Optimizing Result Prefetching in Web Search Engines with Segmented Indices

Extended Abstract

Ronny Lempel Shlomo Moran

Department of Computer Science
The Technion,
Haifa 32000, Israel
email: {rlempel,moran}@cs.technion.ac.il

Abstract

We study the process in which search engines with segmented indices serve queries. In particular, we investigate the number of result pages which search engines should prepare during the query processing phase.

Search engine users have been observed to browse through very few pages of results for queries which they submit. This behavior of users suggests that prefetching many results upon processing an initial query is not efficient, since most of the prefetched results will not be requested by the user who initiated the search. However, a policy which abandons result prefetching in favor of retrieving just the first page of search results might not make optimal use of system resources as well.

We argue that for a certain behavior of users, engines should prefetch a constant number of result pages per query. We define a concrete query processing model for search engines with segmented indices, and analyze the cost of such prefetching policies. Based on these costs, we show how to determine the constant which optimizes the prefetching policy. Our results are mostly applicable to *local index* partitions of the inverted files, but are also applicable to processing of short queries in *global index* architectures.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

1 Introduction

The sheer size of the WWW and the efforts of search engines to index significant portions of it [14] have caused many search engines to partition their inverted index of the Web into several disjoint segments (partial indices). The partitioning of the index impacts the manner in which the engines process queries. Most engines also use some form of query result caching, where results of queries that were served are cached for some time. In particular, query results may be *prefetched* in anticipation of user requests. Such scenarios occurs when the engine retrieves (for a certain query) more results than will initially be returned to the user.

We examine efficient prefetching policies for search engines. These policies depend on the architecture of the search engine (which, in turn, affects its query processing scheme) and on the behavior patterns of search engine users.

1.1 Search Engine Users

Users submit queries to search engines. From a user's point of view, an engine answers each query with a linked set of ranked result pages, typically with 10 results per page. All users browse the first page of results (the results deemed by the engine's ranking scheme to be the most relevant to the query), and some scan additional result pages, usually in the natural order in which those pages are presented.

Three studies have analyzed the manner in which users query search engines and view result pages: a study by Jansen et al. [7], based on 51,473 queries submitted to the search engine *Excite*[1]; a study by Markatos [15], based on about a million queries submitted to Excite; and a study by Silverstein et al. [19], based on about a billion queries submitted to the search engine *AltaVista*[2]. Three findings which these studies share are particularly relevant to this work:

[1] http://www.excite.com/
[2] http://www.altavista.com/

- The queries submitted to WWW search engines are very short, averaging less than 2.4 terms per query, with over half of the queries containing just one or two terms. These results were reported by both [19] and [7]. While the two studies define *query terms* somewhat differently, the reported term counts may be loosely interpreted as the number of words per query.

- Users browse through very few result pages. The mentioned studies differ in the reported distribution of page views, but agree that at least 58% of the users view only the first page (the top-10 results), and that no more than 12% of users browse through more than 3 result pages.

- The number of distinct information needs of users is very large, as can be seen from the huge variety of queries submitted to search engines. However, popular queries are repeated many times, and the 25 most popular queries account for over 1% of all queries submitted to the engines.

1.2 Caching and Prefetching of Search Results

It is commonly believed that all major search engines perform some sort of search result caching and prefetching. Caching of results was noted in Brin and Page's description of the prototype of the search engine *Google*[3][3] as an important optimization technique of search engines. Markatos [15] demonstrated that caching search results can lead to hit ratios of close to 30%.

In addition to storing results that were requested by users in the cache, search engines may also *prefetch* results that they predict to be requested shortly. An immediate example is prefetching the second page of results whenever a new query is submitted by a user. Since studies [19, 7] indicate that the second page of results is requested shortly after a new query is submitted in at least 15% of cases, search engines may prepare and cache two (or more) result pages per query.

1.3 Index Structure and Query Processing Models

Inverted indices, or inverted lists/files, are regarded as the most widely applied indexing technique [18, 8, 20, 17, 16], and are believed to be used by the major search engines. As search engines index hundreds of millions of Web pages [14], the size of their inverted indices is measured in terabytes.

Ribeiro-Neto and Barbosa [17] mention three hardware configurations that can handle large digital libraries: a powerful central machine, a parallel machine, or a high-speed network of machines (workstations and high end desktops). However, when considering the size of the indices which search engines maintain, the growth rate of the Web and the large number of queries which search engines answer each day, using a network of machines is considered to be the most cost-effective and scalable architecture [6, 17]. Such networks operate in a *shared-nothing* memory organization [18] where each machine has its own processing power (one or several CPUs), its own memory and its own secondary storage. The machines communicate by passing messages via the high speed network that connects them.

There are two well-studied schemes of partitioning an inverted index across several machines:

- *Global index organization.* In this scheme, the inverted index is partitioned by terms. Each machine holds posting lists for a distinct set of terms (the terms may be partitioned by lexicographic order, for example). The posting list for term t holds entries for all documents that include t.

- *Local index organization.* In this scheme, the inverted index is partitioned by documents. Each machine is responsible for indexing a distinct set of documents, and will hold posting lists for all terms that appeared in its set of documents.

Some works, such as those by Ribeiro-Neto and Barbosa [17] and Tomasic and Garcia-Molina [20] have compared the (run-time) efficiency of the above schemes. Parallel generation of a global index has been studied in [18], while a system which crawls the Web and builds a distributed local index was presented in [16]. Cahoon et al. [4] evaluated the computational performance of local indices under a variety of workloads, and Hawking [6] examined scalability issues of local index organizations. The prototype of Google was reported as using global index partitioning [3].

Many of the above mentioned works [4, 18, 6, 17, 20] describe essentially the same model for processing queries in systems with segmented indices:

- User queries arrive to a certain designated machine, which we will call the *Query Integrator*, or QI. This machine was called *home site* in [20], *central broker* in [18] and [17], *user interface* (or UIF) in [6] and *connection server* in [4].

- The QI issues each query to the separate index segments, in a manner which depends on the partitioning scheme of the index. With local index partitioning, the QI will send the query (as submitted by the user) to all segments. With global index partitioning, the QI sends each segment a partial query consisting only of the set of terms whose posting lists are stored in the segment.

- The QI waits for the relevant segments to return their result sets, and merges these result sets with respect to the system's ranking scheme. Again,

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

the two index partitioning schemes imply different merge operations.

With local index partitioning, it is usually assumed that each segment has the ability to calculate the global score of each document in its local index with respect to all queries. Since the result sets that are returned by different segments are disjoint, merging the various result sets is straightforward.

With global index partitioning, each (relevant) segment returns a ranked document list that may overlap lists returned by other segments, and where each score reflects only the score of the document with respect to the partial query which that segment received. The QI may need to perform set operations on the partial result sets (for queries containing boolean operators), and might need to weigh the scores returned from each segment differently (for example, according to the different *idf* values of the terms in each partial query).

- The QI returns the merged results to the users.

We consider a *cache-augmented* process, in which the QI maintains a query-result cache. Upon receiving a query from a user, the QI first checks if the cache contains results for that query. If so, the cached results are returned to the user, without forwarding the query to any of the segments. If the query cannot be answered from the cache, the QI processes the query as described above, and upon completion, caches the merged results[4].

1.4 This work

When considering the query processing model described above in the context of Web search engines, we note that merged results are returned to users in small batches (typically 10 at a time), in decreasing order of relevance (as ranked by the search engine). The QI, however, may prepare more results than are to populate the first batch, and cache them for future use. This raises the issue of optimizing the number of prefetched results in systems where the cost of processing uncached queries increases with the number of results that are fetched: prefetching a large number of results per query will be costly at first, but may pay off should the user request additional batches of results (since these will already be cached). Note that with the cost of prefetching we also associate the cache space that is occupied by the prefetched results. Assuming a fixed-size cache, increasing the number of prefetched results per query may decrease the number of queries whose results can be simultaneously cached. This may lead to lower cache hit ratios, and to an increase in the load of the engine.

Another issue arising from the query processing model, is the relationship between the number of results which the QI decides to prefetch per query, and the number of results which it should ask of each segment. As an example, consider an engine which uses local partitioning into m segments, and whose policy is to prefetch n results per query. How many top results (denoted by l) should the QI collect from each segment for each query? It may happen that all of the top results reside on a particular segment. Therefore, in order to be *certain* that indeed all top n results are obtained, it is necessary to collect the top-n results of each segment (setting $l = n$). However, assuming that documents are partitioned randomly and independently into the segments, the QI may be able to collect considerably less results from each segment and still, with very high probability, obtain all of the top-n results. Thus, when optimizing the number of prefetched results n, the behavior of l with respect to n must also be considered.

The tradeoff between the amount (and cost) of result prefetching and the possibility of serving subsequent queries from the cache is the main topic of this paper. As popular search engines process millions of queries every day, efficient prefetching policies can help reduce both the hardware requirements and the response time of the engines.

The rest of this paper is organized as follows. Section 2 formally presents the problems studied and the notations used throughout this paper (the notations are summarized there in Table 1). We model both the search engine's query service process and the users' behavior. We then define the cost of prefetching a given number of results in terms of a cost function which is analyzed and optimized in later sections. Section 3 presents an algorithm which optimizes the prefetch cost function for two special cases. The first case deals with inverted indices that fit on a single machine. This single machine scenario also models serving single-term queries (which are quite common on the Web) with a globally-partitioned index. The second special case deals with a scenario when the engine guarantees that the users receive absolutely optimal results, using worst-case assumptions on the distribution of relevant documents in local-index partitions. The main body of work is contained in Section 4, which presents algorithms that solve and approximately solve the optimization problem for locally partitioned indices with an arbitrary number of segments, among which the documents are randomly distributed. Sections 5 tackles the combinatorial problem of setting the number of results which should be retrieved from each segment in order to provide quality merged results to the users. Section 6 discusses the practical impact that our results may have on search engine engineering. Conclu-

[4] The maintenance of the cache is not considered in this work. In particular, we do not examine how cached entries are replaced or how the freshness of the results is maintained.

sions and suggestions for future research are brought in Section 7.

2 Notations and Formal Model

2.1 The User

Our work requires a model for the way search engine users view result pages of their searches. Two studies [19, 7] have reported on several aspects of such user behavior by examining the query logs of search engines. To our purposes, the analysis of AltaVista's log [19] did not report in sufficient detail the exact distribution of result pages views (citing percentages of users viewing $1, 2$ and 3 pages only). In addition, the statistics reported in that paper only considered requests for additional results which arrived within 5 minutes of the previous request made by the same user. The study of Excite's users [7] brings a more elaborate distribution of result page views per query. 58% of the page views were of the first result page, 19% of the views were of the second result page, and the views of result pages 3-9 (21.3% of the views) conformed to a Geometric distribution with a parameter between 0.288 and 0.427.

We chose to model the number of result pages which users view per query as a Geometric random variable $u \sim \mathcal{G}(1-p)$. According to this model, users view result pages in their natural order, and the probability of a user viewing exactly result pages $1, \ldots, k$ (not viewing result pages $k+1$ and beyond) equals $(1-p)p^{k-1}$. In other words, upon viewing a result page, the user requests the next page with probability p.

An important property of the Geometric distribution is the fact that it is memoryless:

$$Pr(u \geq s+t \mid u \geq s) = p^t \quad \forall s,t \in \mathbb{N}$$

Assume that the complexity of retrieving ranked results is also "memoryless", meaning that the complexity of retrieving the results that rank in places $n, n+1, \ldots, n+(k-1)$ depends only on the number of results retrieved, k. As we will see, this assumption holds when the identity of the result that ranks in place $n-1$ is known. Then, the memoryless behavior of the users and the memoryless cost of retrieval implies that the optimal number of result pages r_{opt} that should be prefetched for a query is independent of the number of result pages requested so far: any time a query cannot be served from the cache, the QI should prepare the next r_{opt} result pages.

2.2 The Index Architecture and the Complexity of Processing Queries

The model to which we refer in most of this paper is that of a *local* index partitioning scheme in a *shared-nothing* network. The index is partitioned among m segments. We assume that documents (URLs) are partitioned into segments by a random process which assigns each document to a segment according to the uniform distribution, and independently of all other documents. Such a partitioning can be achieved by hashing every URL into a fixed-size document ID, and mapping these IDs into segments. Such a scheme was mentioned in [1] in the context of building URL repositories, and the same technique can be applied when assigning pages to the segments of an inverted index. Since the number of documents considered is in the hundreds of millions while m is considerably smaller (much less than the square root of the number of documents), the segments will contain roughly the same number of documents (with high probability). The query processing model is as described in Section 1.3. Throughout the discussion we consider the processing of a "broad topic" query that matches C documents in each segment, where C is much larger than the number of the results a user will actually browse.

Let A denote the number of results which the engine presents in each result page (a typical value is $A = 10$). Since results should be prefetched in *page units*, the number of prefetched results per query should be a multiple of A. In what follows we examine the cost of prefetching $n = rA$ results per query, so that in subsequent sections we will be able to optimize the value of r - the number of prefetched result pages. We will denote a user's query by a pair (t,k), where t is the search topic and $k \geq 1$ is the (ordinal) number of result page requested. A query can either start a search of a new topic (and then $k = 1$), or ask for additional results in an existing search ($k > 1$). The following discussion addresses both query types.

Preliminaries

Upon receiving a query (t,k) which cannot be answered from the cache, the QI needs to fetch n results for t. The first task is to set the value of l, the number of results to retrieve from each of the m segments.

Let $\mathcal{B}(n)$ denote the set of n documents that the engine should ideally retrieve for the query: the n documents that attain the best scores for t (according to the engine's ranking function), out of all documents that have not been retrieved for queries $(t,k'), k' < k$. Let $\mathcal{R}(l,m)$ denote the set of documents that will be *retrieved* for the query t when each of the m segments returns its l most relevant (and previously unretrieved) matches for t. Ideally, $\mathcal{R}(l,m)$ would contain $\mathcal{B}(n)$, but ensuring that means setting l to equal n. [5] Instead, we assume that the engine employs the following *quality policy*, based on a probability q: The QI sets the value of l with respect to n such that

$$Pr[\mathcal{B}(n) \subseteq \mathcal{R}(l,m)] \geq q$$

[5] This special case is discussed in Section 3.

In other words, the QI should collect enough (previously unretrieved) quality results from each segment so that with probability q, the top-n retrieved results will indeed be the best n (previously unretrieved) results for t in the entire index. The relationship between n and l will be studied in Section 5. For the time being, it suffices to note that by the assumption that documents are uniformly distributed among the segments, the above probability depends only on the values of n, m and l, and is independent of the topic t.

Let $\tilde{l}_q(n,m)$ denote the minimal number of documents which should be retrieved by each of the m segments so that the quality criterion is satisfied:

$$\tilde{l}_q(n,m) \stackrel{\triangle}{=} \min\{\ l \mid Pr[\mathcal{B}(n) \subseteq \mathcal{R}(l,m)]\ \geq\ q\}$$

Collecting results

The QI sends each segment the topic t and a request for its $\tilde{l}_q(n,m)$ top results for the query. Whenever $k > 1$ (this is not the first batch of results to be retrieved for t), segment i also receives $s_i(t,k-1)$, the score of the lowest ranking document that it had contributed to the results of $(t,1),\ldots,(t,k-1)$. [6] We now estimate the cost of serving such requests. By our assumption, the query matches C documents in each segment, where C is much larger than the number of results users will actually browse through, and consequently is much larger than $\tilde{l}_q(n,m)$ (since $\tilde{l}_q(n,m)$ is bounded by n, and n is bounded by the number of results that users browse through). We assume that identifying the C-sized set of candidate documents can be done in a time that is linear in C. This assumption holds for the inverted index structure when the number of query terms is very small, as is the case with broad topic queries on the Web (see Section 1.1). Recall that each segment receives the score of the lowest ranking document that it has retrieved so far for the query, and can thus discard previously retrieved results from the set of candidates. The top-scoring $\tilde{l}_q(n,m)$ documents of the remaining candidates are then found. Each segment will thus spend $\Theta(C + \tilde{l}_q(n,m)\log C)$ processing steps (per query) in order to return $\tilde{l}_q(n,m)$ sorted results to the QI.

Merging results

The QI receives m sorted result sets of length $\tilde{l}_q(n,m)$. Reading and buffering these sets takes $\Theta(m\tilde{l}_q(n,m))$ operations. It then partially merges the results until it identifies the top $n = rA$ retrieved results that will populate the r result pages. By using *Tree Selection Sorting* [12] with the m sorted result lists hanging from the leaves of the tree, the merge can be accomplished in time $\Theta(2m + n\log m)$. The overall complexity of this step is thus $\Theta(m\tilde{l}_q(n,m) + 2m + n\log m)$.

[6] We assume that the results of the query $(t,k-1)$ are still cached when the query (t,k) arrives.

Caching results

The r result pages are cached, and the first of those pages is returned to the user. The m scores $s_1(t,k),\ldots,s_m(t,k)$ are also noted. The overall space complexity is thus $\Theta(rA+m)$.

The complexity of the query processing model

Our model requires two messages to be passed between the QI and each of the segments: the QI sends the query to each segment, and each segment returns $\tilde{l}_q(n,m)$ results to the QI. The total number of results received by the QI is $m\tilde{l}_q(n,m)$, and this amount of data impacts its time complexity. Had we allowed more rounds of communication, we could have managed by sending the QI only $m + (n-1)$ results, lowering the complexity of the merge step above to $\Theta(m + n + n\log m)$. We chose not do so since minimizing communication rounds between machines (even at the expense of sending larger messages) is likely to improve performance in distributed computations [6].

Note that the complexity of the retrieval model described above is indeed "memoryless" (see discussion in Section 2.1). The model implies the following computational loads on the various resources of the engine, when following a policy of prefetching r result pages per query:

- The QI performs $\Theta(m\tilde{l}_q(rA,m) + 2m + rA\log m)$ computation steps.
- Each index segment performs $\Theta(C + \tilde{l}_q(rA,m)\log C)$ computations.
- The cache space required is $\Theta(rA+m)$.

Additionally, we introduce two non-negative coefficients α and β which will allow us to assign different weights to the three resources which are consumed during query processing. Specifically, α will multiply the computations of the QI and β will multiply the cache space required [7]. Tuning the values of α and β can emphasize memory (cache) limitations, computational bottlenecks (the QI vs. the segments) and response time per query. More on this in Section 6.2.

We are now ready to formulate $W(r)$, the expected cost (or *work*) of a policy which prefetches r pages for geometric users with parameter p. Result pages $ir+1, ir+2, \ldots, (i+1)r$ will be termed as the i'th *batch* of result pages. For ease of notation, we introduce $l_q(r,m) \stackrel{\triangle}{=} \tilde{l}_q(rA,m)$.

[7] The computational loads were expressed using the $\Theta(\cdot)$ notation. For concreteness and simplicity, we will consider the given expressions as the *exact* complexities. This allows us to avoid tedious notations, and does not affect the ensuing analysis (and results) of the paper.

$$
\begin{aligned}
W(r) &= \text{Caching overhead} + \\
&\quad \sum_{i=1}^{\infty} [\, Pr(\text{preparing batch } i) \cdot \\
&\qquad (\text{batch preparation complexity}) \,] \\
&= \beta(Ar+m) + \\
&\quad \sum_{i=0}^{\infty} p^{ir} [C + l_q(r,m)\log C + \\
&\qquad \alpha(rA\log m + m l_q(r,m) + 2m)] \\
&= \beta(Ar+m) + \frac{C + l_q(r,m)\log C}{1-p^r} + \\
&\quad \frac{\alpha(rA\log m + m l_q(r,m) + 2m)}{1-p^r}
\end{aligned}
$$

Rearranging the terms, and ignoring the constant additive term βm (which does not depend on r and will not affect the optimization of $W(r)$), we get

$$
W(r) = \beta A r + \frac{(C+2\alpha m) + (\log C + \alpha m) l_q(r,m) + (\alpha A \log m) r}{1-p^r}
$$

To ease the notation, we define the following constants: $a = \beta A$, $b = (C + 2\alpha m)$, $c = (\log C + \alpha m)$ and $d = \alpha A \log m$. With this notation,

$$
W(r) = ar + \frac{b + c l_q(r,m) + dr}{1-p^r}
$$

C, the number of documents per segment which match a query, is typically a large number, while A and m are typically much smaller. Thus, when the proportionality constants α and β are both about 1, typical values of b are large (tens of thousands and beyond), while a, c, d are relatively small (typically less than 100).

Our mission: Given an m-way locally segmented index, geometric-p users and some quality criterion q, determine r_{opt}, an integral value of r which minimizes $W(r)$. In doing so, determine $l_q(r_{opt}, m)$. The QI will then prepare r_{opt} result pages whenever a query cannot be answered from the cache, asking each of the m segments to retrieve its top $l_q(r_{opt}, m)$ results (that score below a certain threshold) for the query being processed. We will strive to obtain exact or almost exact values of r_{opt} and $l_q(r_{opt}, m)$.

3 Simple Special Cases

In this section we show that the problem for a single segment ($m = 1$) and the problem for multiple segments with $q = 1$ behave similarly, and in both cases r_{opt} can be found in $\Theta(\log r_{opt})$ steps.

- When the index is stored in a single segment, we can ignore the terms in the complexity function $W(r)$ which deal with the merging of results from different segments (namely the terms involving α). In addition, $l_q(r,1) = rA$ regardless of q's value. Thus, $W(r)$ becomes:

$$
W(r) = ar + \frac{C + (A\log C) r}{1-p^r}
$$

  Note that when an index is partitioned globally (each segment holds posting lists for a distinct set of terms), single-term queries are effectively queries to a single segment as described above. Studies [19, 7] indicate that the percentage of single-term queries on the Web is quite large ($25\% - 30\%$).

- For the case where $q = 1$ we again have $l_1(r,m) = rA$, and the complexity function $W(r)$ takes the following form:

$$
W(r) = ar + \frac{b + (cA + d) r}{1-p^r}
$$

Both cases imply a complexity function of the form

$$
W(r) = ar + \frac{b' + d'r}{1-p^r} \quad , \quad b', d' > 0
$$

The derivative of $W(r)$ is negative at zero and increases for all $r > 0$. Therefore, $W(r)$ (for positive values of r) decreases at first until reaching its (unique) minimal value, and then increases. Relying on this behavior, an optimal integral value of r ($r_o pt$) can be found by applying the following procedure:

1. Find the minimal natural number n such that $W(2^n) < W(2^{n+1})$.
2. Find an optimal value of r, using binary search, in the range $2^{n-1}, \ldots, 2^{n+1}$.

Since n will not exceed $1 + \lceil \log r_{opt} \rceil$, the complexity of finding r_{opt} is $\Theta(\log r_{opt})$.

Symbol	Denotes
a	shorthand for βA
b	shorthand for $(C + 2\alpha m)$
c	shorthand for $(\log C + \alpha m)$
d	shorthand for $\alpha A \log m$
m	number of segments in index
p	probability of viewing result page k when viewing page $k-1$
q	quality criterion of QI
r	number of result pages to fetch
r_{opt}	optimal integral value of r
A	number of results per result page
C	number of relevant results per segment
$W(r)$	work required for fetching r result pages per query
$l_q(r,m)$	number of results to fetch from each segment so that the best rA results are collected with probability at least q; equals $\tilde{l}_q(rA,m)$
α	multiplies the computations of the QI in $W(r)$
β	multiplies the required caching space in $W(r)$

Table 1: summary of notations

4 Solution for an m-way Segmented Local Index

In this section we study the problem of setting the optimal value of r given the quality criterion q ($q < 1$), the engine's architecture parameters A, C and m, and the complexity parameters α and β. Subsection 4.1 presents an algorithm for determining the optimal value of r, which minimizes the retrieval complexity function $W(r)$. Subsection 4.2 presents an approximation algorithm, which finds a value of r for which $W(r)$ is approximately optimal.

4.1 Optimizing r in Indices With m Segments

First, recall the complexity function from Section 2.2:

$$W(r) = ar + \frac{b + cl_q(r,m) + dr}{1 - p^r}$$

Clearly, the behavior of $W(r)$ depends on the behavior of $l_q(r,m)$. While we will show how to precisely calculate $l_q(r,m)$ in Section 5, for the purpose of this subsection it suffices to note that if $r' \geq r$ then $l_q(r',m) \geq l_q(r,m)$.

In order to facilitate the search for r_{opt}, we now set forth to find, for every value of r, an upper bound on the set $\{\bar{r} \mid W(\bar{r}) \leq W(r)\}$[8].

Definition 1 *A function $g(r)$ will be called W-restrictive if for all $r' \geq g(r)$, $W(r') > W(r)$.*

For example, $g_1(r) \stackrel{\triangle}{=} \frac{W(r)}{a}$ is W-restrictive, since for all $r' \geq g_1(r)$, we have $W(r') > ar' \geq ag_1(r) = W(r)$. Consequently, r_{opt} is not larger than $g_1(1)$.

We will use W-restrictive functions to bound our search space for r_{opt}. For this we now seek a W-restrictive function that is better than g_1, providing tighter bounds on the size of the search space. The following Proposition is proved in the full version of this paper:

Proposition 1 *The function*

$$g(r) = r + \frac{p^r\,(b + cl_q(r,m) + dr)}{(1 - p^r)(a + d)}$$

is W-restrictive.

Note that the above function reflects all the architectural parameters of the search engine's index, and also the user's behavior (represented by p) and the desired quality criterion q.

Figure 1 displays Algorithm OP for setting the optimal value of r. All of the steps except the calculation of $l_q(r,m)$ in 2(a) are trivial; that calculation is the topic of Section 5. The correctness of the algorithm follows from the W-restrictiveness of g(r) (Proposition 1), since we do not need to iterate through values of r for which $W(r)$ is known to be higher than values we have already seen.

[8] Since $\lim_{r\to\infty} W(r) = \infty$, the set $\{\bar{r} \mid W(\bar{r}) \leq W(r)\}$ is finite for all r.

1. Initializations: $W_{min} \leftarrow W(1)$, $r_{opt} \leftarrow 1$, limit $\leftarrow \infty$, $r \leftarrow 2$.
2. While $r <$ limit:
   (a) Calculate $l = l_q(r,m)$, and use the value to set $W \leftarrow W(r)$, $g \leftarrow g(r)$.
   (b) If limit $> g$: limit $\leftarrow g$.
   (c) If $W < W_{min}$: $W_{min} \leftarrow W$, $r_{opt} \leftarrow r$.
   (d) $r \leftarrow r + 1$
3. print r_{opt}.

Figure 1: Algorithm OP for optimizing the prefetch policy

The complexity of the algorithm

Algorithm OP needs to be executed relatively few times, when configuring the prefetching policy of the search engine (see discussion in Section 6.2). Therefore, its own complexity does not impact the performance of the engine. Nevertheless, we now prove that its running time is polynomial. We do so by bounding r_{max}, the maximal number of iterations which OP may require throughout its course. For this, let

$$\tau = \frac{a + d}{b + cA + d}$$

Note that by our assumptions on the relative values of a, b, c and d (see Section 2.2), τ is a small constant. Since $l_q(1,m) \leq A$, we have that $g(1) \leq 1 + \frac{p}{(1-p)\tau}$ is a bound on the number of iterations. Thus, r_{max} is bounded by $1 + \frac{2p}{1-p}$ whenever $\tau \geq 0.5$, and by $1 + \frac{1}{1-p}$ whenever $p \leq \tau$. Next, we bound r_{max} when $p > \tau$ and $\tau < 0.5$.

Lemma 1 $r_{max} \leq 3\left\lceil \frac{\log\tau}{\log p} \right\rceil$ *whenever* $p > \tau$, $\tau < \frac{1}{2}$.

Proof: Let $r = \left\lceil \frac{\log\tau}{\log p} \right\rceil$. Then, since $1 > p > \tau$, $r > 1$ and $p^r = 2^{r\log p} \leq 2^{\log\tau} = \tau$. Thus,

$$\begin{aligned} g(r) &= r + \frac{p^r}{(1-p^r)} \frac{b + cl_q(r,m) + dr}{a + d} \\ &\leq r + \frac{\tau}{(1-\tau)} \frac{b + cl_q(r,m) + dr}{a + d} \\ &\leq r + \frac{\tau}{(1-\tau)} \frac{b + crA + dr}{a + d} \\ &\quad \text{(since obviously } l_q(r,m) \leq rA\text{)} \\ &< r + \frac{\tau}{(1-\tau)} \frac{r}{\tau} = \frac{2-\tau}{1-\tau} r \leq 3r = 3\left\lceil \frac{\log\tau}{\log p} \right\rceil \end{aligned}$$

□

To complete the analysis of the complexity of algorithm OP for finding r_{opt}, we show in Section 5

that calculating the values of $l_q(r,m)$ for all $r \in \{1,\ldots,r_{max}\}$ requires $\mathcal{O}(m^2A^2r_{max}^2)$ steps (regardless of the value of q). Since we have already bounded r_{max} by simple functions of m, p and τ, bounds on the complexity of the algorithm follow.

Table 2 brings sample results of the algorithm. For every combination of m and p, r_{opt} and r_{max}^{act} (the highest value of r for which $W(r)$ was actually calculated during execution) are shown. Figure 2 plots $W(r)$ as a function of r, as calculated during the algorithm for three values of m with $p = 0.5$. For all displayed results, we used $q = 0.99, \alpha = \beta = 1, A = 10$ and $C = 2^{13}$.

$m \backslash p$	0.3	0.5	0.7
5	4(5)	7(9)	11(15)
25	4(5)	6(8)	12(14)
50	4(5)	6(8)	10(13)

Table 2: $r_{opt}(r_{max}^{act})$ values as a function of m and p

4.2 Approximating the Optimal Solution

In the previous subsection we have shown how to determine r_{opt}, the number of pages which minimizes the complexity function $W(r)$. However, if we are willing to settle for nearly optimal solutions, namely finding values of r for which $\frac{W(r_{opt})}{W(r)} \geq 1-\epsilon$ for small values of ϵ, we can use the following algorithm:

1. Let $r_{max} \stackrel{\triangle}{=} \left\lceil \frac{\log \epsilon}{\log p} \right\rceil$.
2. Find the value of r in the range $\{1,\ldots,r_{max}\}$ which minimizes $W(r)$.

Note that r_{max} depends on the user's behavior (as modeled by p) but is independent of the engine's architecture and quality policy (which are modeled by a, b, c, d and q). Furthermore, the above algorithm is applicable to *any* work function $\tilde{W}(r)$ such that $(1-p^r)\tilde{W}(r)$ is an increasing function of r. Note that $W(r)$ satisfies this condition, since

$$(1-p^r)W(r) = (1-p^r)ar + b + cl_q(r,m) + dr$$

where a, b, c, d are positive constants, and the functions $(1-p^r), l_q(r,m)$ are nondecreasing functions of r.

The correctness of the approximation algorithm relies on the following Proposition.

Proposition 2 *Let $W(r)$ be any positive function such that $(1-p^r)W(r)$ is an increasing function of r. Let $r,t \in I\!N$ such that $\frac{W(t)}{W(r)} \geq \frac{1}{1-p^r}$. Then, for all $r' \geq t$, $W(r') > W(r)$.*

Proof: Since $(1-p^t)W(t) \geq W(r)$, we have for $r' \geq t$

$$W(r') > (1-p^{r'})W(r') \geq (1-p^t)W(t) \geq W(r)$$

□

Corollary 1 *Let $0 < s < t$. Then $W(s) < \frac{W(t)}{(1-p^s)}$.*

Proof: Since $(1-p^r)W(r)$ increases with r, we get

$$W(s) < \frac{W(s)}{(1-p^t)} < \frac{W(t)}{(1-p^s)}$$

□

Corollary 2 *For all s,*

$$\min\{W(1),\ldots,W(s)\} < \frac{W(r_{opt})}{1-p^s}$$

Proof: If $1 \leq r_{opt} \leq s$, the claim holds. Otherwise, the result is implied by Corollary 1, with $t = r_{opt}$. □

Substituting $s = r_{max} = \lceil \frac{\log \epsilon}{\log p} \rceil$ in the last Corollary yields the approximation algorithm:

$$\begin{aligned} \min\{W(1),\ldots,W(s)\} &< \frac{W(r_{opt})}{1-p^{\lceil \frac{\log \epsilon}{\log p} \rceil}} \\ &= \frac{W(r_{opt})}{1-2^{\log p \lceil \frac{\log \epsilon}{\log p} \rceil}} < \frac{W(r_{opt})}{1-\epsilon} \end{aligned}$$

Table 3 shows the values of r_{max} for $p = 0.1, 0.2, \ldots, 0.9$ and $\epsilon = 0.1, 0.01$ and 0.001. As mentioned earlier, calculating the values of $l_q(r,m)$ for all $r \in \{1,\ldots,r_{max}\}$ requires $\mathcal{O}(m^2A^2r_{max}^2)$ computational steps, and thus the time complexity of the approximation algorithm is $\mathcal{O}(m^2A^2\left\lceil \frac{\log \epsilon}{\log p} \right\rceil^2)$.

Finally, we note that the results of this subsection may be used in practice to improve the running time of Algorithm OP (figure 1), by checking (between steps 2(b) and 2(c)) whether $\frac{W}{W_{min}} \geq \frac{1}{1-p^r}$, and setting limit $\leftarrow r$ if so (thus terminating the algorithm). Proposition 2 asserts that all future iterations with larger values of r will result in greater values of $W(r)$, and so OP can safely terminate and output the current value of r_{opt}.

5 Calculating $l_q(r,m)$

This section brings recursive formulae with which $l_q(r,m)$ can be calculated in a time which is polynomial in m, r and A.

We model the distribution of the top results in the segments by the following random process: $n \stackrel{\triangle}{=} rA$ different balls (the top results for a query) are thrown randomly and independently into m different cells (the segments), where n_i balls are inserted to cell i ($\sum_{i=1}^{m} n_i = n$). We model the querying process by taking $\min\{l, n_i\}$ balls from cell i for $i = 1,\ldots,m$. Denote by $e_{n,m,l}$ the number of *excess* balls that remain in the cells after the querying process is completed. In Section 5.1 we calculate the probability that $e_{n,m,l} = 0$. This corresponds to the case where no cell

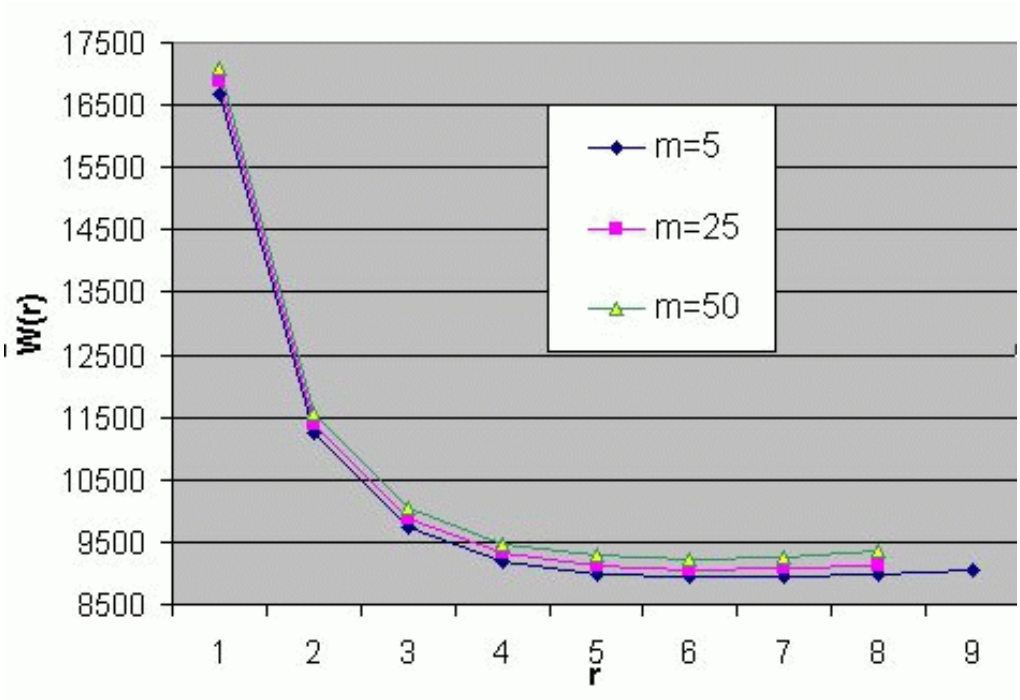

Figure 2: $W(r)$ as a function of r, for $m = 5, 25$ and 50 ($p = 0.5$)

$\epsilon \backslash p$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.1	1	2	2	3	4	5	7	11	22
0.01	2	3	4	6	7	10	13	21	44
0.001	3	5	6	8	10	14	20	31	66

Table 3: r_{max} as a function of p and ϵ

contains more than l balls, so that the QI indeed managed to collect the top n results from the segments. In the full version of this work we also calculate the probability that $e_{n,m,l} = k$. This corresponds to the case where the QI managed to collect just $n - k$ of the top n results. We study this case since the QI may choose to employ a relaxed quality policy, requiring that with high probability, *most* (but not necessarily all) of the top results are returned to the user. Subsection 5.2 briefly reviews previous work on related issues.

We first present a rough bound on $l_q(r, m)$ which may suffice when precise calculations are not essential. Clearly, $\frac{n}{m}$ is a lower bound on $l_q(r, m)$ for all $q > 0$. We show that for $q = 1 - \frac{1}{2^\lambda}$, $l_q(r, m)$ need not be larger than $\max\{\lceil \lambda + \log m \rceil, \lceil \frac{2en}{m} \rceil\}$ [9]:
The probability that *exactly* i of the n results are inserted to a given segment is $\binom{n}{i} \left(\frac{1}{m}\right)^i \left(1 - \frac{1}{m}\right)^{m-i}$. Since $\binom{n}{i} \leq \left(\frac{ne}{i}\right)^i$,

$$\binom{n}{i}(\frac{1}{m})^i(1-\frac{1}{m})^{n-i} < (\frac{ne}{i})^i(\frac{1}{m})^i = (\frac{ne}{mi})^i \ .$$

[9] Sharper (known) asymptotic bounds on $l_q(r, m)$ are discussed in Section 5.2.

Hence, the probability that more than ℓ results are inserted into a given segment is bounded by

$$\sum_{i=\ell+1}^{n} (\frac{ne}{mi})^i \ < \ \sum_{i=\ell+1}^{\infty} (\frac{ne}{m\ell})^i \ = \ (\frac{ne}{m\ell})^\ell \frac{\frac{ne}{m\ell}}{1-\frac{ne}{m\ell}}$$

Whenever $\ell \geq \max\{(\lambda + \log m, \frac{2en}{m}\}$, this last expression is bounded from above by $\left(\frac{1}{2}\right)^{\lambda + \log m} = \frac{1}{m2^\lambda}$. Thus, by the union bound, the probability that at least one of the m segments contains more than ℓ results is smaller than $\frac{1}{2^\lambda}$. The results follow.

5.1 Precise Calculation of $l_q(r, m)$

We now turn to the precise calculation of $l_q(r, m)$. For this we will calculate the probability

$$P(n, m, l) = Pr[e_{n,m,l} = 0] \ ,$$

the probability of throwing n different balls into m different cells so that no cell contains more than l balls.

The size of the problem space is m^n. We will actually be counting $N(n, m, l)$, the number of ways to throw n different balls into m different cells so that no

$m\backslash r$	1	2	3	4	5	6	7	8	9	10	11	12
5	6	10	13	16	19	22	24	27	30	32	35	37
25	4	5	6	7	8	9	10	10	11	12	13	13
50	3	4	5	5	6	6	7	8	8	8	9	9

Table 4: $l_q(r,m)$ for $q = 0.99, A = 10$ and various values of r and m

cell contains more than l balls, and then

$$P(n,m,l) = N(n,m,l)/m^n$$

The following recursive formulae may be used to calculate the $N(n,m,l)$ values:

$$N(n+1,m,l) = m \cdot \sum_{j=0}^{l-1} \binom{n}{j} N(n-j,m-1,l)$$

$$N(n,m+1,l) = \sum_{j=0}^{l} \binom{n}{j} N(n-j,m,l)$$

However, the recursion that most naturally fits in Algorithm OP from Section 4.1 is:
$N(n,m,l) =$

$$\sum_{j=0}^{m} \binom{m}{j} \binom{n}{l,\ldots,l,n-jl} N(n-jl,m-j,l-1)$$

First, we choose some j cells to have exactly l balls. We then choose the balls to populate those cells (the multinomial coefficient has j l-terms). The remaining $n-jl$ balls are distributed to the remaining $m-j$ cells, with each such cell collecting no more than $l-1$ balls.

As r grows in subsequent iterations OP, so will the value of $l_q(r,m)$. This recursion naturally uses results of $N(n,m,l)$ from previous iterations in later iterations. As for the initial values:

1. For all m, l, $N(0,m,l) = 1$.
   Whenever $n > 0$, $N(n,0,l) = N(n,m,0) = 0$.
2. For all $n > 0, m > 0$:
   - Whenever $l < \lceil \frac{n}{m} \rceil$, $N(n,m,l) = 0$.
   - $N(n,m,\lceil \frac{n}{m} \rceil) = \binom{m}{k} \frac{n!}{\lceil \frac{n}{m} \rceil^k \lfloor \frac{n}{m} \rfloor^{m-k}}$,
     where $k \triangleq n \bmod m$.

Denoting by $n_{max} \triangleq r_{max} A$ the value of n in the last iteration of OP (and by l_{max} the value of l found in that iteration), the total time spent calculating values of $l_q(r,m)$ is $\theta(n_{max} m^2 l_{max}) = \mathcal{O}(n_{max}^2 m^2)$. Table 4 shows sample values of $l_q(r,m)$.

5.2 Previous Work

The stochastic properties of the process which randomly throws n balls into m cells have been studied extensively. Two good references are [13] and [10]. Among the properties studied was the distribution of the maximum number of balls in a cell, which we will denote by $L(n,m)$. For example, for $n \geq m$ (more balls than cells), $L(n,m) = \Theta(\frac{\ln m}{\ln \lceil 1 + \frac{m}{n} \ln m \rceil} + \frac{n}{m})$ with probability $1 - o(1)$ [5]. When $n = m$, $L(n,m)$ behaves asymptotically as $(1+o(1))\frac{\ln n}{\ln \ln n}$ with probability $1 - o(1)$ [2]. In [13], the distribution of $L(n,m)$ is examined with regard to the behavior of the ratio $\frac{n}{m \ln m}$ as $n, m \to \infty$. Separate results are obtained for the three cases $\frac{n}{m \ln m} \to 0$, $\frac{n}{m \ln m} \to \lambda > 0$, and $\frac{n}{m \ln m} \to \infty$. In [9] it was shown that the distribution of $L(n,m)$ may be approximated by the the distribution of

$$n \cdot \max \frac{s_j}{\sum_{j=1}^{m} s_j} ,$$

where each s_j is an independent χ^2 variable with $\frac{2(n-1)}{m}$ degrees of freedom.

6 From Theory to Practice

This section attempts to bridge the gap between theory and practice by highlighting the possible practical implications of our model and results.

6.1 The Complexity Function $W(r)$

We first revisit two assumptions we have made while formalizing $W(r)$ (Section 2). These assumptions pertain to the manner by which users view result pages and to the memoryless query processing scheme.

1. "Users view search result pages according to a memoryless geometric process". While this assumption is extremely simplistic, the studies cited in Section 2.1 indicate that it might reasonably approximate the aggregate behavior of users.
2. "When a request for result page k arrives, result page $k-1$ is still cached". We used this assumption to send each segment the score of the lowest result it had contributed to page $k-1$. This, in turn, allowed us to formulate a memoryless query processing scheme. While ignoring cache management issues in this work, the following consideration justifies the intuition behind this assumption: the aim of *any* policy that prefetches r pages (numbered $k, \ldots, k+r-1$) when processing a request for result page k of some query, is to rapidly answer (from the cache) subsequent requests for pages $k+1, \ldots, k+r-1$ of that query. Thus, the prefetching policy implicitly assumes that the

life expectancy of cached entries will allow page $k + r - 1$ to be cached until it is requested. In other words, *every* policy that prefetches r pages assumes that pages will be cached long enough for $r-1$ subsequent requests. We require pages to be cached for r subsequent requests.

The above assumptions allowed us to formulate an exact complexity function to our concrete query processing model. At the end of Section 2.2, the complexity function was abbreviated to the form

$$W(r) = ar + \frac{b + cl_q(r,m) + dr}{1 - p^r} \ .$$

We claim that this abbreviated form (and our results) can accommodate any retrieval model that incurs the following costs when prefetching r pages:

- Cache space that is linear in r, the number of prefetched result pages.
- Retrieval complexity that is the sum of (1) a term that depends on the query's breadth (number of matching results), (2) a term that is linear in $l_q(r,m)$, and (3) a term that is linear in r.

Thus, our results may apply to index structures and query processing schemes that differ from our model. Furthermore, the results of Section 4.2 apply to *any* complexity function $\tilde{W}(r)$ where $(1-p^r)\tilde{W}(r)$ is an increasing function of r. Finally, the results of Section 5, where we determined the number of results that should be retrieved from each segment ($l_q(r,m)$), are applicable to any search engine that uses a locally segmented index in which documents are partitioned uniformly and independently.

6.2 Implementing a Prefetching Policy

Implementing a prefetching policy for engines with locally segmented indices in the framework of this research requires the following two preprocessing steps:

- Setting the parameters: an approximate value of p is derived from analyzing the engine's query logs, the parameter q is set according to the quality policy, and the values of α, β are set according to the engine's resources. Systems with small caches should set β to a high value; when the QI is heavily loaded, α should be set to a high value; etc.
- For a range of query breadths (a range of values for the parameter C), an algorithm (either optimizing or approximating r_{opt}) is executed. The QI and each segment are then loaded with tables containing the values of $r_{opt}(C)$ and $l_q(r_{opt}(C), m)$ for values of C in the range.

Upon receiving a query t, each segment estimates that query's breadth (the value of C_t that corresponds to t). This can be done in two ways:

- Many local index implementations incorporate *global* term statistics in each segment in order to facilitate term-based scoring [1]. These statistics may help estimating the breadth of certain types of queries.
- By our assumption, each of the m segment contain approximately the same number of results for broad topic queries (when $C \gg m$). Thus, a segment can process the query, and use the number of matches it finds as an estimate of C.

After estimating C_t, the segment forwards $l_q(r_{opt}(C_t))$ results to the QI, which merges the retrieved results to produce $r_{opt}(C_t)$ result pages.

7 Conclusions and Future Work

This work examined how search engines should prefetch search results for user queries. We started by presenting a concrete query processing model for search engines with locally segmented inverted indices. We argued that for a model which assumes that the number of result pages that users view is distributed geometrically, the optimal engine policy is to prefetch a constant number of result pages r. We expressed the computational cost of a policy that prefetches r pages, and suggested an algorithm for finding the optimal value of r (which minimizes the expected cost). We also suggested how to find values of r which imply policies whose cost is approximately optimal.

Several extensions of this work are the following:

- The model presented in this paper ignores overlaps in the information needs of different users. We did not consider, for example, that popular queries may be submitted by multiple users during a short time span, increasing the probability of at least one user requesting additional results. By taking query popularity into account, we may find that popular queries warrant more result prefetching than rare queries do.
- This work did not address cache replacement policies; in particular, we did not suggest which result pages should be removed from the cache upon prefetching results for a new query. As noted in [11] (in the context of buffering of posting lists), knowledge of the access patterns to the query cache should be considered when setting the replacement policy. For example, users usually browse result pages in their natural order. Thus, assuming that the first two result pages of some query are cached and that one of them must be evicted, it seems natural to remove the second page of results (and not the first, as an LRU policy might suggest).
- Most of the results in this paper are applicable to locally segmented indices. Only single-term

queries to global indices are considered. Additional research is required in order to extend our results to multi-term queries to global indices.

Acknowledgments

We thank Andrei Broder [10] and Farzin Maghoul from AltaVista for useful discussions and insights on the problems covered in this paper.

References

[1] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the web. *ACM Transactions on Internet Technology*, 1(1):2–43, 2001.

[2] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM Journal of Computing*, 29(1):180–200, 1999.

[3] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Proc. 7th International WWW Conference*, 1998.

[4] Brendon Cahoon, Kathryn S. McKinley, and Zhihong Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Transactions on Information Systems*, 18(1):1–43, 2000.

[5] Artur Czumaj and Volker Stemann. Randomized allocations processes. *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages 194–203, 1997.

[6] David Hawking. Scalable text retrieval for large digital libraries. *First European Conference on Digital Libraries*, 1997.

[7] Bernard J. Jansen, Amanda Spink, and Tefko Saracevic. Real life, real users, and real needs: A study and analysis of user queries on the web. *Information Processing and Management*, 36(2):207–227, 2000.

[8] Byeong-Soo Jeong and Edward Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, 1995.

[9] N. L. Johnson and D. H. Young. Some applications of two approximations to the multinomial distribution. *Biometrika*, 47:463–469, 1960.

[10] Norman L. Johnson and Samuel I. Kotz. *Urn Models and their Application*. John Wiley & Sons, Inc., 1977.

[11] Björn THór Jónsson, Michael J. Franklin, and Divesh Srivastava. Interaction of query evaluation and buffer management for information retrieval. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA*, pages 118–129, June 1998.

[12] Donald E. Knuth. *The Art of Computer Programming, Volume 3*. Addison-Wesley Publishing Company Inc., 1998.

[13] Valentin F. Kolchin, Boris A. Sevast'yanov, and Vladimir P. Chistyakov. *Random Allocations*. V. H. Winston & Sons, 1978.

[14] Steve Lawrence and C. Lee Giles. Searching the world wide web. *Science*, 280, April 1998.

[15] Evangelos P. Markatos. On caching search engine query results. *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, May 2000.

[16] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. *Proc. 10th International WWW Conference*, 2001.

[17] B. Ribeiro-Neto and R. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proc. ACM Digital Libraries Conference, 1998.*, pages 182–190, 1998.

[18] B. A. Ribeiro-Neto, J. P. Kitajima, G. Navarro, C. R. G. Sant'Ana, and N. Ziviani. Parallel generation of inverted files for distributed text collections. *Proc. 18th International Conference of the Chilean Computer Science Society*, 1998.

[19] Craig Silverstein, Monika Henzinger, Hannes Marais, and Michael Moricz. Analysis of a very large altavista query log. Technical Report 1998-014, Compaq Systems Research Center, October 1998.

[20] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proc. Second International Conference on Parallel and Distributed Information Systems*, pages 8–17, 1993.

[10] Andrei Broder is currently with IBM Research