ResIn: A Combination of Results Caching and Index Pruning for High-performance Web Search Engines

Gleb Skobeltsyn[†], Flavio P. Junqueira[‡], Vassilis Plachouras[‡], Ricardo Baeza-Yates[‡]

[†]Ecole Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland

[‡]Yahoo! Research
Barcelona, Spain

gleb.skobeltsyn@epfl.ch, fpj@yahoo-inc.com, vassilis@yahoo-inc.com, rbaeza@acm.org

ABSTRACT

Results caching is an efficient technique for reducing the query processing load, hence it is commonly used in real search engines. This technique, however, bounds the maximum hit rate due to the large fraction of singleton queries, which is an important limitation. In this paper we propose *ResIn* - an architecture that uses a combination of *res*ults caching and *in*dex pruning to overcome this limitation.

We argue that results caching is an inexpensive and efficient way to reduce the query processing load and show that it is cheaper to implement compared to a pruned index. At the same time, we show that index pruning performance is fundamentally affected by the changes in the query traffic that the results cache induces. We experiment with real query logs and a large document collection, and show that the combination of both techniques enables efficient reduction of the query processing costs and thus is practical to use in Web search engines.

Categories and Subject Descriptors: H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval – *Search Process*; H.3.4 [Information Storage and Retrieval]: Systems and Software;

General Terms: Algorithms, Experimentation

Keywords: Web search, Caching, Pruning, Query logs

1. INTRODUCTION

Web search engines rely upon large complex systems to deliver query results to users. Such systems are large because they use thousands of servers, interconnected through different networks, and often spanning multiple data centers. Servers in these systems are often grouped according to some functionality (*e.g.*, front-end servers, back-end servers, brokers), and requiring that the groups of servers interact increases the amount of time and resources needed to process each query. It is therefore crucial for high-performance Web search engines to design mechanisms that enable a reduction of the amount of time and computational resources involved in query processing to support high query rates and ensure low latencies.

SIGIR'08, July 20–24, 2008, Singapore.

Typically, search engines use caching to store previously computed query results, or to speed up the access to posting lists of popular terms [2]. Results caching is an attractive technique because there are efficient implementations, it enables good hit rates, and it can process queries in constant time. Moreover, as the query results are available after a query is processed by the index, having a results cache is simply a matter of adding more memory to the system to hold such results temporarily.

One important issue with caches of query results is that their hit rates are bounded. Due to the large fraction of infrequent and singleton queries, even very large caches cannot achieve hit rates beyond $50-70\%$, independent of the cache size. To overcome this limitation, a system can make use of posting list caching or/and employ a pruned version of the index, which is typically much smaller than the full index and therefore requires fewer resources to be implemented. Without affecting the quality of query results, such a static pruned index is capable of processing a certain fraction of queries thus further decreasing the query rate that reaches the main index [18].

In this paper, we describe an architecture, *ResIn*, that uses both a results cache and a static pruned index. We show that there are several benefits of using them together, including higher hit rates and reduced resource utilization. Although index pruning has been studied in the literature, to our knowledge, we are the first to present results on the impact of results caching on index pruning in an architecture that is highly relevant for practical Web search systems. In such an architecture, the stream of queries that has to be processed by the pruned index differs significantly from the the original query stream because many queries are filtered out by the results cache. This difference affects the performance of index pruning and the way to optimize it.

In this paper we consider several index pruning techniques such as *term pruning* - a complete removal of posting lists of certain terms, *document pruning* - removing certain portions of the posting lists, and the combination of both.

Apart from introducing the *ResIn* architecture, we make the following contributions:

- We show that results caching has important advantages when compared to index pruning. In particular, results caching guarantees high cache hit rates with a constant cache capacity independently of the size of the document collection;
- We compare the properties of the original query stream and the query stream after a results cache, and show how the differences affect the applicability of index pruning;
- We compare the efficiency of various static index pruning techniques when a pruned index is used separately or in

combination with a results cache;
- We propose a different method of combining term and document pruning that outperforms the one of Ntoulas and Cho [18].

2. RELATED WORK

A number of solutions that aim at reducing query processing costs for Web search engines have been proposed in the literature, including various caching and index pruning techniques.

Eviction policies that maximize the hit rate for a given cache capacity have been studied in a number of different domains, including search engine architectures. In the context of **results caching**, Fagni *et al.* [15] introduce Static Dynamic Cache (SDC). SDC achieves higher cache-hit rates than the baseline LRU by devoting a fraction of the storage for a static cache containing frequent queries precomputed in advance. Further improvements are considered in [3]. However, when the cache capacity increases, the hit rate approaches its upper bound determined by the fraction of unique queries in the query log.

The authors of [20] explore users' repeat search behavior and categorize different reformulations of similar queries in the query logs. Baeza-Yates *et al.* [2] investigate the impact of results caching and static caching of posting lists in the context of Web search engine architecture. The impact of compression on caching efficiency is addressed in [22]. Finally, Long and Suel [17] introduce a 3-level caching architecture that includes on-disk caching of the posting lists for popular term combinations.

Index pruning reduces significantly the fraction of the index needed for query processing. With *static pruning*, the system generates a pruned index beforehand, whereas *dynamic pruning* proceeds on a per-query basis saving resources and reducing latency by dynamically skipping non-relevant parts of the index.

In the context of this paper, we call *term pruning* a complete removal of posting lists of certain terms (*e.g.*, stop words removal or the approach described in [4]), whereas *document pruning* refers to ignoring only certain portions of the posting lists (*e.g.*, [10]).

In particular, some document pruning techniques were inspired by the algorithms of Fagin *et al.* [14], known as Threshold algorithms. The intuition behind these algorithms is that there is no need to scan complete inverted lists if only a top-k fraction of the intersection is requested. Instead, the lists are sorted according to some score-dependent value and it is likely that the top-k query results can be found in the top-portions of the posting lists.

In the context of text search engines this idea was first exploited by Carmel *et al.* [10]. They introduced static document pruning that is able to reduce the size of the index by up to 50-70% but at the price of a certain precision loss. The authors of [13] extend Carmel's lossy pruning by taking into account co-occurrences of words that appear close to each other in documents. Long and Suel also employ a similar pruning technique [16].

Anh and Moffat propose impact-ordered inverted lists and a lossy dynamic pruning scheme tailored for such an index organization [1]. Tsegay *et al.* [21] extend this approach by considering in-memory caching of pruned posting lists.

Alternatively, lossy index pruning based on removal of collection-specific stop words is discussed in [4]. Büttcher and Clarke [8, 9] use a compact pruned index that can fit in the main memory of back-end servers. While in [8] they combine term and document pruning, the approach described in [9] advocates pruning the least important terms for each document individually. Query processing with the full index maintained in main memory is discussed in [19].

The approach of Ntoulas and Cho [18] is close to ours as it investigates static index pruning with the aim of reducing the amount of resources needed to handle a given query-rate without sacrificing the result quality (lossless pruning). It also employs term pruning, document pruning, and the combination of both. For a real query log and a large document collection the authors report relatively good hit rates (60-70%) achieved with the pruned index of 10-20% of the original index size.

Notice that contrary to our architecture, none of the index pruning techniques mentioned above use results caching while evaluating their performance.

3. RESIN ARCHITECTURE

In a Web search engine, users submit queries through a front-end server. Upon receiving a new query, such a server forwards it to back-end servers for processing. Each of the back-end servers maintains an index for a subset of the document collection and resolves the query against this subset. The index comprises posting lists for all terms in the sub-collection, where each posting list contains *(document reference, term frequency)* pairs. Once the servers finish processing the query, they return results to the front-end server that displays them to the user. A broker machine is usually responsible for aggregating the results from a number of back-end servers, and returning these results to the front-end server.

It is a natural design choice to place at least one other server in between the front-end and the broker to cache final top-k query results as the broker has to send them to the front-end server in any case.

DEFINITION 1. ***Results cache*** *is a fixed-capacity temporary storage of previously computed top-k query results. It returns the stored top-k results if a query is a* hit *or reports a* miss *otherwise.*

An important advantage of implementing the results cache is decreasing the number of queries that hit the back-end servers, and thus reducing the number of servers needed to handle the query traffic.

However, the hit rate of the results cache cannot increase beyond the limit imposed by singletons, which often constitute a large fraction of the query traffic. Hence, we look into techniques that enable increasing the hit rate further.

DEFINITION 2. ***Pruned index*** *is a smaller version of the main index, which is stored on a separate set of servers. Such a static pruned index resolves a query and returns the response that is* equivalent *to what the full index would produce or reports a* miss *otherwise.*

We consider pruned index organizations that contain either shorter lists (document pruning), fewer lists (term pruning), or combine both techniques. Thus, the pruned index is typically much smaller than the main index and requires fewer servers to maintain it.

Figure 1 shows the *ResIn* architecture where the results cache and the pruned index are placed between the front-end and the broker. In such an architecture, a query is forwarded to the main index only if both the results cache and the pruned index could not answer it, thus substantially reducing the load on the back-end serves.

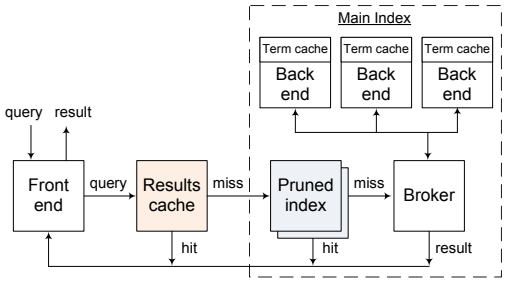

Figure 1: Query processing scheme with the *ResIn* architecture.

Having a pruned index in a different network rather than in the one connecting the main index servers is not a good design choice because ensuring the pruned index is up-to-date requires transferring possibly large portions of the index. Thus, we place the pruned index along with the back-end servers holding the main index.

A recent approach from Ntoulas and Cho [18] employs a similar architecture to reduce the query-rate at the back-end servers without sacrificing the result quality. The authors, however, do not include a results cache, which is a crucial element in our architecture, and employ only a pruned index for this purpose. Note that in reality none of these architectures are really novel as architectures based on clusters for Web search have been proposed before [7]. The importance of the approach proposed by Ntoulas and Cho as well as ours comes from the evaluation of techniques such as caching and pruning.

A typical experimental setup that is used in the literature to investigate the efficiency of an index pruning approach usually considers an original query log or a small set of TREC queries (*e.g.*, [1, 4, 9, 10, 16, 18, 21]). In Section 5.2 we will show that some statistical properties of query logs change significantly when results caching is used.

4. EXPERIMENTAL SETUP

Test queries. We used a large query log of the Yahoo! search engine. The log contains more than 185M queries submitted at the **.uk** front-end of the search engine. We use this query-log to simulate query processing with a results cache and generate a "miss-log" of queries that are not filtered out by the results cache. Henceforth, we use *all queries* to denote the queries from the original query log, and *misses* to denote the queries from the miss log. Throughout the rest of the paper we will compare properties of both logs and use them to test various pruning techniques.

Document collection. To investigate the efficiency of index pruning techniques we also used the UK document collection [5, 11]. It contains 77.9M Web-pages crawled from the **.uk** domain in May 2006. We used the Terrier platform[1] to index the collection on 8 commodity machines, which resulted in a distributed compressed index of approximately 40Gb without positional information.

5. RESULTS CACHING

Results cache is a temporary storage for previously computed partial query results. We assume here dynamic caching: when the system processes the query response containing top-k results and returns it to the user, it stores these results in the cache, such that, for future requests of the same query it returns these results instead of asking the back-end servers to process the same query again. Results cache uses an eviction policy to ensure that it never exceeds a maximum capacity. A query produces a *hit* if it was found in the cache and a *miss* otherwise.

In this section we analyze the performance of results caching using a real query log and study the differences between the original query stream and the stream of misses after the results cache.

5.1 Results Cache Performance

To evaluate the performance of the results cache we implemented the LRU policy and tested it with our query log. We varied the cache capacity from $1M$ to $15M$ entries, where each entry contains a query and its top-k results.

Figure 2 shows cache-hit rates obtained with different cache capacities for our query log. For each point we warm up the cache with the first $40M$ queries and then measure the average cache hit for the remaining $145M$ queries.

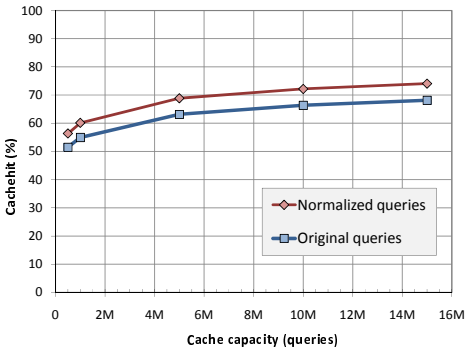

Figure 2: Cache hit achieved with a large results cache using the LRU eviction policy.

We also normalized each query by converting it to lower case, removing special symbols and sorting the terms in each query in alphabetical order. Despite the simplicity of this normalization[2], it performs similarly to normalization procedures in current search engines. Figure 2 shows that the normalization increases the cache hit by roughly $4-5\%$. This happens because semantically similar, but lexicographically different queries have the same normalized versions, *e.g.*, queries written in a different case, containing symbols ignored by search engines, *etc.*

Due to the presence of singleton queries the performance of any results cache is bounded by the fraction of potential hits $hits_{max} = (1 - \frac{distinct\_queries}{all\_queries})$. For the normalized version of our query log this value is equal to 75.04%.

According to our experiments, sophisticated caching policies such as LFU, SDC [15] or AC [3] substantially outperform LRU when the cache capacity is small, due to the optimized space usage. However, the improvement is marginal when the cache capacity is big enough such that the hit rate approaches its upper bound. In such a case, LRU becomes the best option due to its simplicity.

Let us assume that one entry in the results cache requires around 2 Kb in order to store top-20 results including document URLs, titles and snippets. This number can be further reduced by using compression. In this case one machine with $16Gb$ of memory can host a results cache with the capacity of approximately $10M$ queries. Thus, we simulated

[1]http://ir.dcs.gla.ac.uk/terrier

[2]*E.g.*, it ignores phrase queries and treats URLs as sets of terms.

a $10M$ results cache with the LRU eviction policy and used it to process all $185M$ normalized queries. We warm up the cache with the first $40M$ queries, and then record all query misses for the remaining $145M$ queries. As a result, we obtained a "miss log" containing $41M$ (mostly singleton) queries. Notice that the miss log contains queries that have to be processed by the back-end servers, and that query processing has to be optimized for such queries.

A very important property of the results cache is that its hit rate remains constant as the size of the document collection grows because it depends only on the query log properties, in particular on the number of unique queries. This property makes results caching a very efficient technique to reduce the query load on the back-end servers, as it requires a relatively small amount of storage and processing. The size of a pruned index, however, typically grows with the size of the collection as we discuss further in Section 6.

5.2 All Queries vs. Misses

First, we confirm that the average number of terms in a query increases from 2.4 for all queries to 3.23 for misses. Fractions of queries with different number of terms are shown in Figure 3. In particular, it shows that the majority of single term queries become hits in the results cache and thus very few of them have to be processed by the index.

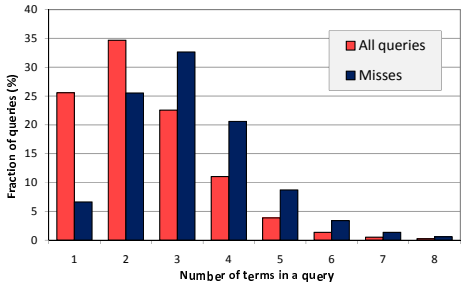

Figure 3: Fraction of queries with a given number of terms among all queries and misses.

Another crucial difference between all queries and misses is the distribution of the sizes of query results. We use the Yahoo! search engine to collect the (estimated) results set sizes for $2,000$ randomly chosen queries in each set. Figure 4 shows that the result set sizes for query misses are approximately two orders of magnitude smaller than for all queries. In particular, almost half of the misses return less than $5,000$ results, which is extremely small compared to the total number of documents on the Web indexed by the Yahoo! search engine. Figure 4 also shows similar results for the (much smaller) UK document collection of 78M documents described in Section 4.

We say that a query is *discriminative* if it returns relatively few results. Usually it happens because the query contains at least one rare term or the number of terms in the query is large. Both cases suggest that such a query is unlikely to be popular in the query log and therefore results caching performs poorly for discriminative queries. Furthermore, query misses contain a considerable fraction of misspells as they are typically not filtered out by the results cache. In our test set of misses, we detected about 20% of misspelled queries.

We characterize each query term with two properties: popularity and frequency. A term is considered *popular* if it is likely to appear often in *queries*. That is, term popularity is proportional to the number of occurrences of the term in

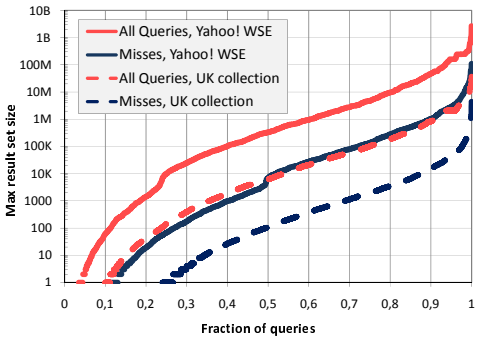

Figure 4: Query result size distributions for: 1) the Yahoo! Web search engine, and 2) the UK document collection (78M documents).

the query log. A term is considered *frequent* if it is likely to appear in *documents*. Frequency (or document frequency) is hence proportional to the number of occurrences of the term in the document collection.

Figure 5 shows that terms that are popular in the original query log remain popular in the miss log as well. We extracted all terms from the original query log and sorted them by popularity. For each term we also computed its popularity in the miss log (0 if the term does not appear there). Each point on the plot shows the average popularity of $1,000$ consecutive terms normalized by the size of the query log (185M for the original query log and 41M for the miss log). Notice that averaging over $1,000$ consecutive terms generates a smoother curve for misses while having one point per term would compromise visualization.

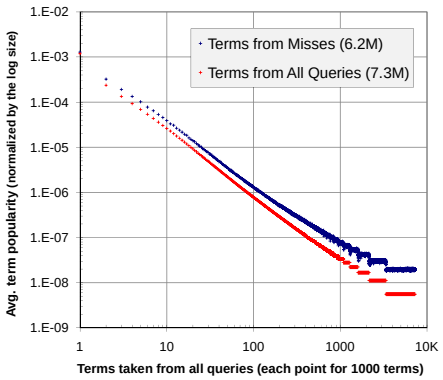

Figure 5: Term popularity distribution for all queries and for misses.

From the figure we conclude that popular terms remain popular in the miss log as well. Unpopular terms typically come from queries that are not hits in the results cache, so their absolute popularity is similar in both logs, but the normalized popularity is higher for the miss log because the miss log size is smaller.

This result indicates that, despite significant changes to the query stream after the results cache, techniques that are based on term popularity such as term caching and term pruning work similarly for all queries and for misses. We investigate this observation in detail in the following sections.

6. INDEX PRUNING

In this section we consider term and document pruning techniques as well as the combination of both in the context of the architecture of Figure 1.

To perform index pruning experiments we used the original query log containing $185M$ normalized queries and the miss log containing $41M$ normalized queries. Since it is computationally expensive to run experiments with such workloads, we randomly selected a test set of $10,000$ queries from the last $5M$ queries in the original query log. Similarly, we used the last $1M$ queries in the miss log to generate a test set of $10,000$ misses. We will refer to these test sets as *all queries* and *misses* in the following sections. The remaining portions of the logs (first $180M$ queries and $40M$ misses) were used to compute term popularities.

6.1 Term pruning

Term pruning selects the most profitable terms and includes their full posting lists into the pruned index. Thus, in order to verify whether a query can be processed by such a pruned index, we only need to check that *all* terms from the query are included in the pruned index. Notice that such a pruning technique assumes that each document in the result set of a query has to contain all terms from the query (conjunctive query processing).

We implemented a term pruning algorithm that estimates a term profit as: $profit(t) = \frac{pop(t)}{df(t)}$, where $pop(t)$ is the popularity of the term t and $df(t)$ is its document frequency[3]. Hence, the profit of a term is proportional to its popularity and inversely proportional to the space required to store its posting list. We extracted the list of all terms found in the original query log and computed their profits.

The optimization problem of selecting the terms that maximize the hit rate for a given pruned index size constraint is known to be NP-complete, as it can be reduced to the well known knapsack problem. Thus, we follow the standard heuristics that have been shown to perform well in this scenario [2, 18]. We sort the list of terms by profits in descending order and pick the top terms as long as the sum of the posting list sizes for selected terms remains below the given size constraint. Figure 6 shows the performance of such a pruning strategy for all queries and for misses.

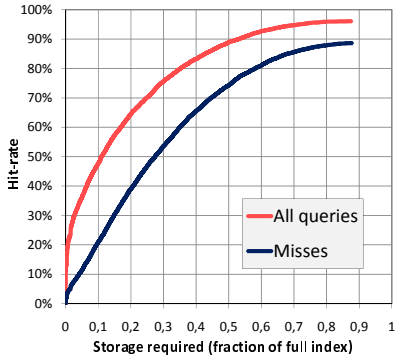

Figure 6: Hit rate with the term pruned index.

The hit rate for all queries grows quickly in the beginning because of single term queries and queries that contain few popular terms (typical hits in the results cache). In case of misses though, the growth is nearly linear in the beginning, still being able to handle nearly half of the queries with only a quarter of the index. Thus, even with misses, we can reduce hardware costs by considering a pruned version of the index. For example, suppose that one full cluster processes half of the query load, for some design of index cluster, thus requiring two clusters to process all queries. By using a pruned index, we can have instead one full cluster and one other cluster for the pruned index that is one-fourth of the original cluster.

Figure 7-a compares the efficiency of the results cache and the term pruned index when they are used separately. It shows that with a fixed amount of storage available, the hit rate of the pruned index decreases linearly with the growth of the document collection size (a collection of size x corresponds to the set of 78M documents from the .uk domain described in Section 4). The storage required for the results cache, however, depends only on the query log properties and thus remains constant. Therefore, for a Web-scale document collection, the relative results cache size would amount to a small fraction of the full index. Results caching hence is the preferable solution when the amount of available storage is limited and the document collection is large.

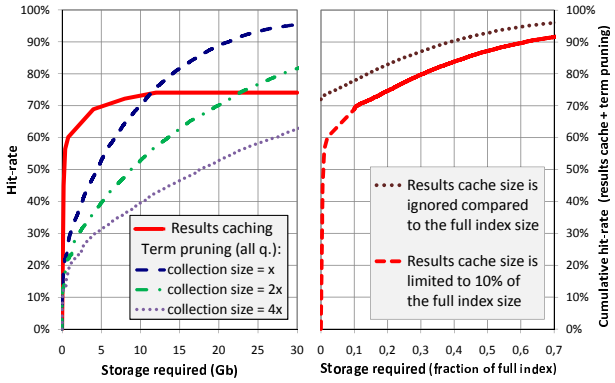

Figure 7: Comparison of results caching and term pruning used separately (a), cumulative hit rate with both techniques used together (b).

However, the hit rate of the results cache is bounded and to avoid this limitation we study the combination of results caching and index pruning. Figure 7-b shows the cumulative hit rate obtained by the results cache and the pruned index together following the architecture of Figure 1. The top dotted line in Figure 7-b shows the maximum cumulative hit rate in such a scenario assuming a sufficiently large index size such that the results cache storage cost can be neglected. We also measured the cumulative hit rate with the real (and relatively small) UK document collection. In this case the results cache size was limited to maximum 10% of the full index size (approx. 4 Gb). The dashed portion of the bottom line in Figure 7-b until 10% of the index size corresponds to the hit rate produced by the results cache and the rest is obtained by the pruned index additionally to the results cache.

Figure 7 proves the importance of results caching in our architecture and shows that the combination of results caching and index pruning delivers very good hit rates.

Recall that term popularities in the original query log and in the miss log are very similar (see Figure 5). Hence, the term pruned index has to include a significant number of popular terms associated with large posting lists. Indeed, with the next experiment we confirm that a large fraction of misses contains at least one frequent term, although misses typically return few results.

In Figure 8 we use the test set of misses and the UK document collection to plot the correlation between the query

[3]In fact, the denominator $df(t)$ in the profit formula implies a non-zero storage cost for t's posting list.

result set size and the document frequency of the *most* and the *least* frequent term in a query. The former one is denoted as $MaxDF$ (top plot), while the latter one is $MinDF$ (bottom plot). On both plots, we sort queries by the size of their result set in increasing order, and the dashed line shows the size in log scale. The remaining lines reflect the probability of $MaxDF$ ($MinDF$) being above one of the thresholds: 0, $1K$, $100K$ and $1M$ elements. Each point is computed for 250 consecutive queries with about the same result set size, which can be estimated from the dashed line.

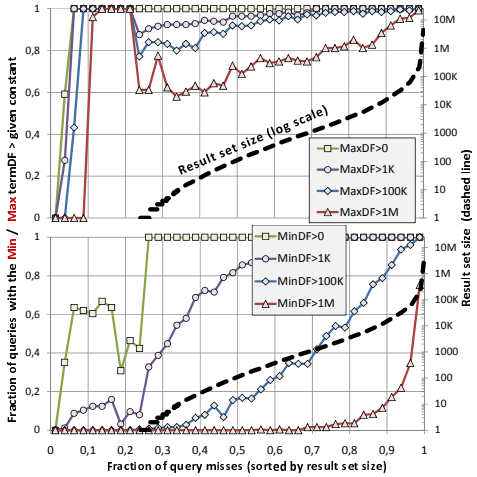

Figure 8: Fraction of *misses* with *df* of the most frequent (top) and the least frequent (bottom) term in a query above a given threshold.

Figure 8 suggests that $MinDF$ correlates with the result set size of the query (bottom plot), whereas $MaxDF$ is constantly high for most of the misses (top plot). Hence, the term pruned index has to include large posting lists in order to guarantee a reasonable hit rate.

6.2 Document pruning

Including full (and often large) posting lists in the pruned index might seem redundant, so we studied the document pruning option. Document pruning removes the least important entries from the posting lists. It is based on the observation that if posting lists are sorted such that more relevant documents are stored first, the top-k results for a query are likely to be computed without traversing the whole lists, but by examining the top portions of them only [14, 18]. Thus, posting lists are usually sorted by an attribute that reflects the probability of the document to be included in the top results, such as score, term frequency or impact [1].

When static document pruning is used, only top-portions of posting lists are included in the pruned version of the index and are used for query processing. A query can be answered from such a pruned index only if it produces exactly the same top-k results as the full index would. It is in general difficult to test this condition, and the only acceptable solution we are aware of is starting processing the query using the pruned index – if the top-k correct results are identified, then the query is a *hit*. Otherwise, the query is a *miss* and it has to be forwarded to and processed again by the main index, thereby affecting latency. To determine the correctness of the results, we compute a maximum score threshold for all potentially relevant documents whose scores cannot be computed exactly due to truncation. Then, we verify whether all top-k results have scores above the threshold [14, 18].

Document pruning depends on the ranking function as it significantly influences how quickly the top-k results for a query can be found while examining the top portions of the posting lists. Following Craswell *et al.* [12], we used the following weighting formula:

$$score(d,q) = \sum_{\forall t \in q} (bm25(t,d) + \omega \frac{pr(d)}{pr(d)+k}),$$

where $bm25(t,d)$ is the non-normalized BM25 score of the document d for a term t and $pr(d)$ is the query independent score of the document d (pagerank computed from the graph of the UK collection [6]). We do not have relevance judgments for our document collection to estimate the values of k and ω, so we fix $k = 1$ and vary the value of ω to study the impact of the pagerank weight on the document pruning performance. Intuitively, the higher the weight ω is, the better document pruning performs. In the following experiments we set ω to 0 (no pagerank), 10 and 20.

Notice that the way the query independent rank is included in the final score affects the index construction algorithm. Our formula guarantees that the top-k results obtained from the pruned index and having scores above the maximum score threshold are exactly the same as if they were computed from the full index. The ranking function presented in [18] requires a slightly modified index construction so that this condition holds.

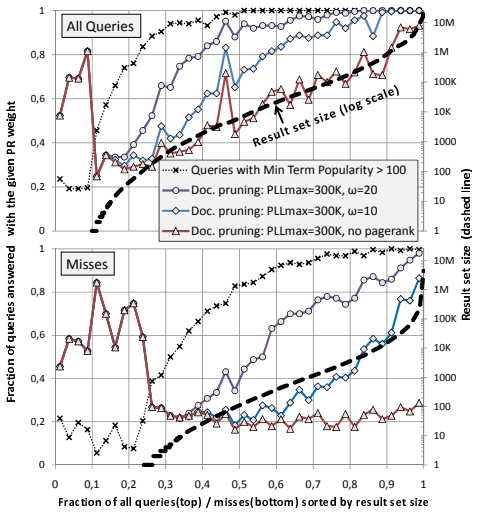

Figure 9: Fraction of *all queries* (top) and *misses* (bottom) that can be resolved from posting lists truncated to top $PLL_{max} = 300K$ entries.

Figure 9 shows the fraction of queries whose correct top-10 results are found in all pruned posting lists relevant to the query. The maximum Posting List Length (PLL_{max}) specifies the pruning threshold and is set to $300K$ entries. From the figure, we can see that document pruning works better with non-discriminative queries (i.e., queries with a large result set size). Furthermore, the fraction of queries that require no more than top-$300K$ entries in the posting lists grows with the increase of the pagerank weight ω.

The same plot shows the fraction of queries with the *least* popular term having more than 100 occurrences in the query log. We could see that non-discriminative queries contain popular terms only, while unpopular terms appear in queries that return few results. This observation indicates that popular terms tend to be frequent as well.

Figure 9 suggests that document pruning performs much

better for all queries than for misses. Indeed, Figure 10 shows that while working well for all queries, document pruning requires very high values of ω to outperform term pruning with misses. Each point of Figure 10 is obtained by computing the hit rate[4] of the document pruned index with a number of different PLL_{max} values and selecting the one that delivers the maximal hit rate. The dashed lines correspond to term pruning for comparison.

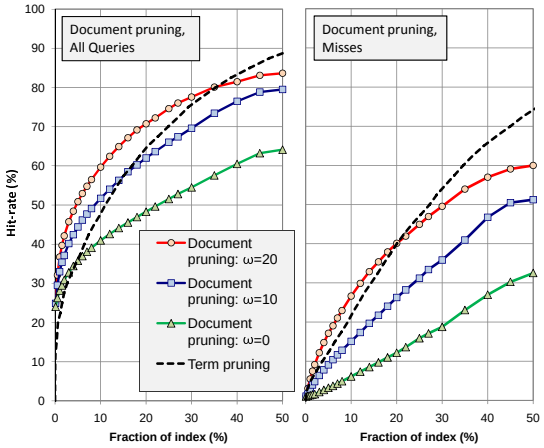

Figure 10: Document pruning for *all queries* (left) and *misses* (right) compared with term pruning.

In fact, the efficiency of document pruning correlates to the size of the query result. When the query result is small, it is likely that we have to scan large parts of the posting lists (or even whole lists) to obtain enough documents. It can be very inefficient because frequent terms with long posting lists are common in misses (see Figure 8). Thus, the worse performance of document pruning with misses can be explained by the fact that the result set sizes are approximately two orders of magnitude smaller compared to all queries (see Figure 4).

6.3 Term+Document pruning

Following Ntoulas and Cho [18] we study the combination of term and document pruning in our scenario. The intuition behind combining the two is the following: while including profitable terms, only at most top-PLL_{max} entries from each posting list should be copied to the pruned index. If the PLL_{max} constant is chosen such that the hit rate is maximal for a given pruned index size, term+document pruning would deliver at least as good hit rate as term pruning. Indeed, in the case when PLL_{max} is set to ∞ the hit rate with such a pruned index will be exactly the same as for term pruning.

To select terms for such a pruned index we adopt the same profit function as for term pruning $profit_1(t) = \frac{pop(t)}{df(t)}$ because it was used in [18]. However, taking into account the observations made in the previous section, we introduce a new profit function $profit_2(t) = \frac{pop(t)}{min(df(t), PLL_{max})}$ specifically tailored to term+document pruning. The rationale behind it is that the storage cost of including a term t in the pruned index is proportional to $df(t)$ when $df(t) < PLL_{max}$, but depends only on the PLL_{max} parameter if $df(t) \geq PLL_{max}$.

[4]In fact, we compute an *upper bound* for the hit rate – the fraction of queries for which the correct top-10 results can be found in each pruned posting list relevant to the query.

Figure 11 shows the hit rate[4] of term+document pruning with the two profit functions defined above for all queries and for misses. Each point is obtained by computing the performance of the term+document pruned index with a number of different PLL_{max} values and selecting the one that delivers the maximal hit rate. Notice that when the maximum hit rate is achieved with $PLL_{max} = \infty$, the corresponding point coincides with the dashed line, which means that there is no improvement over term pruning.

Figure 11 suggests that the new $profit_2$ function (left plot) substantially outperforms $profit_1$ (right plot), especially with high pagerank weights. For example, the table below shows the hit rate values obtained with the pruned index 10 times smaller than the full index for both profit functions and different values of ω:

	Term pruning only	Term+document pruning					
		$profit_1$			$profit_2$		
		ω=0	ω=10	ω=20	ω=0	ω=10	ω=20
All queries	47.7	47.7	54.9	61.4	49.7	61.7	**69.3**
Misses	21.7	21.7	21.7	28.6	21.7	26.2	**37.7**

Figure 11 also shows that for all queries term+document pruning brings substantial benefits compared to term pruning only. However, with misses it yields only a small increase of hit rate when the pagerank weight is very high. For example, the table above shows that the hit rates of term+document pruning with misses noticeably outperform the baseline 21.7% of term pruning with $\omega = 20$ only. Furthermore, term+document pruning induces high processing costs and latencies compared to term pruning, therefore becoming rather unusable with misses.

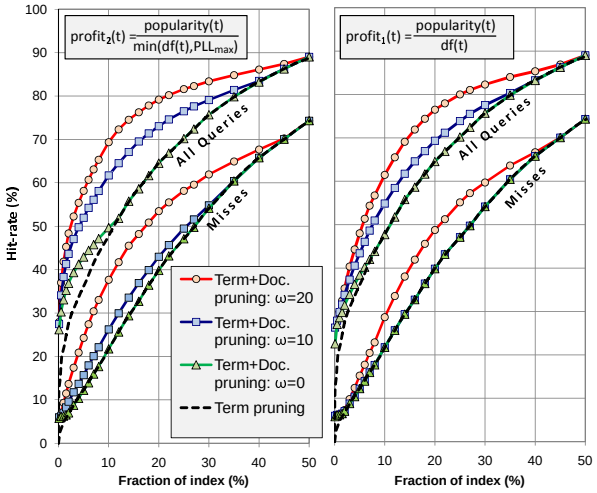

Figure 11: Term+document pruning for *all queries* and for *misses* with both profit functions.

To perform the last experiment, we fixed the size of the pruned index to 10% of the full index (i.e., the size of the pruned index grows *linearly* with the size of the document collection) and the pagerank weight ω to 20 (the maximum value in our experiments). We measure the hit rate of the pruned index for fractions of the document collection indexed on 1 and 4 machines respectively, as well as for the full index on 8 machines.

Figure 12 shows that the hit rate is approximately the same for the term pruned index when its size is 10% of the full index size (dashed lines) because the length of the posting lists increases linearly with the collection size.

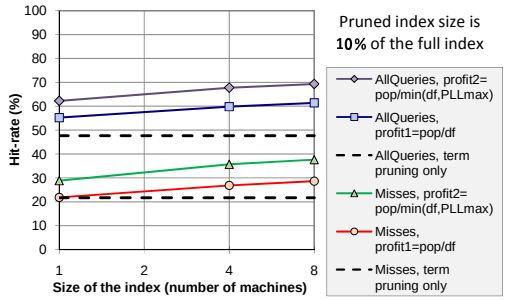

Figure 12: Index pruning efficiency for different sizes of the document collection.

The hit rate for term+document pruning grows with the size of the document collection. This happens because document pruning works better with non-discriminative queries, which match a large number of documents. However, in practice, if such a pruned index had to be distributed among several servers following the standard document partitioning scheme, each server would have to compute the top-k results for each query from its (small) fraction of the index. Thus, the resulting hit rate suffers because each server matches a small set of documents in its local index.

6.4 Discussion

The original stream of queries contains a large fraction of non-discriminative queries that usually consist of few frequent terms (*e.g.*, navigational queries). Since frequent terms tend to be popular, those queries are likely to repeat in the query log and therefore are typical hits in the results cache. At the same time, these queries would be good candidates to be processed with the document pruned index due to their large result set size. Therefore, document pruning does not perform well anymore when the results cache is included in the architecture. The same conclusion can be drawn from the fact that misses return much fewer results than original queries on average.

However, individual term popularites are similar for all queries and for misses. Therefore the contents of the term pruned index does not change much. A larger fraction of short queries explain higher hit rates with the term pruned index for all queries than for misses.

7. CONCLUSION

In this paper we have presented the *ResIn* architecture for Web search engines that combines results caching and index pruning to reduce the query workload of back-end servers. With *ResIn*, we showed that such a combination is more effective than previously proposed approaches, for example being capable of handling up to 85% of queries with four times less resources than the full index requires.

Results caching is an effective way to reduce the number of queries processed by the back-end servers, because its performance is bounded by the amount of singleton queries in the query stream, and not by the size of the index. The results cache significantly changes the characteristics of the query stream: the queries that are misses in the results cache match approximately two orders of magnitude fewer documents on average than the original queries. However, the results cache has little effect on the distribution of query terms. These observations have important implications for implementations of index pruning: the results cache only slightly affects the performance of term pruning, while document pruning becomes less effective, because it targets the same queries that are already handled by the results cache.

When combining term and document pruning, we substantially increased the hit rates by considering a better term profit function. We have also found that document pruning is more effective when query independent scores, such as pagerank, have high weights in the final scoring formula.

Overall, *ResIn* has allowed us to gain significant insight into the dependence between different techniques for reducing the load on the back-end servers in a Web search engine, and it has allowed us to improve existing techniques by testing them in a more realistic setting.

8. REFERENCES

[1] V. N. Anh and A. Moffat. Pruned Query Evaluation Using Pre-computed Impacts. In *SIGIR*, 2006.

[2] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The Impact of Caching on Search Engines. In *SIGIR*, 2007.

[3] R. Baeza-Yates, F. Junqueira, V. Plachouras, and H. F. Witschel. Admission Policies for Caches of Search Engine Results. In *SPIRE*, 2007.

[4] R. Blanco and A. Barreiro. Static Pruning of Terms in Inverted Files. In *ECIR*, 2007.

[5] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience*, 34(8), 2004.

[6] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW*, 2004.

[7] E. A. Brewer. Lessons from Giant-scale Services. *IEEE Internet Computing*, 5(4), 2001.

[8] S. Büttcher and C. L. A. Clarke. Efficiency vs. Effectiveness in Terabyte-scale Information Retrieval. In *TREC*, 2005.

[9] S. Büttcher and C. L. A. Clarke. A Document-centric Approach to Static Index Pruning in Text Retrieval Systems. In *CIKM*, 2006.

[10] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, and A. Soffer. Static Index Pruning for Information Retrieval Systems. In *SIGIR*, 2001.

[11] C. Castillo, D. Donato, L. Becchetti, P. Boldi, S. Leonardi, M. Santini, and S. Vigna. A Reference Collection for Web Spam. *SIGIR Forum*, 40(2), 2006.

[12] N. Craswell, S. Robertson, H. Zaragoza, and M. Taylor. Relevance Weighting for Query Independent Evidence. In *SIGIR*, 2005.

[13] E. S. de Moura, C. F. dos Santos, D. R. Fernandes, A. S. da Silva, P. Calado, and M. A. Nascimento. Improving Web Search Efficiency via a Locality Based Static Pruning Method. In *WWW*, 2005.

[14] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, 2001.

[15] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the Performance of Web Search Engines: Caching and Prefetching Query Results by Exploiting Historical Usage Data. *ACM Trans. Inf. Syst.*, 24(1), 2006.

[16] X. Long and T. Suel. Optimized Query Execution in Large Search Engines with Global Page Ordering. In *VLDB*, 2003.

[17] X. Long and T. Suel. Three-level Caching for Efficient Query Processing in Large Web Search Engines. In *WWW*, 2005.

[18] A. Ntoulas and J. Cho. Pruning Policies for Two-Tiered Inverted Index with Correctness Guarantee. In *SIGIR*, 2007.

[19] T. Strohman and W. B. Croft. Efficient Document Retrieval in Main Memory. In *SIGIR*, 2007.

[20] J. Teevan, E. Adar, R. Jones, and M. A. S. Potts. Information Re-retrieval: Repeat Queries in Yahoo's Logs. In *SIGIR*, 2007.

[21] Y. Tsegay, A. Turpin, and J. Zobel. Dynamic Index Pruning for Effective Caching. In *CIKM*, 2007.

[22] J. Zhang, X. Long, and T. Suel. Performance of Compressed Inverted List Caching in Search Engines. In *WWW*, 2008.