Cache-Based Query Processing for Search Engines

B. BARLA CAMBAZOGLU, Yahoo! Research
ISMAIL SENGOR ALTINGOVDE, L3S Research Center
RIFAT OZCAN and ÖZGÜR ULUSOY, Bilkent University

In practice, a search engine may fail to serve a query due to various reasons such as hardware/network failures, excessive query load, lack of matching documents, or service contract limitations (e.g., the query rate limits for third-party users of a search service). In this kind of scenarios, where the backend search system is unable to generate answers to queries, approximate answers can be generated by exploiting the previously computed query results available in the result cache of the search engine. In this work, we propose two alternative strategies to implement this cache-based query processing idea. The first strategy aggregates the results of similar queries that are previously cached in order to create synthetic results for new queries. The second strategy forms an inverted index over the textual information (i.e., query terms and result snippets) present in the result cache and uses this index to answer new queries. Both approaches achieve reasonable result qualities compared to processing queries with an inverted index built on the collection.

Categories and Subject Descriptors: H.3.3 [**Information Storage Systems**]: Information Retrieval Systems

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Web search engine, result caching, result aggregation, query view, system availability

ACM Reference Format:
Cambazoglu, B. B., Altingovde, I. S., Ozcan, R., and Ulusoy, Ö. 2011. Cache-Based query processing for search engines. ACM Trans. Web 6, 4, Article 14 (November 2012), 24 pages.
DOI = 10.1145/2382616.2382617 http://doi.acm.org/10.1145/2382616.2382617

1. INTRODUCTION

In everyday life, a common way of accessing digital information is to issue queries to different kinds of information retrieval systems, such as DataBase (DB) systems, medium and large-scale search engines, digital libraries, and Peer-to-Peer (P2P) search systems. In the general context of search systems, it is common practice to cache query results to reduce query processing and/or data transfer costs. For instance, database systems might cache queries and their results at the client side to avoid the data transfer cost between server and client as well as avoiding the query processing cost at the server. Result caching is also important for search engines. By caching the results of a large number of frequently and/or recently issued queries

This work is partially supported by EU FP7 Project CUBRIK (contract no. 287704).
Authors' addresses: B. B. Cambazoglu, Yahoo! Research, Barcelona, Spain; I. S. Altingovde, L3S Research Center, Hannover, Germany; R. Ozcan (corresponding author) and Ö. Ulusoy, Bilkent University, Ankara, Turkey; email: rozcan@turgutozal.edu.tr.

DOI 10.1145/2382616.2382617 http://doi.acm.org/10.1145/2382616.2382617

at cache servers, the query processing cost incurred on the backend servers and the response times experienced by the end-users are significantly reduced [Baeza-Yates et al. 2008].

In general, a result cache serves results only for hits, that is, queries whose results are readily available in the cache. However, there are also a number of proposals in the literature to exploit a result cache to fully or partially answer misses, that is, queries whose results are not cached. The earliest approaches are proposed for DB systems, under the names of query folding [Qian 1996] and semantic caching [Dar et al. 1996]. For instance, if two queries have previously retrieved the results for "all cars made by Toyota" and "all cars produced after year 2000", it suffices to intersect these cached results to answer a new query requesting "Toyota cars made after year 2000". These approaches exploit the fact that database queries usually include atomic predicates that are combined with various logical operators and the answer set of a query obtained from a traditional DB system is always complete, that is, all data satisfying the query conditions are retrieved. In previous studies, cache-based result generation is shown to reduce response time and save significant data transfer costs, especially in mobile database applications. More recently, the idea of generating approximate results from a result cache for miss queries is also applied to P2P systems [Zimmer et al. 2008] and similarity search in metric spaces [Falchi et al. 2008].

Obviously, search engines[1] have some fundamental differences with respect to DB systems. First, queries are usually a simple conjunction of a few keywords without any complex predicates. This renders semantic caching techniques from the DB literature less useful. Second, search engines present and cache very few best-matching results (typically, about 10 per query), but not complete result sets as in traditional DBs. This implies that it is not possible to guarantee correct results for most of the queries. Finally, the result transfer cost is not significant, unlike DB and P2P applications. Given also the highly competitive search market, it is more beneficial to process at the backend the queries that lead to a cache miss, instead of attempting to generate potentially lower-quality results using the result cache.

Despite the aforementioned observations, there are several scenarios where cache-based result generation may be useful or even necessary. For example, a backend search system may be unavailable for a certain period of time, due to a particular reason such as power failure, natural disaster, hardware and software failure, network partition, operator error, or denial-of-service attack [Oppenheimer et al. 2003]. At first thought, the unavailability problem may be seen as a relatively less important issue given the low likelihood of it to happen in practice. Unavailability incidents, however, especially if they take more than a few minutes, may imply major consequences for a commercial Web service. In addition to financial losses (potentially in the order of millions of dollars for a large company), the reputation of the Web service is hampered. Cache-based query processing provides a low-cost solution to generate partial answers to queries issued during the downtime period. As another scenario, consider a search system that does not own a query processing backend and postprocesses results from other search engines. Metasearch engines and some verticals are examples of such systems. These systems may prefer to answer certain queries using their own result caches instead of forwarding to component subsystems, which might incur financial costs (e.g., payments per-click or per-result [Chidlovskii and Borghoff 2000]). In this particular example, the backend system is not available for query processing due to some financial constraints based on company policies. As a final scenario, it might happen that the backend search system finds no matching results for a given query. In

[1]We broadly mean small- to large-scale Web search engines, metasearch engines, verticals, and others.

this case, a synthetic query result can be generated using the cache. In Section 2, we present these scenarios in more detail.

We emphasize that our cache-based query processing strategy is not designed to substitute a backend search system that is fully available to serve queries. The main idea here is to use the entries in the result cache (e.g., rankings or snippets) as a last-resort option to answer user queries during periods in which obtaining the search results from the backend search system is impossible (physical unavailability) or infeasible (logical unavailability). All techniques we develop in this work rely on the existence and continuous availability of a large, general-purpose result cache. In most cases, this is a reasonable assumption as search engines maintain large result caches that can readily serve frequently and/or recently submitted queries, thus reducing the query traffic volume hitting the backend search system and the associated computational costs [Cambazoglu et al. 2010a]. We provide some preliminaries and discuss our problem setting more concretely in Section 3.

Given the availability of some precomputed result entries in the cache, the research problem is to form answers for queries whose results are not found in the result cache (i.e., cache misses). In this work, we investigate two alternative approaches to create synthetic answers for unseen queries using previously cached result entries. For a given query, the first approach identifies the cached queries that are most similar to the query and uses their rankings to create an aggregate ranking. The second approach builds an inverted index using the textual information present in the result cache and evaluates new queries over this inverted index. The proposed cache-based query evaluation techniques are described in Section 4.

We evaluate our strategies via simulations over a real-life query log and a document collection. Details of our experimental setup are given in Section 5. Our experiments indicate that cache-based query evaluation is indeed a promising approach that can substitute query processing on architecturally sophisticated backend search systems in case of unavailability. In particular, we observe that cache-based query processing can correctly identify and serve, on average, up to one-third of the top 10 search results generated by a backend search system. Detailed experimental results are provided in Section 6. We survey the related work in Section 7. A number of issues that are related to the proposed work are discussed in Section 8. We conclude the article in Section 9.

2. MOTIVATING SCENARIOS

The cache-based query processing strategy proposed in this work is most useful when the backend search system is unable to generate any results for some queries due to physical or logical unavailability. In what follows, we present several scenarios and motivate some potential unavailability problems.

2.1. Physical Unavailability of the Backend

In large-scale Web search engines, the backend search clusters are usually geographically replicated to improve the availability of the search service. However, most small-scale or medium-scale search services (e.g., digital libraries) cannot afford investing on this kind of costly multisite or cloud computing solutions. For such services, it may not even be feasible to build replicas of their main search clusters due to high maintenance and operational costs (e.g., costs due to power consumption and cooling). While it might seem tempting to replicate a pruned version of the index, earlier works show that for correctly answering 20% of the cache misses, at least 10% of the entire index is required [Skobeltsyn et al. 2008]. Since physical unavailability situations are rare, such mechanisms may not be cost effective to implement.

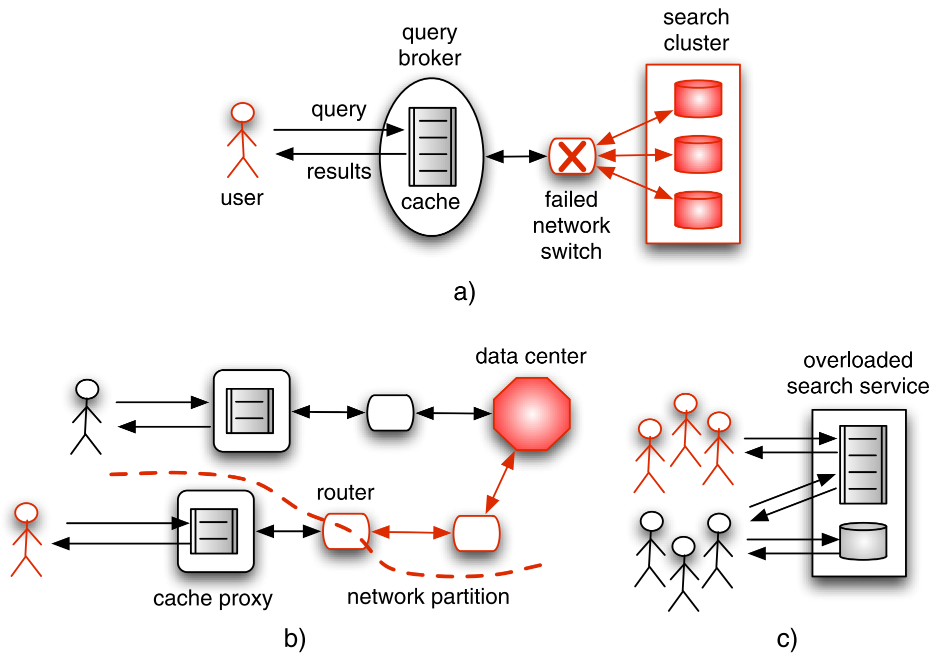


Fig. 1. Use case scenarios: (a) network switch failure in a search site; (b) network partition in a centralized search engine setting with regional proxies; and (c) bursty query traffic. In the illustrations, the red drawings indicate failed system components and users who receive lower-quality search results.

In contrast, cache-based query processing is a low-cost, last-resort availability solution, especially suitable to medium-scale or low-budget search services. In particular, for short durations of physical unavailability (e.g., a few minutes), the system can switch to the cache-based query processing mode and let its users know that presented results are partial. This is clearly preferable to displaying a blank result page with a "service unavailable" message. It can decrease user frustration as the partial results may keep users busy at the site until the system comes back to full force. In the following, we discuss three use case scenarios (Figure 1) to further motivate our research problem, without restricting ourselves to a particular architecture. In all scenarios, we assume that a result caching module is still functional and accessible during the downtime of the main query processing system.

Hardware failure. A medium-scale search engine commonly contains a search backend composed of multiple query processing nodes (e.g., a PC cluster) and a central query broker (Figure 1(a)). For improved fault tolerance, the inverted index is typically document-based partitioned over the processing nodes [Barroso et al. 2003]. The search process is embarrassingly parallel, that is, the query is issued to all nodes, each generating a local top k result set for the query. The central broker is responsible for merging the local result sets into a final top k result set as well as caching the final results in its cache. In case of a hardware failure that prevents the access to the backend (in our example, a failed network switch), some queries can be served using the information in the result cache of the query broker.

Network partition. Some search engines maintain regional cache proxies to reduce the round-trip network latency between the search site and users (Figure 1(b)). The query results are first looked up in the cache located in the region of the user. The main search site is contacted to get the answer only if the query results are not cached in the proxy. In case of a network partition that breaks the connection between proxy and main search site, queries can be answered by regional cache proxies using the proposed strategy.

Burst in query traffic. Most search services have hardware investments to sustain a certain peak query processing throughput. However, there may be occasional bursts in the query traffic of a search service, and the query traffic rate may exceed the sustainable query throughput. In such cases, it may not be possible to process all queries within reasonable response times. In case of heavy traffic, the cache-based query processing strategy can be used to serve the query traffic volume that exceeds the peak throughput sustainable by the search service (Figure 1(c)).

2.2. Logical Unavailability of the Backend

Usage restrictions. Consider a metasearch system that generates the results of user queries by aggregating the search results retrieved from other search systems. In this scenario, it is likely that each query submitted to the component search systems will incur a financial cost to the metasearch system [Chidlovskii and Borghoff 2000], or the metasearch system, due to its contract, may be limited to submit only a fixed number of queries to the other search systems. In the former case, generating cache-based results for at least some queries may be tempting for financial reasons. In the latter case, this becomes mandatory after the metasystem reaches its query limit and cannot send more queries to its search subsystems.

No matching results. Some portion of the query traffic volume leads to no results in search services, that is, some queries do not match any documents in the index. A potential solution is to reevaluate the query by dropping query terms until some results are retrieved (i.e., as in long query reduction [Kumaran and Carvalho 2009]). However, this may be computationally expensive and the query response times may increase. Cache-based query processing can be applied to such queries to return approximate answers with little computational overhead.

3. PRELIMINARIES

3.1. Background

Result caching. A result cache is a cost-effective solution for reducing the backend query traffic in search services. Typically, a result cache stores the results of frequently or recently issued queries. The first is motivated by the power-law distribution in query traffic, that is, a small fraction of queries that are highly repetitive (see Figure 1 in Cambazoglu et al. [2010a]). The second is motivated by the presence of a large fraction of queries having low interarrival times, that is, a high likelihood for the same query to repeat soon (see Figure 2 in Cambazoglu et al. [2010a]).

In general, each cache entry stores a result page corresponding to a query. A result page is typically composed of 10 results that are ranked in decreasing order of their relevance to the query. A result (document summary) is made up of a title, a URL, and a snippet. Basically, a snippet is formed of a few text pieces extracted from the parts of the document that match the query terms. Depending on the search service, some other information may be available in cache entries.

Inverted index. An inverted index is composed of a set of inverted lists $\mathcal{L} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_V\}$ and an index pointing to the start of the lists. Here, $V = |\mathcal{V}|$ is the

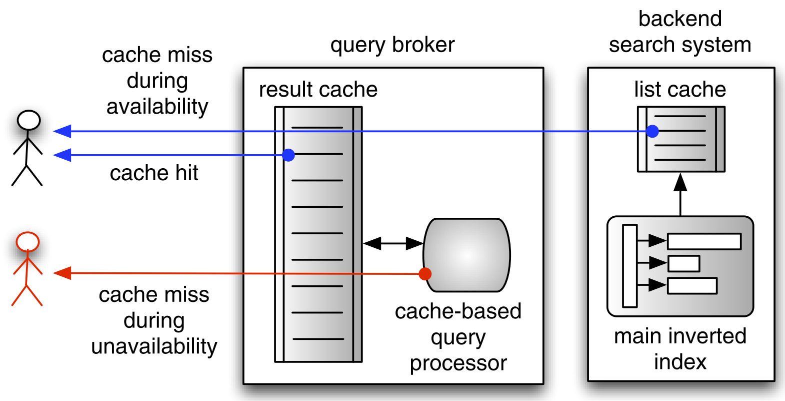


Fig. 2. The system model assumed in the article.

size of the vocabulary $\mathcal{V}$ of the indexed document collection $\mathcal{D}$. Each inverted list $\mathcal{I}_i \in \mathcal{L}$ is associated with a term $t_i \in \mathcal{V}$ and contains entries (called postings) for the documents containing the term it is associated with. A posting $p \in \mathcal{I}_i$ typically consists of a document id field $p.d = j$ and a term frequency field $p.f = f(t_i, d_j)$ for a document d_j in which term t_i appears. In query processing, $f(t_i, d_j)$ is used to compute a score contribution for d_j according to some relevance measure.

Query processing. While processing a user query $q = \{t_{x_1}, t_{x_2}, \ldots, t_{x_{|q|}}\}$ using an inverted index, the postings in the corresponding inverted lists of the query terms are traversed and the score contributions obtained from the postings are separately accumulated for each document. After all lists are processed, the documents are sorted in decreasing order of their final scores, and the highest ranked k documents are returned to the user. Interested readers may refer to Zobel and Moffat [2006] for more details.

3.2. System Model

We describe our system model by a simple but representative scenario: a two-level architecture in which the first level (query broker) contains a cache of query results and the second level (search backend) contains a query processing system using an inverted index (Figure 2). In this architecture, as long as the second-level system is available, queries are processed as follows. Queries are first looked up in the cache. In case of a cache hit, the results are immediately served by the cache. In case of a cache miss, queries are processed in the second-level system. The results are generated using the entire index and also cached in the first level. Typically, the backend search nodes employ various caches for efficiency, such as caches of inverted lists [Baeza-Yates et al. 2008] or even their intersections [Long and Suel 2005]. However, since such caches are located in the backend [Skobeltsyn et al. 2008], that is, decoupled from the broker, they are rendered useless in case of unavailability of the backend.

In most cases, the initial availability period of the search backend would be sufficiently long. Hence, we can assume a result cache that is warm enough, that is, a cache in which the results of frequent queries are mostly available. At a certain time point t, we assume that the search backend becomes unavailable or overloaded until time point t', at which time it recovers. During the interval $[t, t']$, queries that lead to a hit are served from the cache, as before. However, for a certain fraction of queries whose results are not found in the cache,[2] a last-resort ranking system, located in the first level, generates an approximate ranking by using the information stored in the result cache. The research problem is to find suitable cache-based query processing strategies that let the service continue its execution by generating approximate top k results for such queries. Obviously, computing the query results by using only the cache, instead of the backend index, is expected to degrade the search quality. Hence, the developed strategies should try to keep the quality loss as low as possible. Moreover, they should not be expensive, leading to a throughput bottleneck for the search service.

4. CACHE-BASED QUERY PROCESSING

We propose two alternative cache-based query processing approaches for computing the result page of a query q^*. The first approach creates an aggregate ranking using the rankings of cached queries that are similar to q^*. The second approach builds an inverted index using some text (in our case, query terms and/or snippets) extracted from the entries in the cache and evaluates q^* using this index. These two approaches are similar in that they both use the previously computed entries in the result cache to provide answers to future queries that lead to cache misses. They mainly differ in the way they create the result rankings returned to the user.

Before going into the detail of the approaches, let us introduce some notation. Given a query q, a top k list τ^q is a one-to-one mapping from the members of its top k set $\mathcal{T}_{\tau^q}$ to ranks in $\{1, 2, \ldots, |\mathcal{T}_{\tau^q}|\}$. The rank of a document $d \in \mathcal{T}_{\tau^q}$ is denoted by $\tau^q(d)$. We use $\mathcal{Q}_\mathrm{C}$ to denote the set of queries that are issued to the search engine during the availability period and whose top k results are stored in the result cache.

4.1. Aggregating Rankings of Similar Queries

This approach first generates a set $\mathcal{Q}_{q^*}$ of queries that are identified as similar to q^* and whose results are found in the cache. For every query $q_i \in \mathcal{Q}_{q^*}$, the ranking τ^{q_i} associated with the query is fetched from the cache. Thus, assuming $n = |\mathcal{Q}_{q^*}|$, we obtain a set $\mathcal{R}_{q^*} = \{\tau^{q_1}, \tau^{q_2}, \ldots, \tau^{q_n}\}$ of n rankings for queries in $\mathcal{Q}_{q^*}$. The problem now reduces to the problem of combining these rankings into a final ranking $\hat{\tau}^{q^*}$ that is as similar as possible to the actual ranking τ^{q^*}.

4.1.1. Selecting Similar Queries.

Although some complex techniques are available for identifying queries similar to a target query q^* (e.g., query clustering), we prefer three computationally inexpensive approaches. As similar queries, the first approach selects the cached queries whose terms form a subset of those in q^*, that is,

$$\mathcal{Q}^{\mathrm{sub}}_{q^*} = \{q : q \in \mathcal{Q}_\mathrm{C} \wedge q \subset q^*\}. \tag{1}$$

The second approach selects cached queries that contain all the terms in q^* plus one more term,[3] that is, the superset

$$\mathcal{Q}^{\mathrm{sup}}_{q^*} = \{q : q \in \mathcal{Q}_\mathrm{C} \wedge q^* \subset q \wedge |q| = |q^*| + 1\}. \tag{2}$$

The third approach simply combines the two preceding sets, that is,

$$\mathcal{Q}^{\mathrm{sub+sup}}_{q^*} = \mathcal{Q}^{\mathrm{sub}}_{q^*} \cup \mathcal{Q}^{\mathrm{sup}}_{q^*}. \tag{3}$$

We note that the first approach requires performing $2^{q^*} - 2$ lookups in the cache for all nonempty subsets of q^*. This is computationally inexpensive as $|q^*|$ is typically very

[2]In case of the unavailability of the search service, this implies every query that leads to a cache miss.

[3]We also tried to select queries that are larger supersets of q^*, but these query sets did not perform well.

small.[4] The second and third approaches require finding a subset of cached queries that include all terms in q^*. A feasible solution to form the query sets for these two approaches is to build an inverted index over the terms of all cached queries so that, given a query term, we can efficiently identify all queries containing that term. By processing q^* using this index and computing the union of the related inverted lists, we can quickly form $\mathcal{Q}_{q^*}^{\text{sup}}$ or $\mathcal{Q}_{q^*}^{\text{sub+sup}}$.

4.1.2. Rank Aggregation. We evaluate four simple approaches that produce $\hat{\tau}^{q^*}$ by aggregating the rankings in $\mathcal{R}_{q^*}$. In the literature, as we will discuss in Section 7, there are more elegant aggregation algorithms.[5] Our approaches work on a candidate document set $\mathcal{D}_{\text{q}^*}$, computed by taking the union of the top k sets as

$$\mathcal{D}_{q^*} = \bigcup_{\tau \in \mathcal{R}_{q^*}} \mathcal{T}_\tau. \tag{4}$$

Our techniques compute a score $s(q^*, d)$ for every document $d \in \mathcal{D}_{q^*}$. The aggregate ranking $\hat{\tau}^{q^*}$ is formed by sorting the highest-scoring k documents in decreasing score order.

Simple voting. In this aggregation approach, every document $d \in \mathcal{D}_{q^*}$ receives a unit vote of one from each ranking $\tau^{q'}$ that ranks the document (for all $q' \in \mathcal{Q}_{q^*}$), that is,

$$s(q^*, d) = \sum_{q' \in \mathcal{Q}_{q^*}} v(d, \tau^{q'}), \tag{5}$$

where vote $v(d, \tau^{q'})$ given to d by $\tau^{q'}$ is computed as

$$v(d, \tau) = \begin{cases} 1, & \text{if } d \in \mathcal{T}_\tau, \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

Jaccard-weighted voting. This aggregation approach weights each vote given by a ranking $\tau^{q'}$ using the similarity of the corresponding query q' to q^*, that is,

$$s(q^*, d) = \sum_{q' \in \mathcal{Q}_{q^*}} w(q', q^*) \times v(d, \tau^{q'}), \tag{7}$$

where weight $w(q', q^*)$ is computed by the Jaccard similarity between q' and q^* as

$$w(q', q^*) = \frac{|q' \cap q^*|}{|q' \cup q^*|}. \tag{8}$$

The intuition here is that queries with higher term overlap are more likely to contribute relevant results. Therefore, their votes are scaled in proportion to their similarity.

IDF-weighted voting. This is another weighted voting approach, where $w(q', q^*)$ is computed as[6]

$$w(q', q^*) = \frac{\text{IDF}(q')}{\text{IDF}(q^*)}, \tag{9}$$

[4]The average number of terms in a Web search query (cache miss) is about three.
[5]Our motivation for selecting simpler aggregation techniques is solely due to performance constraints.
[6]This equation assumes $q' \subset q^*$. Otherwise, the reciprocal of (9) is used.

where IDF(q) denotes the sum of the inverse document frequencies of terms in q.

Borda count. The previous approaches do not take the rank of documents into account in the aggregation process. To incorporate the rank information, we use a modified version of Borda's rank aggregation technique [Borda 1781]. The original algorithm of Borda simply maintains a vector of ranks r_i of size n for each document d_i found in at least one ranking $\tau_j \in \mathcal{R}_{q^*}$. That is, $\overrightarrow{r_i}[j]$ indicates the rank of document d_i in ranking associated with q_j, that is, $\overrightarrow{r_i}[j] = \tau_j(d_i)$. Borda's approach assumes that all rankings are full, in other words, they provide a rank for all documents. In our case, however, the rankings are partial, that is, there may be missing $\overrightarrow{r_i}[j]$ entries as a document d_i may not appear in ranking τ_j associated with q_j. For documents that are not available in a ranking, that is, for all $d_i \notin \mathcal{T}_{\tau_j}$, we compute $\overrightarrow{r_i}[j]$ as

$$\overrightarrow{r_i}[j] = |\mathcal{T}_{\tau_j}| + \left\lceil \frac{|\cup_{\tau \in \mathcal{R}_{q^*}} \mathcal{T}_\tau| - |\mathcal{T}_{\tau_j}|}{2} \right\rceil. \quad (10)$$

This is a neutral approach that assigns an average rank to missing documents. We also tried optimistic and pessimistic versions that assign ranks using (11) and (12), respectively.

$$\overrightarrow{r_i}[j] = |\mathcal{T}_{\tau_j}| + 1 \quad (11)$$

$$\overrightarrow{r_i}[j] = |\cup_{\tau \in \mathcal{R}_{q^*}} \mathcal{T}_\tau| \quad (12)$$

Since the results were not significantly different, we adopted the formula in (10) in our experiments.

After the rank vectors are formed, the final aggregate ranking can be determined in different ways. A typical approach is to compute the average rank of all documents and then rank the documents in increasing order of these averages. Instead of arithmetic mean, geometric mean, median, L$_2$ norm, or other functions can also be used. In our work, we adopted arithmetic mean as it yielded the best results. We compute the aggregate rank r'_i of a document d_i as

$$r'_i = \frac{\sum_{j=1}^{n} \overrightarrow{r_i}[j]}{n}. \quad (13)$$

We also tried different weighted versions of Borda, but the results were not significantly different. Hence, we omit them.

4.2. Building an Index over the Cache

An alternative approach is to generate representative term sets that will substitute the original terms of the documents during the unavailability period. As soon as the search backend becomes unavailable, the frontend starts building a temporary inverted index over the cache using the terms available in the cached results. Alternatively, the index can be periodically rebuilt (e.g., on a daily basis) without waiting for the unavailability of the main index. During the unavailability periods, the results of queries that are cache misses are computed over this index and returned to the user. This index can be built in different ways using the terms stored within the result cache entries.

4.2.1. Index Construction. We discuss two approaches for building the inverted index and a hybrid approach that combines these two approaches.

Query views. In this approach, we simply iterate over the cache and, for each cached document, construct a set of terms to represent it. The terms are extracted from the queries that matched the document in the past. More specifically, for each document d in the result cache, we identify every query $q \in \mathcal{Q}_{\mathrm{C}}$ such that $d \in \mathcal{T}_{\tau^q}$. In a sense, we form a query view $V(d)$ for the document [Altingovde et al. 2012; Poblete and Baeza-Yates 2008; Puppin and Silvestri 2006] as

$$V(d) = \bigcup_{\substack{q \in \mathcal{Q}_{\mathrm{C}}, \\ d \in \mathcal{T}_{\tau^q}, \\ t \in q}} t. \tag{14}$$

The motivation in this approach is to reconstruct the original terms of documents using the terms in query views, hoping that such terms will have a good coverage of the original terms. The coverage is expected to be higher for documents that are short or queried by many different terms. After the entire cache is traversed and all query views are created, an inverted index is built using the identified terms. Since the query view index is created using only the information in the result cache, there is no clue about the frequencies of terms in documents. Hence, we simply assume that $p.f = 1$ for all postings in the query view index.

Snippet terms. Another approach is to use the terms in the snippets, which are stored within the cache entries. This approach iterates over the result cache as in the previous approach and combines the snippets extracted for each document into one big document.[7] In a sense, this approach tries to reconstruct the original terms in the document using the pieces of text extracted from the document at different times. A temporary index is then built as before.

In constructing a snippet-based index, frequencies of the terms that appear in a document can be approximated more accurately in comparison to the query-view-based index discussed earlier. For instance, if a term appears in three different snippets extracted from a particular document, this implies that the term appears at least three times in the document.

Hybrid. Both of the aforementioned approaches have their own weaknesses in terms of their coverage of the vocabulary. The approach that relies on query views can only capture terms that appear in cached queries. Furthermore, the association between a term and a document remains undiscovered if the document containing the term is never made it into the top k list of any query including the term. The approach that relies on snippet terms has a better coverage of the document content, but it cannot capture external terms that are used to index the document (e.g., anchor text). In addition, especially for long queries, even if all terms are present in the full text of the document, the snippet generation algorithm may prefer not to include some query terms in the snippet. Therefore, herein, we also try a simple hybrid approach that uses both the terms in the query view and the snippets.

4.2.2. Query Processing. The queries that lead to a cache miss during the unavailability period are processed using the index constructed by the aforementioned approaches. For query processing, it is possible to employ either a conjunctive (AND) or disjunctive (OR) processing mode. In our work, we adopt the conjunctive mode of processing.

[7]Typically, the snippet computed for a document depends on the terms in the query. Therefore, there may be many different snippets computed for the same document.

Table I. Properties of the Query Sets (the reported values are for normalized queries)

Query set type	Query set size	Avg. query frequency	Avg. query length
Training	0.5M	9.74	2.56
Training	1.5M	4.05	2.79
Training	2.5M	2.84	3.05
Training	3.5M	2.29	3.12
Test	10K	1.17	3.19

5. EXPERIMENTAL SETUP

Document collection. We obtained a list of URLs from the Web directory provided by the open directory project.[8] We downloaded about 2.2 million of these URLs. The obtained pages formed our collection, which is around 37GB in uncompressed HTML format.

Query log. We use the AOL query log [Pass et al. 2006], which contains 20 million queries issued by about 650,000 people during a period of 12 weeks. Queries are normalized by case-folding, sorting their terms in alphabetical order, and removing punctuation, stop-words, and term repetitions. We consider only the queries whose terms appear in the vocabulary of the collection. This guarantees that the selected queries are meaningful for the collection. A similar approach is adopted also by some other works [Ntoulas and Cho 2007].

Training and test query sets. We use the first six weeks of the query log (about 8M queries, 3.5M of which are unique) as the training set to populate the result cache. We assume a static result cache that contains the most frequent queries in the training set, up to a certain cache capacity. In this article, we experiment with four different cache capacities: 0.5M, 1.5M, 2.5M, and 3.5M queries. We select the test queries (10K queries) from the queries that immediately follow the training queries in submission order. The test queries are restricted to those that lead to a cache miss and match at least one result document in the collection.

In Table I, we provide some properties of the training and test queries. As the table shows, the properties of cache misses (test set) are different than those of cached queries (training set). Similar findings are also reported in Skobeltsyn et al. [2008]. We note that our test set is large enough to be able to obtain statistically significant results as we will show in Section 6.3. Similar query set sizes are used in other works [Altingovde et al. 2012; Skobeltsyn et al. 2008].

Indexing and query processing. Our document collection is indexed by the Zettair search engine[9] without applying stemming to the terms in the documents. For query processing, we use a custom text retrieval system that we developed.

Snippet generation. For training queries, we retrieve the top 10 documents along with the snippets. To generate the snippets, we use the algorithm proposed in Turpin et al. [2007]. In a nutshell, this algorithm divides the document into sentences and assigns scores to each of these sentences based on some heuristics, for example, based on whether the sentence belongs to the title or some heading, or whether it contains any of the query terms or not. Then, a weighted combination of these scores is computed for the final ranking of sentences. The snippet is constructed by selecting the

[8]Open directory project, available at `http://www.dmoz.org`.
[9]Zettair, available at `http://www.seg.rmit.edu.au/zettair/`.

highest-scoring three sentences. If a sentence is longer than 50 characters, we only extract the fragment around the query term(s) in the sentence. As typical in commercial Web search engines, the snippet size is limited to 150 characters.

Evaluation. The main goal of our techniques is to construct synthetic query results that are as similar as possible to those generated by processing queries with an index built on the collection. Hence, we assume that the top k result sets obtained using the backend index form a ground truth. The same approach is adopted by other works as well [Carmel et al. 2001]. We evaluate the test queries using the backend index and obtain their top 10 results. Then, we obtain the top 10 results produced by the proposed approaches and compute the precision at 10 (P@10) and Mean Average Precision (MAP) metrics with respect to the ground truth. Both P@10 and MAP values are generated by the trec_eval toolkit, the standard tool used by TREC for evaluating an ad hoc retrieval run. For both metrics, averages are computed over 10K test queries with the `-c` flag as in typical TREC evaluations.

6. EXPERIMENTS

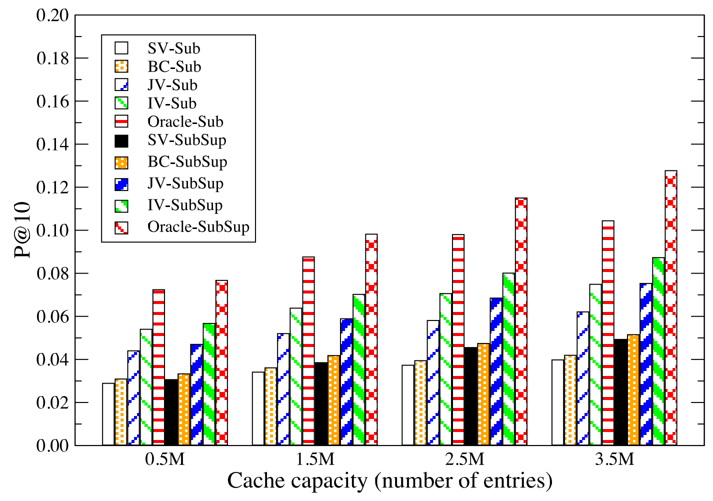
6.1. Performance Comparison

Figure 3 shows the P@10 and MAP values achieved by different aggregation-based approaches: simple voting (`SV`), Jaccard-weighted voting (`JV`), IDF-weighted voting (`IV`), and Borda count (`BC`). The suffixes `-Sub` and `-SubSup` refer to the subset and subset+superset query set selection approaches given in (1) and (3), respectively. We omit the results of the superset approach (2) because the results were significantly worse than the others. We also evaluate an oracle ranker (`Oracle`), which selects all relevant results from the aggregated rankings. For instance, when applying `Oracle-Sub` on a given query q^*, we obtain the union of all documents retrieved by the queries in $\boldsymbol{Q}_{q^*}^{\mathrm{sub}}$, rank all relevant documents before irrelevant documents, and return the top 10 documents. The oracle provides an upper bound on the relevance values that can be achieved by the proposed aggregation-based approaches.

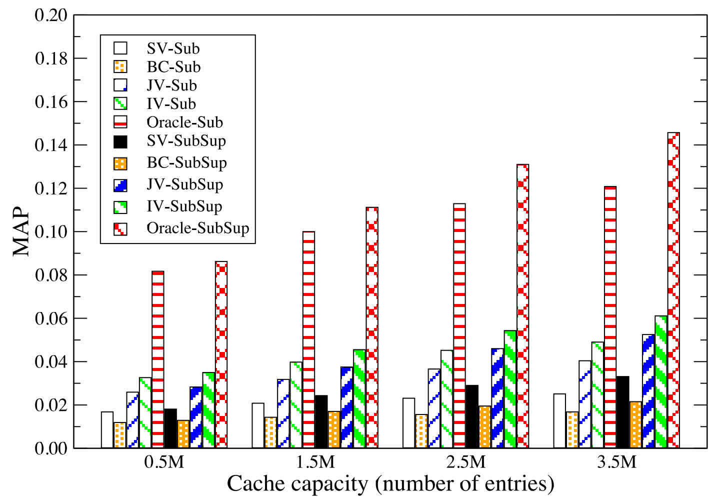
According to Figure 3(a), as expected, all approaches obtain better P@10 values as the cache capacity increases. In general, for all approaches, using both subsets and supersets performs better than using only the subsets. Among the competing approaches, the best performing is `IV-SubSup` with a precision of 0.087 at 3.5M cache entries. The upper bound given by `Oracle-SubSup` is 0.128. We observe a similar trend for the MAP values in Figure 3(b), except for the `BC` approach, which performs relatively poorly.

Figure 4 compares the winner of the aggregation-based approaches, that is, `IV-SubSup`, with the index-based approaches: query view (`QV`), snippet terms (`ST`), and hybrid (`QV+ST`). We observe that the index-based approaches perform considerably better than `IV-SubSup`, especially when the cache capacity is increased. Except for the smallest cache capacity of 0.5M entries, `QV` performs considerably better than `ST`. There is also significant performance improvement when both approaches are combined under `QV+ST`, which attains the highest relevance values.

Herein, we also conduct a user study in order to evaluate the user-perceived relevance of results that we obtained using the best-performing `QV+ST` strategy. A random sample of queries are selected from the 10K test query log and adult queries are manually filtered. Ten queries are assigned randomly to each of the 14 graduate students from the Computer Engineering Department of Bilkent University. We merged and shuffled the top 10 results obtained by using the backend index and those obtained via the `QV+ST` strategy. The participants are informed about the shuffling of results. Each participant evaluated the obtained 20 results (at most) for 10 queries and decided



(a) P@10



(b) MAP

Fig. 3. The search result quality of different rank aggregation approaches.

whether each result is relevant or not relevant to the query. According to the results of our user study, the average number of relevant results is 3.2 in case of query evaluation using the backend index and 2.5 for our cache-based query processing strategy. This shows that the user-perceived relevance achieved by the QV+ST strategy is not considerably worse than that attained by processing queries using the backend index.

6.2. Performance with Varying Parameters

In this section, we analyze the impact of different parameters on the performance. All experiments are conducted assuming a cache capacity of 3.5M entries.

Query length. Figure 5 shows the performance variation as the query length increases. It is interesting to observe that the IV-SubSup and ST approaches suffer

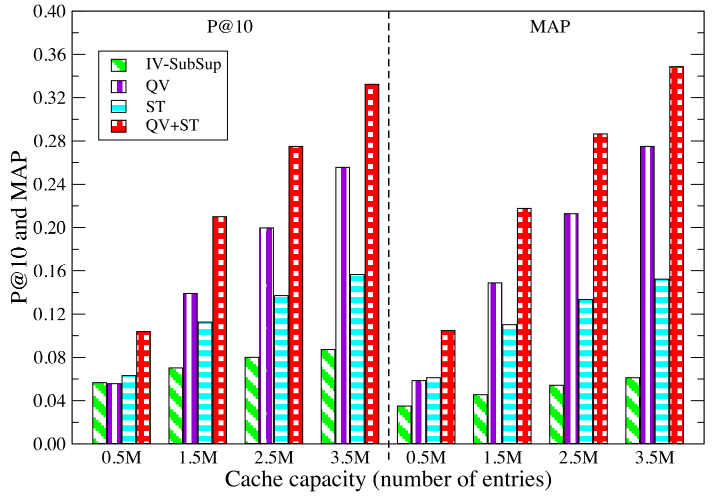


Fig. 4. P@10 and MAP values achieved by the best aggregation-based approach (IV-SubSup) and the three index-based approaches (QV, ST, and QV+ST).

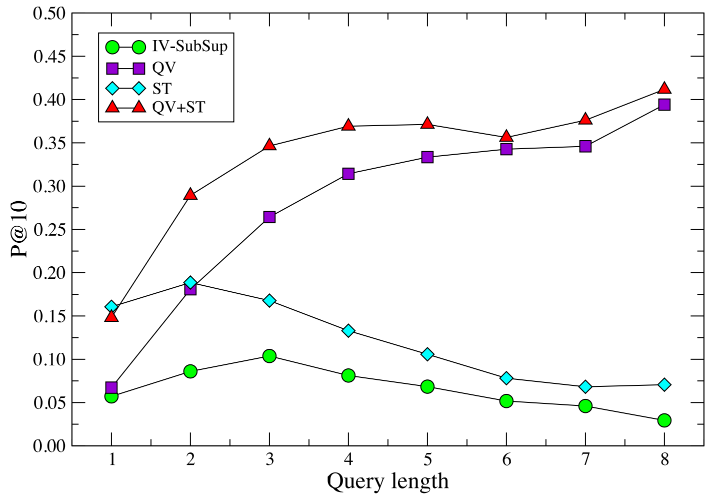


Fig. 5. P@10 with varying query length.

increasing query length, whereas QV and QV+ST seem to perform better on longer queries. The optimal query length for IV-SubSup is three. At this length, the likelihood of finding queries that satisfy the subset or superset relations is higher. This is because, for very short queries (e.g., one or two terms), it is not possible to find many subsets while, for very long queries, it is harder to find supersets. The inferior performance of ST with increasing query length is relatively easier to explain. As the query length increases, it is more likely that the query includes a term that is unlikely to appear in short snippets. The reason QV performs relatively worse on shorter queries may be because shorter queries usually include more common terms, which also appear in many query views. This would yield a large candidate set to rank. As discussed in Section 4.2, the snippet index captures term frequencies more accurately for such terms, whereas QV has less information in ranking large candidate sets.

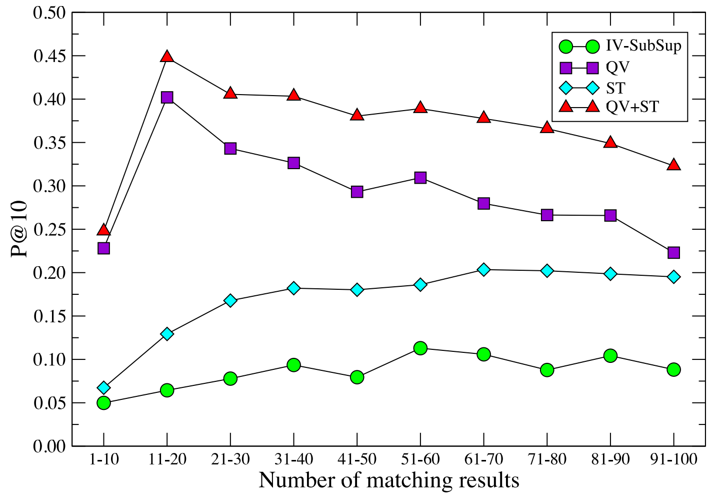


Fig. 6. P@10 with varying number of matching results.

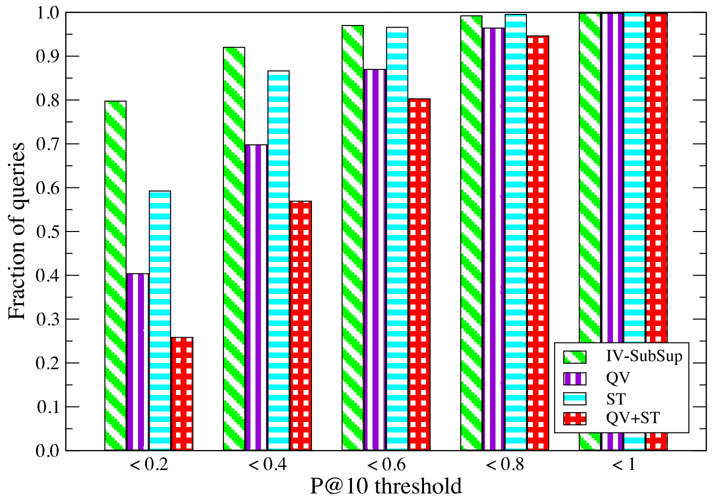


Fig. 7. The fraction of queries whose P@10 value is less than a certain threshold value.

Result set size. Figure 6 shows the performance as the number of results matching the query increases. In general, a high number of matching results imply a general query, whereas low numbers imply a more specific query. ST seems to perform better as there are more results matching the query. This finding is consistent with that in Figure 5, as short queries are more likely to match many documents. QV achieves its peak performance around 10 results, after which the performance begins to degrade as the number of matching results increases.

Distribution of P@10. Figure 7 shows the fraction of queries whose P@10 value is less than a certain threshold value. As in the previous experiments, QT+ST has a relatively better performance than the other approaches. In particular, this approach can return at least two relevant results for more than three-fourth of the queries.

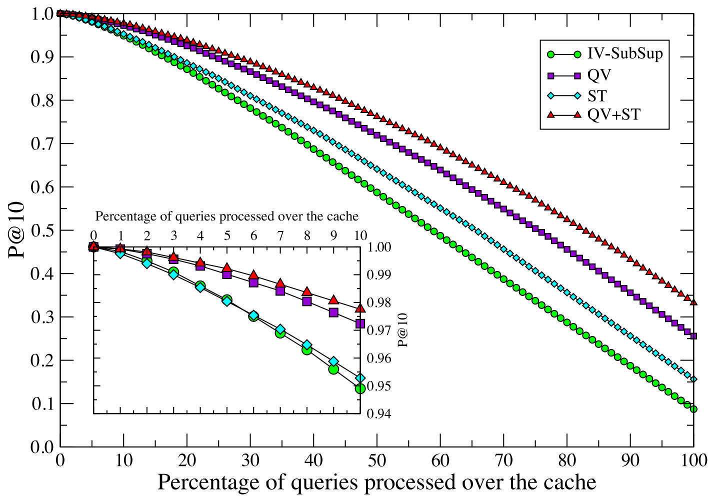


Fig. 8. P@10 values as more queries are processed by the cache-based strategy.

Query degradation. We now assume a scenario where the search service is overloaded by queries and hence a certain fraction of queries have to be processed by our cache-based processing strategy. To select such queries, we use an oracle that prioritizes queries for which our approaches produce the best answers. This way, we obtain an upper bound on the P@10 values that can be achieved by any approach. In Figure 8, the x axis shows the fraction of queries that are processed by the cache-based query processing strategy. The remaining queries are evaluated using the backend index and hence they have a P@10 of one. The y axis shows the P@10 values that are computed over all queries. The general trend of P@10 values is displayed in the outer plot. For better visibility, the P@10 values at lower percentages, which are more representative values for our use case scenarios, are displayed in the inner plot.

The previous experiment assumes an oracle strategy that has a perfect knowledge of the result quality. For a more practical analysis, we also build a machine learning model, based on gradient boosted decision trees [Friedman 2000], to predict the result quality that can be attained by the cache-based query processing strategy for a given query. Our model uses various features extracted from the query string and other sources (e.g., query length, query frequency, IDF values, and number of matching results in the cache). The output of the model is a P@10 value predicted for a given query. The model is trained using the first 9K queries and tested over the remaining 1K queries.

We repeat the previous experiment using the learned model, that is, we simply prioritize queries in decreasing order of the P@10 values predicted by the model. The results are reported in Figure 9 for the best-performing strategy `QV+ST`, assuming oracle, ML-based, and random query prioritization. According to the inner plot in the figure, even when 10% of the queries are degraded, the average precision loss of the system is not very high, thus making the cache-based query processing approaches suitable for query result degradation under heavy query workloads. We also note that there is still some performance gap between the ML-based query prioritization strategy and the oracle prioritization strategy. This gap may be difficult to close as the problem we have at hand is somewhat similar to the query difficulty prediction problem, which is not trivial at all [Carmel et al. 2006].

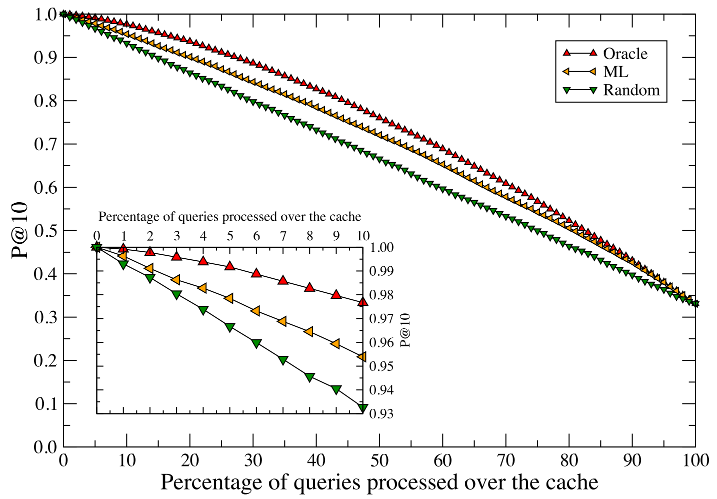


Fig. 9. P@10 values as the percentage of queries processed by the `QV+ST` strategy increases (assuming oracle, ML-based, and random query prioritization).

6.3. Statistical Significance

We investigate the significance (at the 0.05 confidence level) of performance differences among different approaches by a one-way ANOVA analysis, followed by Tukey's post hoc test. In particular, for each cache size configuration and for all results in Figures 3 and 4, we compare the output of the 10K test queries for each strategy using this statistical analysis method. Herein, we only discuss the findings for P@10 results, for which there seems to be more cases with close score values. Significance tests on MAP results, which are not discussed here, also verify the conclusions we draw about the strategies.

We find that, for rank aggregation approaches reported in Figure 3, there are only a few cases with insignificant differences. For instance, for a cache size of 3.5M queries, only three cases out of 45 comparisons[10] yield insignificant P@10 results: the differences between pairs (`BC-Sub`, `SV-Sub`), (`BC-SubSup`, `SV-SubSup`), and (`IV-Sub`, `JV-SubSup`) are not statistically significant at the 0.05 confidence level. Note that this confirms our conclusion that `BC` and `SV` algorithms perform similarly (for respective cases of `Sub` and `SubSup`) while `JV` is significantly better than the former two, and `IV` is significantly better than all three strategies. Furthermore, the difference between `Oracle` and `IV` is also significant, implying that there is still some room for further improvement.

For the results plotted in Figure 4, for each cache configuration, we observe that all differences among the competing strategies are statistically significant. The only exception is P@10 for the (`IV-SubSup`, `QV`) case with the smallest cache size (0.5M) as the performance of these strategies are found not statistically significantly different at the 0.05 level.

6.4. Web-Scale Experiments

The success of our cache-based query processing strategy depends on the likelihood of finding the results of test queries in the cache. For an accurate performance evaluation, there are two issues that must be considered. First, it is important to appropriately choose the ratio between the test and training set sizes. Second, the

[10]There can be 45 pairwise comparisons among 10 strategies.

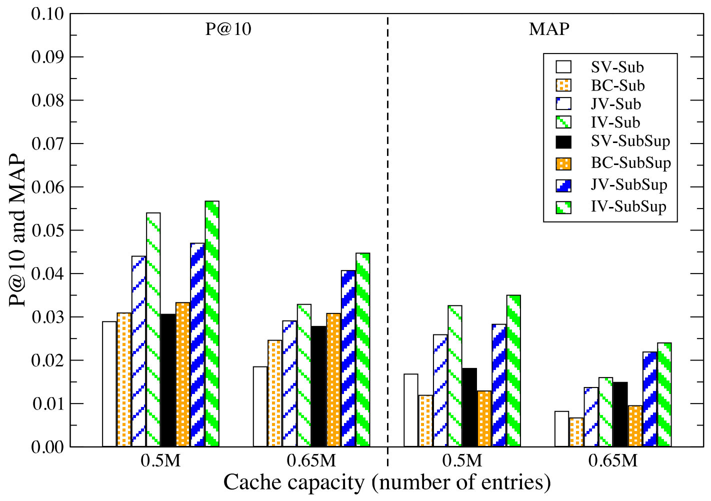


Fig. 10. Results for Web-scale experiments.

relative sizes of the document collection and the cache capacity should be carefully determined.

Regarding the first issue, we assume that the result cache is filled by a large number of training queries while the backend search system is available. Test queries are issued during a certain period of unavailability. Since all scenarios in Section 2 assume a short period of physical unavailability or a small number of queries with no answers, we anticipate that the test set size should be a tiny fraction of the training set size. This justifies our choice of using a small test set size in the experiments.

Regarding the second issue, clearly, larger cache capacities yield better results (see Figures 3 and 4). However, it is not obvious how the ratio between the collection size and cache capacity should be chosen. For instance, if we had a much larger collection, could we obtain the same performance by using a cache size in practical limits, or would we need an unrealistically large cache? In this section, we make an attempt to answer this question by using the largest possible collection, the Web. To this end, we sample around 660K unique queries (in submission time order) from the AOL query log and obtained their top 10 answers using the Yahoo! Web search API. Note that, given the usage restrictions of the API (5,000 queries per day), this is a fairly large number of queries. Next, we reserve the first 650K queries as the training set (i.e., cached queries) and the remaining 10K queries as the test set, that is, cache misses. Using this setup, we repeat the experiments conducted for the aggregation-based approaches.

In Figure 10, we compare the performance of two different caches containing the results obtained from (i) our collection using 500K training queries and (ii) the Web using 650K training queries. The figure shows that P@10 and MAP values are comparable for both caches while the former cache leads to slightly better performance. This is an encouraging result as it shows that, even for very large collections, our techniques can yield competitive performance at reasonably low cache capacities. We anticipate that, to obtain the results given in Figures 3 and 4, the cache size of a Web search engine has to be much larger than what we used in our experiments. Our findings imply that the required cache size would be in practical limits. Indeed, since search engines are known to cache hundreds of millions of queries, we believe that the performance figures in Section 6.1 or even better results can be obtained in large-scale search engines.

7. RELATED WORK

Result caching. Markatos [2001] conducted the first study on result caching in search engines. He investigated a number of alternatives for both static and dynamic result caching. A number of works extended this work by taking into account combinations of a result cache with a posting list cache [Baeza-Yates et al. 2008; Saraiva et al. 2001] or other types of caches [Li et al. 2007; Long and Suel 2005; Marin et al. 2010; Ozcan et al. 2012]. Prefetching of search results is discussed in Lempel and Moran [2003]. Result caching techniques are further improved in some recent works [Altingovde et al. 2011; Baeza-Yates et al. 2007; Fagni et al. 2006; Gan and Suel 2009; Ozcan et al. 2008, 2011]. In Alici et al. [2011, 2012], Blanco et al. [2010], Cambazoglu et al. [2010a], and Jonassen et al. [2012], freshness is shown to be currently the main issue in result caching in commercial search engines and various methods are proposed for this problem.

Similarity caching. A number of recent works propose to compute approximate answers for similarity search queries in metric spaces by using previously cached result objects [Chirerichetti et al. 2009; Falchi et al. 2008; Houle et al. 2010; Pandey et al. 2009]. In these works, each result object is stored in the cache together with its features that are used in the similarity computations and a user query is an object in the same feature space. Consequently, the similarity between a given query object and a cached result object can be accurately computed. In our context, however, the user queries are keyword based and the result cache only stores some basic information (e.g., URL, title, and snippets) about the documents. Hence, it is not possible to accurately compute the relevance of a query to the cached result entries. For this reason, the cache-based query processing techniques proposed in the context of similarity search are not applicable to our problem setting.

Semantic caching. In Ferrarotti et al. [2009], a location cache, which stores the ids of the nodes that generate a query result, is used as a semantic cache [Chidlovskii and Borghoff 2000]. For an unseen user query, the search nodes that produce answers to queries that are similar to the user query are determined, and the user query is processed only on those nodes. In this approach, the cached information is used to reduce the processing overhead of the backend, but not for generating a top k result set directly from the cache as we do.

In database literature, there is a wealth of works on semantic caching [Adali et al. 1996; Chen and Roussopoulos 1994; Dar et al. 1996; Keller and Basu 1994; Miranker et al. 2002; Qian 1996]. Most approaches exploit the fact that the answer set of a query obtained from a traditional database system is always complete. Hence, cache-based result generation can reduce the query response time as well as the data transfer costs in database systems. However, as we discussed in Section 1, the requirements of a search engine are quite different: queries are usually simple conjunctions of keywords and cached query results are only a tiny fraction (e.g., top 10) of the entire result set for a given query. Hence, the techniques from the semantic caching literature in the context of databases are not directly applicable in the context of search engines. Nevertheless, in Cheong et al. [2001], ideas similar to the aforementioned works are employed for cache-based processing of keyword queries in mediator systems. The result cache and queries are represented as a variation of disjunctive formal form to allow decomposition of a query into a “hit” subquery and a “miss” subquery whose results are retrieved from the backend. Since the techniques in Cheong et al. [2001] do not take into account the fact that only the top results are cached in a large-scale search engine, they are not applicable to our setting.

To best of our knowledge, there is no prior work on the use of the result caches for generating partial query results when the backend of a search engine is physically or logically unavailable. However, we are aware of a recent work [Papadakis 2010] that proposes evaluating queries using the result entries in the cache for efficiency purposes. The strategy in Papadakis [2010] works as follows. Given a query, an exact set cover is tried to be found for the query using cached queries. If an exact set cover cannot be found, the query is processed at the backend. Otherwise, the top k rankings for the queries in the exact set cover are used to generate the results of the query.

The strategies proposed in our work differ from the approach of Papadakis [2010] mainly in three ways. First, our strategies can generate results for a larger fraction of queries, whereas Papadakis [2010] requires finding an exact set cover for the query within the cache. For most tail queries, which are the target of our work, this may not be possible. Therefore, the approach in Papadakis [2010] is not well suited to our unavailability context as it requires using the backend for such queries. Second, the adopted result generation approach [Ntoulas and Cho 2007] is not suitable for aggregation of very short result lists (e.g., 10 results in our case). Even with statically pruned indexes that have much larger posting lists, the performance is limited. Finally, the work in Papadakis [2010] is only applicable to backend ranking functions that are additive and decomposable (e.g., there is no way we can apply their technique to the rankings we obtained through the Yahoo! search API). The techniques we propose in the article are more general, as we have no assumption about the nature of the backend ranking function and do not rely on the scores computed by the backend.

Rank aggregation. The rank aggregation problem appears in a variety of applications, including metasearch engines [Aslam and Montague 2001], long query reduction [Kumaran and Carvalho 2009], and word association problem [Dwork et al. 2001]. For the partial rank aggregation problem, there are elegant solutions based on Markov chains or minimum weight maximum bipartite graph matching [Dwork et al. 2001]. In this work, due to its simplicity and efficiency, we preferred using Borda's algorithm [Borda 1781]. Moreover, the index-based approaches can clearly beat even the oracle algorithm `Oracle-SubSup`, which forms an upper bound on the performance of any aggregation-based technique. For a comparison of the rank aggregation algorithms, the interested reader may refer to Schalekamp and van Zuylen [2009].

8. DISCUSSION

Score-based aggregation. The techniques we used for rank aggregation assume that no information regarding the relevance of the documents to the queries is available in the cache. In practice, however, the relevance scores computed for the documents can be stored in the cache. If the scores are compatible, they can be used in rank aggregation to generate better rankings. Unfortunately, most scoring functions used in practice do not yield scores that are comparable across different queries. Hence, even if the score information is available, it may not be used to improve the quality of aggregate ranking. In this work, to be as general as possible, we assumed that the relevance scores are not available.

Global statistics. Popular ranking functions (e.g., BM25) use two types of information: IDF values of the terms in the index and document lengths (i.e., the number of terms in the document). We assume that, during the unavailability period, such statistics are not available to our techniques. Hence, we try to approximate these values in the index creation phase, using the statistics obtained from the text in the cache. In other words, we do not assume the existence of any global knowledge about collection statistics and rely only on the content of the result cache. In a separate experiment, we evaluated our index-based approaches using the original IDF scores and document lengths. The observed results were not significantly different. This indicates that our approximations are accurate enough and eliminates the need to maintain global statistics in the cache.

Computational cost. In terms of the computational cost, aggregation-based approaches are more appealing. For instance, aggregating a subset of results requires generating and locating in the cache all subsets of a query, which is a relatively inexpensive operation as the cost of locating a query result in the cache is around 60–70μs [Baeza-Yates et al. 2008; Fagni et al. 2006]. However, as the experimental results demonstrate, these aggregation techniques are inferior to index-based strategies in terms of result relevance. In index-based approaches, on the other hand, query processing time may be an issue, especially for large caches.

Storage overhead. Unlike the aggregation-based approaches, the index-based approaches require some storage. In general, a query-view-based index is more compact and yet more effective than a snippet-based index. The combined index yields substantially better results, but is larger. However, we can still argue that the average document length in such an index would be much smaller in comparison to the index constructed on the original collection. For instance, the average document length computed during the creation of the largest combined index (for the 3.5M-entry cache) is less than one-fifth of the original collection size. Hence, it may be feasible to create an index for hundreds of millions of queries and store it on many commodity computers that could store the result cache in a distributed fashion. The index may be constructed periodically or when the server is idle.

Sophisticated ranking algorithms. In order to compute the relevance of a user query to the documents in the collection, Web search engines rely on sophisticated learning algorithms that incorporate many features [Cambazoglu et al. 2010b]. The extracted features can be query dependent or query independent. Query-dependent features include simple statistical features such as BM25 or term proximity features. Query-independent features include those that solely depend on the document (e.g., link analysis metrics, click popularity) or the characteristics of the user (e.g., location of the user). In our problem setting, one possibility is to store the query-independent features of each result document in the cache entry, together with its descriptive information. During the rank aggregation phase of our techniques, these features can be used to boost the rank of more important documents or for result regionalization purposes. The incorporation of query-independent features such as PageRank into the ranking can be expected to improve the relative effectiveness of our cache-based query processing techniques because this will boost the ranks of popular documents, which are more likely to be present in the result cache.

Result freshness. Search engines typically bound the staleness of result cache entries by associating each cache entry with a Time-To-Live (TTL) value, that is, a result entry is considered to be stale after its TTL expires [Alici et al. 2012; Cambazoglu et al. 2010a]. Hence, at any point in time, a certain fraction of the entries in the cache are in an expired state. Our techniques may or may not use such stale entries when generating the top k results for previously unseen queries. If such stale cache entries are used in result generation, we have a high coverage of results with poor result freshness. Alternatively, if they are not used, we have low coverage, but the results are more fresh. Unfortunately, the trade-off between relevance and freshness in terms of user satisfaction is not well-explored in literature. Therefore, any further discussion on the topic will not go beyond speculation.

9. CONCLUSIONS

In this article, we devised query processing strategies that use the result entries found in the result cache of a search engine to answer previously unseen user queries. We described several use case scenarios to motivate these strategies, where the main idea is to generate partial answers whenever accessing the search backend is impossible (physical unavailability) or infeasible (logical unavailability). The proposed strategies are grouped under two headings: aggregation-based and index-based strategies. In the former type of strategies, the search results for a given query are computed by combining the results of similar queries in the cache via rank aggregation techniques. In the latter type of strategies, an index is built over the textual content in the cache (more specifically, over the terms in the cached queries or the terms in the snippets) and the search results are generated by processing new queries over this index.

In general, the aggregation-based strategies yielded relatively poor performance. The main weakness of the aggregation-based strategies lies in the high number of unique entries in the aggregated search results, that is, it is not very common to have a document that repeats many times in the aggregated results. Therefore, when aggregating the search results, the ranks of the results play a more important role than how many times they appear in the aggregated rankings (a result typically appears in only one ranking). We note that the sophisticated aggregation strategies cannot help much here as there is very limited opportunity for accumulating multiple “votes” for a result entry. The index-based strategies, on the other hand, yielded fairly good results. Especially the hybrid strategy, which builds the index on both the terms of the cached queries and the snippet terms, is found promising. The experiments demonstrated that this strategy can achieve P@10 values as high as 0.35 and could retrieve at least two relevant results for more than 75% of the queries.

ACKNOWLEDGMENTS

We thank Ender Demirkaya for his assistance in implementing some of the methods discussed in this article. We would also like to thank to anonymous referees, whose comments greatly improved the quality of the work.

REFERENCES

ADALI, S., CANDAN, K. S., PAPAKONSTANTINOU, Y., AND SUBRAHMANIAN, V. S. 1996. Query caching and optimization in distributed mediator systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 137–148.

ALICI, S., ALTINGOVDE, I. S., OZCAN, R., CAMBAZOGLU, B. B., AND ULUSOY, O. 2011. Timestamp-Based result cache invalidation for web search engines. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 973–982.

ALICI, S., ALTINGOVDE, I. S., OZCAN, R., CAMBAZOGLU, B. B., AND ULUSOY, O. 2012. Adaptive time-to-live strategies for query result caching in web search engines. In *Proceedings of the 34th European Conference on Advances in Information Retrieval*. 401–412.

ALTINGOVDE, I. S., OZCAN, R., CAMBAZOGLU, B. B., AND ULUSOY, O. 2011. Second chance: A hybrid approach for dynamic result caching in search engines. In *Proceedings of the 33rd European Conference on Advances in Information Retrieval*. 510–516.

ALTINGOVDE, I. S., OZCAN, R., AND ULUSOY, O. 2012. Static index pruning in web search engines: Combining term and document popularities with query views. *ACM Trans. Inf. Syst. 30*, 1, 2:1–2:28.

ASLAM, J. A. AND MONTAGUE, M. 2001. Models for metasearch. In *Proceedings of the 24th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 276–284.

BAEZA-YATES, R., JUNQUEIRA, F. P., PLACHOURAS, V., AND WITSCHEL, H. F. 2007. Admission policies for caches of search engine results. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval*. 74–85.

BAEZA-YATES, R. A., GIONIS, A., JUNQUEIRA, F., MURDOCK, V., PLACHOURAS, V., AND SILVESTRI, F. 2008. Design trade-offs for search engine caching. *ACM Trans. Web 2*, 4, 20:1–20:28.

BARROSO, L. A., D., J., AND HOLZLE, U. 2003. Web search for a planet: The Google cluster architecture. *IEEE Micro 23*, 2, 22–28.

BLANCO, R., BORTNIKOV, E., JUNQUEIRA, F., LEMPEL, R., TELLOLI, L., AND ZARAGOZA, H. 2010. Caching search engine results over incremental indices. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 82–89.

BORDA, J. C. 1781. Memorie sur les elections au scrutin. In *Histoire de l'Academic Royale des Sciences*.

CAMBAZOGLU, B. B., JUNQUEIRA, F. P., PLACHOURAS, V., BANACHOWSKI, S., CUI, B., LIM, S., AND BRIDGE, B. 2010a. A refreshing perspective of search engine caching. In *Proceedings of the 19th International Conference on World Wide Web*. 181–190.

CAMBAZOGLU, B. B., ZARAGOZA, H., CHAPELLE, O., CHEN, J., LIAO, C., ZHENG, Z., AND DEGENHARDT, J. 2010b. Early exit optimizations for additive machine learned ranking systems. In *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining*. 411–420.

CARMEL, D., COHEN, D., FAGIN, R., FARCHI, E., HERSCOVICI, M., MAAREK, Y., AND SOFFER, A. 2001. Static index pruning for information retrieval systems. In *Proceedings of the 24th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 43–50.

CARMEL, D., YOM-TOV, E., DARLOW, A., AND PELLEG, D. 2006. What makes a query difficult? In *Proceedings of the 29th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 390–397.

CHEN, C.-M. AND ROUSSOPOULOS, N. 1994. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Proceedings of the 4th International Conference on Extending Database Technology*. 323–336.

CHEONG, J.-H., GOO LEE, S., AND CHUN, J. 2001. A method for processing boolean queries using a result cache. In *Proceedings of the 12th International Conference on Database and Expert Systems Applications*. 974–983.

CHIDLOVSKII, B. AND BORGHOFF, U. 2000. Semantic caching of web queries. *VLDB J. 9*, 1, 2–17.

CHIRERICHETTI, F., KUMAR, R., AND VASSILVITSKII, S. 2009. Similarity caching. In *Proceedings of the 28th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. 127–136.

DAR, S., FRANKLIN, M. J., JÓNSSON, B. T., SRIVASTAVA, D., AND TAN, M. 1996. Semantic data caching and replacement. In *Proceedings of the 22nd International Conference on Very Large Data Bases*. 330–341.

DWORK, C., KUMAR, R., NAOR, M., AND SIVAKUMAR, D. 2001. Rank aggregation methods for the Web. In *Proceedings of the 10th International Conference on World Wide Web*. 613–622.

FAGNI, T., PEREGO, R., SILVESTRI, F., AND ORLANDO, S. 2006. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst. 24*, 1, 51–78.

FALCHI, F., LUCCHESE, C., ORLANDO, S., PEREGO, R., AND RABITTI, F. 2008. A metric cache for similarity search. In *Proceedings of the 6th Workshop on Large-Scale Distributed Systems for Information Retrieval*. 43–50.

FERRAROTTI, F., MARIN, M., AND MENDOZA, M. 2009. A last-resort semantic cache for web queries. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*. 310–321.

FRIEDMAN, J. H. 2000. Greedy function approximation: A gradient boosting machine. *Ann. Statist. 29*, 1189–1232.

GAN, Q. AND SUEL, T. 2009. Improved techniques for result caching in web search engines. In *Proceedings of the 18th International Conference on World Wide Web*. 431–440.

HOULE, M. E., ORIA, V., AND QASIM, U. 2010. Active caching for similarity queries based on shared-neighbor information. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*. 669–678.

JONASSEN, S., CAMBAZOGLU, B. B., AND SILVESTRI, F. 2012. Prefetching query results and its impact on search engines. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*.

KELLER, A. M. AND BASU, J. 1994. A predicate-based caching scheme for client-server database architectures. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*. 229–238.

KUMARAN, G. AND CARVALHO, V. R. 2009. Reducing long queries using query quality predictors. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 564–571.

LEMPEL, R. AND MORAN, S. 2003. Predictive caching and prefetching of query results in search engines. In *Proceedings of the 12th International Conference on World Wide Web*. 19–28.

LI, H., LEE, W.-C., SIVASUBRAMANIAM, A., AND GILES, C. L. 2007. A hybrid cache and prefetch mechanism for scientific literature search engines. In *Proceedings of the 7th International Conference on Web Engineering*. 121–136.

LONG, X. AND SUEL, T. 2005. Three-Level caching for efficient query processing in large web search engines. In *Proceedings of the 14th International Conference on World Wide Web*. 257–266.

MARIN, M., GIL-COSTA, V., AND GOMEZ-PANTOJA, C. 2010. New caching techniques for web search engines. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. 215–226.

MARKATOS, E. 2001. On caching search engine query results. *Comput. Comm. 24*, 2, 137–143.

MIRANKER, D. P., TAYLOR, M. C., AND PADMANABAN, A. 2002. A tractable query cache by approximation. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation*. 140–151.

NTOULAS, A. AND CHO, J. 2007. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 191–198.

OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. 2003. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*. 1–16.

OZCAN, R., ALTINGOVDE, I. S., AND ULUSOY, O. 2008. Static query result caching revisited. In *Proceedings of the 17th International Conference on World Wide Web*. 1169–1170.

OZCAN, R., ALTINGOVDE, I. S., AND ULUSOY, O. 2011. Cost-Aware strategies for query result caching in web search engines. *ACM Trans. Web 5*, 2, 9:1–9:25.

OZCAN, R., ALTINGOVDE, I. S., CAMBAZOGLU, B. B., JUNQUEIRA, F. P., AND ÖZGÜR ULUSOY. 2012. A five-level static cache architecture for web search engines. *Inf. Process. Manag. 48*, 5, 828–840.

PANDEY, S., BRODER, A., CHIERICHETTI, F., JOSIFOVSKI, V., KUMAR, R., AND VASSILVITSKII, S. 2009. Nearest-Neighbor caching for content-match applications. In *Proceedings of the 18th International Conference on World Wide Web*. 441–450.

PAPADAKIS, M. 2010. Set cover-based results caching for best match retrieval models. M.S. thesis, University of Crete.

PASS, G., CHOWDHURY, A., AND TORGESON, C. 2006. A picture of search. In *Proceedings of the 1st International Conference on Scalable Information Systems*.

POBLETE, B. AND BAEZA-YATES, R. 2008. Query-Sets: Using implicit feedback and query patterns to organize web documents. In *Proceedings of the 17th International Conference on World Wide Web*. 41–50.

PUPPIN, D. AND SILVESTRI, F. 2006. The query-vector document model. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*. 880–881.

QIAN, X. 1996. Query folding. In *Proceedings of the 12th International Conference on Data Engineering*. 48–55.

SARAIVA, C. P., DE MOURA, E. S., ZIVIANI, N., MEIRA, W., FONSECA, R., AND RIBEIRO-NETO, B. 2001. Rank-Preserving two-level caching for scalable search engines. In *Proceedings of the 24th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 51–58.

SCHALEKAMP, F. AND VAN ZUYLEN, A. 2009. Rank aggregation: Together we're strong. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments*. 38–51.

SKOBELTSYN, G., JUNQUEIRA, F. P., PLACHOURAS, V., AND BAEZA-YATES, R. 2008. ResIn: A combination of results caching and index pruning for high-performance web search engines. In *Proceedings of the 31st International ACM SIGIR Conference on Research and Development in Information Retrieval*. 131–138.

TURPIN, A., TSEGAY, Y., HAWKING, D., AND WILLIAMS, H. E. 2007. Fast generation of result snippets in web search. In *Proceedings of the 30th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 127–134.

ZIMMER, C., BEDATHUR, S., AND WEIKUM, G. 2008. Flood little, cache more: Effective result-reuse in P2P IR systems. In *Proceedings of the 13th International Conference on Database Systems for Advanced Applications*. 235–250.

ZOBEL, J. AND MOFFAT, A. 2006. Inverted files for text search engines. *ACM Comput. Surv. 38*, 2, 1–56.

Received June 2011; revised June 2012; accepted June 2012