Batch Query Processing for Web Search Engines

Shuai Ding*
Polytechnic Institute of NYU
Brooklyn, NY, USA
sding@cis.poly.edu

Josh Attenberg
Polytechnic Institute of NYU
Brooklyn, NY, USA
josh@cis.poly.edu

Ricardo Baeza-Yates
Yahoo Research
Barcelona, Spain
rby@yahoo-inc.com

Torsten Suel
Polytechnic Institute of NYU
Brooklyn, NY, USA
suel@poly.edu

ABSTRACT

Large web search engines are now processing billions of queries per day. Most of these queries are interactive in nature, requiring a response in fractions of a second. However, there are also a number of important scenarios where large batches of queries are submitted for various web mining and system optimization tasks that do not require an immediate response. Given the significant cost of executing search queries over billions of web pages, it is a natural question to ask if such batches of queries can be more efficiently executed than interactive queries.

In this paper, we motivate and discuss the problem of batch query processing in search engines, identify basic mechanisms for improving the performance of such queries, and provide a preliminary experimental evaluation of the proposed techniques. Our conclusion is that significant cost reductions are possible by using specialized mechanisms for executing batch queries in Web search engines.

Categories and Subject Descriptors

H.3.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Search and Retrieval

General Terms

Algorithms, Experimentation

Keywords

Web Search, Query Processing, Batch Query Processing, Result Cache Updates

*Part of this work was done while this author was visiting Yahoo! Research Barcelona, under the Yahoo! internship program

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WSDM'11, February 9–12, 2011, Hong Kong, China.
Copyright 2011 ACM 978-1-4503-0493-1/11/02 ...$10.00.

1. INTRODUCTION

Over the last decade, web search technology has developed into a multi-billion dollar industry that serves answers to hundreds of millions of users each day. Users, in turn, have come to rely on search engines for an ever-increasing share of their information and other needs, in the process supplanting many other printed, electronic, and human information sources. The current search engines index tens of billions of web pages and petabytes of data and aim to provide a single interface into many different types of information including web sites, books, news, videos, product information, technical documents and images.

While search engines generate significant income, primarily through advertising, they also incur large costs in terms of hardware, power, and engineering. The current large-scale search engines, including Google, Yahoo!, Microsoft Bing, and Baidu, are based on tens to hundreds of thousands of computer servers, representing multi-billion dollar investments, and requiring many millions more in maintenance effort and power consumption. In addition, a competitive search engine must employ teams comprising hundreds of researchers, engineers and developers just to keep up with the ever-increasing data size, user load, and user expectations. These teams perform significant analysis and mining of the underlying data and modeling of users behavior in order to increase the quality of the search results, improve the targeting of the accompanying advertising, and optimize the efficiency of the system.

Beyond the search engine companies themselves, there currently exists an entire ecosystem of companies, including new media, e-commerce sites, advertising networks, and social networking sites that depend on or closely interact with the search engines. These companies also perform significant data mining and analysis to optimize their business. While some of these companies can afford to acquire significant amounts of data from the web, others use data and mechanisms provided by the search engines, which due to increasing cost and consolidation are more and more becoming gatekeepers to much of the data on the web [4, 29].

Current search engine query processing systems are designed with the goal of returning results in fractions of a second. This is clearly very important since most of the queries processed by the search engines are interactive queries—those that are posed by human users. However, there are also a large numbers of queries that do not require inter-

active results; this includes queries issued from within the search companies in order to test, mine, and optimize the search engines, as well as queries issued by outside parties in order to mine information from the web. Our assumption here, discussed further below, is that such queries will increase significantly in volume and that it makes sense to consider optimized query processing mechanisms for such queries in order to achieve higher efficiency than the systems designed for interactive queries.

Thus, the topic of this paper is how to efficiently execute batches of queries that do not require interactive answers. We discuss some basic approaches that allow more efficient processing of such queries, describe how to optimize these approaches, and then provide an experimental evaluation that shows significant performance gains compared to the processing of interactive queries.

1.1 Motivation

Above, we have asserted that there are significant numbers of queries that do not require interactive responses. We now try to discuss and justify this claim in more detail. In particular, we describe three important scenarios where such queries arise:

(1) **Cache Updates:** We start with a very specific, but important, example of non-interactive queries. All major search engines cache the results of common search queries, and a significant amount of work has focused on optimizing caching mechanisms [5, 25, 21, 30, 32, 16]. Recently, work has focused on the need to also update and refresh results stored in these caches [8] since in practice performance is limited more by updates to the document collection than by the size of the cache. In particular, it makes sense to periodically recompute and store results to common queries, by issuing batches of queries whose results would soon expire to the search engine. This has benefits over caching methods based on simple invalidation in that batches can be scheduled to avoid peak loads and in that user queries do not experience delays due to cache misses. In addition, as we will demonstrate, queries can be processed more efficiently, that is, with greater throughput, in batches.

(2) **Internal Testing and Mining:** As described above, search engines employ multiple teams that perform various mining and testing tasks with the intent of improving the search engine. Many of these tasks are performed on separate clusters running, e.g., mapReduce [13], Hadoop[1], Dryad [18], or similar systems. However, there are also many tasks that involve issuing batches of queries to the search engine. For example, search engines perform significant amounts of internal quality testing that involves issuing large numbers of queries. Common query processing optimizations, such as index tiering and early termination [26, 23] and indexing of subqueries [24, 9], use queries to determine the best index organization. Data mining operations may issue queries for various tasks such as tuning of ad matching mechanisms and of crawl policies, or filtering and classification of content. While there is not a lot of published work on the internal day-to-day operations and engineering of search engines, in our own experience it is quite common to issue significant batches of queries for mining and testing purposes. (Additional automatic queries with interactive response requirements may be triggered by various browsing and online ad matching operations, but these are not the topic of this paper.)

(3) **Queries by Outside Parties:** As mentioned, there are now many outside entities who are closely connected with major web search engines. Amongst these parties are researchers, companies requiring keyword-focused random access to web content, and third-party search engines, where results are initially gleaned from query interfaces of larger search engines. This programmatic access of web scale inverted indices has become so important, that all of the major search engines have APIs for automated access. For instance, Yahoo! provides a service called BOSS (Build your Own Search Service) that third parties (almost) unlimited automated access to the Yahoo! search service. In May 2009, Yahoo! reported that about 30 million queries per day were issued through BOSS,[2] and a number of companies including Hakia (semantic search), OneRiot (social search), and Linguaseek (cross-language search) rely on BOSS. Some of these applications require interactive responses, but much of this traffic does not need to be processed in real time. Additionally, the capabilities of a web search engine can be leveraged as a tremendous resource to aid data acquisition in the context of building, maintaining, and evaluating models for data mining and machine learning [4, 17, 29]. Here, by issuing specific queries, including those derived through data-driven processes and those generated by expert background knowledge, to a web search engine, a far greater repository of information becomes accessible than may otherwise be possible.

In summary, we have argued that there are many scenarios that involve issuing large batches of queries to search engines. As arbiters of the web's data, we expect third party demands on increase significantly in the coming years as data on the web balloons in size and as innovative uses for large data are devised. Combined with the ongoing consolidation in the core search ecosystem, driven in part by the significant cost of engineering and maintaining a petabyte general-purpose search engine implies that there will an ever increasing demand by parties interested in issuing queries over the web, but fewer and fewer companies able to afford to acquire their own data and platform for executing such queries.

We believe that there are two main obstacles that currently limit use of such queries. One obstacle, which is *not* the focus of this paper, is that IR systems are designed with humans rather than machines in mind. Thus, while relational databases are heavily standardized in their operations and designed for the explicit purpose of supporting multiple layers of software on top, search engines are focused on providing quality results to human users, and it can be difficult to built additional tools on top of this foundation.

The other obstacle is limited access to queries. While some limitations are due to concerns by search engines that

[1]http://hadoop.apache.org

[2]These queries are not counted in the reported traffic and market share figures for search engines.

outside parties could reverse engineer and then subvert their ranking systems, another major factor is the cost of providing large numbers of queries to outside parties. This is the problem we hope to address here, by making batches of queries cheaper than normal, interactive queries. Of course, once more companies start using such queries, they may also over time learn how to overcome the first obstacle, the difficulty of building on top of search engines.

1.2 Optimizing for Batches of Queries

Having discussed possible applications of batched search engine queries, we now have to argue that there is a reason to believe that large batches of queries can be processed more efficiently than interactive queries, thus justifying special optimized query processing mechanisms for batched queries.

First, a trivial way to make batched queries cheaper to schedule them during off-peak times. But beyond this simple approach, we are looking here at ways to make batched queries fundamentally cheaper to execute, that is, faster or by using fewer hardware resources. We identify and evaluate three major sources of savings and approaches in this paper:

(1) **Query Reordering:** Given a batch of queries, we can execute the queries in any order, resulting, e.g., in significantly better caching behavior.

(2) **Clairvoyance:** Since we know all the queries in the batch, we can use clairvoyant algorithms for caching lists and partial results. It is known that for list caching, there is still a significant gap between the best online solution and the clairvoyant algorithm [16].

(3) **Reusing Partial Results:** Both clairvoyance and reordering can then be exploited to get better reuse of partial results. Reuse of partial results corresponding to subqueries was evaluated, e.g., in [24, 9, 20] but it turns out that it is quite difficult to ascertain which subqueries will reappear often enough to justify storing the results. Reordering and clairvoyance, as we will show, can very significantly improve this mechanism over the online case.

Before proceeding to the contributions of this work, we note some limitations of our research. One orthogonal idea that we do not pursue in this paper is to interleave the batch queries with the interactive query stream to essentially wait for and piggy-back onto identical or similar queries in the query stream. This is an interesting idea for future work, but requires a somewhat different experimental setup and evaluation measures.

Finally, we note some assumptions and limitations of this work. Most of our approaches assume that query results are based on an intersection of the query terms. Current search engines tend to use intersections whenever suitable, but also use slightly more general Boolean filters for many types of queries. Furthermore, current search engines use very complicated ranking models based on hundreds of predictive covariances, and often execute queries in several phases, first a simple first-cut ranking function based on a traversal of significant parts of the relevant index structures, followed by the computation of a more elaborate ranking function on a small subset of promising candidate documents. In this paper, we focus on the first phase. We note that some applications of batch queries, say those testing the quality of search results, require second-phase results, and our techniques would only give savings for the cost of the first phase. In other applications, first-phase results might in fact be sufficient to achieve the given mining task[3]. Finding savings in the second phase is a problem for future research.

The remainder of the paper is organized as follows: Section 2 gives background and related work, Section 3 and 4 describe the technical details, Section 5 gives experimental results and finally Section 6 concludes.

2. BACKGROUND AND RELATED WORK

Web search engines are facing formidable performance challenges due to data sizes and query loads. The major engines have to process tens of thousands of queries per second over tens of billions of documents. To solve this performance problem major search engines employ large clusters of servers. However, given suitable mechanisms for load balancing, the problem of optimizing overall throughput can be reduced to the single-node case, i.e., how to maximize the number of queries per second that can be processed on each machine within a reasonable response time.

One important technique for optimizing performance in search engines is caching. Caching takes advantage of a computer system's hierarchical architecture and enables fast access to recently used data. It has been studied extensively in search engines on three different levels: *Result caching* [25, 22, 21, 30, 34], which deals with the case where identical queries are issued repeatedly by keeping a cache of recently returned results, *list caching* [30], which keeps inverted lists corresponding to frequently used terms in memory, and *intersection caching* [24, 9] where results of frequent subqueries are cached. Baeza-Yates *et al.* studied the trade-off between result caching and list caching [5], Skobeltsyn *et al.* combined pruned indexes and result caching in [32], and Kumar *et al.* considered top-k aggregation algorithms in the context of intersection caching [20]. Recently, Barla Cambazoglu *et al.* studied techniques for refreshing the content of a result cache [8], which in fact is one important motivation of this paper.

In our batch processing problem, we are given a large stream of distinct queries, as any duplicate queries can be trivially resolved using caching. Thus, only list caching and intersection caching are potentially useful in this context. However, the list caching problem in batch query processing is quite different from the traditional list caching problem because we can shuffle the order of the queries to take advantage of locality and improve caching performance, and we can use a clairvoyant cache eviction policy [7] since we know all future queries. While there is no previous work on the problem of reordering the query stream to improve cache performance in search engines, there is work in the context of web server and proxy caching, where the so-called *r-reordering problem* [15, 2] was shown to be NP-hard.

A powerful extension to traditional posting list caching is so-called *intersection caching* [24]. The basic idea of this setting is to cache commonly used subqueries (usually pairs of terms) in order to speed up query processing by alleviating the need for repeated computation. In [20], pruning techniques are combined with intersection caching to achieve further performance gains. Intersection caching offers increased

[3]e.g. tasks involving queries demanding *all* documents matching a key-phrase

potential benefits in addition to algorithmic challenges in the context of batch query processing; here we have the power to foresee future sub-queries and to alter their order. In a heavy-tailed query distribution such as a typical query log, intersection cache-hits may a high payoff, but have an extremely low rate of occurrence [9]. Further separating our work from the prior work on intersection caching, here we only consider in-memory caching, ignoring the possibility of precomputing intersections and storing them on disk for later retrieval.

Query processing in search engines has been extensively studied; for a basic overview, see [37, 6]. For recent research on performance optimization such as index compression and pruning, see [26, 36]. We assume we are given a collection of N documents, where each document is uniquely identified by a document ID (docID) between 0 and $N-1$. The collection is indexed by an inverted index structure, used by all major web search engines, which allows efficient retrieval of documents containing a particular set of words (or terms). An inverted index consists of many inverted lists, where each inverted list I_w contains the docIDs of all documents in the collection that contain the word w. Each inverted list I_w is typically sorted by document ID, and usually also contains for each docID the number of occurrences of w in that document and maybe the locations of these occurrences in the page or other extra information (title, font, etc). Inverted indexes are usually stored in highly compressed form on disk or in main memory, such that each list is laid out in a contiguous manner.

A query $q = \{t_0,\ t_1,\ t_2, \cdots\}$ is a set of terms (words). The most common way to rank the results is based on comparing the terms contained in the documents and the query. More precisely, documents are modeled as unordered bags of words and a ranking function assigns a score to each document with respect to the current query. There are many different ranking functions proposed, such as Okapi's BM25 [27] or the Cosine measure [33]. The techniques proposed in this paper are not limited to a particular class of ranking function, except that we do assume query processing based on intersections of terms. Many ranking functions studied by the traditional IR community do not require the result document to contain all the query terms. However, most search engines prefer conjunctive semantics for queries, and try to consider only documents containing all query terms whenever possible. This is due to many reasons including collection size, user expectations and preponderance of short queries. Given an inverted index, a query is executed by computing the scores of all documents in the intersection of the inverted lists for all the query terms. This is efficiently done in a *document-at-a-time* approach where we simultaneously scan each of the inverted lists and compute the score for any document that is encountered in all lists. It is shown in [19] that this is more efficient than a *term-at-a-time* approach (we process the inverted list one after another). So in this paper we use the *document-at-a-time* approach (Note that all the proposed techniques can be applied to the *term-at-a-time* approach). Another important optimization of query processing is pruning technique, which attempts to determine the top-K results without a complete scan of all the inverted lists, by presorting the lists according to their contribution to the score. There has been a lot of work on this issue over the last two decades, see, e.g., [3, 23, 28]. We note that our approach is based on a complete scan of the list intersections, and that integration with early termination techniques is left for future work.

Overall, our work makes the following contributions:

1. We introduce, motivate, and study batch query processing in search engines, which we argue will become increasingly important.
2. We study two important aspects of the batch query processing problem; one assumes I/O is the main bottleneck for the system, and the other assumes CPU is. We show both are challenging algorithmic problem in their own right. For each aspect we design algorithms that appear to work quite well in practice.
3. We perform an extensive experimental evaluation of our approaches on real-life large data sets. Our results show significant improvements compared with the baseline.

3. I/O SAVINGS

In a setting with limited memory and a large, web-scale index, disk accesses may become the primary bottleneck when performing query processing. In these cases, the time to answer a query may be dominated by random disk seek required to gather those posting lists not resident in memory. In such an environment, the limitations on throughput imposed by disk accesses can seriously hinder performance. However, the batch query processor has several tools available to alleviate this hindrance that are unavailable to the online search engine. Amongst these are query reordering and clairvoyant cache eviction strategies.

3.1 Query Reordering

Almost any caching policy benefits from locality – while having a particular term occur at a uniform spread throughout a query stream would incur numerous disk accesses and cache evictions, having the same terms occur sequentially or near sequentially may only necessitate a single disk lookup. It is this intuition that motivates our first family of batch query optimizations – reordering queries in such a way that the impact of disk reads is minimized.

We illustrate two basic classes of heuristics for I/O improving query reordering, with the intent of improving term locality in the query stream for better cache eviction performance and therefore reduced disk-induced latency.

- **Sorting-Based Reordering** This technique simply sort the queries alphabetically with the hope that the queries will be clustered.
- **Cluster-Based Reordering** This technique imparts a top-down or bottom-up agglomerative clustering on queries based on the frequency of terms present in each query. The global query frequency of all terms in a query set is first computed. Terms are now ordered by their query-frequency. Traversing from most to least frequent, queries are clustered based on the presence or absence of terms. See [10] for details.

We note that there are many other heuristics that one can think of, such as the TSP-based technique proposed in [14] and so forth. For the batch query processing, a fast heuristic is more preferable as this is an online process, so we believe that our proposed techniques give a good tradeoff between the effort used on re-ordering the query stream and the improvements we get.

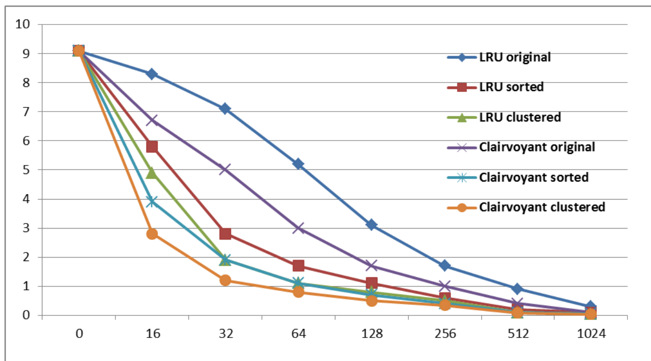

Figure 1: Caching performance using LRU and Clairvoyant, with different query stream reordering, on our 1.16 million query stream. The x-axis is the size of the cache in millions, and the y-axis is the size of data needs to be transferred from disk in TB.

3.2 Cache eviction

Traditionally caching technique can not explore the clairvoyant feature because the query stream in the future is unknown. But in our case where the future of the query stream is known, clairvoyant caching technique will be realistic. The basic idea for clairvoyant caching is to evict things whose next use will occur farthest in the future and it is shown to be optimal under certain cost model. For details please refer to [7]. Figure 1 shows the different caching performance for original query stream and reordered query stream, on the 1.16 million distinct queries from Excite query log (for details see Section 5). From Figure 1 we can see that after reordering the query stream, the caching performance is greatly improved. Moreover, clustered-based reordering outperforms sorting-based reordering. Also clairvoyant caching technique is much better compared with our baseline caching technique LRU.

4. REUSE OF SUB-QUERIES

In many other search engines, the inverted index fits in the main memory and the CPU computation is the main bottleneck for query processing. Actually this is the real case for many of the commercial search engines such as Google or Yahoo! [12]. So in this section we discuss the techniques to improve query processing in this case.

If one assumes that the inverted index is able to reside in main memory, the problem faced in batch processing is: given a large sequence of distinct queries, we want to find efficient techniques to process these queries in a batch mode. The general idea for speeding up the query processing is to store and re-use the sub-queries on the fly. For instance, two queries share the same sub-query such as '*free software download*' and '*free ebook download*' can be re-phased as '*free download* $>$ X' then 'X *software*' and 'X *ebook*', where X will be interpreted as an intermediate results for the corresponding pair. A similar idea is studied in [24] and [9]. Unlike prior work, the batch setting is able to leverage future knowledge via a clairvoyant algorithm, alleviating the difficult task of modeling sub-query frequency estimation.

In this work, we use a very simple model for the cost of query processing, just the sum of the inverted list length for all the terms in this query as in [24, 9]. For instance, the cost for the two-term query '*free download*' will be modeled as list-length(*free*) plus list-length(*download*). By using this model, we always get benefit by storing the sub-query for '*free download*' and then re-use it more than once, because the list length for the intersection of the sub-query '*free download*' is always not longer than that of either of the terms. Note, however, that this simplifying independence assumption is often violated in reality and typical query processing systems have to take the *document-at-a-time* technique and skipping behavior of the query processing into account. For the two queries '*free software download*' and '*free ebook download*', if the inverted list for '*ebook*' and '*software*' are much shorter compared with '*free*' and '*download*', we can potentially do much worse by computing and storing the result for the sub-query '*free download* ' and then re-use it for only a few times. compared with the straight forward approach. The reason is that in the original query most of the inverted lists in '*free download*' will be skipped when doing *document-at-a-time* because they are quite long compared with the third list, but if you materialize it most of the lists in '*free download*' will not be skipped, resulting in a worse performance. But we will see that in general on large scale (many queries) our model estimates the query cost quite well.[4]

The rest of this section will be organized as follows: we first describe our query processor in section 4.1, then we divide our problem into two parts: in section 4.2 we assume there is a fixed amount of memory budget and try to get the maximal benefit, without evicting materialized sub-queries. Then in section 4.3 we reorder and evict the materialized sub-queries to reduce further the amount of main memory consumption. Our framework is shown in Figure 2. From Figure 2 we can see that in the first phase, we are given a memory budget and try to take advantage of the clairvoyant to rewrite the query stream so that the maximal benefit is achieved. In the second phase we reorder the query stream passed from the first phase and evict stale sub-query to further reduce the memory consumption.

4.1 Query Processor

In this section we describe the query processor used in our system. We re-implement the query processor based on the description in [35], we use Pfordelta compression [35] to compress the inverted list, with *document-at-a-time* and a 128 fixed block size for skipping. Besides the basic functionality, the processor is able to cache the intermediate result for sub-queries and evict a sub-query as well. The basic operations in our query processor includes:

1. $T_a\ T_b\ T_c \cdots\ >\ X$

   executes the query with terms $T_a\ T_b\ T_c \cdots$ and caches the intermediate results into X. Note that the terms $T_a\ T_b\ T_c \cdots$ can be the terms from the original query, or cached intermediate results itself.

2. $T_a\ T_b\ T_c \cdots$

   executes the query with these terms, that can also be cached intermediate results.

3. *Delete-X*

   deletes the cached sub-query represented by X

[4]Our techniques are not limited to this simple model; the design and evaluation of improved cost models are topics of future work.

With the modified query processor, we can rewrite the query stream following the above grammar and our query processor will automatically cache the sub-queries and reuse it. An example is shown in Figure 3. From Figure 3 we can see that when a pair is materialized and reused, it replaces the corresponding terms in the query, reducing the sum of list length because the intersection size is always not longer than the sum of the corresponding terms.

For the precise format of the cached sub-query, recall that an inverted list is a sequence of postings sorted by document ID, thus a posting in the cached intersections list simply contains the document ID in the intersection and information about corresponding score for the ranking purpose for the sub-query. Note that in this work we are not using overlapped sub-queries, as our initial experiments showed that they give no extra benefit because we have to maintain an extra data structure (to make sure the scores do not overlap).

Then, in the next section, we discuss how we rewrite the query stream given a fixed memory budget so that we can take full advantage of the sub-queries.

4.2 Fixed Memory Budget

In this section we assume that there is a fixed amount of main memory (S) that we can use, and we can materialize as many pairs as long as we do not exceed the memory limitation. The cost for a query is modeled as the sum of the lists length for its terms. Given a query stream $\{ Q_1, Q_2, Q_3, \cdots, Q_n\}$ where Q_i contains terms $\{T_1, T_2, \cdots\}$, the computation cost for the query stream without rewriting is:

$$C = \sum_{i=1}^{n} cost(Q_i) = \sum_{i=1}^{n} \sum_{j=1}^{|Q_i|} listLength(T_j) \tag{1}$$

When we materialize a pair with terms T_i and T_j, we store its result into a temporal buffer X in memory, the computation cost for the producing of the pair is:

$$cost(T_i \quad T_j > X) = listLength(T_i) + listLength(T_j) \tag{2}$$

Also we need to consume $listLength(X)$ memory space to store the temporal result. When we re-use it, the corresponding computation cost will be modeled as $listLength(X)$ rather than $listLength(T_i) + listLength(T_j)$. Note that we are not using overlapped pairs, so when we materialize the pair and re-use it in certain queries, the maximal benefit for other potential good pairs will decrease if they share certain terms with this pair. For instance, if we rewrite the query $\{T_i\ T_j\ T_k\}$ into $\{X\ T_k\}$, materializing $\{T_j\ T_k\}$ will not give any extra benefit for this particular query.

Now we can formally define the problem: (1) given a query stream $\{ Q_1, Q_2, Q_3, \cdots \}$ with original cost C, which is modeled as the sum of all the lists lengths for the terms in the query sequence; (2) given a max buffer size S and a candidate pool filled with all the pairs $\{ p_1, p_2, p_3, \cdots\}$ which are potentially beneficial (we will talk about how to get this candidate pool later); and (3) assuming that when a pair p_i is chosen it will be used in the queries and replace the corresponding terms to reduce the cost of those queries by reducing the list lengths, the reduction is called $benefit(p_i)$; then we want to select a set of pairs $\{ p_1, p_2, p_3, \cdots \}$ with $size(p_1)$, $size(p_2)$, $\cdots$ which is the intersection size for the pair (memory cost), and with $cost(p_1)$, $cost(p_2)$, $\cdots$ which is the sum of lists lengths for the terms in the pair (computation cost), so that:

$$\sum size(p_i) \leq S \tag{3}$$

and minimize

$$C + \sum cost(p_i) - \sum benefit(p_i) \tag{4}$$

Note that this problem is different from the one in [24] because there the cost for producing the pair is ignored. This problem is in general NP-hard as we prove below.

PROOF. Assume there is a polynomial algorithm P for the above problem, we set S as large as needed, and assume

$$\sum cost(p_i) = C(p) \ . \tag{5}$$

Then we construct the following generalized maximum coverage problem [11]:

$$\sum cost(p_i) \leq C(p) \tag{6}$$

and maximize

$$\sum benefit(p_i) \tag{7}$$

We claim that if the polynomial algorithm P gives an optimal result for (4) then it also gives the optimal result for (7) with the constrain (6), which is a generalized maximum coverage problem and NP-hard by itself, which is a contradiction. □

Now we describe our algorithm for the query stream rewriting. Given a query stream we first produce the candidate pool and the bipartite graph:

- **Candidate Pool** We simply scan the query stream and put all the pairs which appear more than once into the pool. As most of the queries are short, this phase is fast and the time cost can be ignored even for a large query stream.

- **Bipartite graph** The relationship between pairs of sub-query in the candidate pool and the queries can be seen as a bipartite graph as shown in Figure 4. This relationship can be easily established from the scan above and the cost can be neglected.

Then our algorithm works as follows (*Clairvoyant Greedy*):

1. Set available space as S. For all the pairs in the candidate pool, compute its potential max benefit score (the pairs will be overlapped so the max benefit is not likely to achieve). Store them in a heap.
2. Pick the pair with the maximal benefit score per caching size as the current pair, extract it from the pool. Update the potential max benefit score for all the other pairs which share terms in the same query Q with the current pair based on the bipartite graph.
3. Remove the links for the pairs which share the same terms in Q with the current pair in the bipartite graph, update the available space by decreasing it by the size of the current pair.
4. If there is space available, repeat. Otherwise, exit.

Essentially our algorithm is a greedy one, which chooses the best choice by choosing the pair with largest ratio of benefit by space. Note that the third step is important, which breaks the outdated link in the bipartite graph. By doing this, we will not over-decrease the benefit score in the future which makes the algorithm unfair.

As mentioned, all previous work only studies the usage of pairs of terms. However, in practice common patterns also appear in 3-term combinations and more-term combinations, such as '*cheerleader champions 1999*' or '*Chicago advertising agencies*'. To take advantage of 3-term combinations and more-term combinations, we only need to change the candidate pool and add the 3-term combinations and more-term combinations to the pool in our algorithm mentioned above, then greedily choose the best one and so on. The problem here for batch query processing is that will be too expensive to scan the query stream again and again and naively count all the 3-term combinations and more-term combinations because it will be exponentially growing, and also if we keep one entry for each of the possible more-term combination for checking, the memory usage will be huge. The algorithm we use to get 3-term combinations and more-term combinations candidate is similar to 'Apriori' as in [1], basically we do a bottom-up search based on what we already have in the pool. In general, we start from 2-term combinations in the candidate pool, add an entry for checking for the 3-term combinations only if all its 2-term appear in the pool already. For example, we will add an entry for '*cheerleaders champions 1999*' only if '*cheerleaders champions*' and '*champions 1999*' and '*cheerleaders 1999*' are all in the pool already. By doing this the cost for pool-filling will be reduced greatly.

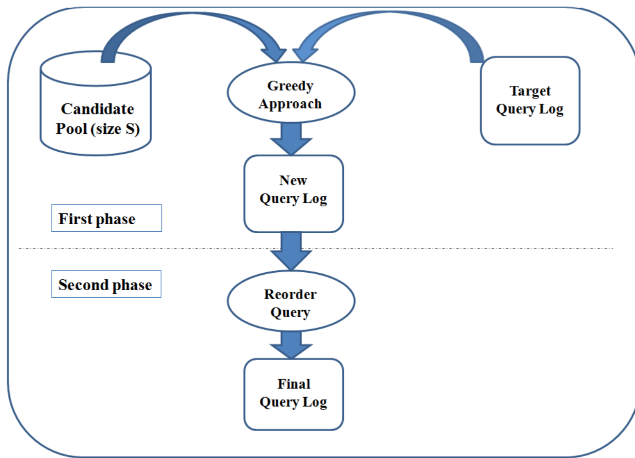

Figure 2: Framework for query rewriting.

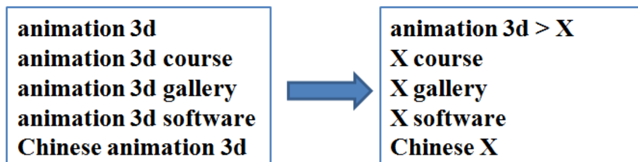

Figure 3: An example for query rewriting.

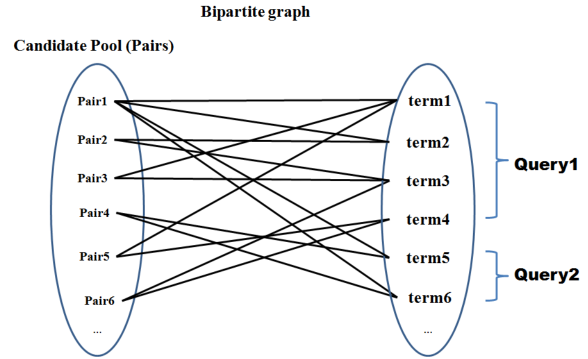

Figure 4: Bipartite graph for pairs and queries.

4.3 Reordering to Reduce Memory Usage

In this section, we get the query stream passed from our clairvoyant greedy algorithm, basically the query stream looks like the one in Figure 3, containing many materialized pairs. Note that in the previous section we take full advantage of the clairvoyance and greedily materialize the sub-query which will be useful in the future query using a fixed amount of memory. Actually after a pair is materialized and re-used, it can be removed from the memory, using the "DELETE" operation in our query processor. In this case the memory usage will be reduced further. As we already know the future of each appearance for the materialized pair, we can use clairvoyant caching to evict pairs when we want to set the memory buffer smaller. Further more, we can re-order the queries so that the materialized sub-queries are clustered and then we can further reduce the memory usage using clairvoyant caching. Actually this is the same as the caching problem explored in the first part, where the basic unit for caching is a materialized pair instead of a single term. From the first part we know the best heuristic is the cluster technique, so we use it to cluster the materialized pairs. Then we use clairvoyant caching to evict the sub-queries when the buffer size is set smaller.

5. EXPERIMENTAL EVALUATION

We now present our experimental setup and give some results for our algorithms.

Data sets and experimental setup: For our experiments we use a subset of 10 million pages selected at random from a crawl of about 120 million web pages crawled by the PolyBot web crawler [31]. This subset size corresponds to a scenario where the pages are evenly handed out over a distributed search engine. The uncompressed size of the pages was more than 100G. After indexing the compressed size of the inverted index is 4.2 G using Pfordelta compression technique without position information. For ranking we use Okapi BM25.

Queries are taken from a large log of queries issued to the Excite search engine on 1999. Totally there are $2,477,283$ queries in the original query log and $1,161,793$ distinct queries, where $840,156$ of the queries are singleton queries. We remove the stop words and non-Ascii characters in the query log. The distribution for queries is shown in Figure 5, and follows a Zipf distribution. Figures 6 and 7 plot the distribution for pairs and 3-term frequencies in the query

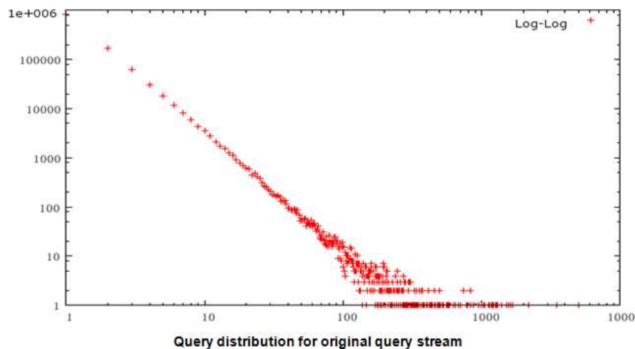

Figure 5: Distribution of queries frequency (log-log scale).

stream without counting duplicated queries. It matches with the results mentioned in [9], where not only the query frequency in the original query stream follows a power law distribution, but also the frequency for pairs and 3-term combinations follow the same distribution. Actually this indicates that *caching a few good pairs will give most of the benefit.* From Figures 6 and 7 we can see that the slope for 3-term combination frequency is larger than that for pairs, as expected.

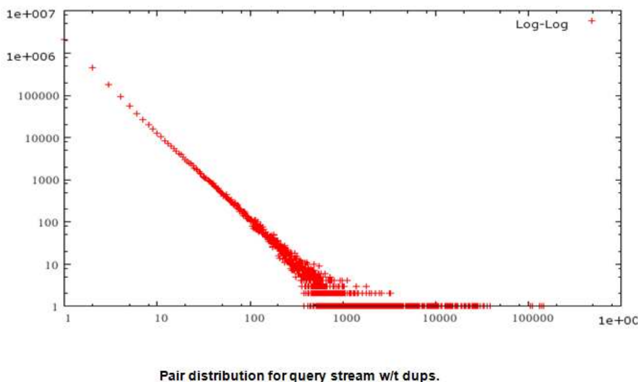

Figure 6: Distribution of pairs frequency (2-term sub-query, log-log scale).

Next we show that with our clairvoyant greedy algorithm, the performance for batch query processing can be improved. Figure 8 shows the performance gain with different caching space. The x-axis is the memory budget with respect to the total compressed inverted index size. From Figure 8 we reenforce the notion that a small number of sub-queries give the most of the benefit. And even with only 8% of the index size as cache size we can get up to 26% speed up in the processing time on the 1.1 million query stream. This is mostly due to the fact that the distribution of the subqueries (pairs) follows a Zipf distribution as already mentioned, so as we increase the buffer size the candidate become less promising. Also from Figure 8 we see that using list length as the query cost model works pretty well, as it gives a good estimate of the performance gain in processing time.

To compare with the techniques used in [24], we imple-

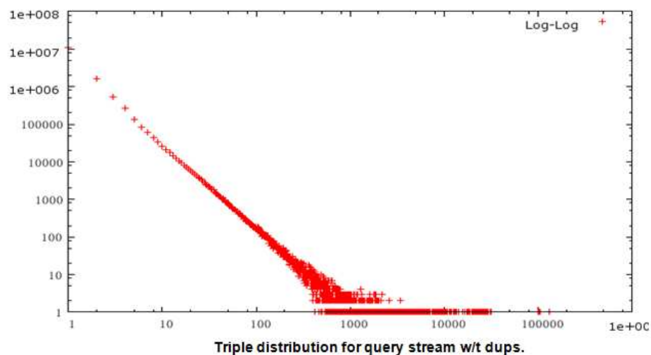

Figure 7: Distribution of 3-term combination frequency (log-log scale).

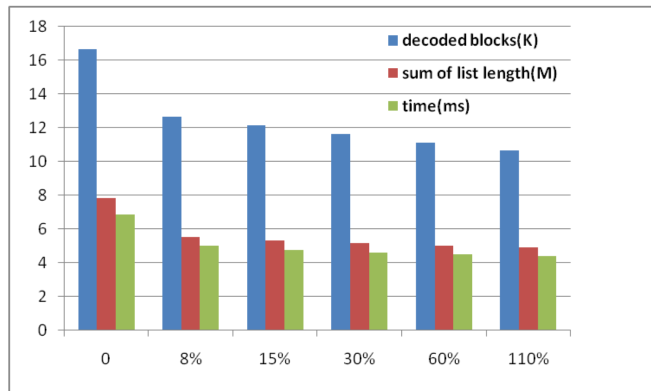

Figure 8: Performance gains with caching pairs (2-term sub-queries) using our greedy algorithm, with different cache size with respect to the inverted index size. Zero means no sub-queries are cached and re-used, 110% means the cache size is even larger than the inverted index size. We average the numbers over our 1.1 million queries.

mented their intersection caching algorithm using strict admission. Note that due to the different compression technique and skipping strategies, the absolute performance gain will vary compared with [24]. We compare the average time for each query in our query stream, using 30% inverted index size as the caching buffer. Our algorithm is then better (4.6 ms per query) than the strict admision case (6.05 ms per query) and the naive algorithm (6.86 ms per query), mainly due to the use of the clairvoyant technique that only stores the results for pairs that will be useful in the future. That is a 24% improvement over [24].

Figure 9 shows the performance gain with different number of queries, with unbounded memory size. Here we show the improvement based on the measure of the sum of list length instead of real query processing time for convenience. From Figure 9 we can see that for batch query processing, we need more queries to get more benefit.

Figure 10 shows that after putting 3-term combination and 4-term combination into our candidate pool, extra benefit is achieved compared with using 2-term sub-query only. But as we can see, the improvement is not huge (less than 10%), the main reason is that even a three term combina-

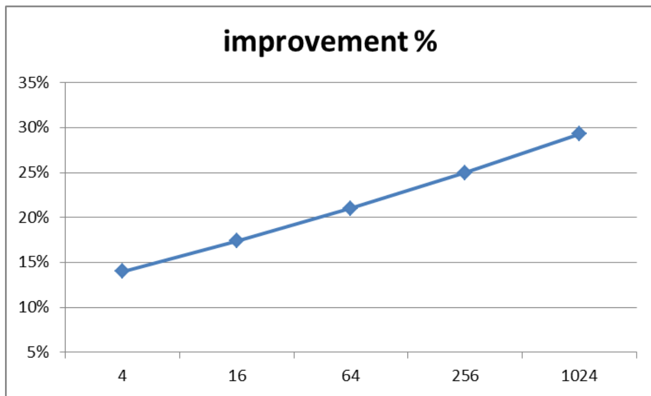

Figure 9: Performance gain with different numbers of queries (in thousands). The benefit is showed as the sum of lists lengths.

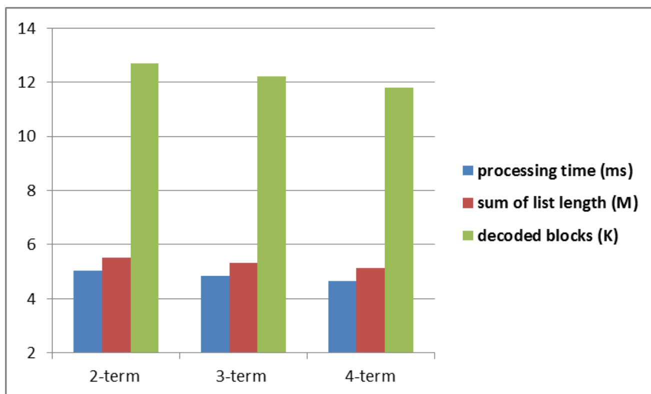

Figure 10: Performance with 2-term, 3-term and 4-term sub-queries, averaged on our 1.1 million query stream. For 5-term sub-query the additional benefit is tiny.

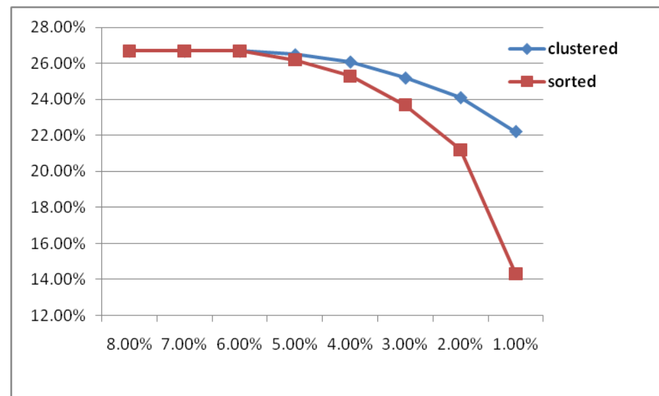

Figure 11: Performance gains with clairvoyant caching on clustered and sorted rewritten query stream, x-axis is the buffer size wrt the total inverted index size, y-axis is the improvement in query processing time compared with the naive query processing.

tion such as '*cheerleader champion* 1999' is very promising, using only 2-term sub-query will not completely lose the benefit because it is very likely that '*cheerleader champion*' or '*cheerleader* 1999' will be materialized.

Note that in Figure 8 we know that even with a buffer size of 8% of the inverted index we can get a good improvement compared with not using sub-queries. Actually we can reduce the memory usage further by re-ordering the appearance of the sub-queries and making them clustered, then use a clairvoyant caching technique as mentioned in Section 4.3. Figure 11 shows the result. We can see that if we use the cluster technique to cluster the materialized sub-queries and then use clairvoyant caching for evicting the sub-queries, actually we can use even less than 8% inverted index size to get the same amount of benefit. Moreover, as we decrease the cache buffer size further, the eviction of sub-queries starts to "hurt" our performance gains, and also we can see that the cluster technique outperforms the sorting technique.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we present *Batch Query Processing* as a new and important problem for search engines. We study the possible techniques in *Batch Query Processing* for improving the I/O and CPU performance. Our experiments show significant improvements compared with the baseline approach.

There are some limitations and interesting future works. Firstly, how to combine our work with the pruning technique is an open problem to us. Secondly, for the case where the inverted index is fit into main memory, it is interesting to explore if there is a way to maximize the query processing benefit and reduce the memory consumption at the same time. In other words, it is interesting to combine the problem mentioned in Section 4.2 (which is NP-hard) and Section 4.3 (which is also NP-hard). Thirdly, it would be interesting to have a technique which optimizes both the I/O and CPU at the same time.

Another interesting future work will be to apply batch query processing technique for the online case given that the largest search engine gets tens of thousands of queries per second. Let us assume that the probability of a query q (term pair tp) appearing in a query stream is p_q (p_t). It would be worth to delay answering the query if appears again quickly enough. However caching does the same function without needing to delay the query. Hence, caching or reusing would be interesting if a query or a term-pair appears soon enough. The fraction of those queries would be p_q^2 or p_t^2 for terms. Modeling the query or term pair distribution as a power law of parameter α and summing for all possible queries or term-pairs, it is not difficult to show that the fraction of frequent queries in any interval of time is upper bounded by $\zeta(2\alpha)/\zeta(\alpha)^2$ where where ζ is the Zeta Riemann function. Using the power-law parameters for our data set we obtain an upper bound for this fraction of 0.56 for queries and 0.47 for terms. As the hit-rates for caching are close to, or well above, these numbers, respectively, we can infer that the query distribution should become more biased to be able to make query or term-pair reuse interesting in the online case or that the query load should be much larger than what it is today. Nevertheless, in the future the query load will be large enough and hence our work will make sense also for the online case.

7. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th VLDB Conference*, pages 487–499, 1994.

[2] S. Albers. New results on web caching with request reordering. In *Proc. of Sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, 2004.

[3] V. Anh and A. Moffat. Compressed inverted files with reduced decoding overheads. In *Proc. of the 21th Annual int. ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 290–297, 1998.

[4] J. Attenberg and F. Provost. Why label when you can search? strategies for applying human resources to build classification models under extreme class imbalance. In *KDD*, 2010.

[5] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdoch, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *Proc. of the 30th Annual int. ACM SIGIR Conference on Research and Development in Information Retrieval*, 2007.

[6] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[7] L. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 1966.

[8] B. B. Cambazoglu, F. P. Junqueira, V. Plachouras, S. Banachowski, B. Cui, S. Lim, and B. Bridge. A refreshing perspective of search engine caching. In *19th International World Wide Web Conference*, 2010.

[9] S. Chaudhuri, K. Church, A. C. Konig, and L. Sui. Heavy-tailed distribution and multi-keyword queries. In *Proc. of the 30th Annual int. ACM SIGIR Conference on Research and Development in Information Retrieval*, 2007.

[10] C.-S. Cheng, C.-P. Chung, and J. J.-J. Shann. Fast query evaluation through document identifier assignment for inverted file-based information retrieval systems. In *Inf. Processing and Management*, 2006.

[11] R. Cohen and L. Katzir. The generalized maximum coverage problem. *Information Processing Letters*, 2008.

[12] J. Dean. Challenges in building large-scale information retrieval systems. In *Second ACM International Conference on Web Search and Data Mining*, April 2009.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

[14] S. Ding, J. Attenberg, and T. Suel. Scalable techniques for document identifier assignment in inverted indexes. In *19th International World Wide Web Conference*, April 2010.

[15] T. Feder, R. Motwani, R. Panigrahy, and A. Zhu. Web caching with request reordering. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA'02)*, 2002.

[16] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *18th International World Wide Web Conference*, 2009.

[17] P. Ipeirotis, L. Gravano, and M. Sahami. Probe, Count, and Classify: Categorizing Hidden-Web Databases. In *SIGMOD*, 2001.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[19] M. Kaszkiel, J. Zobel, and R. Sacks-Davis. Efficient passage ranking for document databases. *ACM Transactions on Information Systems*, 17:406–439, 1999.

[20] R. Kumar, K. Punera, T. Suel, and S. Vassilvitskii. Top-k aggregation using intersections of ranked inputs. In *Second ACM International Conference on Web Search and Data Mining*, April 2009.

[21] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the 28th VLDB Conference*, 2002.

[22] R. Lempel and S. Moran. Optimizing result prefetching in web search engines with segmented indices. In *12th International World Wide Web Conference*, 2003.

[23] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *Proceedings of the 29th VLDB Conference*, pages 129–140, 2003.

[24] X. Long and T. Suel. Three level caching for efficient query processing in large web search engines. In *14th International World Wide Web Conference*, 2005.

[25] E. Markatos. On caching search engine query results. In *5th International Web Caching and Content Delivery Workshop*, 2000.

[26] A. Ntoulas and J. Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proc. of the 30th Annual int. ACM SIGIR Conference on Research and Development in Information Retrieval*, 2007.

[27] Okapi **bm**25. http://en.wikipedia.org/wiki/Okapi_BM25/.

[28] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. of the American Society for Information Science*, 47(10):749–764, 1996.

[29] S. Rajan, D. Yankov, S. Gaffney, and A. Ratnaparkhi. A large-scale active learning system for topical categorization on the web. In *WWW*, 2010.

[30] P. Saraiva, E. de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. RibeiroNote. Rank-preserving two level caching for scalable search engines. In *Proc. of the 24th Annual int. ACM SIGIR Conference on Research and Development in Information Retrieval*, 2001.

[31] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *Proc. of the Int. Conf. on Data Engineering*, pages 357–368, 2002.

[32] G. Skobeltsyn, F. Junqueira, V. Plachouras, and R. Baeza-Yates. Resin: A combination of results caching and index pruning for high-performance web search engines. In *Proc. of the 31th Annual int. ACM SIGIR Conference on Research and Development in Information Retrieval*, 2008.

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

[34] Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *IEEE Infocom*, 2002.

[35] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *18th International World Wide Web Conference*, 2009.

[36] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *17th International World Wide Web Conference*, 2008.

[37] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 2006.