Noname manuscript No.
(will be inserted by the editor)

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

Gabriel Tolosa · Luca Bechetti · Esteban Feuerstein · Alberto Marchetti-Spaccamela

Received: date / Accepted: date

Abstract Modern information retrieval systems use several levels of caching to speedup computation by exploiting frequent, recent or costly data used in the past. Previous studies show that the use of caching techniques is crucial in search engines, as it helps reducing query response times and processing workloads on search servers. In this work we propose and evaluate a static cache that acts simultaneously as list and intersection cache, offering a more efficient way of handling cache space. We also use a query resolution strategy that takes advantage of the existence of this cache to reorder the query execution sequence. In addition, we propose effective strategies to select the term pairs that should populate the cache. We also represent the data in cache in both raw and compressed forms and evaluate the differences between them using different configurations of cache sizes. The results show that the proposed *Integrated Cache* outperforms the standard posting lists cache in most of the cases, taking advantage not only of the intersection cache but the query resolution strategy too.

Keywords information retrieval systems · caching · integrated cache · performance improvement

Gabriel Tolosa
University of Buenos Aires & National University of Luján, Argentina
E-mail: tolosoft@unlu.edu.ar

Luca Bechetti
Sapienza University of Rome, Italy
E-mail: becchetti@dis.uniroma1.it

Esteban Feuerstein
University of Buenos Aires, Argentina E-mail: efeuerst@dc.uba.ar

Alberto Marchetti-Spaccamela
Sapienza University of Rome, Italy
E-mail: alberto@dis.uniroma1.it

1 Introduction

Modern large-scale information retrieval systems such as Web Search Engines (WSE) exploit sophisticated techniques for efficiency and scalability purposes: they crawl and index tens of billions of documents (thus managing a huge inverted index file) and must answer queries in fast (in a few hundred milliseconds) to satisfy users' expectations.

It is known that the main contributions to the cost of a query are processing time (C_{cpu}) and disk access times (C_{disk}); if many parallel machines cooperate to answer the query, a communication cost (C_{comm}) must be also considered. 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 [Cambazoglu et al (2010)] and higher precision levels. On the other hand, C_{disk} involves fetching from hard disk the posting lists (usually compressed) of all the query terms. Finally, C_{comm} involves moving data and synchronization messages between different nodes in the system through the network.

Usually, the architecture of a WSE is formed by a front-end node called *broker* and by a large number of machines (*search nodes* organized in a cluster that process queries in parallel [Barroso et al (2003)].

Each *search node* holds only a fraction of the document collection and maintains a local inverted index that is used to obtain a high query throughput. Given a cluster of p search nodes and a collection of C documents and assuming that these are evenly distributed among the nodes, each one maintains an index with information related to only C/p documents. Besides, to achieve the strong performance requirements, WSE usually implement different optimization techniques such as posting list compression [Zhang et al (2008)], list pruning [Macdonald et al (2011)], results prefetching [Jonassen et al (2012)] and caching [Baeza-Yates et al (2007)]. In these cases, caching is one of the most important and crucial tools to achieve fast response times and to increase query throughput.

The main goal of a cache is to speedup computation by allowing fast access to a suitably chosen (frequent, recently used or costly) set of data. The typical architecture of a search engine involves different cache levels (Figure 1): 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 re-computation 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, which are very expensive compared to CPU processing times. If a list is found in cache, then disk cost is avoided.

Another approach is *Intersection Caching* that requires caching portions of a query (e.g., pairs of terms), as initially proposed in [Long and Suel (2005)] and extended in [Ding et al (2011)]. The idea in this case is to exploit term co-occurrence patterns, e.g., by keeping the intersection of the postings lists

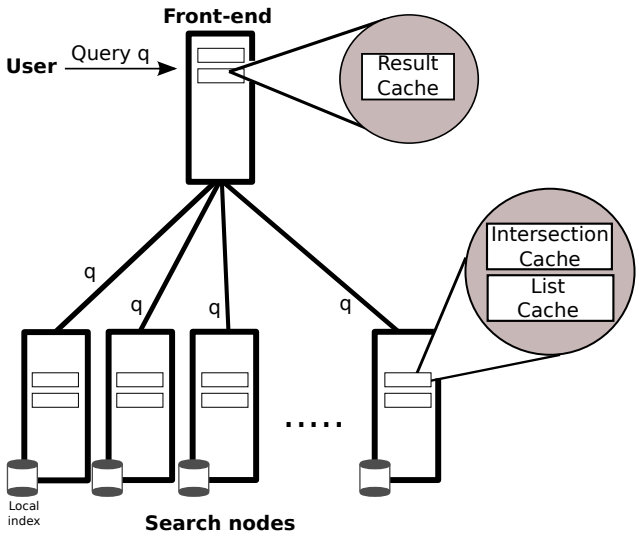

Fig. 1: Search Engine Architecture considered in this work

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.

In the case of industry-scale web search engines, the entire inverted index is usually stored in main memory [Dean (2009)]. Under this configuration, the List Cache becomes useless but the Intersection Cache is still useful [Feuerstein and Tolosa (2014)] 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 memory while the remaining fraction is stored in secondary storage (i.e. HDD of SDD). In this cases, effective caching architectures becomes essential to mitigate disk latencies, so List and Intersection Caches are both 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. 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 or more 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 this aim, we adapt a data structure previously proposed by Lam et al. [Lam et al (2009)]. 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. However, as queries submitted to a search engine have significantly varying costs in terms of several aspects (e.g., CPU processing time, disk access time, etc.) and the frequency of the query is not an indicator of its cost we consider cost-aware caching strategies [Cao and Irani (1997)] [Young (1998)] which provide further gains, as shown in [Ozcan et al (2011)] and [Feuerstein and Tolosa (2013)]. Roughly, cost-aware caching selects the item to be evicted taking into account the costs of each intersection. The approach is based on the observation that cache misses do not have the same cost and caching policies that only consider frequency of the items may not always lead to optimum performance.

Although this approach already reduces the size of the resulting data structure, compressing the inverted index (namely, each of its posting lists) is a crucial tool used to improve query throughput and fast response times in WSEs. Data is usually kept in compressed form in memory (or disk) leading to a reduction in its size that would typically be about 3 to 8 times, depending on index structure, stored information and compression method. Usually, document identifiers, frequency information and positions are stored separately and can be compressed independently, even with different methods. These techniques have been studied in depth in the literature [Witten et al (1999)], [Manning et al (2008)], [Baeza-Yates and Ribeiro-Neto (2011)]. Among the compression methods for posting lists we mention the classical Elias [Elias (2006)] and Golomb [Golomb (1966)] encoding and the more recent ones Simple9 [Anh and Moffat (2005)], and PForDelta [Zukowski et al (2006)] encodings. In this work, we also evaluated the compressed version of our proposal using the state-of-the art PForDelta method which improves the tradeoff between compression ratio and decompression speed.

1.1 Our Contribution

This paper includes and extends our preliminary work [Tolosa et al (2014)], where we introduced the *Integrated Cache*, a static cache that resides at search nodes and replaces both list and intersection caches in a single memory space. To this aim, we use a particular data structure that makes an efficient use of memory space. Moreover, we design a specific cache management strategy that avoids the duplication of cached terms and we adapt a query resolution

strategy that tries to maximize the hit ratio, introduced in [Feuerstein and Tolosa (2014)].

We also consider different strategies to populate the integrated cache and we propose three strategies that consider both frequency of term co-occurrence and postings list size. Furthermore, we propose a novel 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 different characteristics and a well known query log over a simulation framework. Rather than hit ratio, we consider as a performance metric the overall time needed by the different strategies to process all queries. Experimental evidence shows that substantial savings are possible using the proposed approach.

We boost our Integrated Caching framework with a compressed version, and we extend the evaluation comparing this new framework against list caching. Some structural characteristics of the input imply that compression methods yield lower compression ratios in this application. However, the evaluation shows that compression introduces savings that are still highly useful to reduce the overall processing time.

The remainder of this paper is organized as follows: in the next section, we review related work. In Section 3, we provide some background about query processing and related data structures along with a description of the datasets used in the analysis and experiments. In Section 4 Integrated Cache is presented; methods for selecting the items to be placed in the cache are discussed in Section 5. The subsequent section is devoted to the experimental setup and the experimental results are summarized in Section 7. Finally, we introduce the conclusions of our work in Section 8 followed by future work.

2 Related Work

There is a large body of work devoted to caching in text search systems. In the sequel we summarize the work more relevant to our proposal.

2.1 Posting Lists Caching

In [Baeza-Yates et al (2007)] Baeza et al. analyze the problem of simply caching posting lists (combined with results caching) that achieves higher hit rates than simply caching query results. They also propose an algorithm called $Q_{tf}D_f$ that selects the terms to put in cache according to their $\frac{frequency(t)}{size(t)}$ ratios. 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 and simulate different architectures both in LAN and WAN environments. Inverted index compression and list caching techniques are explored in [Zhang

et al (2008)]: several inverted list compression algorithms and caching policies are compared. Namely [Zhang et al (2008)] compares LFU, LRU, Optimized Landlord, Multi-Queue, ARC.

2.2 Results Caching

Markatos (2001)] is the first paper introducing 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 in the policy and proposes LRU-2S (two stages LRU) that tries to capture both variables. In 2006, Fagni et al. [Fagni et al (2006)] 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 [Gan and Suel (2009)] 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 [Young (1998)]. The main result is the study of the weighted case with the goal of optimizing the processing cost. In [Ozcan et al (2011)], cost aware strategies for result caching are extensively evaluated. Authors observe that cache misses have different costs, and popularity-aware caching policies can not always minimize the total savings. To overcome this limitation, they propose to incorporate the query costs into the caching policies and evaluate them in both static, dynamic and hybrid cases.

2.3 Multilevel Caching

Saraiva et al. propose a two-level caching scheme that combines caching of search results with the caching of frequently accessed postings lists [Saraiva et al (2001)]. Subsequently, Long and Suel extend the idea to caching intersections of pairs of terms that are often co-used [Long and Suel (2005)]: they introduce a three-level caching architecture for a web search engine (results+intersections+posting lists). The cache is disk-based (about 20% of the total index space) and the main idea is to save processing cost for high co-occurrent pairs of terms. They apply a greedy algorithm to select which items to store in cache, and then evaluate a landlord-based policy. In a more recent work, Ozcan et al. [Ozcan et al (2012)] introduce a 5-level static caching architecture.

Further studies regarding cost-aware intersection caching are presented in [Feuerstein and Tolosa (2013)] and [Feuerstein and Tolosa (2014)]. 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.

2.4 Posting List Compression

Efficient access to the lists data structure is a key aspect for a search system, mainly when the index resides in hard disk. Many compression techniques have been developed and evaluated to deal with long lists of integers that represent the document identifiers and complementary information associated with each term. For example, classic techniques such us Variable-Byte Encoding [Williams and Zobel (1999)] and Simple 9 [Anh and Moffat (2005)] or PForDelta [Zukowski et al (2006)] and its optimized versions are commonly used.

In [Zhang et al (2008)] the authors explore the combination of inverted index compression and list caching to improve search efficiency. They compare several inverted list compression algorithms and list caching policies separately and finally study the benefits of combining both, exploring how this benefit depends on hardware parameters (i.e. disk transfer rate and CPU speed).

In [Catena et al (2014)] the authors analyze the performance of modern integer compression schemes across different types of posting information (document identifiers, frequencies and positions). They analyze the space and time efficiency of the search engine compressing different types of posting information. They show that the simple Frame of Reference [Goldstein et al (1998)] codec achieves the best query response times in all the cases, slightly outperforming PForDelta.

3 Preliminaries

In the sequel we provide the background about the data structures and algorithms used to solve a query in a distributed search system. We also provide the details of the datasets used in the subsequent studies and the evaluation of the proposed algorithms. In general, we consider a query $q = \{t_1, t_2, t_3, ..., t_n\}$ as a set of terms that represents the users information need, expressed as the intersection $\bigcap_1^n \ell_i$, where ℓ_i corresponds to the posting list of term t_i.

3.1 Background

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 [Zobel and Moffat (2006)]. Basically, it contains 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* according to different solving strategies. As we observed, the inverted index is usually stored in compressed form.

Gap encoding (DGap) is used when the lists are sorted by DocID, [Witten et al (1999)]. This basically works as follows: the first document identifier is represented as it is, whereas the remaining identifiers are represented as the difference with the previous one. For instance: let $\ell_i = \{22, 28, 48, 49, 50, 51, 52, 67\}$, its DGap'ed version becomes $\ell'_i = \{22, 6, 20, 1, 1, 1, 1, 5\}$. This representation is particularly useful because the compression methods benefit from sequences of small integers because they may be represented more compactly (i.e., using short codes). Often, inverted lists are logically divided into blocks (i.e. 128 DGaps each) and skip lists are used [Melink et al (2001)] to speed up its traversal when searching for a particular DocID. This enables the decompression of only those blocks that are relevant to a particular search. There is a considerable body of literature on index construction ([Baeza-Yates and Ribeiro-Neto (2011), Witten et al (1999), Zobel and Moffat (2006)]).

Query Processing: The processing of queries in a distributed search system as shown in Figure 1 is usually done as follows [Cambazoglu et al (2010)]. First 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 a distributed version of the inverted index resides. Each search node fetches the posting lists of the query terms (from disk or cache), orders them 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 document-identifiers is sent to the broker which merges it with the lists of other nodes to obtain the final answer. Loading lists from disk is a time consuming task that is critically important 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.

A second phase of the computational process consists typically of taking, at the broker level, the top ranked answers to make the final result page. This is composed of titles, snippets and URL information for each resulting item. As the result page is typically made of 10 documents the cost of the processing may be considered constant.

To solve a query there exist two main strategies, namely Term-at-a-time (TAAT) and Document-at-a-time (DAAT) [Turtle and Flood (1995)]. In the TAAT approach, the posting lists of the query terms are sequentially evaluated, starting from the shortest to the longest one. 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 [Culpepper and Moffat (2007)] is an efficient strategy for DAAT processing while the WAND strategy [Broder et al (2003)] is a dynamic pruning approach that allows fast query processing for both conjunctive and disjunctive queries.

However, the use of an *Intersection Cache* enables other possibilities such as the S4 strategy introduced in [Feuerstein and Tolosa (2014)]. 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. Concretely, this strategy works as follows:

1. The query is first decomposed in all possible two-term combinations (or intersections, I_{ij}).
2. All intersections I_{ij} that are present in the cache are assigned to a candidate-set, C.
3. C is sorted according to the size of the intersections, from shortest to longest ($C\prime$).
4. A *new* query is built by picking first the intersections from $C\prime$.
5. The remaining terms (those that are not found in any cached intersection) are added to the query.

Each time a query is evaluated we need to check $\binom{n}{2}$ candidate pairs. However, the distribution of the number of terms in a query shows that 95% have up to 5 terms, so the computational cost overhead incurred doing this (across all the queries in the log) becomes negligible.

As an example consider the query $q = \{t_1, t_2, t_3, t_4\}$ and suppose that $(t_2 \cap t_3), (t_3 \cap t_4)$ are cached intersections. We set $A \leftarrow (t_2 \cap t_3)$ and $B \leftarrow (t_3 \cap t_4)$ and rewrite the query as $(A \cap B) \cap t_1$ which only needs to fetch and intersect the posting list of t_1. It is shown that the S4 strategy allows a performance improvement up to 30% combining it with cost aware cache policies.

3.2 Datasets

We use real datasets of documents and queries to carry out the corresponding analysis of compression performance, establish baselines, populate the cache and evaluate our proposal. We select two different document collections with different characteristics: the first one is a subset of a large crawl of the UK web obtained by Yahoo! in 2005. The second collection is a crawl derived from the Stanford WebBase Project[1] [Hirai et al (2000)]. We select a rather recent sample (March, 2013), bigger that the previous collection. In the following we refer to these collections as UK and WB respectively. Table 1 summarizes collection statistics.

We use the AOL Query Log [Pass et al (2006)] that contains around 20 million queries. All queries are processed using standard approaches: terms are converted to lower case, stopwords are not eliminated and no stemming algorithm is applied. Besides, we eliminate all queries that do not match the

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

Dataset	Documents	Total Terms	Unique Terms	Size GB	Index Size GB
UK	1,479,139	834,510,076	6,493,453	29.1	2.1
WB	7,774,632	9,143,511,516	110,838,794	241.0	23.0

Table 1: Collection Statistics. "Size" corresponds to the uncompressed documents in HTML format.

vocabulary of each document collection. From the resulting query-set we select a subset of 6M queries to compute statistics and around 2.7M queries as the test set (AOL-1). Table 2 summarizes query-log statistics. Then, we filter the file keeping only unique queries. This allows to isolate the effect of the Result Cache simulating that it captures all query repetitions (in the case of having a cache of infinite size), thus giving a lower bound on the performance improvement due to our cache. This second test file is about 800K queries (AOL-2).

Queries		Pairs	
Total	**Unique**	**Total**	**Unique**
11,972,277	5,722,691	42,301,175	10,011,656

Table 2: Number of total and unique queries and pairs in the query log.

This dataset has the benefits of being publicly available (and widely used) and corresponds to real user queries but the indexed collections do not belong to the same data source. However, the UK collection is contemporary with the query log (it was crawled in 2005 while the AOL Query log was released in 2006). Besides, we try to improve the coherence between documents and queries by eliminating all queries that do not match the vocabulary of each document collection. This data setup is appropriate to assess efficiency (see [Webber and Moffat (2005)].

4 Integrated Cache

In [Lam et al (2009)] the authors propose the paired data representation to design an integrated cache that behaves as list and intersection cache at the same time. The authors propose an index compression technique based on pairing posting lists of frequently co-occurring terms obtaining a new *paired* list in the inverted index, thus obtaining a compact representation that may reduce query processing time. This data structure is introduced as a compression technique for inverted indexes combined with Gamma Coding and Variable Byte Coding [Baeza-Yates and Ribeiro-Neto (2011)] schemes.

Our main contribution is to extend the approach to an integrated cache of both lists and intersections. We use the "Separated Union" (SU) [Lam et al (2009)] representation to maintain an in-memory data structure, which

replaces both the list and intersection caches. Regarding space savings, the main idea is to keep in cache those pairs of terms that maximize the hit ratio of the List Cache and the savings of the most valuable precomputed intersections. We also avoid the repetition of single term lists when these can be reconstructed using information held in previous entries. This leads to extra space savings and a more efficient use of memory, at the expense of some extra computational cost.

This caching scheme is particularly useful in the case of retrieval systems that store only a fraction of the inverted index in main memory (according to the available computing resources), leaving the remaining portion in secondary storage. Under this consideration, both List and Intersection Caches are helpful to reduce disk access latencies and processing time, so we move a step forward by combining them with the main goal of using the memory space in a more efficient way. However, the success of the Integrated Cache relies on good strategies to select and combine individual terms that help to save memory space while using *good* terms (according to some metric, such as their individual access frequency). The methods we propose to select the term-pairs to populate the cache work under these considerations. We are going to introduce them in short.

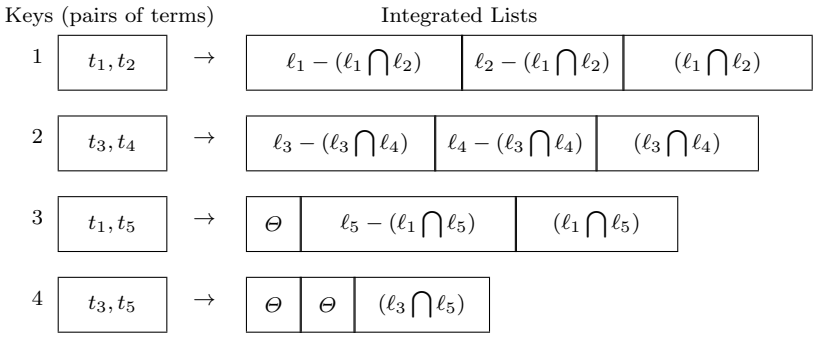

Fig. 2: Data Structure used for the *Integrated Cache*.

Before presenting the details of the data structure we illustrate the idea with an example. Figure 2 shows the SU data representation (entries 1 and 2) and the improvements we propose (entries 3 and 4). Let's assume ℓ_i represents the inverted list of t_i. Line 1 shows the entry for terms t_1 and t_2; the line contains the DocIDs for the first term only $(\ell_1 - (\ell_1 \bigcap \ell_2))$, then the postings of the second term only $(\ell_2 - (\ell_1 \bigcap \ell_2))$, and, finally, the last area with the postings common to both terms (i.e. the intersection $(\ell_1 \bigcap \ell_2)$). Line 2 is analogous for terms t_3 and t_4. Note that obtaining the posting list of any single term requires an extra computational cost for merging lists of the entry; however this is cheaper than loading it from disk.

Line 3 shows the entry that contains a previously cached term, t_1, (that already appears in the first intersection). To reduce the memory requirement of the line we avoid the repetition of part of the postings; namely, we propose to reconstruct the full posting list of t_1 from the first entry and include in the entry a redirection (Θ). All redirections are kept in a simple lookup table whose structure consists of $(t_i \rightarrow (pair))$ and can be accessed in $O(1)$ time using a hash function. This table is created while filling the static cache during the initialization step, using Algorithm 1. At running time, it is possible to test the cache looking for a pair (t_i, t_j) or a single term (t_i) using Algorithm 2. If we want to cache a single term, lets say t_1, the list is completely stored in the first area and the remaining two are kept empty $(t_1 \rightarrow |\ell_1|\phi|\phi|)$.

Algorithm 1: Insert item in cache

Input: IC: Integrated Cache, (t_1, t_2): Key (pair of terms), L (posting lists), RT: Redirection table
Output: IC: Integrated Cache, RT: Redirection table
$pair \leftarrow (t_1, t_2)$
$IC\{pair\} \leftarrow L$
if *(!exists(RT, t_1))* **then**
  $RT\{t_1\} \leftarrow (pair)$
end
if *(!exists(RT, t_2))* **then**
  $RT\{t_2\} \leftarrow (pair)$
end
return IC, RT;

The proposed data structure replaces both the intersection and posting list caches at the search node level. It is initially intended to support conjunctive (AND) queries but it also enables the possibility of solving disjunctive queries (OR) by simply joining the corresponding posting lists. For instance, to solve the query $q = \{t_1, t_2, t_3\}$ in disjunctive way (using the same data shown in Figure 2) it is sufficient to join the lists of t_1 and t_2 (line 1) with the list of t_3 (line 2). Clearly, if one of the lists is not present in the cache, it must be fetched from disk.

The query resolution strategy first decomposes the query in pairs of terms. Each pair is checked in the Integrated Cache and the final resolution order is given by first considering the pairs that are present in the cache and afterwards intersecting them with the remaining ones. "Separate" terms (i.e. terms that are not present in any pair in the cache) are also checked in the Integrated Cache as a single term using the redirection table.

For example, given a query $q = \{t_1, t_3, t_4, t_6\}$, their corresponding posting lists (ℓ_i) and the configuration of the Intersection Cache as shown in Figure 2, the query is solved in the following way:

1. The pairs are checked in cache and the final resolution order becomes: $(((\ell_3 \bigcap \ell_4) \bigcap \ell_1) \bigcap \ell_6)$.
2. The intersection $(\ell_3 \bigcap \ell_4)$ is recovered from cache.

3. The posting list of t_1 is recovered from the first entry of the cache using the redirection table (working as a posting list cache).
4. The posting list of t_6 is recovered from disk.

Algorithm 2: Test item in cache

Input: IC: Integrated Cache, RT: Redirection table, p: search pattern ($p = (t_i, t_j)$ when looking for an intersection or $p = t_i$ in the case of a single term)
Output: List r: The results list (if p is found in cache) or [ ] (otherwise)

```
r = [];
if (isPair(p)) then
    if (found(IC, p)) then
        r ← IC{p}[(ℓ_i ⋂ ℓ_j)]
    end
else
    if (found(RT, p)) then
        (t_x, t_y) ← (RT{p})
        if (p == t_x) then
            r ← IC{(t_x, t_y)}[(ℓ_x \ (ℓ_x ⋂ ℓ_y))] ⋃ IC{(t_x, t_y)}[(ℓ_x ⋂ ℓ_y)]
        else
            r ← IC{(t_x, t_y)}[(ℓ_y \ (ℓ_x ⋂ ℓ_y))] ⋃ IC{(t_x, t_y)}[(ℓ_x ⋂ ℓ_y)]
        end
    end
end
return r;
```

Starting from (3), the corresponding list is intersected with the resulting one of the previous step. In this example, t_1 incurs in the cost of joining the contents of the first and third areas ($(\ell_1 - (\ell_1 \bigcap \ell_2))$ and $(\ell_1 \bigcap \ell_2)$, respectively) of the first entry in the integrated cache to reconstruct its full posting list. Finally, only term t_6 incurs in disk access cost.

4.1 Compressing the Integrated Cache

In the standard inverted index postings lists are represented separately while in the Integrated Cache representation postings lists are paired[2] in each entry. This leads to some differences which affect the compression ratio when compressing the data structure. As we observed, DGap encoding is usually used to represent posting lists because this compression techniques is effective in compressing lists of relatively small integers.

In the case of the Integrated Cache the original sequence is modified as shown in the following example. Let $\ell_i = \{10, 11, 12, 13, 15, 18\}$ and $\ell_j = \{11, 15, 21, 23\}$ be the posting lists of terms t_i and t_j respectively. In the case of

[2] The framework also supports the representation of three-term intersections at the expense of that structure of the integrated lists becomes more complex.

an inverted index, the DGap'ed representation becomes: $\ell'_i = \{10, 1, 1, 1, 2, 3\}$ and $\ell'_j = \{11, 4, 6, 2\}$ which are then compressed. However, the Integrated representation of a pair of terms (t_i, t_j) is as follows:

$(\ell_i \setminus \ell_j) = \{10, 12, 13, 18\}$
$(\ell_j \setminus \ell_i) = \{21, 23\}$
$(\ell_i \bigcap \ell_j) = \{11, 15\}$

and its DGap'ed version $(t_i, t_j)'$ becomes:

$(\ell_i \setminus \ell_j)' = \{10, 2, 1, 5\}$
$(\ell_j \setminus \ell_i)' = \{21, 2\}$
$(\ell_i \bigcap \ell_j)' = \{11, 4\}$

The comparison of both representations shows that ℓ'_i and ℓ'_j can be better compressed using DGaps because these lists contain smaller values than $(t_i, t_j)'$.

In order to get a better understanding of this effect, we index the WB dataset and compute the DGap distribution of both standard compressed posting lists and the integrated (also compressed) representation. In this analysis, we fill up the Integrated Cache according to a competitive strategy used to select *useful* pairs of terms (in the next section we describe this approach in detail). Figure 3 plots the distribution of DGap values of both representations. Both series of data follow a power-law distribution $f(x) = Cx^{-\beta}$ with parameter $\beta = 1.34$ and $\beta = 2.17$ for lists and integrated respectively. In the figure, we can observe that DGap'ed lists have more runs of 1's (and lower values) than integrated ones.

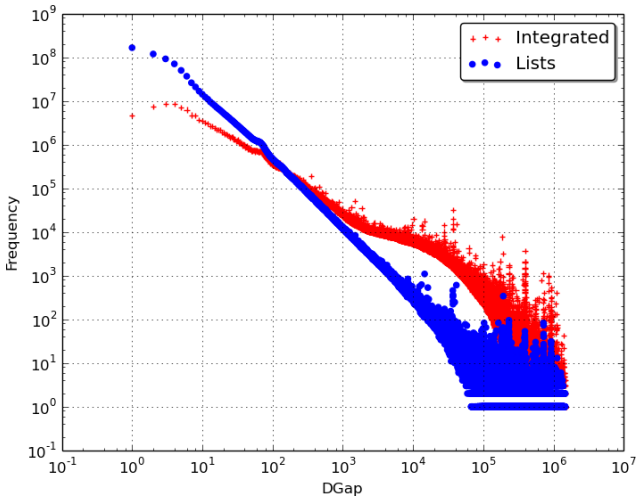

Fig. 3: DGap value distributions for the Integrated and standard Posting Lists

However, the performance comparison of the two proposals requires to analyze the compression ratios in both cases. Let ℓ_i denote the posting list of term t_i and I_{ij} an entry in the Integrated cache that represents the pair $x = (t_i, t_j)$. Recall that this representation becomes:

$$I_{ij} = \ell_i - (\ell_i \bigcap \ell_j) \cup \ell_j - (\ell_i \bigcap \ell_j) \cup (\ell_i \bigcap \ell_j)$$

Then, we denote their compressed forms as $C(\ell_i)$ and $C(I_{ij})$ respectively. To compare the space used for each representation we use the ratio:

$$\delta_x = \frac{|C(I_{ij})|}{|C(\ell_i)| + |C(\ell_j)|} \tag{1}$$

When $\delta_x < 1$, compressing the Integrated entry is more space-efficient than compressing the posting lists separately. Otherwise, it uses more space than the sum of both compressed lists. We should keep in mind that the Integrated representation stores a precomputed intersection between the terms that may be still highly useful to save computation time. It is quite clear that the Integrated representation is more space-efficient when the intersection $(\ell_i \bigcap \ell_j)$ is bigger.

However, there is a drawback when compressing I_{ij}. The split of each list ℓ_i in two pieces $(\ell_i - (\ell_i \bigcap \ell_j))$ and $(\ell_i \bigcap \ell_j)$ separates sometimes consecutive DocIDs, leading to shorter runs of 1's in the DGap representation. As a result of this split, $C(I_{ij})$ gets a lower compression ratio than $(C(\ell_i) + C(\ell_j))$ in some cases. Therefore, we analyze the δ ratio as a function of the size of the "Intersection Size" (IS) ratio defined as:

$$IS_{ij} = \frac{|\ell_i \bigcap \ell_j|}{|\ell_i| + |\ell_j|} \tag{2}$$

For each pair of terms in the dataset we compute both IS_{ij} and δ ratios. Figure 4 shows the results. Values of $\delta_x > 1$ correspond to a worse compression rate of the integrated representation with respect to the compression of the separated lists. Looking in detail those values we find that 97% of them occur when the IS_{ij} ratio is smaller than 0.15. This means that when $IS_{ij} > 0.15$, the integrated representation is still more space-efficient than the compression of the lists (ℓ_i and ℓ_j). This observation shows that the size of the intersection is a good criterion to select pairs of terms to insert into the Integrated cache (the bigger $\ell_i \bigcap \ell_j > 0.15$, the better).

We also compute the efficiency in the space usage for the Integrated Cache with respect to the separated posting lists in both uncompressed and compressed representations. To this aim, we join the top-m posting lists in different groups (where m grows in powers of 10) according to competitive strategies (as we previously mentioned) and sum the length of the standard (separated) posting lists ($|\ell_i + \ell_j|$) and the integrated representation ($|Iij|$) respectively. Finally, we calculated the ratio between both. Table 3 shows the uncompressed representations. The entries in the table show that the Integrated Cache makes better use of the space (that decreases according to the downsizing of the IS_{ij}

ratio). The compressed versions of both representations show a similar behaviour (Table 4). However, the ratio is slightly worse in this case due to the lower compression performance of the integrated representation ($C(I_{ij})$) as a consequence of the DGap value distribution (described above). The performance of compressing individual lists is about 8.8% better (on average) than the compression of the Integrated Cache.

# of entries	$\sum \lvert\ell_i\rvert + \lvert\ell_j\rvert$	$\sum \lvert I_{ij}\rvert$	*ratio*
10	9.797.866	5.060.928	0.5165
100	44.227.078	23.899.513	0.5404
1.000	143.357.823	108.769.302	0.7587
10.000	457.975.054	404.064.223	0.8823
100.000	706.712.370	653.568.221	0.9248

Table 3: Sum of posting lists lengths for the uncompressed representation. For different groups of the top-k lists we show the sum of the lengths of: a) individual lists, b) the integrated representation and, c) the corresponding ratio.

# of entries	$\sum C\lvert\ell_i\rvert + C\lvert\ell_j\rvert$	$\sum \lvert C(I_{ij})\rvert$	*ratio*
10	720.395	409.170	0.5680
100	4.570.464	2.685.710	0.5876
1.000	19.833.629	16.817.892	0.8479
10.000	81.141.977	77.938.818	0.9605
100.000	133.297.699	129.155.629	0.9689

Table 4: Sum of posting lists lengths for the compressed representation. For different groups of the top-k lists we show the sum of the lengths of: a) individual lists, b) the integrated representation and, c) the corresponding ratio.

The implementation of the uncompressed version of the Integrated Cache reserves eight bytes for each posting in the pure terms area (DocID and frequency uses four bytes each) while the intersection area requires twelve bytes because it stores the frequencies of both terms. DocIDs and frequencies are stored separately to favor the compression process. This allows us to select different compression codecs for each type of data.

5 Selecting Term Pairs

In this section we describe different approaches used to select the pairs of terms to fill up the cache. We propose a static posting list cache populated with those lists that maximize a particular function. We consider the $Q_{tf}D_f$ algorithm [Baeza-Yates et al (2007)] that is one of the best algorithm for maximizing hit rate and a variant of it in which each postings list is weighted according to the $f(t_i) \times |\ell_i|$ product. In this expression, $f(t_i)$ is the raw frequency of term t_i in a query log training set and $|\ell_i|$ is the length of the posting list of term t_i in the reference collection. Hereafter, we refer to this metric as FxS.

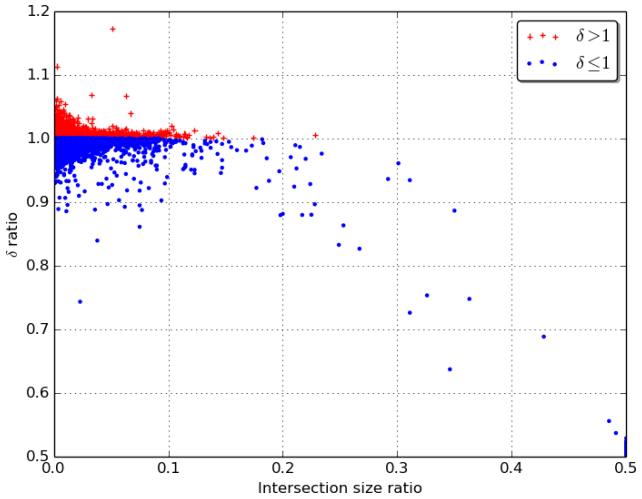

Fig. 4: Integrated cache compression performance: δ_x ratio scatter plot. x-axis in log scale to get a clearer view.

We consider several strategies to select the "best" intersections (bigrams) to keep in cache. According to the analysis introduced in the previous section the "best" intersections to populate the cache are those that maximize the size of the intersection $\ell_i \bigcap \ell_j$. We also consider the FxS product that weight each posting list in the baseline method.

5.1 Greedy Methods

We first consider a simple approach that orders posting lists according to their FxS products (base list) and then merges lists by pairing together consecutive term pairs as $(1^{st}, 2^{nd}), (3^{rd}, 4^{th}), ..., ((n-1)^{th}, n^{th})$. The construction time of this method is $O(n)$ because it requires a single scan through the list of terms. We refer to this method as *PfBT-seq*. Note that this approach groups terms considering their individual FxS products, that becomes a cost effective approach, but it does not take into account the size of their intersections.

The second approach (*PfBT-quad*) computes the intersection of each possible bigram and then selects the pairs that maximize $|t_i \bigcap t_j|$ without repetitions of terms.

As this algorithm is time consuming ($O(n^2)$), we run it considering only sub-groups of lists that we estimate to fit in cache (according to their size). This approach works as follows: given a cache of size B we pick a number of lists from the base list (FxS, sorted in decreasing order) such that their total size exceeds B ($\sum |\ell_i| > B$) and compute *PfBT-quad* on this set to reduce the number of combinations to work on. For example, 1GB cache holds roughly 1000 individual lists for a given collection. So, we compute the top-500 pairs according to the size of their intersection (the bigger, the better) and then, if

there is still remaining space, we fill it with pairs picked sequentially (using the same criteria 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) selected from the base list and computes the intersection of each possible pair. The intention behind this approach is to bound the computational cost required to get the best candidate term pairs. Finally, it selects the definitive term pairs using the same criterion as before.

The last greedy approach we propose, is one that chooses the pairs of terms that maximize the value $w_{ij} = |\ell_i \bigcap \ell_j| \times f(t_i, t_j)$, where $f(t_i, t_j)$ is the frequency of the pair (t_i, t_j) in the query log. This approach offers two differences with regard to the previous ones: On one hand, it weights each pair considering its historical frequency. On the other hand, it allows that some terms may be repeated in different pairs (e.g. term t_5 in line 3 of Fig. 2). However, the redirection strategy in the implementation of the Integrated Cache avoids the use of extra space in the memory. This last greedy approach is named *PfBT-greedy-sf*.

5.2 Term Pairing as a Matching Problem

The fifth approach (*PfBT-mwm-is*) considers the term pairing as an optimization problem, reducing it to the Maximum Weighted Matching (MWM). 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 $t_i, t_j \in T$, there is edge $e_{ij} \in E$ exists with weight $|t_i \bigcap t_j|$ if and only if $|t_i \bigcap t_j| \geq 0$. We cache the pairs of terms corresponding to edges of the maximum weighted matching in G.

Following the same idea than before (*PfBT-quad*), we compute the matching considering only sub-groups of lists that we estimate to fit in cache (according to their size), by picking a number of lists from the baseline (FxS) whose total size exceeds the considered cache size B ($\sum |\ell_i| > B$). Using this strategy the matching is computed on a subset of a priori *good* lists (according to their FxS product). In our experiments we use the matching algorithm proposed by Edmonds [Edmonds (1965)] for general graphs. This is an exact algorithm that runs in $O(n^3)$ time; however the size of our graphs (thousands of vertices) makes it computationally tractable. The algorithm is executed offline: the solution gives the set of pairs of terms that are used to populate the static cache.

This approach is similar to that proposed in [Lam et al (2009)] but we apply a slightly different weighting criteria. While in that work the weight e_{ij} measures the benefit of pairing two terms (in number of bits) and considering the encoding method used for representing the lists, we directly define the weight as the size of the intersection $|t_i \bigcap t_j|$.

We also consider a variation of the previous MWM approach using the same objective function as the *PfBT-greedy-sf* approach. That is, the weight

of each edge of the graph is here $w'_{ij} = |\ell_i \bigcap \ell_j| \times f(t_i, t_j)$, where $f(t_i, t_j)$ is the frequency of the pair (t_i, t_j) in the query log. We name this last approach *PfBT-mwm-sf*.

6 Experimental setup

We use Zettair[3] 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 to handle the vocabulary.

As we mentioned before, the Integrated Cache reserves eight bytes for each posting in the pure terms area (DocID and frequency use four bytes each) while the intersection area requires twelve bytes because it stores the frequencies of both terms (DocIDs and frequencies are stored separately). In this case, we compress each part of the Integrated Cache in two steps using different codecs: by one hand, DocIDs are compressed using the PForDelta codec while frequencies are compressed using variable-byte encoding, as in Zettair. This process requires to add pointers to the beginning of each compressed part to allow the reconstruction of the original lists.

We use the cost estimation methodology introduced in [Feuerstein and Tolosa (2013)] to evaluate the cost of solving a set of queries. In this experiments, we consider conjunctive (AND) queries. The cost of processing a query in a node is modeled in terms of disk fetch and CPU times: $C_q = C_{disk} + C_{cpu}$. The first parameter, 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 compute C_{cpu} we run a list intersection benchmark using the well-known Zipper algorithm. It scans both lists as in a binary merging operation to solve the intersection and runs in $O(n + m)$ time. As we studied the effects of DGap encoding and compression ratio in the Integrated Cache we decided to use the mentioned algorithm because it harmonizes well with the two techniques.

To validate this methodology we run a comparison against a real system implemented on the top of the Zettair search engine, using a posting list cache and a small sample of 100.000 queries. The results lead to differences around 2.5% on average between the simulation and the real system and the correlation is $R^2 = 0.9991$ which we consider acceptable. The results are shown in Figure 5.

We provide a simulation-based evaluation of the proposal using both document collections. The total amount of memory reserved for the cache ranges from 100MB to 1GB for the UK data set, and from 100MB to 16GB for the WB data set. A cache of 16 GB stores about 60% and 70% of the indexes respectively. For each query we log the total costs incurred using a static version

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

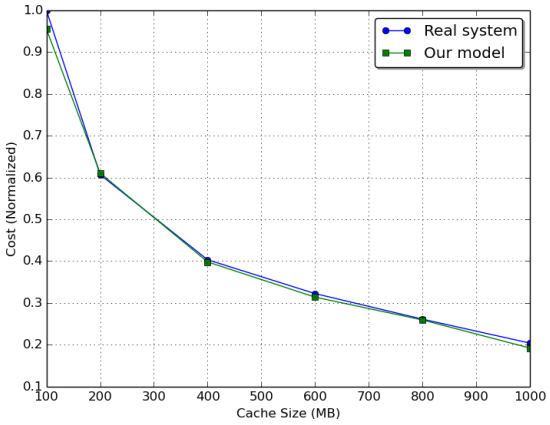

Fig. 5: Comparison between the simulation methodology and a real system

of the List Cache, filling it with the most valuable posting lists and the four proposed methods to fill data using *Integrated Cache*; in both case we use the FxS metric for comparison. We set $w = 10$ for the *PfBT-win* method.

It is important to note that we do not use the cache hit-ratio as an evaluation metric because of the intrinsic nature of the S4 strategy that prevents how to establish the correct relationship between hits and misses, thus modeling the accurate behaviour of the cache. Finally, we normalize the final costs with respect to the highest one to get a clearer comparison.

7 Results

7.1 Integrated Cache with uncompressed data

We compared all approaches using the two document collections and the two query sets. For the AOL-1 query set we test all the approaches against the baseline, the standard posting list cache sorted according the FxS score. In our setup, experimental evidence shows that FxS outperforms $Q_{tf}D_f$ when measuring cost (instead of hit-rate). All evaluated strategies outperform the baseline and the best strategy is PfBT-mwm-sf. This improvements increase up to 55% and 66% for the UK (Figure 6) and WB (Figure 7) collections, respectively. The greedy method PfBT-greedy-sf performs efficiently for cache sizes up to 4 GB (WB collection) with an average improvement close to 30%.

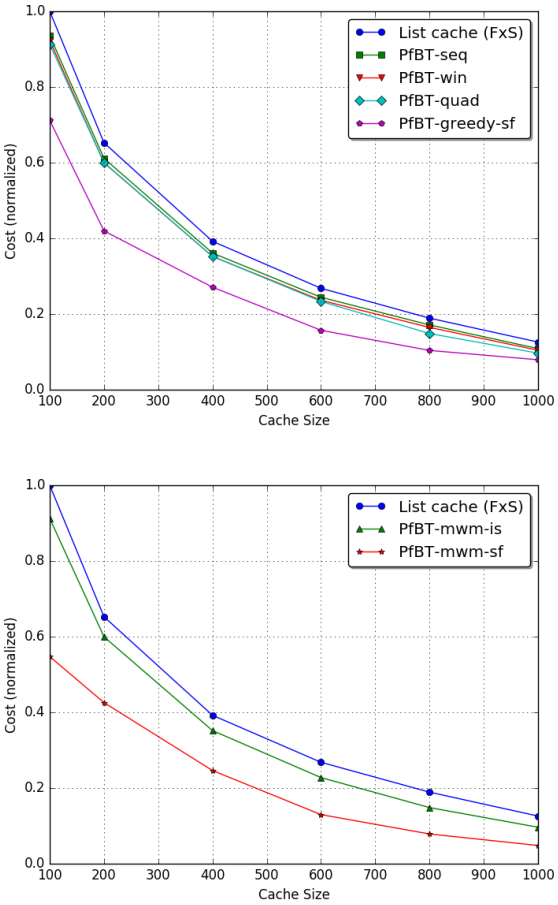

Fig. 6: Performance of the different approaches using the the AOL-1 query set and the UK collection: greedy methods (top) and MM-based (bottom).

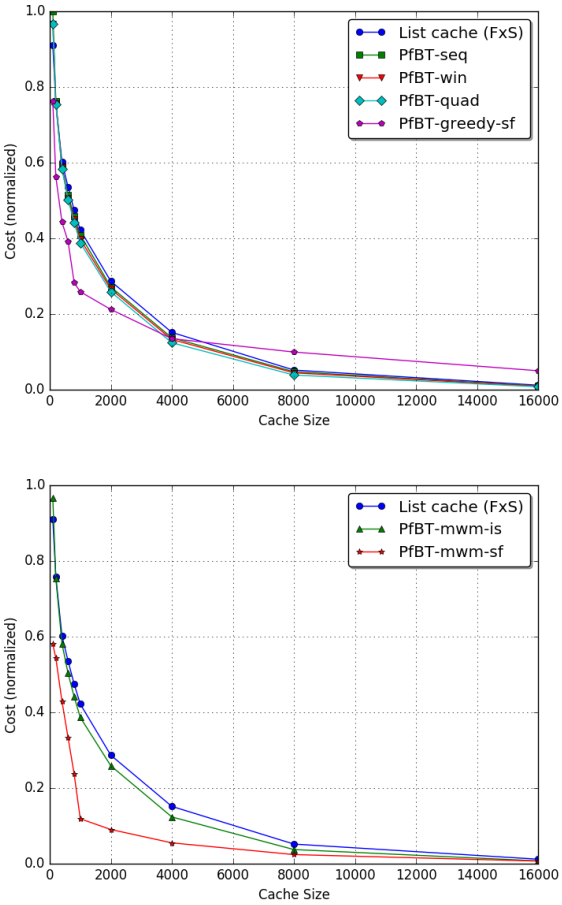

Fig. 7: Performance of the different approaches using the the AOL-1 query set and the WB collection: greedy methods (top) and MM-based (bottom).

We complement the analysis comparing average query times for the baseline method with respect to the greedy approach that obtains the peak performance improvement (PfBT-greedy-sf) and the best maching-based method (PfBT-mwm-sf), using the WB collection and four cache sizes. Figure 8 shows the results. The best method reduces both average query time and time dispersion in all cases, while the greedy method becomes efficient up to a cache size 4GB and gets a worse performance for bigger ones, as shown in Figure 7.

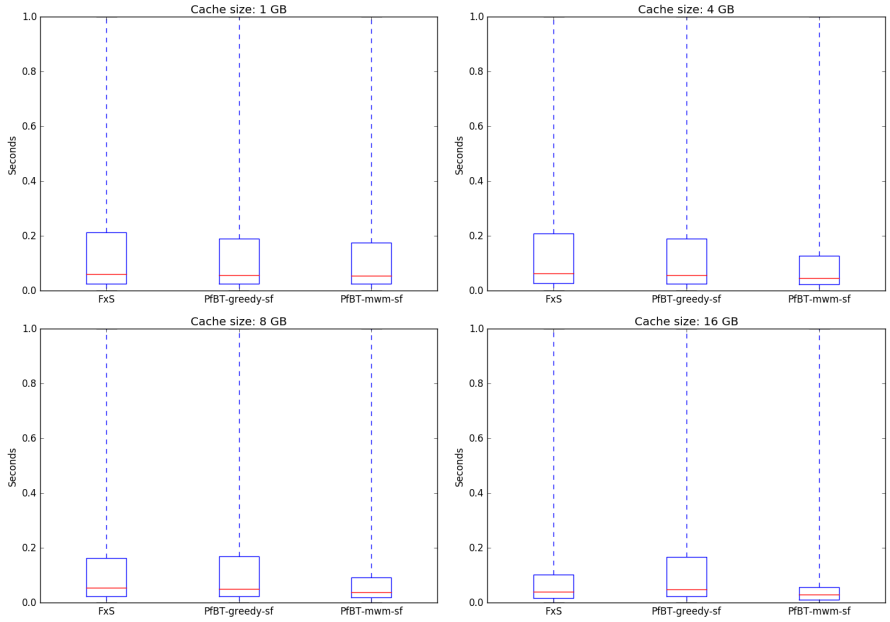

Fig. 8: Average query times for baseline, best greedy and best matching-based methods using the WB collection, the AOL-1 query log and four cache sizes.

In the same experiment, we analyze the impact of the Integrated Cache in a separate way, that is, when requesting intersections or individual terms to the cache. We found that 97% of cache hits corresponds to individual terms while the remaining 3% corresponds to intersections. This is not surprising because the algorithms used to select the pairs to populate the cache *joins* terms that maximize the length of their intersection to optimize the cache memory occupancy. However, 94% of the total cost saved due to cache hits corresponds to individual terms while the remaining 6% is given by hits in intersections. This is an interesting result that reinforces the idea that measuring only cache hits do not necessarily reflects the impact of caching in the total cost savings. So, the Integrated Cache not only offers the advantages of a traditional List Cache, but adds extra savings working as an Intersection Cache, as well.

In a second experiment we used the dataset of unique queries (AOL-2) and the best strategy obtained from the previous test (*PfBT-mwm-sf*). Improve-

ments 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). This is because this collection has longer posting lists and only a few are loaded in smaller caches. These results are shown in Fig. 9.

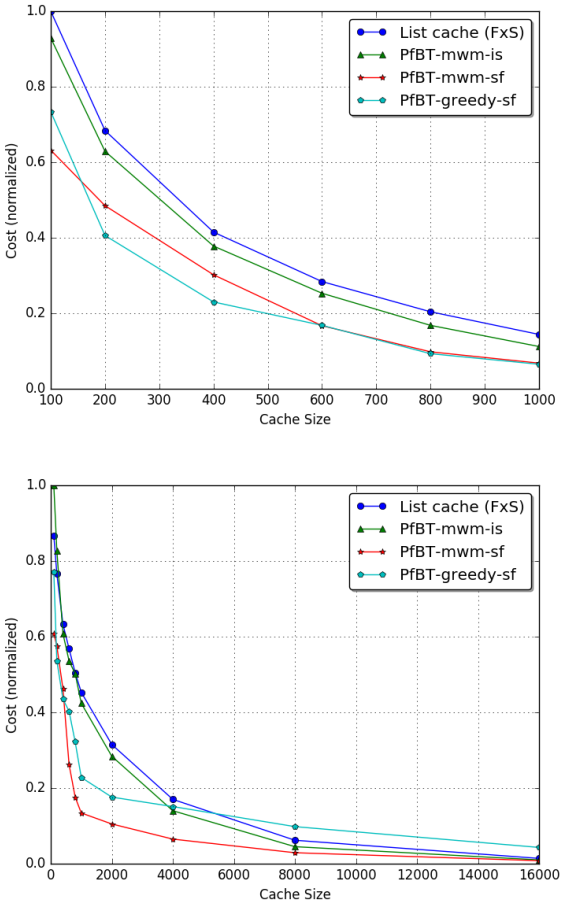

Fig. 9: Performance of the best approaches using the the AOL-2 query set for the UK collection (top) and WB collection (bottom).

7.2 Integrated Cache with compressed data

Compression techniques are used to reduce the size of each inverted list thus enabling the possibility of storing more data in memory. For this reason we evaluate the performance of our method when inverted lists are compressed. In this case we compare the baseline method (FxS) against three methods for selecting term pairs: the best greedy algorithm and the two MM-based approaches.

In these experiments we used the WB collection which contains longer posting lists than UK that may still require lots of space in the cache when compressed. Figure 10 (top) shows the results for the AOL-1 query set. The best strategy in this case becomes *PfBT-mwm-sf* with improvements up to 70% in the best case (cache sizes between 800 MB and 1GB). *PfBT-greedy-sf* is also much better than the baseline but worse than *PfBT-mwm-is* for cache sizes greater than 2 GB.

In the case of the second query set (AOL-2) the behaviour is different (Figure 10, bottom). The best strategy is *PfBT-mwm-is* in all cases (up to 57% better than the baseline). The second strategy *PfBT-mwm-fs* only outperforms the baseline for cache sizes greater than 800 MB, while the *PfBT-greedy-sf* do not perform well (on average).

7.3 Integrated Cache and Result Cache

To obtain a complete evaluation of the effectiveness of Integrated Cache we consider another set of experiments in which the Integrated Cache is combined with a Result Cache. For this reason we add a dynamic LRU-based Result Cache to our simulation framework. LRU is a popular recency-based approach that chooses the least recently used items for replacement when the cache is full and works well in different domains.

In this experiments, we assume a Result Cache of limited size, contrasting with the experiments that use the filtered AOL-2 query set that simulates a cache of infinite size. We consider two cache sizes (250K and 500K) for the experiments and compare the same strategies used in the previous section. Figure 11 shows the results for cache sizes of 250K and 500K entries.

All the strategies outperform the baseline for the result cache of 250K entries. Both new strategies that include the frequency of the pair in its objective function (*PfBT-mwm-sf* and *PfBT-greedy-sf*) are the best on average except for the last case (16GB of cache size) where the simplest approach gets the better results. According to [Skobeltsyn et al (2008)], more sophisticated strategies are better when the cache capacity is small, due to the optimized space usage. However, when the cache capacity is big enough a simpler strategy is still useful because the hit rate of the result cache approaches to its upper bound.

Although there are some differences in the remaining case (500K entries), the results show that *PfBT-mwm-sf* and *PfBT-greedy-sf* are again the best

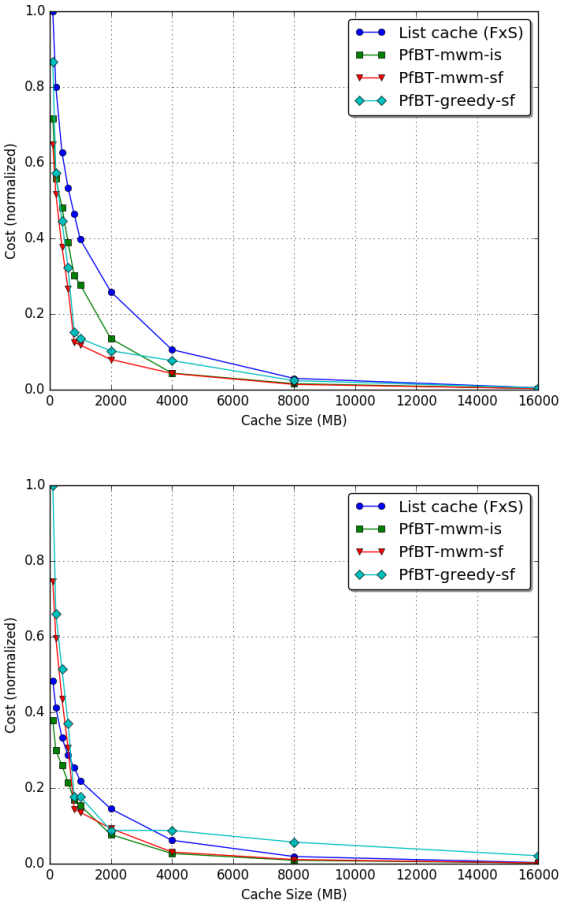

Fig. 10: Performance of the different approaches using compressed data (WB collection) for AOL-1 (top) and AOL-2 (bottom) query sets.

strategies. The best performance improvement is achieved with an Integrated Cache of 4GB, achieving an improvement close to 70%. When considering all the settings and cache sizes, *PfBT-mwm-sf* becomes the best strategy with a performance improvement around 49% (averaged over the entire range of cache sizes).

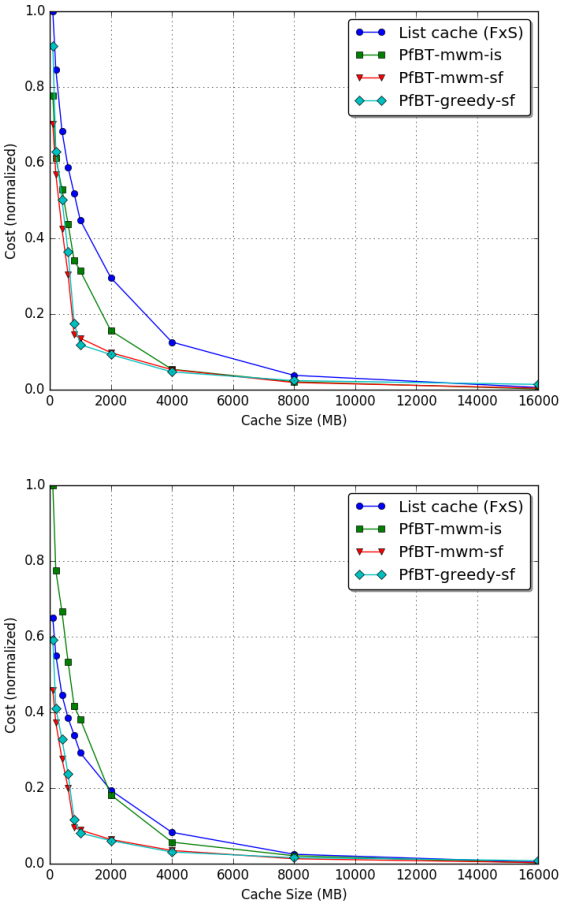

Fig. 11: Performance of the different approaches using compressed data (WB collection) and the AOL-1 query set, combined with a Result Cache of 250K entries (top) and 500K entries (bottom).

8 Conclusions and Future Work

We have proposed *Integrated Cache* a method to improve the performance of a search system using the memory efficiently to store the lists of pairs of terms based on a paired data structure along with a resolution strategy that takes advantage of an intersection cache. We consider several heuristics to populate the cache, including one based on casting the problem as a maximum weighted matching one. We provide an evaluation of our architecture using two different document collections and subsets of a real query log considering several scenarios. We also represent the data in cache in both raw and compressed forms and evaluate the differences between them using different configurations of cache sizes. Besides, we consider the existence of a Result Cache in the architecture of the search system that filters out a significant number of repeated queries. The experimental results show that the proposed method outperforms the standard posting lists cache in most of the cases, taking advantage not only of the intersection cache but the query resolution strategy too.

Several interesting open problems remain. First, it would be interesting to extend the approach by considering trigrams and other combinations of terms. Besides, the evaluation of different compression encoders according to the distribution of the DocIDs in the integrated representation is an issue for future work. It would be interesting to model this problem analyzing the space-time tradeoff. Another interesting open question concerns the design and implementation of a dynamic version of this cache. Here, the admission and eviction policies should contemplate not only the terms but also the pairs.

Acknowledgements This work was partially supported by EU-IRSES project EUSACOU 247574, by EU FET project MULTIPLEX 317532 and byUBACyT Project 20020120100058 "Herramientas algorítmicas avanzadas para aplicaciones de búsqueda en Internet - Parte 2".

References

Anh VN, Moffat A (2005) Inverted index compression using word-aligned binary codes. Inf Retr 8(1):151–166

Baeza-Yates R, Ribeiro-Neto B (2011) Modern Information Retrieval: The Concepts and Technology behind Search, 2nd edn. Addison-Wesley Prof., Inc.

Baeza-Yates R, Gionis A, Junqueira F, Murdock V, Plachouras V, Silvestri F (2007) The impact of caching on search engines. In: Proc. of the 30th annual Int. Conf. on Research and Development in Information Retrieval

Barroso LA, Dean J, Hölzle U (2003) Web search for a planet: The google cluster architecture. IEEE Micro 23(2):22–28

Broder AZ, Carmel D, Herscovici M, Soffer A, Zien J (2003) Efficient query evaluation using a two-level retrieval process. In: Proceedings of the Twelfth International Conference on Information and Knowledge Management, ACM, New York, NY, USA, CIKM '03, pp 426–434

Cambazoglu BB, Zaragoza H, Chapelle O, Chen J, Liao C, Zheng Z, Degenhardt J (2010) Early exit optimizations for additive machine learned ranking systems. In: Proc. of the third ACM Int. Conf. on Web search and data mining

Cao P, Irani S (1997) Cost-aware www proxy caching algorithms. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems, USENIX Association, Berkeley, CA, USA, USITS'97, pp 18–18, URL http://dl.acm.org/citation.cfm?id=1267279.1267297

Catena M, Macdonald C, Ounis I (2014) On inverted index compression for search engine efficiency. In: de Rijke M, Kenter T, de Vries A, Zhai C, de Jong F, Radinsky K, Hofmann K (eds) Advances in Information Retrieval, Lecture Notes in Computer Science, vol 8416, Springer International Publishing, pp 359–371

Culpepper JS, Moffat A (2007) Compact set representation for information retrieval. In: Proc. of the 14th International Conf. on String Processing and Information Retrieval, Berlin, Heidelberg, SPIRE'07, pp 137–148, URL http://dl.acm.org/citation.cfm?id=1778666.1778679

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

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

Edmonds J (1965) Maximum matching and a polyhedron with 0, 1 vertices. J of Res the Nat Bureau of Standards 69 B:125–130

Elias P (2006) Universal codeword sets and representations of the integers. IEEE Trans Inf Theor 21(2):194–203

Fagni T, Perego R, Silvestri F, 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

Feuerstein E, Tolosa G (2013) 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

Feuerstein E, Tolosa G (2014) 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, New York, LSDS-IR'14

Gan Q, Suel T (2009) Improved techniques for result caching in web search engines. In: Proc. of the 18th Int. Conf. on World wide web, WWW '09, pp 431–440

Goldstein J, Ramakrishnan R, Shaft U (1998) Compressing relations and indexes. In: Proceedings of the Fourteenth International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, ICDE '98, pp 370–379, URL http://dl.acm.org/citation.cfm?id=645483.656226

Golomb S (1966) Run-length encodings. In: IEEE Trans Info Theory, vol 12

Hirai J, Raghavan S, Garcia-Molina H, Paepcke A (2000) Webbase: A repository of web pages. In: Proc. of the 9th International World Wide Web Conf. on Computer Networks, North-Holland Publishing Co., URL http://dl.acm.org/citation.cfm?id=347319.346288

Jonassen S, Cambazoglu BB, Silvestri F (2012) Prefetching query results and its impact on search engines. In: Proc. of the 35th Int. Conf. on Research and Development in Information Retrieval, USA, SIGIR '12, pp 631–640

Lam HT, Perego R, Quan NT, Silvestri F (2009) Entry pairing in inverted file. In: Proc. of the 10th International Conf. on Web Information Systems Engineering, Springer-Verlag, Berlin, Heidelberg, WISE '09, pp 511–522

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

Macdonald C, Ounis I, Tonellotto N (2011) Upper-bound approximations for dynamic pruning. ACM Trans Inf Syst 29(4):17:1–17:28

Manning CD, Raghavan P, Schütze H (2008) Introduction to Information Retrieval. Cambridge University Press, New York, NY, USA

Markatos E (2001) On caching search engine query results. Comput Commun 24(2):137–143

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

Ozcan R, Altingovde IS, 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, Sengor Altingovde I, Barla Cambazoglu B, Junqueira FP, Ulusoy O (2012) A five-level static cache architecture for web search engines. Information Processing & Management 48(5):828–840

Pass G, Chowdhury A, Torgeson C (2006) A picture of search. In: Proc. of the 1st International Conf. on Scalable Information Systems, ACM, InfoScale '06

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

Skobeltsyn G, Junqueira F, Plachouras V, Baeza-Yates R (2008) Resin: A combination of results caching and index pruning for high-performance web search engines. In: Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, New York, NY, USA, SIGIR '08, pp 131–138

Tolosa G, Becchetti L, Feuerstein E, Marchetti-Spaccamela A (2014) Performance improvements for search systems using an integrated cache of lists+intersections. In: Proceedings of the 21st International Symposium on String Processing and Information Retrieval - Volume 8799, Springer-Verlag New York, Inc., New York, NY, USA, SPIRE 2014, pp 227–235

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

Webber W, Moffat A (2005) In search of reliable retrieval experiments. In: ADCS 2005, Proceedings of the Tenth Australasian Document Computing Symposium, December 12, 2005, pp 26–33

Williams HE, Zobel J (1999) Compressing integers for fast file access. The Computer Journal 42:193–201

Witten IH, Moffat A, Bell TC (1999) Managing Gigabytes (2Nd Ed.): Compressing and Indexing Documents and Images. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

Young NE (1998) On-line file caching. In: Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, SODA '98, pp 82–86, URL http://dl.acm.org/citation.cfm?id=314613.314658

Zhang J, Long X, Suel T (2008) Performance of compressed inverted list caching in search engines. In: Proc. of the 17th Int. Conf. on World Wide Web, USA, WWW '08, pp 387–396

Zobel J, Moffat A (2006) Inverted files for text search engines. ACM Comput Surv 38(2)

Zukowski M, Heman S, Nes N, Boncz P (2006) Super-scalar ram-cpu cache compression. In: Proceedings of the 22Nd International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, ICDE '06, pp 59–