Performance Improvements for Search Systems using an Integrated Cache of Lists+Intersections

Gabriel Tolosa, Luca Becchetti, Esteban Feuerstein, Alberto Marchetti-Spaccamela

Abstract. Modern information retrieval systems use sophisticated techniques for efficiency and scalability purposes. Among the most frequent such techniques is the implementation of several levels of caching. The main goal of a cache is to speedup computation by exploiting frequent, recent or costly data used in the past. In this study we propose and evaluate a static cache that works simultaneously as list and intersection cache, offering a more efficient way of handling cache space. In addition, we propose effective strategies to select the term pairs that should populate the cache. Simulation using two datasets and a real query log reveal that the proposed approach improves overall performance in terms of total processing time, achieving savings of up to 40% in the best case.

1 Introduction

Modern high scale information retrieval systems such as Web Search Engines (WSE) utilize sophisticated techniques for efficiency and scalability purposes to deal with their requirements: they crawl and index tens of billions of documents (thus managing a huge inverted index file) and must solve queries in minimum fractions of time (i.e. milliseconds) to satisfy users expectations. In such a scenario, caching becomes one of the most important and crucial tools to achieve fast response times and to increase query throughput.

Basically, the total cost of serving a query is defined as the sum of processing time (C_{cpu}) and disk access times (C_{disk}). C_{cpu} involves decompressing the posting lists, computing the query-document similarity scores and determining the top-k documents that form the final answer set. In most cases a conjunctive semantic is considered because intersections produce shorter lists than unions, which leads to smaller query latencies [3] and higher precision levels. On the other hand, C_{disk} involves fetching from hard disk the (usually compressed) posting lists of all the query terms.

The main goal of a cache is to speedup computation by storing frequent, recent or costly data. The typical architecture of a search engine involves different cache levels (Fig. 1): essentially, caching involves both query result pages (*Result cache*) at the broker level and the posting lists of terms that appear in the queries (*List Cache*) at search node level. The first level tries to minimize recomputation of results for queries that appeared in the past, thus also reducing the workload of back-end servers. The latter attempts at reducing the amount of disk fetch operations, that are very expensive compared to CPU processing times. If a list is found in cache disk cost is avoided.

A further approach involves caching portions of a query (i.e., pairs of terms), as initially proposed in [14] and extended in [6]. This approach is named *Intersection Caching* and is implemented at search node level as well. The idea in this case is to exploit term co-occurrence patterns, e.g., by keeping the intersection of the postings lists of frequently co-occurring pairs of terms in the memory of the search node, in order to not only save disk access time, but CPU time too.

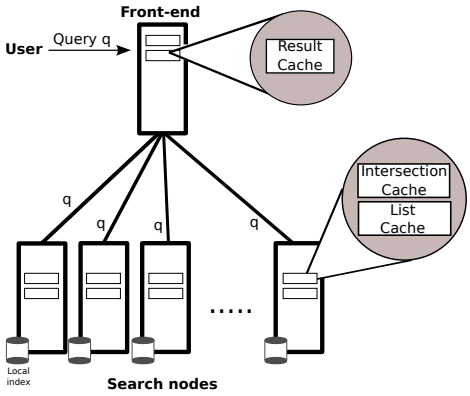

Fig. 1. Search Engine Architecture considered in this work

Some industry-scale search engines systems store the entire index in main memory [5]. This means that the *List Cache* becomes useless, but the intersection cache is still useful [9] because it allows to save CPU time (i.e. the cost of intersecting two posting lists). For more general cases such as medium-scale systems, only a fraction of the index is maintained in cache. Here, lists and intersections caches are both very helpful to reduce disk access and processing time. We consider this scenario in our work.

All types of caches (query results, posting lists, intersections), may be managed using static or dynamic policies, or both. In the static case, the cache is filled with items previously selected from a training set and its content remains unaltered until the next update. In the dynamic case, the cache content changes in an online fashion with respect to the query stream, as new entries may be inserted and existing entries may be evicted. A great deal of relevant prior work on caching is found in the literature. Section 2 summarizes some work related to ours.

As we mentioned earlier, list and intersection caches are implemented at search node level. Usually, these are independent and try to give benefits from two different perspectives. The List Cache achieves a greater hit rate because the frequency of individual terms is higher than that of pairs of terms, but each hit in the later entails a higher benefit because the posting lists of two terms are involved and also some computation is avoided.

Based on the observation that many terms co-occur frequently in different queries our motivation is to build a cache that may capture the benefits of both approaches in just one cache (instead of two). To do this, we implement a data

structure previously proposed by Lam et al. [13]. The original idea is to merge the entries of two frequently co-occurring terms to form a single entry to store the inverted files more compactly. We adapt this structure to a static cache in which the pair of terms (bigram) selected offer a good balance between hit rate and benefit, leading to an improvement in the total cost of solving a query. We investigate different ways of choosing and combining the terms.

1.1 Our Contribution

In this work, we explore the possibility of reserving the whole memory space allocated to caching at search nodes to an integrated cache, in order to reduce query processing time. More precisely, as our main contribution we propose a static cache (named *Integrated Cache*) that replaces both list and intersection caches using a data structure previously used for pairing terms in disk inverted indexes. This data structure already makes an efficient use of memory space, but we design a specific cache management strategy that avoids the duplication of cached terms and we adopt a query resolution strategy (named S4 in [9]) that tries to maximize the hit ratio. As a further contribution, we consider different strategies to populate the integrated cache: besides a strong baseline, we consider three greedy strategies that consider both frequency of term co-occurrence and postings list size. Furthermore, we propose a novel, principled strategy, that relies on casting the problem of selecting term pairs as a maximum weighted matching, a well-known combinatorial optimization problem. We evaluate the proposed framework against a competitive list caching policy using two real web crawls with pretty different characteristics and a well-known query log over a simulation framework. Rather than hit ratio, we consider the overall time needed by the different strategies to process all queries as a performance metric. Experimental evidence shows that substantial savings are possible using the proposed approach.

The remainder of this paper is organized as follows: in the next section, we review the related work in the literature. In Section 3, we provide some background about query processing and related data structures. In Section 4 Integrated Cache proposal is presented while the methods for selecting the terms to fill the cache are introduced in section 5. The following two sections are devoted to the experimental setup and the methodology, respectively. The experimental results are summarized in Section 8. We conclude and point to future research directions in Section 9.

2 Related Work

There is a large body of work devoted to caching in text search systems, however this is still an active research area because both the data to process and the number of users are continuously growing.

Posting Lists Caching: Baeza et al. [1] analyze the problem of posting list caching (combined with results caching) that achieves higher hit rates than caching query results. Also, they propose an algorithm called $Q_{tf}D_f$ that selects

the terms to put in cache according to its $\frac{frequency(t)}{size(t)}$ ratio. The most important observation is that the static $Q_{tf}D_f$ algorithm has a better hit rate than all dynamic versions. They also present a framework for the analysis of the trade-off between caching query results and caching posting lists.

In [24] inverted index compression and list caching techniques are explored. Authors compare several inverted list compression algorithms and caching policies. They compare LFU, LRU, Optimized Landlord, Multi-Queue, ARC. The last three policies try to balance recency (LRU) and frequency (LFU) and in practice perform similarly.

Results Caching: The work in [16] is the first approach on the problem of result caching. Based on the analysis of a query log and the amount of locality observed, the author proposes to consider both frequency and recency into the policy and proposes LRU-2S (two stages LRU) that tries to capture both variables. In 2006, Fagni et al. [7] propose SDC (Static and Dynamic Cache) to handle both long term popular queries and shorter query bursts in smaller periods of time. SDC divides the cache space in two parts: a static one that is filled (offline) with the results of the most frequent queries (computed on the basis of a query log), and a dynamic part that is managed with LRU.

Gan and Suel [11] study the problem of weighted result caching based on the observation that most previous work focuses only on optimizing the hit ratio while the processing costs of queries vary according to lists' sizes, terms' popularities, and so on. They propose weighted versions of LFU, LRU and SDC policies and a Landlord strategy. The main result is the study of the weighted case with the goal of optimizing the processing cost. In [18], cost aware strategies for result caching are extensively evaluated. This is based on the observations that cache misses have different costs and caching policies based on popularity can not always minimize the total cost. So, authors propose to incorporate the query costs into the caching policies and evaluate them in both static, dynamic and hybrid cases.

Intersection Caching: The first proposal on intersection caching appears in [14] where the authors introduce a three-level caching architecture for a web search engine (results+intersections+posting lists). In this case, cache is disk based (about 20% of the total index space) and the main idea is to save processing cost for high co-occurent pairs of terms. They apply a greedy algorithm to select which items to store in cache, and then evaluate a landlord-based policy.

Deeper studies regarding cost aware intersection caching are presented in [8] and [9]. Basically, these works focus on two different scenarios: with the inverted index residing on disk and in main memory. The authors also propose and evaluate different query resolution strategies specially designed to take advantage of the *Intersection Cache* and explore both static, dynamic and hybrid policies.

Multilevel Caching: Saraiva et al. [21] propose a two-level caching scheme that combines caching of search results with the caching of frequently accessed postings lists. Long and Suel extend this idea to caching also intersections of pairs of terms that are often co-used [14]. In a more recent work, Ozcan et al. [19] introduce a 5-level static caching architecture.

3 Background

In this section we provide basic background about the data structures and algorithms used to solve a query in a distributed search system. In general, a query $q = \{t_1, t_2, t_3, ..., t_n\}$ is a set of terms that represents the users information need.

3.1 Inverted Indexes

The main data structure used in information retrieval systems is the inverted index. This data structure enables full-text indexing and retrieval using free text queries and phrases [25]. They store the set of all unique terms in the document collection (vocabulary) associated to a set of entries that form a posting list. Each entry represents the occurrence of a term t within a document d. Usually, a posting is composed by a document identifier (DocID) and a payload that is used to store information about the occurrence of t within d (frequency, positions, etc.). Each posting list is sorted in increasing order of DocID or score depending on the solving strategies. Often, skip-lists [15] are used to index the lists [17] to speed up their traversal when searching for a particular DocID. There is a considerable body of literature on index construction, please refer to [2, 23, 25]. In the remainder, we denote by t both a term and the corresponding list.

3.2 Query Processing

The processing of queries in a distributed search system as depicted in Figure 1 is usually handled as follows [3]: the broker machine receives the queries and looks for it in its result cache. If the result is found the answer is immediately returned to the user with no extra computational cost. Otherwise, the query is sent to the search nodes in the cluster where an inverted index resides.

Each search node fetches the posting lists of the query terms (from disk or cache), reorders these in ascending order of their lengths, executes the intersection of the lists and finally ranks the resulting set. After that, a list containing the top-k documents identifiers is sent to the broker which merges it with the lists of other nodes to obtain the final answer. This is a time-consuming task that is critical for the scalability of the system because of disk accesses (partially affected by the size of the document collection). To mitigate this situation different cache levels are used to reduce disk costs. This phase is the focus of our work, where the proposed integrated cache may be used to improve performance.

To solve a query there exist two main strategies, namely Term-at-a-time (TAAT) and Document-at-a-time (DAAT) [22]. In the TAAT approach, the posting lists of the query terms are sequentially evaluated, starting from the shortest to the longest one. This basically computes the result as $R = \cap_{i=1}^{n} t_i = (((t_1 \cap t_2) \cap t_3) \ldots \cap t_n)$. On the other hand, in the DAAT approach the posting lists are traversed in parallel for each document and only the current k-th best candidates are maintained in memory. The Max Successor algorithm [4] is an efficient strategy for DAAT processing. However, the presence of an *Intersection Cache* enables other possibilities such as the S4 strategy introduced in

[9]. This basically tests all the possible two-term combinations in the cache in order to maximize the chance of a hit and rewrites the query according to this result. In that work, the authors show that the S4 strategy allows a performance improvement up to 30% combining it with cost aware cache policies.

4 Integrated Cache

Our proposal consists on using the paired data representation presented in [13] to build an integrated cache that works as list and intersection cache at the same time. In that work, an index compression technique based on pairing posting lists of frequently co-occurring terms was proposed. The idea is to merge the lists of two frequently co-occurring terms to build a new *paired* list in the inverted file. This is obviously a more compact representation that may speed-up the query processing time. This data structure is introduced as a compression technique for inverted files combined with Gamma Coding and Variable Byte Coding [2] schemes. As a complementary application, the authors show the use of this data structure to build a static list cache. Our proposal extends this approach to an integrated cache of lists+intersections.

In our approach, we use the "Separated Union" representation to maintain an in-memory data structure, that replaces both the list and intersection. Regarding space savings, the main idea is to keep in cache those pairs of terms that maximize the high hit ratio of the *List Cache* and the savings of the most valuable precomputed intersections. We also propose to avoid the repetition of single term lists when these can be reconstructed using information held in previous entries. This leads to an extra space saving and a more efficient use of the memory, exchanged for some extra computational cost.

The idea is best illustrated with an example. In Fig. 2 we show the SU data representation [13] (entries 1 and 2) and the "extra" improvements we propose (lines 3 and 4) to get an even more efficient storage. In line 1, the entry for terms t_1 and t_2 is shown. This contains the DocIDs for the first term only $(t_1 - (t_1 \bigcap t_2))$, then the postings of the second term only $(t_2 - (t_1 \bigcap t_2))$, and finally the last area with the postings common to both terms (i.e. the intersection $(t_1 \bigcap t_2)$). Entry in line 2 is similar to the previous one. In line 3, we show an entry that contains a previously cached term, t_1 (i.e. in the first intersection). To avoid the repetition of part of the postings we propose to reconstruct the full posting list of t_1 from the first entry (with a computational cost overhead) and include in the entry a redirection (ϕ). Although we incur in an extra cost to reconstruct the posting list, this is cheaper than loading it from disk.

5 Selecting the Pairs of Terms

We consider several strategies to select the "best" intersections (bigrams) to keep in cache. To this purpose, each postings list is weighted according to the $freq(t_i) \times |t_i|$ product, where $freq(t_i)$ is the raw frequency of term t_i in a query log training set and $|t_i|$ is the length of the posting list of term t_i in the reference

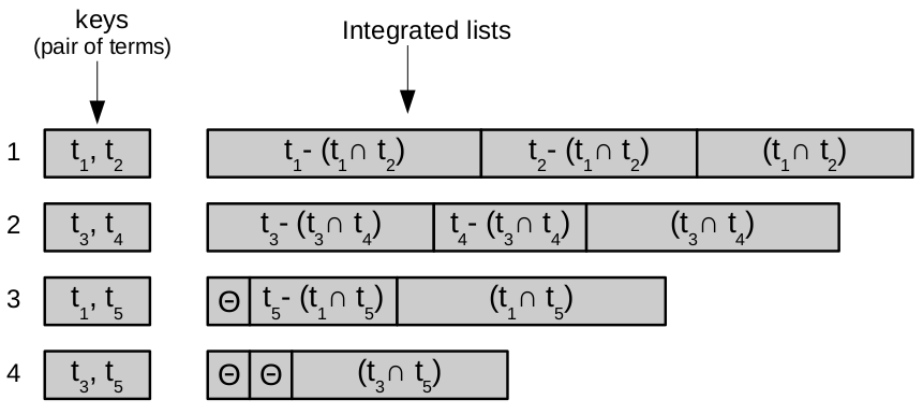

Fig. 2. Data Structure used for the Integrated Cache

collection. Experimental evidence shows that this approach outperforms solely frequency based selection criteria. Hereafter, we refer to this metric as FxS.

Greedy methods. We start with a naive approach that considers the ordering of the best posting lists sorted according to FxS product and we form up bigrams pairing together two consecutive terms as $(1^{st}, 2^{nd}), (3^{rd}, 4^{th}), ..., ((n-1)^{th}, n^{th})$. We refer to this method as *PfBT-seq*. This approach only groups good terms but doesn't consider the size of their intersection (while if the size of the intersection is larger, the space saving is larger too).

The second approach (*PfBT-cuad*) computes the intersection of each possible bigram (for all term lists) and then selects the pairs that maximize $(t_i \bigcap t_j)$ without repetitions of terms. This algorithm is time consuming $(O(n^2))$ so we run it considering only sub-groups of lists that we estimate may fit in cache (according to its size). For example, the 1GB cache holds roughly 1000 lists for a given collection, so we compute the 500 best pairs and then we fill the remaining space with pairs picked sequentially (as in *PfBT-seq*).

The third approach (named *PfBT-win*) is a particular case of the previous one that tries to maximize the space saving among a group of posting lists. It sets a window of w terms (instead of all terms) and computes the intersection of each possible pair. Finally, it selects the pairs using the same criterion as before.

Term pairing as a matching problem. Our last method considers the term pairing as an optimization problem, reducing it to the Maximum Weighted Matching (MWM). In graph theory, a matching is a subset of edges such that none of the selected edges share a common vertex. This is similar to [13] but we apply a different weighting criterion. We formalize the problem as follows: Let $G(T, E)$ be a graph with vertex set the set T of terms and such that, for every $t_i, t_j \in T$, edge $e_{ij} \in E$ exists if and only if $|t_i \bigcap t_j| > 0$. Moreover, we weight each edge e_{ij} by the size of the intersection $|t_i \bigcap t_j|$. The MWM is a matching in G that maximizes the sum of the weights of the matched edges.

We use the method proposed by Galil [10] to solve the MWM problem. This is an exact algorithm based on the *blossom* method by Edmonds. Although this algorithm runs in $O(n^3)$ time, the size of our graphs (thousand of vertices at most) makes it computationally tractable. In our experiments we refer to this method as *PfBT-mwm*.

Note that all the above strategies select *pairs* of terms to fill the cache, so there will not be situations like the ones depicted in lines 3 and 4 of Fig. 2. We plan to consider other strategies (for example dynamic) in the future that will take benefit of this idea.

6 Experimental Setup

6.1 Datasets

We select two completely different document collections to evaluate the *Integrated Cache*. Our goal is to simulate two scenarios whose behaviors may be rather distinct. The first document corpus is a subset of a large web crawl of the UK obtained by Yahoo! in 2005. It includes 1.479.139 documents, with 6.493.453 distinct index terms and takes 29 GB of disk space in HTML format (uncompressed). We refer to this corpus as UK. The second collection is a crawl derived from the Stanford WebBase Project[1] [12]. We select a new sample (march, 2013) that contains about 7.774.632 documents and takes 241 GB of disk space in uncompressed HTML format. Table 1 shows statistics about the two collections.

	UK	**WB**
# documents	1.479.140	7.774.631
avg(document size) (bytes)	20.555	29.793
avg(terms)/doc	564	1.176
avg(unique terms)/doc	218	384

Table 1. Collections statistics

6.2 Query log

To evaluate the proposal we use the well known AOL Query Log [20] that contains around 20 million queries. We select a subset of 6 millions queries to compute statistics and around 2.7 millions queries as the test set (AOL-1). Then, we filter the file keeping only unique queries. This allows to isolate the effect of the *Result Cache* simulating that it captures all the query repetitions (in the case of having a cache of infinite size), thus giving a lower bound of the performance improvement due to our cache. This second test file is about 800K queries (AOL-2).

[1] http://dbpubs.stanford.edu:8091/ testbed/doc2/WebBase/

7 Methodology

We use Zettair[2] to index the collections and to obtain real fetching times of the posting lists. The size of the (compressed) index for the UK collection is about 1.8 GB, which grows up to 23 GB for WB. Zettair compresses posting lists using a variable-byte scheme with a B+Tree structure that efficiently handles the vocabulary. Each entry in the posting lists includes both a DocID and the terms frequency in that document (used for ranking purposes).

Our integrated cache implementation reserves eight bytes for each posting in the pure terms area (the DocID and the frequency uses four bytes each) while the intersection area occupies twelve bytes because it stores the separated frequencies of both terms.

7.1 Cost model

We model the cost of processing a query in a node in terms of disk fetch and CPU times: $C_q = C_{disk} + C_{cpu}$. C_{disk} is calculated fetching all the terms from disk using Zettair (we retrieve the whole posting list and measure the corresponding fetching time). To calculate C_{cpu} we run a list intersection benchmark on the same machine. This cost estimation methodology is also used in [8]. We adopt this approach because running real tests is too expensive considering our dataset.

We run the experiments on a machine with a Intel(R) Core(TM)2 Quad CPU (Q9550) processor running at 2.83 GHz and 8 GB of main memory. The hard disk is a mid-range SATA with 320 GB. The operating system is Debian Linux (kernel 3.2) and all the programs are coded in C++ and compiled with gcc 4.6.3.

8 Experiments

In this section, we provide a simulation-based evaluation of the proposal using both document collections.

Experimental setup. The total amount of memory reserved for the cache ranges from 100MB to 1GB for the UK collection while we increase the size up to 16GB for the WB collection. This sizes allow to store about 60% and 70% of the UK and WB indexes respectively.

For each query we log the total cost incurred using a static version of the *List Cache* filled with the top-k most valuable posting lists according to the FxS metric (this is our *baseline*). Then, we evaluate the *Integrated Cache* filling it with data from the proposed four approaches that are also based in the FxS metric (to allow a fair comparison against the list cache). We set $w = 10$ in our experiments for the *PfBT-win* method. A deeper analysis of the optimal value of w will be part of future work. Finally, we normalize the final costs to get a more clear comparison.

Results. Figure 3 shows cost results for the AOL-1 query set. All evaluated strategies outperform the baseline and the best strategy is PfBT-mwm. Figure

[2] http://www.seg.rmit.edu.au/zettair/

4 shows the improvement that *Integrated Cache* achieves while increasing cache sizes. Improvements range from 8% to 23% for the biggest cache size. However, for the WB collection the behavior is slightly different. For small cache sizes, the improvements are not significant but they grow up from 1 GB of cache space up to 38% for the best case. This is because this collection has longer posting lists and only a few are loaded for the small caches.

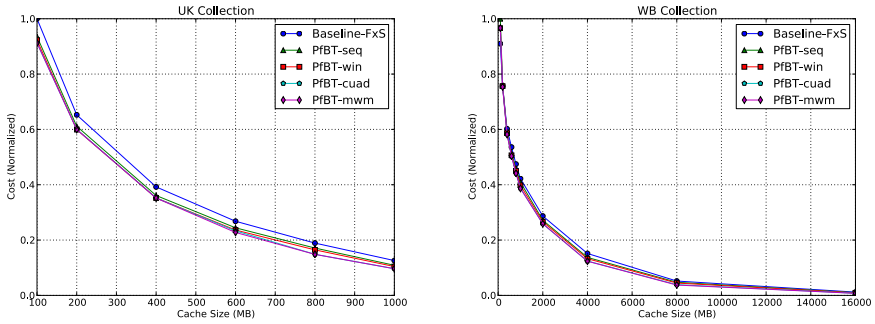

Fig. 3. Performance of the different approaches using the the AOL-1 query set

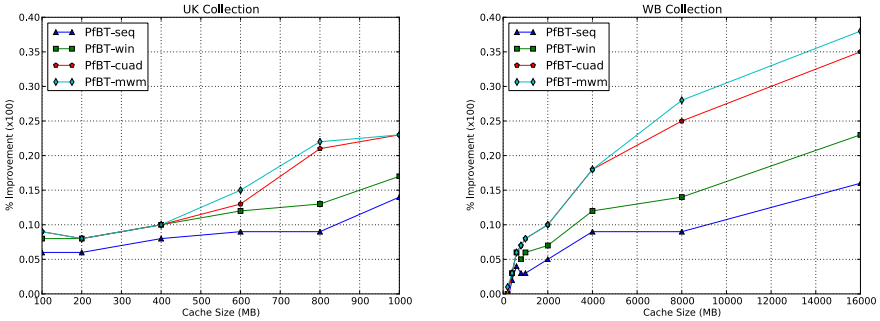

Fig. 4. Improvements of the different approaches (AOL-1 query set)

In the second experiment (see Figure 5) using the dataset of unique queries (AOL-2) we evaluate the best strategy according to the previous results (PfBT-mwm). As we expected, performance is slightly worse, because no repeated queries are present. Improvements range from 7% up to 22% for the UK collection. The behavior is again different for the WB collection. For smaller cache sizes, the performance is worse (or just slightly better) up to 1GB cache and it increases up to 30% in the best case (16GB).

9 Conclusion and Future Work

In this paper, we proposed an integrated cache for lists+intersections for text search systems. We used a paired data structure along with a resolution strategy

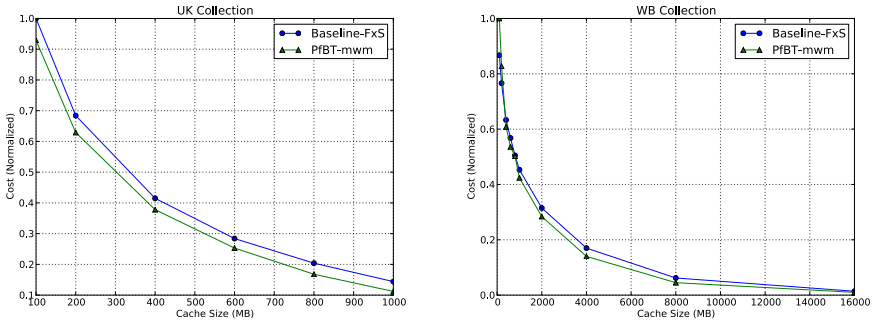

Fig. 5. Performance of the PfBT-mwm approach using the the AOL-2 query set

that takes advantage of the intersection cache. We considered several heuristics to populate the integrated cache, including one based on casting the problem as a maximum weighted matching one. We provided an evaluation using two completely different document collections and two subsets of a real query log. We showed that the proposed *Integrated Cache* outperforms the solely posting lists cache up to a 40%.

There are several interesting remaining open problems. First, we plan to extend this proposal to consider trigrams or more complex combinations of terms. Another interesting open question concerns the design and implementation of a dynamic version of this cache. Here, the access and eviction policies should consider not only the terms but also the pairs. It is not clear how to best apply standard replacement algorithms in an online fashion.

References

[1] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *Proc. of the 30th annual Int. Conf. on Research and Development in Information Retrieval*, SIGIR '07, pages 183–190, USA, 2007.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval: The Concepts and Technology behind Search*. Addison-Wesley Prof., Inc., 2nd edition, 2011.

[3] B. B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *Proc. of the third ACM Int. Conf. on Web search and data mining*, WSDM '10, pages 411–420, USA, 2010.

[4] J. S. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. of the 14th International Conf. on String Processing and Information Retrieval*, SPIRE'07, pages 137–148, Berlin, Heidelberg, 2007.

[5] J. Dean. Challenges in building large-scale information retrieval systems: Invited talk. In *Proc. of the Second ACM International Conf. on Web Search and Data Mining*, WSDM '09, pages 1–1, New York, NY, USA, 2009. ACM.

[6] S. Ding, J. Attenberg, R. Baeza-Yates, and T. Suel. Batch query processing for web search engines. In *Proc. of the Fourth ACM International Conf. on Web Search and Data Mining*, WSDM '11, pages 137–146, New York, NY, USA, 2011.

[7] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historicalusage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, Jan. 2006.

[8] E. Feuerstein and G. Tolosa. Analysis of cost-aware policies for intersection caching in search nodes. In *Proc. of the XXXII Conf. of the Chilean Society of Computer Science*, SCCC'13, 2013.

[9] E. Feuerstein and G. Tolosa. Cost-aware intersection caching and processing strategies for in-memory inverted indexes. In *In Proc. of 11th Workshop on Large-scale and Distributed Systems for Information Retrieval*, LSDS-IR'14, New York, 2014.

[10] Z. Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Comput. Surv.*, 18(1):23–38, Mar. 1986.

[11] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *Proc. of the 18th Int. Conf. on World wide web*, WWW '09, pages 431–440, 2009.

[12] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. Webbase: A repository of web pages. In *Proc. of the 9th International World Wide Web Conf. on Computer Networks*, pages 277–293, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[13] H. T. Lam, R. Perego, N. T. Quan, and F. Silvestri. Entry pairing in inverted file. In *Proc. of the 10th International Conf. on Web Information Systems Engineering*, WISE '09, pages 511–522, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *Proc. of the 14th Int. Conf. on World Wide Web*, WWW '05, pages 257–266, USA, 2005.

[15] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[16] E. Markatos. On caching search engine query results. *Comput. Commun.*, 24(2):137–143, Feb. 2001.

[17] S. Melink, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. *ACM Trans. Inf. Syst.*, 19(3):217–241, July 2001.

[18] R. Ozcan, I. S. Altingovde, and O. Ulusoy. Cost-aware strategies for query result caching in web search engines. *ACM Trans. Web*, 5(2):9:1–9:25, May 2011.

[19] R. Ozcan, I. Sengor Altingovde, B. Barla Cambazoglu, F. P. Junqueira, and O. Ulusoy. A five-level static cache architecture for web search engines. *Information Processing & Management*, 48(5):828–840, Sept. 2012.

[20] G. Pass, A. Chowdhury, and C. Torgeson. A picture of search. In *Proc. of the 1st International Conf. on Scalable Information Systems*, InfoScale '06, New York, NY, USA, 2006. ACM.

[21] P. C. Saraiva, E. Silva de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *Proc. of the 24th annual Int. Conf. on Research and Development in Information Retrieval*, SIGIR '01, pages 51–58, USA, 2001.

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

[23] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes (2Nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[24] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. of the 17th Int. Conf. on World Wide Web*, WWW '08, pages 387–396, USA, 2008.

[25] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006.