual term A_{p_s} whose posting list contains the sampled postings of A. Then the posting list for A_{p_s} contains roughly $10M/5000 = 2000$ documents. We return the documents satisfying the query A_{p_s} **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_s, we can create the posting list of A_{p_s} online in time proportional to $|A_{p_s}|$; hence, this method allows us to examine only 2000 postings, a clear gain over the $10M$ postings examined by the naive approach.

Now let us 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 $p_s = 1000/15M = 1/15000$. The naive approach is to check every document in A **OR** C and insert it into the sample with probability p_s, 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_s} in time proportional to $|A_{p_s}|$. However, a document may satisfy the query even if it does not contain A, so we create also a virtual term C_{p_s} in the same manner, and return documents in A_{p_s} **OR** C_{p_s}. Thus the total number of postings examined is $|A_{p_s}| + |C_{p_s}| = 20M/15000 \simeq 1333$, so we have 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_s} with probability p_s, and, similarly, if it contains only the term C then it is inserted in C_{p_s} with probability p_s. But if a document contains both terms, the probability to be contained in either A_{p_s} or C_{p_s} is $2p_s - p_s^2$. Hence, every document containing both A and C and contained in A_{p_s} **OR** C_{p_s} must be rejected from the sample with probability $1 - p_s/(2p_s - p_s^2)$. This will ensure that every document in A **OR** C is included in the sample with probability exactly p_s.

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 [6], AltaVista [9], and IBM's Trevi [13].

Consider a query Q, which 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 such as phrases or proximity operators. Every such query contains a number of simple terms, say $T_1, T_2, \ldots, T_t$, 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 $p_s = k/m$. Assume, moreover, for the moment that we know m, and therefore we know

the sampling probability p_s—in Section 2.3 we show how to handle this.

The way to sample the results is simple in concept. In a nutshell, we use rejection sampling to sample uniformly from the union of the posting lists $T_1, T_2, \ldots, T_t$, conditional on the sample satisfying the query.

For every term T_i (but not for terms **NOT** T_i) we create a *pruned posting list* of document entries that contains every document from the posting list of T_i with probability p_s, 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_s. 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_s. We can create a geometrically distributed random variable with parameter p_s, in constant time, by using the formula

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

where U is a real random variable uniformly distributed in the interval $[0, 1]$ (see [11]).

The random skip is then performed by executing a *jump*$(r,\ X)$ operation, where r is the last document considered. (Recall from the discussion of Section 1.4 that the data structure used for postings allows for efficiently skipping documents in the posting lists and thus in our model 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 $Q(T_1, T_2, \ldots)$ with the query

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

By this construction, every document that appears in some posting list has probability at least p_s 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 $1-(1-p_s)^r$. 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 $p_s/(1-(1-p_s)^r)$, so that overall the probability that the document is accepted becomes exactly p_s.

There are several remarks to be made about this technique:

- First we want to stress its generality, which 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 1.4, 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_s providing the primitive *sample-next*(p_s) method, and the approach described above provides a *sample-next*(p_s) method for arbitrary queries: we first advance to the minimum posting in all pruned posting lists via the primitive *sample-next*(p_s) 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 [13] and JURU [10] search engines 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 $p_s = k/m$. 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 [23] was first to propose efficient algorithms to address that problem, using a technique called *reservoir sampling*. The main idea is that when the ith 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, which instead of checking every element skip over a number of them (see, for example, [23, 18]).

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 [14] in the context of stream processing: We maintain a buffer of size $B > k$ (e.g., B can equal $2k$), and set the initial sampling probability equal to some upper bound for the correct sampling probability, p_0; trivially we can set $p_0 = 1$. 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 $1 - \alpha$, all random choices being independent. 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 B 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.

To recapitulate, we present in Figure 1 a high-level description of the entire algorithm for sampling the results of a general query. Note that, for the sake of simplicity, the description does not give any details, nor does it present the most efficient implementation. For example, in Figure 1, in order to consider the next candidate document we consider only the pruned posting lists; in an actual implementation, we would consider both the pruned lists and the posting lists of the actual terms.

```
1. Function getSample()
2.    /* First some initializations. */
3.    curDoc ← 0
4.    p ← 1
5.    /* We assume that initially the pointers of all
       the terms' posting lists point to DID 0 */
6.
7.    foreach (term T_j)
8.       create a term T_{j,p} with the same posting list
          as T_j
9.
10.   repeat
11.      foreach (j : T_{j,p}.loc() = curDoc)
12.         T_{j,p}.nextPruned(curDoc)
13.      curDoc ← min{T_{j,p}.loc(), j = 1, ..., t}
14.      if (curDoc = lastID)
15.         return /* Finished with all the
             documents */
16.      foreach (j : T_j.loc() < curDoc)
17.         T_j.next(curDoc)
18.      r ← |{j : T_j.loc() = curDoc}|
19.      if (curDoc satisfies Q)
20.         with probability normalizedProbability(r)
             addToSample(curDoc)
21.   end repeat

1. Function T.nextPruned(r)
2.    X ← Geometric(p)
3.    T.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 α = p'/p
9.       p ← p'
```

Figure 1: The General Sampling Scheme. We assume that we want to sample a query Q where terms $T_1, T_2, \ldots, T_t$ appear nonnegated.

To estimate the running time of the algorithm, observe that the number of times that the sampling probability is decreased is bounded by

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

Every time the probability is decreased the expected number of samples removed from the buffer is $(1-\alpha)B$. Thus, assuming that $B = \Theta(k)$, the expected total number of samples considered can

be bounded by approximately

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

Using this fact, and under independence assumptions that are common in information retrieval for the containment of terms in documents, we can show that the expected running time of this sampling scheme is bounded by approximately

$$O\big(k\ln(m/k)\big),$$

for any fixed query, and $k, m \to \infty$. The analysis is similar to the one that we present later in Section 3.3 so we omit it.

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 that we sampled from. This is captured by Theorems 2.1, 2.3, and 2.4.

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. 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 if and only if it has $X_j < p'$.

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 we mentioned, an additional feature of the algorithm is that we can estimate the number of documents matching the query. Theorem 2.3 summarizes the result.

The main tool that we use in the proof of Theorem 2.3 is the concept of a martingale, and here we present the definition and the main result that we are using.

Definition 2.1. *Consider a sequence $\{X_t, t = 0, 1, \ldots\}$ of random variables, and a family of sets of random variables $\{\mathcal{H}_t, t = 0, 1, \ldots\}$, where $\mathcal{H}_{t-1} \subset \mathcal{H}_t$. We say that* the sequence $\{X_t\}$ forms a martingale with respect to $\{\mathcal{H}_t\}$ *if for every $t \geq 0$ the following three properties hold:*

1. *X_t is a function of $\mathcal{H}_t$.*

2. *X_t is integrable, that is,* $\mathbf{E}[|X_t|] < \infty$.

3. $\mathbf{E}[X_{t+1} \mid \mathcal{H}_t] = X_t$.

Intuitively, $\mathcal{H}_t$ corresponds to the history up to time t.

Definition 2.2. *A random variable T taking values in $\{0, 1, 2, \dots\} \cup \{\infty\}$ is called a* stopping time *with respect to $\{\mathcal{H}_t\}$, if for every $t \in \{0, 1, 2, \dots\}$ the indicator function of the event $\{T = t\}$ can be written as a function of the random variables in $\mathcal{H}_t$.*

This means that T is a stopping time if it is decidable whether or not $T = t$ with a knowledge only of the past and present, $\mathcal{H}_t$, and with no further information about the future.

Having defined a martingale and a stopping time, we are now able to present and prove a (nonstandard) version of the *Optional Sampling Theorem* that we use in our proof.

Theorem 2.2 (Optional Sampling Theorem)**.** *Consider a martingale $\{X_t, t = 0, 1, \dots\}$ with respect to $\{\mathcal{H}_t\}$ and assume that T is a stopping time, such that $\mathbf{Pr}(T < \infty) = 1$. Then we have that $\mathbf{E}[X_T] = \mathbf{E}[X_0]$ if there is a constant A independent of t, such that for every $t = 0, 1, 2, \dots$ we have that $\mathbf{E}[|X_{t \wedge T}|^2] < A$, where $t \wedge T = \min\{t, T\}$.*

Proof. Since the process $\{X_t\}$ forms a martingale with respect to $\{\mathcal{H}_t\}$, also the stopped process $\{X_{t \wedge T}\}$ forms a martingale with respect to $\{\mathcal{H}_t\}$, so, in particular, $\mathbf{E}[X_{t \wedge T}] = \mathbf{E}[X_0]$ for $t = 0, 1, 2, \dots$.

The fact that $\mathbf{E}[|X_{t \wedge T}|^2] < A$ implies that the sequence $\{X_{t \wedge T}\}$ is uniformly integrable, and the fact that $\mathbf{Pr}(T < \infty) = 1$ implies that $X_{t \wedge T} \to X_T$ almost surely. Therefore (see, for example, [24, page 131]),

$$\lim_{t \to \infty} \mathbf{E}[X_{t \wedge T}] = \mathbf{E}[X_T],$$

which concludes the proof. □

We are now in position to state and prove that our algorithm provides an unbiased estimator for the number of matches.

Theorem 2.3. *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 with probability α; 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 are K_t documents in the sample, the sampling probability is p_t, and we have considered m_t candidate documents. Thus if the algorithm stops after f steps, $K_f = K$, $p_f = p^*$, and $m_f = m$. If $m_t = m$, we also define $m_{t+1} = m_t = m$, $K_{t+1} = K_t$, and $p_{t+1} = p_t$. Now we define a sequence of random variables $\{X_t, t = 0, 1, \dots\}$ as follows. We let $X_0 = 0$ and for $t \geq 1$ we have

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

We now show that the sequence $\{X_t\}$ is a martingale with respect to $\{\mathcal{H}_t\}$, where $\mathcal{H}_t = (K_0, p_0, m_0, K_1, p_1, m_1, \dots, K_t, p_t, m_t)$. This will finally imply that $\mathbf{E}[K_f/p_f] - \mathbf{E}[m_f] = 0$, which is what we want to prove.

It's clear that X_t is a a function of $\mathcal{H}_t$, while for the integrability notice that

$$\mathbf{E}[|X_t|] = \mathbf{E}\left[\left|\frac{K_t}{p_t} - m_t\right|\right] \leq \frac{B}{\alpha^t} + m < \infty,$$

where we used the fact that $K_t \leq B$, and $m_t \leq m$.

It remains to show that $\mathbf{E}[X_{t+1} \mid \mathcal{H}_t] = X_t$, the basic martingale property. If $m_t = m$, then the property holds trivially; if $m_t < m$ we consider two cases: First, if $K_t < B$ then the sampling probability does not change ($p_{t+1} = p_t$) but we consider a new document, which is inserted with probability $p_{t+1} = p_t$. Therefore, if we let Z be the indicator of the event that at time $t+1$ a document is accepted, we get

$$\begin{aligned}
\mathbf{E}[X_{t+1} \mid \mathcal{H}_t] &= \mathbf{E}\left[\frac{K_t + Z}{p_{t+1}} - m_{t+1} \,\middle|\, \mathcal{H}_t\right] \\
&= \mathbf{E}\left[\frac{K_t + Z}{p_t} - (m_t + 1) \,\middle|\, \mathcal{H}_t\right] \\
&= \frac{K_t + p_t}{p_t} - m_t - 1 = X_t.
\end{aligned}$$

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

$$\begin{aligned}
\mathbf{E}[X_{t+1} \mid \mathcal{H}_t] &= \mathbf{E}\left[\frac{\text{Binomial}(K_t, p_{t+1}/p_t)}{p_{t+1}} - m_{t+1} \,\middle|\, \mathcal{H}_t\right] \\
&= \mathbf{E}\left[\frac{\text{Binomial}(K_t, \alpha)}{\alpha p_t} - m_t \,\middle|\, \mathcal{H}_t\right] \\
&= \frac{K_t \alpha}{\alpha p_t} - m_t = X_t.
\end{aligned}$$

Hence, we conclude that the sequence $\{X_t\}$ forms a martingale with respect to $\{\mathcal{H}_t\}$. We define the stopping time $f = \min\{t : m_t = m\}$. We will apply Theorem 2.2 for the martingale $\{X_t\}$ and the stopping time f, which will allow us to conclude that $\mathbf{E}[X_f] = \mathbf{E}[X_0] = 0$, therefore $\mathbf{E}[K_f/p_f] = E[m_f] = m$.

It is not hard to show that $\mathbf{Pr}(f < \infty) = 1$, but in order to apply Theorem 2.2 we have also to show that the second moment $\mathbf{E}[|X_{t\wedge f}|^2]$ is uniformly bounded (over t) by some constant. To this end, it helps to define for $t = 0, 1, 2, \ldots$ the random variable $Y_t = \log_\alpha p_t$. Then Y_t counts how many times the algorithm resampled from the buffer up to time t. We have

$$\begin{aligned}
\mathbf{E}[|X_{t\wedge f}|^2] &= \mathbf{E}\left[\left|\frac{K_{t\wedge f}}{p_{t\wedge f}} - m_{t\wedge f}\right|^2\right] \\
&\leq \mathbf{E}\left[\frac{K_{t\wedge f}^2}{p_{t\wedge f}^2}\right] + \mathbf{E}[m_{t\wedge f}^2] + 2\,\mathbf{E}\left[\frac{K_{t\wedge f}}{p_{t\wedge f}} m_{t\wedge f}\right] \qquad (2) \\
&\leq B^2\,\mathbf{E}\left[\frac{1}{p_{t\wedge f}^2}\right] + m^2 + 2Bm\mathbf{E}\left[\frac{1}{p_{t\wedge f}}\right],
\end{aligned}$$

where we used the fact that $K_{t\wedge f} \leq B$, and $m_{t\wedge f} \leq m$. We now show that the term $\mathbf{E}\left[\frac{1}{p^2_{t\wedge f}}\right]$ is uniformly bounded. We have

$$\begin{aligned}
\mathbf{E}\left[\frac{1}{p^2_{t\wedge f}}\right] &\leq \sum_{i=0}^{\infty} \mathbf{Pr}\left(\frac{1}{p^2_{t\wedge f}} \geq i\right) \\
&= \sum_{i=0}^{\infty} \mathbf{Pr}\left(\frac{1}{p_{t\wedge f}} \geq \sqrt{i}\right) \\
&= \sum_{i=0}^{(1/\alpha)^{2m}-1} \mathbf{Pr}\left(\frac{1}{p_{t\wedge f}} \geq \sqrt{i}\right) + \sum_{i=(1/\alpha)^{2m}}^{\infty} \mathbf{Pr}\left(\frac{1}{p_{t\wedge f}} \geq \sqrt{i}\right) \\
&\leq \left(\frac{1}{\alpha}\right)^{2m} + \sum_{i=(1/\alpha)^{2m}}^{\infty} \mathbf{Pr}\left(\left(\frac{1}{\alpha}\right)^{Y_{t\wedge f}} \geq \sqrt{i}\right) \\
&= \left(\frac{1}{\alpha}\right)^{2m} + \sum_{i=(1/\alpha)^{2m}}^{\infty} \mathbf{Pr}\left(Y_{t\wedge f} \geq \log_{1/\alpha}\sqrt{i}\right) \\
&\leq \left(\frac{1}{\alpha}\right)^{2m} + \sum_{i=(1/\alpha)^{2m}}^{\infty} \mathbf{Pr}\left(Y_f \geq \log_{1/\alpha}\sqrt{i}\right),
\end{aligned}$$

since Y_t is increasing with t.

We will now bound the probability of the event $\mathcal{E}_i = \{Y_f \geq \log_{1/\alpha}\sqrt{i}\}$. Recall that Y_t counts how many times the algorithm resampled from the buffer up to time t, so, since there are m documents in total, event $\mathcal{E}_i$ implies that in at least $\log_{1/\alpha}\sqrt{i} - m$ resample steps no document was evicted from the buffer. Since in every resampling step each document that is in the buffer stays in the buffer with probability α, and since the buffer contains B documents at every resampling step, the probability of event $\mathcal{E}_i$ is bounded by $\alpha^{B(\log_{1/\alpha}\sqrt{i}-m)}$. Therefore we get

$$\begin{aligned}
\mathbf{E}\left[\frac{1}{p^2_{t\wedge f}}\right] &\leq \left(\frac{1}{\alpha}\right)^{2m} + \left(\frac{1}{\alpha}\right)^{Bm} \sum_{i=(1/\alpha)^{2m}}^{\infty} \alpha^{\log_{1/\alpha} i^{B/2}} \\
&= \left(\frac{1}{\alpha}\right)^{2m} + \left(\frac{1}{\alpha}\right)^{Bm} \sum_{i=(1/\alpha)^{2m}}^{\infty} \frac{1}{i^{B/2}},
\end{aligned}$$

which is bounded for $B > 2$.

This implies that also $\mathbf{E}\left[\frac{1}{p_{t\wedge f}}\right]$ is uniformly bounded, and by Equation (2) the expectation $\mathbf{E}[|X_{t\wedge f}|^2]$ is also uniformly bounded. So, we can apply Theorem 2.2 and get that $\mathbf{E}[X_f] = \mathbf{E}[X_0] = 0$, which finally implies that

$$\mathbf{E}\left[\frac{K}{p^*}\right] = \mathbf{E}\left[\frac{K_f}{p_f}\right] = \mathbf{E}[m_f] = m.$$

Hence, K/p^* is an unbiased estimator for the number of documents satisfying the query. □

Besides having the correct expectation, a good estimator should be close to the correct value with high probability.

Definition 2.3. *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 2.4. *There are constants* c_1, c_2 *such that for any positive* $\epsilon < 1$ *and* $\delta < 1$*, the algorithm above with a buffer size* $B = \frac{c_1}{\epsilon^2} \ln \frac{c_2}{\delta}$ *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 [14]. We assume that the initial sampling probability is $p_0 = 1$ and we have $p_i = \alpha p_{i-1}$, so that $p_i = \alpha^i$. In order to simplify some calculations we assume that $\alpha \leq 3/4$. We set the buffer size to be

$$B = \frac{1+\epsilon}{\alpha} \cdot \frac{3}{\epsilon^2} \ln \frac{8}{\delta}.$$

We use the coupling argument (the one used in Theorem 2.1) and think of the algorithm as sampling from all documents and working by levels. Let Y_0 be the number of all the matching documents, Y_1 be a random variable counting the number of documents that got sampled with probability α, Y_2 be a random variable that counts the documents that further got sampled with probability α and so on. Then Y_i is distributed as Binomial(m, α^i).

Here is the main idea. The final outcome size K equals one of the Y_i's (the one for which we have $Y_{i-1} \geq B$ and $Y_i < B$) and then p^* equals α^i. The idea is that for small i, with good probability, the estimates Y_i/α^i are accurate and if K equals one of those then the algorithm provides a good estimate. Otherwise K equals one of the later Y_i's, but this has small probability.

We define the events

$$\mathcal{B}_i = \left\{\left|\frac{Y_i}{\alpha^i} - m\right| > \epsilon m\right\} = \{|Y_i - m\alpha^i| > \epsilon m \alpha^i\}.$$

We also define

$$\ell = \max\ i \quad \text{s.t.} \quad m\alpha^i \geq \frac{3}{\epsilon^2} \ln \frac{8}{\delta},$$

and therefore we have

$$m\alpha^{\ell+1} < \frac{3}{\epsilon^2} \ln \frac{8}{\delta}.$$

So, the probability that the estimator fails to be within a factor of ϵ close to m is bounded by

$$\sum_{i=1}^{\ell} \mathbf{Pr}(\mathcal{B}_i) + \mathbf{Pr}(Y_\ell \geq B).$$

By applying a Chernoff bound we have that for $i \leq \ell$

$$\mathbf{Pr}(\mathcal{B}_i) \leq 2e^{-\frac{\epsilon^2}{3} m\alpha^i}.$$

If $f(i) = e^{-\frac{\epsilon^2}{3}m\alpha^i}$, then for $i \leq \ell$ we have $f(i)/f(i-1) \geq (8/\delta)^{(1-\alpha)/\alpha}$, which for $\alpha \leq 3/4$ is at least 2, for any $\delta < 1$. So for the summation we have

$$\begin{aligned}\sum_{i=1}^{\ell} \mathbf{Pr}(\mathcal{B}_i) &\leq \sum_{i=1}^{\ell} 2e^{-\frac{\epsilon^2}{3}m\alpha^i} \\ &\leq 4e^{-\frac{\epsilon^2}{3}m\alpha^\ell} \\ &\leq 4e^{-\frac{\epsilon^2}{3}\frac{3}{\epsilon^2}\ln\frac{8}{\delta}} \\ &= \frac{\delta}{2}.\end{aligned}$$

For $\mathbf{Pr}(Y_\ell \geq B)$ we have

$$\begin{aligned}\mathbf{Pr}(Y_\ell \geq B) &= \mathbf{Pr}\left(Y_\ell \geq \frac{1+\epsilon}{\alpha}\frac{3}{\epsilon^2}\ln\frac{8}{\delta}\right) \\ &\leq \mathbf{Pr}\left(Y_\ell \geq \frac{1+\epsilon}{\alpha}m\alpha^{\ell+1}\right) \\ &= \mathbf{Pr}\left(Y_\ell \geq (1+\epsilon)\cdot m\alpha^\ell\right) \\ &\leq e^{-\frac{\epsilon^2}{3}m\alpha^\ell} \\ &\leq \frac{\delta}{8}.\end{aligned}$$

Therefore, the probability that the algorithm fails is less than δ. □

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 [8], which 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 [8] 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

$$\sum_{1 \leq i \leq k} x_i w_i \geq \theta, \tag{3}$$

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 satisfy Equation (3) with X_i indicating the presence of query term T_i in document d. We note however that the situation considered in [8] 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 [8], 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*, *jump*, and *sample-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 $T = \{T_1, \ldots, T_t\}$ be the set of the query terms with associated positive weights, $w_1, w_2, \ldots, w_t$. Our goal is to sample uniformly at random documents from the set of documents $\{j\}$ that satisfy the inequality

$$\sum_{1 \le i \le t} x_{i,j} w_i \ge \theta, \tag{4}$$

where $x_{i,j}$ is given by

$$x_{i,j} = \begin{cases} 1, & \text{if document } j \text{ contains } T_i \\ 0, & \text{otherwise.} \end{cases}$$

We divide T 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 (4) must contain at least one term from S. It is easy to check that the **AND** and the **OR** examples of Section 2.1 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_s, independently of other documents and other terms.

Of course the algorithm does not know p_s 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 Section 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 Section 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 2, while a complete description and a formal proof of correctness can be found in Section 3.4; Figure 3 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 of documents, which is geometrically distributed.

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), 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 Section 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

```
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].DID ≤ 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].DID ≤ 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].DID = 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 α = p′/p
9.     p ← p′
```

Figure 2: Sampling **WAND**.

some checker has to be advanced there is usually more than one choice. The goal is to select the

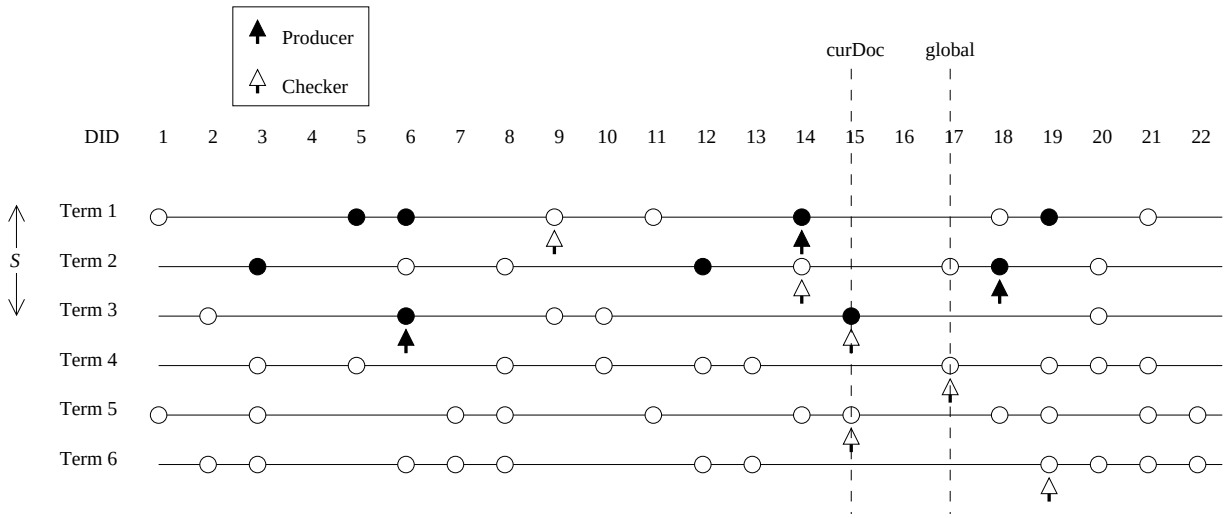

Figure 3: An example of the posting lists. A bullet indicates that the term exists in the corresponding document. A black bullet indicates that the document was sampled (or will be), hence it exists in the pruned list.

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 [9]). 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 (e.g., 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 reduces the running time by using 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 $p_s = k/m$. Consider a query with t terms, and recall that N_i is the total number of documents containing the ith term and that w_i is the weight of the ith 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 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 $p_s \sum_{i \in S} N_i$. Therefore, the running time is expected to be

$$O\left(tp_s \sum_{i \in S} N_i\right) = O\left(t\frac{k}{m} \sum_{i \in S} N_i\right). \tag{5}$$

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, $\pi_{i,1}, \pi_{i,2}, \ldots, \pi_{i,s}$ that d satisfies the query and contains exactly $1, 2, \ldots, s$ terms from S, where $s = |S|$. 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

$$\sum_{j=1}^{s} \frac{\pi_{i,j} p}{1-(1-p)^j} \geq \sum_{j=1}^{s} \frac{\pi_{i,j}}{j} \stackrel{\triangle}{=} \rho_i.$$

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

$$O\bigg(\frac{k}{\rho_i} \ln(m/k)\bigg),$$

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

$$O\bigg(tk \ln(m/k) \sum_{i \in S} \frac{1}{\rho_i}\bigg) = O\big(k \ln(m/k)\big), \tag{6}$$

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 (6) 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

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

Then let $S = \{1, 2, \ldots, s\}$. 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,

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

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 the independence assumptions for the containment of terms in documents; in reality, however, the running time will depend on the actual joint distribution of the query terms, which 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.

3.4 Detailed Description of the Algorithm and Proof of Correctness

Here we present a detailed description as well as a formal proof of the correctness of the algorithm presented in Figure 2. The essence of the proof is to show that a set of four invariants is maintained throughout the execution of the protocol. For $i \in S$, let

$$\mathcal{C}_i = \{\text{DIDs that appear in the pruned list of term } i\},$$

and $\mathcal{C} = \bigcup_{i \in S} \mathcal{C}_i$. The four invariants are the following:

1. All documents with DID $\leq$ *curDoc* have either been considered as candidates, or do not belong to $\mathcal{C}$.

2. For any term $i \in T$, any document containing i with DID $<$ *checker*$[i]$.*DID* has either been considered as a candidate, or does not belong to $\mathcal{C}$.

3. At every given time point, every document with DID $<$ global has either been considered or does not belong to $\mathcal{C}$.

4. For all terms $i \in S$, every document containing i with DID $<$ *producer*$[i]$.*DID* has either been considered or does not belong to $\mathcal{C}_i$.

It is easy to verify that all for of them are true after the initializations (line 9).

We next try to find the next candidate document. We increase *global* to equal the minimum DID that could be a potential candidate, which happens when the sum of the weights of terms whose *checker*s are behind (i.e., point to documents with DID smaller than or equal to) *global* reaches the threshold θ. Since we increase *global* we must verify that invariant 3 is maintained after the execution of line 12. This is indeed true by invariant 2. $[2 \Rightarrow 3]$

The next candidate document must be pointed to by some *producer*. Therefore, at lines 13–14, if all the *producer*s are ahead of *global* we increase *global* to the smallest *producer*. The validity of invariant 3 follows from invariant 4. [4 ⇒ 3]

The next candidate document is the one with the smallest DID ≥ *global* that exists in some pruned list. In order to discover it, we advance all the *producer*s that are behind *global* in their pruned list to their first document with DID ≥ *global* (lines 16–19). Notice that we can stop if we discover a pruned list that contains *global*. [3 ⇒ 4]

By line 20, either some *producer* points to *global*, or we have advanced all the *producer*s past *global*. In the latter case (lines 20–21) we increase *global* to the smallest of the *producer*s. [4 ⇒ 3]

At lines 22–23 we check whether we have traversed all the documents and in that case the sampling is finished. Otherwise *global* points to the next candidate document. Notice that if the DID of that document is ≤ *curDoc*, then the document has been considered in the past (invariant 1) so we can advance one of the *checker*s past *curDoc* (lines 27–31). (Actually *global* cannot be strictly smaller than *curDoc*.) [1 ⇒ 2]

Otherwise we have a candidate new document. The next step (line 33) is to check whether we have discovered enough terms contained in the document, that is, we check whether the sum of the weights of terms whose *checker*s point to *global* exceeds the threshold θ. If this is the case, then we consider the new document as a candidate and we set *curDoc* to its DID [3 ⇒ 1]. Now the document should be inserted to the sample after applying the rejection scheme described in Section 2.3. Since we want the probability of the document being sampled to be exactly p, we must compute the probability that at least some pruned list contains the document. Consider all the terms S that have *producer*s (and associated pruned lists). Assume that the document appears in the posting lists of r such terms. Then for each of those, the probability to appear to the corresponding pruned list is p, therefore the probability to appear in *some* pruned list equals $1-(1-p)^r$. Hence we must normalize and accept the document with probability $p/(1-(1-p)^r)$ (lines 39–40). Previously, at lines 37–38, we advance the *checker*s of all terms in S in order to compute r. [1 ⇒ 2]

In the case that the sum of the weights of terms whose *checker*s point to *global* is less than θ (i.e., in the case that the **if** clause of line 33 evaluates to false) we chose one of the terms whose *checker* is behind *global* (and therefore contributed to the upper bound at line 12) and we advance the corresponding *checker* to the first smallest DID ≥ *global*. [1 ⇒ 2]

When the execution of the algorithm is over, invariant 1 proves that every document in $\mathcal{C}$ gets at some point to be considered as a candidate. At line 39, the set D contains exactly those terms that exist in set S and are contained in document *global*. This follows from the definition of the set D and from invariant 2, combined with the **foreach** loop at line 37. (If a posting list contains the document *global* then its *checker* cannot be ahead of *global*, otherwise invariant 2 would have been violated. It is therefore behind, and the **foreach** loop makes sure that the *checker* will point exactly to *global*.) Therefore we count all the terms in S that exist in document *global*, hence (because of the normalization) the document becomes accepted with probability exactly p.

4 An Alternative Sampling Scheme

In this section we present a different way for adjusting the sampling probability. As we will see, this technique is faster theoretically, but in practice it may not be as efficient, as it may require traversing a term's posting list more than once.

Here is the main idea. For concreteness, we present the algorithm for sampling a **WAND** query,

although the same method can be applied for a general query (where the set S will include all the query terms that appear in the query and are not negated, as in Section 2.2). Recall that N_i is the number of documents in the posting list of the ith term. We let $N = \sum_{i \in S} N_i$. By Equation (5), whenever we sample with probability p, the expected running time is bounded by $O(tpN)$. In this scheme we sample initially with some small probability $p_0 = 1/N$. Then the expected time needed to scan all the lists is $O(t)$. If we sample at least k documents, then we stop and we have a uniform sample (like previously, if we end with more than k documents then we further select a sample of exactly k documents, by sampling from the final set without replacement). Otherwise, we set $p_1 = 2p_0$, and repeat. In general, if in the ith step the buffer did not become full, then we increase the sampling probability $p_{i+1} = 2p_i$ (so $p_i = 2^i/N$), and we restart. Let $\ell = \log_2 \frac{kN}{m}$. We are expected to finish when $p_i \simeq p_\ell = k/m$, so we expect the total time to be about

$$O\left(t\left(\sum_{i=0}^{\ell} p_i N\right)\right) = O\left(t\left(1 + 2 + 4 + \cdots + \frac{k}{m}N\right)\right) = O\left(t\frac{k}{m}N\right).$$

Let us try now to analyze the running time and the performance more rigorously. We prove the following lemma, which bounds the total number of producer advances.

Lemma 4.1. *The expected number of producer advances is bounded by $6kN/m$.*

Proof. We call the traversing of the posting list with probability p_i *round* i. Notice that for the ith round, the total number of producer advances is stochastically dominated by a binomial random variable, Binomial(N, p_i). Let the number of producer advances at round i be X_i (X_i equals 0 if the algorithm stopped before round i) and the total number of producer advances be $X = \sum_{i=0}^{\infty} X_i$. We then have

$$\mathbf{E}[X] = \sum_{i=0}^{\ell+1} \mathbf{E}[X_i] + \sum_{i=\ell+2}^{\infty} \mathbf{E}[X_i] \leq \sum_{i=0}^{\ell+1} \mathbf{E}[\text{Binomial}(N, p_i)] + \sum_{i=\ell+2}^{\infty} \mathbf{E}[X_i].$$

We denote by $\mathcal{A}_i$ the event that "the algorithm has not terminated up to (and including) round i." First notice that conditioning on $\overline{\mathcal{A}_{i-1}}$ we have $X_i = 0$. Also, event $\mathcal{A}_i$ implies that at round i there were fewer than k documents sampled. Since at level i the number of documents that become sampled is distributed as Binomial(m, p_i), the expected number of sampled documents is $m2^i/N$, and for $i \geq \ell + 1$ we get by a Chernoff bound

$$\begin{aligned}
\Pr(\mathcal{A}_i) &= \mathbf{Pr}(\text{Binomial}(m, 2^i/N) < k) \\
&= \mathbf{Pr}\left(\text{Binomial}(m, 2^i/N) < \frac{k}{m2^i/N} m2^i/N\right) \\
&\leq e^{-\frac{1}{2}\frac{m2^i}{N}\left(1-\frac{k}{m2^i/N}\right)^2} \\
&\leq e^{-\frac{1}{8}\frac{m2^i}{N}}.
\end{aligned}$$

From the previous calculation and from the fact that conditional on event $\mathcal{A}_{i-1}$ the random variable X_i is stochastically dominated by a Binomial(N, p_i), we get for $i \geq \ell + 2$

$$\begin{aligned}
\mathbf{E}[X_i] &= \mathbf{E}[X_i \mid \mathcal{A}_{i-1}] \cdot \mathbf{Pr}(\mathcal{A}_{i-1}) + \mathbf{E}[X_i \mid \overline{\mathcal{A}_{i-1}}] \cdot \mathbf{Pr}(\overline{\mathcal{A}_{i-1}}) \\
&\leq N\frac{2^i}{N} e^{-\frac{1}{8}\frac{m2^{i-1}}{N}} + 0,
\end{aligned}$$

and so

$$\begin{aligned}
\mathbf{E}[X] &\leq \sum_{i=0}^{\ell+1} 2^i + \sum_{i=\ell+2}^{\infty} 2^i e^{-\frac{1}{8}\frac{1}{N}m2^{i-1}} \\
&\leq 2^{\ell+2} + 2^{\ell} \sum_{i=\ell+2}^{\infty} 2^{i-\ell} e^{-\frac{1}{8}\frac{1}{N}m2^{i-1}} \\
&= \frac{4kN}{m} + \frac{kN}{m} \sum_{j=2}^{\infty} 2^j e^{-\frac{1}{8}\frac{1}{N}m2^{\ell+j-1}} \\
&= \frac{4kN}{m} + \frac{kN}{m} \sum_{j=2}^{\infty} 2^j e^{-\frac{1}{16}k2^j} \\
&\leq \frac{6kN}{m},
\end{aligned}$$

for $k \geq 6$. □

Since, as we argued right before Equation (5), the number of advances that each checker performs is bounded by the total number of producer advances, and by making use of Lemma 4.1, we have proven the following theorem.

Theorem 4.1. *The expected running time of the algorithm is $O(tkN/m)$.*

Let us compare now the running time of this scheme with the one of the first algorithm. According to Equation (6) the expected running time of the first algorithm under independence assumptions is approximately

$$O\left(tk \ln(m/k) \sum_{i \in S} \frac{1}{\rho_i}\right).$$

Recall from the discussion after Equation (6) that $\rho_i \approx m/N_i$, therefore the expected running time is approximately

$$O\left(tk \ln\left(\frac{m}{k}\right) \frac{N}{m}\right).$$

Therefore, we can see that the first scheme is slower by a logarithmic factor than the second scheme. More importantly, Theorem 4.1 holds without the independence assumptions that are introduced in the analysis of the first scheme. Nevertheless, as we mentioned previously, we expect the second scheme to be less efficient in practice, since it requires accessing the terms' posting lists several times.

Now we show that the second algorithm also provides an (ϵ, δ)-approximation scheme to the number of documents that match the query.

Theorem 4.2. *There are constants c_1, c_2 such that for any positive $\epsilon < 1$ and $\delta < 1$, the algorithm above with a requested sample size $k = \frac{c_1}{\epsilon^2} \ln \frac{c_2}{\delta}$ 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 main idea is to show that the algorithm will not terminate in the first rounds (up to round $\ell - 1$), while if it terminates later it provides a good approximation to the number of documents matching the query.

Let $\mathcal{B}_i$ be the event that "the algorithm terminated at round i," and $\mathcal{C}_i$ the event that "the algorithm terminated at round i and failed to provide an estimation within ϵ to m." Finally, let $\mathcal{B}$ be the event that "the algorithm failed." Then we have

$$\mathbf{Pr}(\mathcal{B}) \leq \sum_{i=0}^{\ell-1} \mathbf{Pr}(\mathcal{B}_i) + \sum_{i=\ell}^{\infty} \mathbf{Pr}(\mathcal{C}_i).$$

In order to bound $\mathbf{Pr}(\mathcal{B}_i)$, we notice that the number of matches at the ith round is a random variable distributed as a Binomial$(m, 2^i/N)$. Therefore,

$$\begin{aligned}
\mathbf{Pr}(\mathcal{B}_i) &\leq \mathbf{Pr}(\text{Binomial}(m, 2^i/N) \geq k) \\
&\leq \mathbf{Pr}\left(\text{Binomial}(m, 2^i/N) \geq \frac{k}{2^i m/N}\frac{2^i m}{N}\right) \\
&\leq e^{-\frac{1}{3}\frac{2^i m}{N}\left(\frac{kN}{2^i m}-1\right)^2}.
\end{aligned}$$

Notice that if $f(i) = e^{-\frac{1}{3}\frac{2^i m}{N}\left(\frac{kN}{2^i m}-1\right)^2}$, then for $i \leq \ell-1$ and $k \geq 2$ we have $f(i)/f(i-1) \geq e^{7k/12} > 2$. Therefore,

$$\sum_{i=0}^{\ell-1} \mathbf{Pr}(\mathcal{B}_i) < 2 \cdot \mathbf{Pr}(\mathcal{B}_{\ell-1}) \leq 2e^{-\frac{k}{6}}.$$

Also,

$$\begin{aligned}
\mathbf{Pr}(\mathcal{C}_i) &\leq \mathbf{Pr}\left(\left|\frac{\text{Binomial}(m, 2^i/N)}{2^i/N} - m\right| > \epsilon m\right) \\
&= \mathbf{Pr}\left(\left|\text{Binomial}(m, 2^i/N) - \frac{2^i m}{N}\right| > \epsilon\frac{2^i m}{N}\right) \\
&\leq 2e^{-\frac{1}{3}\frac{2^i m}{N}\epsilon^2}.
\end{aligned}$$

So,

$$\begin{aligned}
\sum_{i=\ell}^{\infty} \mathbf{Pr}(\mathcal{C}_i) &\leq 2 \cdot \sum_{j=0}^{\infty} e^{-\frac{1}{3}\frac{2^{\ell+j} m}{N}\epsilon^2} \\
&\leq 2 \cdot \sum_{j=0}^{\infty} e^{-\frac{1}{3}2^j k\epsilon^2} \\
&\leq 3e^{-\frac{1}{3}k\epsilon^2}.
\end{aligned}$$

Putting everything together, we get

$$\mathbf{Pr}(\mathcal{B}) \leq 2e^{-\frac{k}{6}} + 3e^{-\frac{1}{3}k\epsilon^2},$$

which for

$$k = \frac{6}{\epsilon^2}\ln\frac{3}{\delta}$$

is less than δ. □

#	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.

5 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 [10].

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*, *jump*, or *sample-next*) 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 by creating 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	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.

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.

5.1 Estimating the Most Frequent Categories of the Search Results

We also evaluated the suitability of our approach for a particular application, namely the discovery of the most frequent categories spanned by the set of documents matched by a given query. We emphasize that we are not testing the uniformity of the samples: our samples are provably uniform; what we test here is whether uniform samples capture popular categories, something which of course depends on the distribution of categories over the result set. To this end 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, for each sample size, how many of the 10 most frequent categories in the full result set are present in the sample; we show the results in Table 3(a).

Furthermore, it is desirable for the frequent categories in the full result set to be also frequent in the sample so that we can identify them. For that, for each query, we check how many of the top-10 frequent categories in the result set are present within the top-10 frequent categories according to the sample, and we show the results in Table 3(b).

There are some facts worth noticing with respect to the results of sampling, some of which are not revealed in the tables. First observe that in most cases, even small sample sizes succeed in sampling documents from the frequent categories (Table 3(a)) but a somehow larger sample size is needed in order to ensure that the frequent categories are frequent in the sample as well (Table 3(b)). 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 3(b),

(a) Number of the top-10 frequent categories that appear in the samples.

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

(b) Number of the top-10 frequent categories that appear in the 10 most frequent sample categories.

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 4: Results from experiments on discovering popular categories using a random sample.

compared with Table 3(a). 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 3(b) (although, notice in Table 3(a) 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 rather small 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 10% chance to be a document contained within the top-10 categories, and negligible probability (0.57%) to be contained within the top 10th category.

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.

5.2 Estimating the Size of the Result Set

Finally we evaluate the quality of the estimator for the size of the result set. Table 5 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

Query	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 5: 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.

greater than 200. Here, however, the fact that the result sets are rather small plays to our advantage: our samples are relatively large, occasionally larger than the result set. Further research is needed to elucidate the case of very large result sets and to compare against the current performance of commercial search engines.

6 Summary

We propose performing sampling on the results of search-engine queries in order to support a plethora of applications, ranging from determining the set of categories in a given taxonomy spanned by the search results to estimating the size of the result set. We develop two general schemes for performing the sampling efficiently, in time essentially proportional to the sampling size, 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 of 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 [9], 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 the first scheme for sampling **WAND** that allows a rigorous analysis and requires fewer or no independence assumptions, remains a challenge.

7 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), which 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 of both the conference and, especially, the journal version.

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] A. Anagnostopoulos, A. Z. Broder, and D. Carmel. Sampling search-engine results. In *WWW '05: Proceedings of the 14th International Conference on World Wide Web*, pages 245–256. ACM Press, 2005.

[4] 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.

[5] J. P. Bagrow and D. ben-Avraham. On the *Google*-fame of scientists and other populations. In *Proceedings of the 8th Granada Seminar on Computational and Statistical Physics, "Modeling Cooperative Behavior in the Social Sciences"*, pages 81–89, 2005.

[6] 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.

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

[8] 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.

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

[10] 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.

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

[12] 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`.

[13] 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.

[14] 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.

[15] 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.

[16] A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *WWW '05: Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, pages 902–903. ACM Press, 2005.

[17] 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.

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

[19] S. Muthukrishnan. Data streams: Algorithms and applications. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-03)*, pages 413–413. ACM Press, 2003.

[20] 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.

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

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

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

[24] D. Williams. *Probability with Martingales*. Cambridge University Press, 1991.

[25] 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.