Distributed Search based on Self-Indexed Compressed Text

Diego Arroyuelo[a], Veronica Gil-Costa[a,c], Senén González[a], Mauricio Marin[a,b], Mauricio Oyarzún[b]

[a] *Yahoo! Research Latin America, Santiago of Chile*
[b] *DIINF, University of Santiago of Chile*
[c] *CONICET, National University of San Luis, Argentina*

Abstract

Query response times within a fraction of a second in Web search engines are feasible due to the use of indexing and caching techniques, which are devised for large text collections partitioned and replicated into a set of distributed memory processors. This paper proposes an alternative query processing method for this setting, which is based on a combination of self-indexed compressed text and posting lists caching. We show that a text self-index (i.e., an index that compresses the text and is able to extract arbitrary parts of it) can be competitive with an inverted index if we consider the whole query process, which includes index decompression, ranking and snippet extraction time. The advantage is that within the space of the compressed document collection, one can carry out the posting lists generation, document ranking and snippet extraction. This significantly reduces the total number of processors involved in the solution of queries. Alternatively, for the same amount of hardware, the performance of the proposed strategy is better than that of the classical approach based on treating inverted indexes and corresponding documents as two separate entities in terms of processors and memory space.

1. Introduction

Major *Web search engines* (WSEs) process millions of user queries per day so they are compelled to be extremely efficient in dealing with intensive and highly dynamic query traffic. Achieving this goal necessarily implies the use of heuristics and optimizations, which as a whole lead to efficient performance. Given the scale of the hardware resources devoted to host search engines in data centers, performance refers not only to query throughput or individual query response time, but also to the average amount of hardware hit by queries. Search engines are built on thousands of computers organized in highly communicating groups which are implemented as clusters of distributed memory processors. Thus, for reasons such as power consumption, it is relevant to devise query processing strategies capable of reducing the use of hardware resources whilst satisfying a target throughput and response time bound.

The query process, from reception to response, consists of several steps, many of which can take place in parallel. Typically, each cluster of processors is assigned the role of executing one single step, which include determining the top-K documents that best match the query, constructing the answer Web page (this involves extracting small pieces of text, the *snippets*, from the resulting documents), plus other steps such as advertising, spelling, suggestions and facets.

This paper proposes mixing into a single task and cluster the steps of top-K determination and snippet extraction, by using recently proposed self-indexed compressed text techniques (i.e., text indexes that compress the text and are able to extract any part of it) in combination with posting lists caching. This is achieved for similar memory space requirements and number of processors of a standard cluster in charge of the top-K determination. We study how already-known sequential compressed self-indexes can be employed to support efficient and fault-tolerant query processing in search engines. Fault-tolerance is supported by a form of replication in which the cluster can be seen as a $P \times D$ array of search node processors, where P indicates the level of text partitioning and D the level of replication of each partition. Replication also increases throughput, since query traffic in each column can be evenly distributed across the D replicas.

Email addresses: darroyue@yahoo-inc.com (Diego Arroyuelo), gvcosta@unsl.edu.ar (Veronica Gil-Costa), sgonzale@dcc.uchile.cl (Senén González), mmarin@yahoo-inc.com (Mauricio Marin), mauricio.silvaoy@usach.cl (Mauricio Oyarzún)

Notice that even though the secondary memory contents of replicas are identical, one is free to decide which contents to upload from secondary to main memory during normal operation. In the case of a processor failure, the remaining $D-1$ processors can reconstruct the main memory contents of the failed processor into their own main memories by just reading data from secondary memory. Occasional imbalance can be easily cleared by temporarily diverting skewed query traffic to replicas selected uniformly at random if necessary.

The compressed text self-indexes that enable our purpose are the *Wavelet Trees* (WT) [6], which have been shown to be competitive with inverted indexes [1], being able to generate the occurrence list of any term and compute the intersection among the occurrence lists of the query terms. Experimental results show that the proposed indexing scheme is able to answer queries efficiently. We emphasize that the advantage is that we are able to quickly perform snippet extraction from the self-indexed text itself, thus reducing both the need of extra storage for text and extra communication from interaction with an additional cluster in charge of snippet extraction.

Our main result is that the WT can be adapted to our $P \times D$ context. That is, by doing term clustering along the D replicas, we can keep in main memory a very optimized realization of the WT, containing only the terms assigned to the respective replica. We keep the inverted lists for the terms assigned to other replicas in secondary memory.

We also show how the capabilities of the WT can be used to dynamically maintain a cache with the most recently used posting lists of an inverted index generated on-demand. Hence, the posting lists of non-used terms can be disregarded, whereas the posting lists of missing terms can be constructed on-the-fly from the WT itself. The result is an scheme that is adaptive to the user queries and with a query throughput comparable to that of a standard inverted index. For disjunctive queries, a type of query supported by major search engines which we believe is a much less frequent one than the conjunctive one, the WT allows us to quickly generate the posting lists containing term frequencies. The cached portions of posting lists resulting from disjunctive queries can be used as entry points in the WT to speed up the generation of these lists as well. Overall, there is room for many other optimizations which are useful depending on the target design of the search engine. In this paper we explore just a few of them.

2. Background and Previous Work

2.1. Inverted Files

The *inverted index* [2, 18] (or inverted file) is a data structure used by all well-known WSEs. It is composed of a *vocabulary table* (which contains the V distinct relevant terms found in the document collection) and a set of *posting lists*. The posting list for term $c \in V$ stores the identifiers of the documents that contain the term c, along with additional data used for ranking purposes. To solve a query, one must fetch the posting lists for the query terms, compute the intersection among them, and then compute the ranking of the resulting intersection set. Hence, an inverted index allows for the very fast computation of the top-K relevant documents for a query, because of the pre-computed data.

2.2. The Architecture of a Distributed WSE

Given the query traffic of current WSEs, and the need for fast responses, the architecture of large-scale search engines is usually distributed. Distributed WSEs are composed of a query receptionist machine (the *broker*) and a set of P *search nodes* (we also call them *processors*), where each search node is replicated D times. There are basically two approaches to the parallelization of inverted files, namely the *document-* and the *term-partitioned* approaches [15]. In the former, the document collection is evenly distributed among the search nodes, so each of them indexes a fraction $1/P$ of the whole collection. In the latter, the vocabulary terms are distributed among the nodes, such that every node stores only the posting lists for these terms. Both methods have advantages and drawbacks, and the choice depends on the particular scenarios [2, 15].

In large document collections, some posting lists contain millions of items. Thus, large index portions must be kept on secondary storage. However, observations from query logs tell us that certain terms in queries are more frequent than others. Also, the most frequently asked terms vary over time. There are burst periods in which a small set of terms are very frequent, to then fade away smoothly. Other terms are steadily popular along time [5]. These observations encouraged a number of cache strategies [5]. The broker keeps a result cache, the search nodes keep a cache for frequently-used posting lists, and the document servers keep a cache for frequently-required Web documents.

2.3. Query Processing in a Distributed WSE

Upon a user query, each of the top-K results obtained is displayed along with a short *snippet* surrounding the query terms. In large-scale systems [2] the index and document collection are organized as two independent clusters of processors: besides the *search nodes*, the *document servers* are used to obtain the snippets. Overall, the broker performs the following tasks:

1. receives the query from the user and, provided that there are no results for it in its result cache, sends the query to the relevant search nodes (to every node in document-partitioned indexes);
2. receives each of the local top-K documents calculated by the search nodes and merges the results to obtain the global top-K results;
3. sends those results and the query to the document server for snippets determination;
4. once the broker receives the snippets from the document server, builds up the result's page (URLs + snippets) and responds to the user.

2.4. The Self-Indexing Technology and WSEs

A current trend in text indexing is that of *compressed full-text self-indexes*, which replace the text with a representation that takes space proportional to that of the compressed text (so the space is reduced for compressible texts), support indexed text searches, and the fast extraction of any text portion [12]. The aim is to fit the index of large texts in main memory, avoiding the high secondary storage costs. Another feature is that self-indexes can search without accessing the text. This has applications in cases where accessing the text at search time is expensive (as in the case of classical WSEs).

A recent work proposes compressed self-indexes [1] as an alternative to inverted indexes, achieving interesting results. However, much work need to be done in order to apply them in practical settings. This paper contributes towards this direction. We propose the use of self-indexing technologies to combine into a single unit each of the P corresponding pairs of search nodes and document servers. The rationale is reducing from $2P$ to P the total number of processors involved in the processing of queries, yet achieving a similar query throughput. Namely, the standard two-stepped procedure (i.e., steps 2 and 3 above) is reduced to one step, in which document ranking and snippet determination are combined into a single operation in each processor. This should reduce query processing time and communication.

2.5. Succinct Data Structures for rank *and* select *queries*

Given a bit sequence $B[1..n]$, we define operation $\mathsf{rank}_1(B,i)$ (similarly $\mathsf{rank}_0(B,i)$) as the number of 1s (0s) occurring in $B[1..i]$. Operation $\mathsf{select}_1(B,i)$ (similarly $\mathsf{select}_0(B,i)$) is defined as the position of the i-th 1 (or i-th 0) in B. Given a sequence $S[1..n]$ over an alphabet $\Sigma = \{1,\ldots,V\}$, we generalize the definition for $\mathsf{rank}_c(S,i)$ and $\mathsf{select}_c(S,i)$ for any $c \in \Sigma$.

3. Self-Indexes for Document Retrieval

Let $\mathcal{D} = \{D_1,\ldots,D_{N_d}\}$ be a *document collection* of N_d documents, where each document is represented as a sequence $D_i[1..n_i]$ of n_i terms from an alphabet Σ of size V. Assuming that '\$'$\notin \Sigma$ is a special separator symbol, we build the sequence $T[1..n] = \$D_1\$D_2\$\cdots\$D_{N_d}\$$ of length $n = 1 + \sum_{i=1}^{N_d}(n_i+1)$ and $n \log V$ bits. Each document D_i is assigned a unique *document identifier* (docid, for short) i.

If we represent T with a rank/select data structure, we can easily obtain both the docid of the document that contains a given position in T and the starting position of a given document j [1]. For instance, we can represent T with a *wavelet tree* [6] (WT), which is a balanced binary search tree, where each different term in the vocabulary corresponds to a leaf A WT supports extracting any text symbols and compute operations rank and select in $O(\log V)$ time [12]. The space usage is $n \log V + o(n \log V)$ bits [6, 12], which is about 1.1–1.2 times the space of the original text [4]. In our application this would produce an index that is bigger than inverted indexes (actually, bigger than the text itself). Therefore, in this paper we use Huffman-shaped WTs [4], achieving $n(H_0(T)+1)+o(n(H_0(T)+1))$ bits of space, where $H_0(T) \leq \log \sigma$ is the zero-order empirical entropy of T, and $nH_0(T)$ is a lower bound to the number of bits needed to compress T based on the symbol frequencies. In practice, the space of a Huffman-shaped WT is about 0.6–0.7 times the text size [4]. We use the Huffman-shaped WT implementation from [1].

A recent work [1] shows that WTs can be used to index document collection and support the following functionality within $n(H_0(T)+1)+o(n(H_0(T)+1))$ bits of space: (*1*) Generate, on-the-fly and on-demand, the posting list of a given query term $t \in \Sigma$; (*2*) answer conjunctive queries of the form $t_1 \wedge \cdots \wedge t_k$, in

theory as efficiently as an inverted index; (*3*) extract a snippet surrounding the occurrences of a given term; (*4*) obtain within-document term frequencies; and (*5*) obtain positional information with no extra space usage. Hence, a WT can be thought as a black box that provides most of the functionality of an inverted index. However, the space usage should be reduced by using a WT. For instance, positional information requires considerable space, and could be compressed to zero-order entropy [16] (i.e., about the same space as the WT). The WT, on the other hand, stores also the text, frequencies and document identifiers within that space. However [1] lacks a complete comparison with inverted indexes in practice.

The work [1] introduces three algorithms to support conjunctive queries, namely a simple worst-case scheme, an adaptive scheme, and a hybrid scheme that combines the previous methods. Because of its simplicity and performance, we will use the former method in this paper (called `SLF` in [1], which stands for Shortest List First, as it generates first the shortest inverted list among those of the query terms, and then quickly check whether the remaining terms occur within these documents).

We compare the performance of a search engine based on WTs, with one bases on inverted indexes. Our aim is to show that though WTs can be slower than inverted indexes for computing conjunctive queries, the former are competitive if we consider the whole query processing, which after performing the intersection includes obtaining the frequencies of the query terms in the resulting documents, to then perform a top-K ranking step, to finally carry out the snippet extraction for the top-K documents. The ranking step is performed using the tf-idf model. Also, we extract snippets of length 5 to both sides of an occurrence (thus, the whole snippet has length 11). For extracting snippets with the inverted index, we assume the following baseline scheme. First, we divide the document collection into blocks, each consisting of b documents. We compress each block using the LZMA compression algorithm (using the LZMA Software Development Kit, `http://www.7-zip.org/sdk.html`). This algorithm is able to decompress about 20–30 MB of text per second. The compressed blocks are maintained in main memory. Given the top-10 results (obtained with the inverted index) of a given query, we decompress the blocks containing these documents, and then perform a text search over these documents (for instance, using a Booyer-More like algorithm). Once we found the first occurrence of the query term, we display a snippet surrounding the occurrence. Notice how different block sizes yield different time/space trade-offs, since smaller blocks produce more overhead in the compression.

Table 1 shows the comparison of space usage between WTs and inverted indexes using three of the most effective compression methods, namely S9, PForDelta, and variable byte encoding (VByte) [3]. For the experiments we used a sample of the UK Web from which we selected at random a total of 429,895 documents. This demands a total space usage of about 1.5 GB (no html data is included in the text). The computer used is an Intel(R) Core(TM)2 Duo CPU at 2.8 GHz, with 64KB of L1 cache, 3,072 KB of L2 cache and 5 GB of RAM, and running version `2.6.31-22-server` of Linux kernel.

Table 1: Comparison of space in MB for WTs and inverted indexes (using S9, VByte and PForDelta compression schemes).

	S9	VByte	PForDelta	WT
Docids (MB)	220	248	614	–
Frequencies (MB)	107	175	209	–
Compressed text ($b = 1$)	570	570	570	–
Total size (MB)	897	993	1,393	706
Compressed text ($b = 150$)	379	379	379	–
Total size (MB)	706	802	1,202	706

We use $b = 1$ for the snippet extraction with inverted indexes, which compresses the document collection to about 570MB, with an average block size (document size in this case) of 3.57 KB. In this case, we are able to extract on average 13,404 snippets of length 11 per second. We also use $b = 150$, so that overall the S9 compressed inverted index uses about the same space than that of the WT.

In Table 2 we compare the query performance for the different steps of the query process. The first line shows the base intersection speed, and then each row shows the performance obtained by adding a step to the previous one. As it can be seen, if we consider the whole query process (i.e., including snippet extraction), WTs are slower than inverted indexes for $b = 1$, however the latter uses 1.27 times more memory than WTs. This space could be used by the WT to store a cache with the most frequently used inverted lists (see Section 5). For $b = 150$, snippet extraction becomes extremely slow, whereas WTs maintain about the same performance. This is because WTs extract about 131,227 snippets per second.

Table 2: Comparison of query performance (in number of queries per second) for different steps of the query process.

	S9	VByte	PForDelta	WT
Intersection speed	3,271	2,763	3,005	146
+ Frequencies extraction	2,103	2,090	2,450	145
+ Top-10 ranking	1,504	1,503	1,587	142
+ Snippet extraction (b=1)	242	241	253	139
+ Snippet extraction (b=150)	19	18	19	139

4. Distributed Web Search Engines Based on Self-Indexes

Inverted indexes must store extra information to provide extra functionality needed by WSEs, such as ranking and positional information (the latter is needed for phrase searching and positional ranking functions [16]), snippet extraction, query caching, etc. Hence, the overall solution requires considerable space, and parts of the index must reside on secondary storage. This results in a undesirable increase of the I/O traffic. Also, major current WSEs receive millions of queries per day, so they have to resort to parallel computing techniques to satisfy this demand. We introduce now different ways to handle wavelet trees on a distributed-memory environment, which shall be used in Section 5 to develop space-efficient distributed in-memory search engines. We assume in the following an array of $P \times 1$ processors.

4.1. Document Partitioned Self-Index

In this simple approach, we divide the document collection among the cluster nodes, such that every node stores about N_d/P documents. Given the documents of a given search node i of the cluster, we construct a text T_i (of length $|T_i|$) from the concatenation of these documents. Then, at each processor i we construct the wavelet tree W_i for T_i. At query time, a given query must be sent to all search nodes in order to answer it, as it is usual in this distribution model [15].

4.2. Term Partitioned Self-Index

We divide the vocabulary such that every search node is responsible for a fraction V/P of the vocabulary. Let V_i denote the vocabulary for processor i. Then, the local text T_i is the global text T but considering only the terms in V_i (in other words, T_i is the projection of T over the symbols in V_i). Hence, the size of the local texts in the processors can be different to each other, since these depend on the occurrences of each individual symbol in V_i.

To retain the global structure of the text, which will help us with the snippet extraction, we store in each processor i a bit sequence $B_i[1..n]$, such that $B_i[j] = \mathtt{1} \Leftrightarrow T[j] \in V_i$. In other words, B_i keeps track of the global text positions that are indexed by processor i.

Though the total (logical) number of bits in all bit vectors is nP, there are only n 1s overall, since any global-text position is indexed by one and only one processor. If we use the `sarray` data structure from [13], the total space usage is $n \log \frac{nP}{n} + 1.92n + o(n) = n \log P + 1.92n + o(n)$ bits. In practice, this should be much smaller than the text size. In order to be able to search and extract snippets with this approach, given a global-text position j, the corresponding local-text position for processor i is $\mathsf{rank}_1(B_i, j)$, which is supported in time $O(\log P + \frac{\log^4 n}{\log(nP)})$ [13]. Given a local-text position j in processor i, the corresponding global-text position is $\mathsf{select}_1(B_i, j)$, which is supported in $O(\frac{\log^4 n}{\log(nP)})$ time [13].

An interesting property is that we do not lose compression, not matter how the term distribution is done. This is because wavelet trees are zero-order compressors, and the overall frequency of these terms is not affected if we distribute them. Moreover, we can achieve further compression, because of two facts. First, the vocabulary at each processor has about V/P terms. Then, $\log \frac{V}{P}$ bits (instead of $\log V$ bits) are enough to represent each symbol, so we should get shorter Huffman codes on average. Besides, we will obtain shallower wavelet trees, hence improving the running time. Second, with a careful term distribution we could achieve further compression, as we will see next.

Experimental Results. We experimentally study this effect. We first sort the terms by their global frequencies, and then test several distribution strategies: (**S1**) the terms are distributed at random; (**S2**) the sorted list of terms is distributed in a round-robin fashion; (**S3**) the terms are distributed such that the next one is assigned to the processor that has the smallest sum of frequencies; and (**S4**) the V/P

most frequent terms are assigned to the first processor, the next V/P in the list are assigned to the second processor, and so on. Some of these (and other) variants have been studied in several related papers [8, 14, 11, 10].

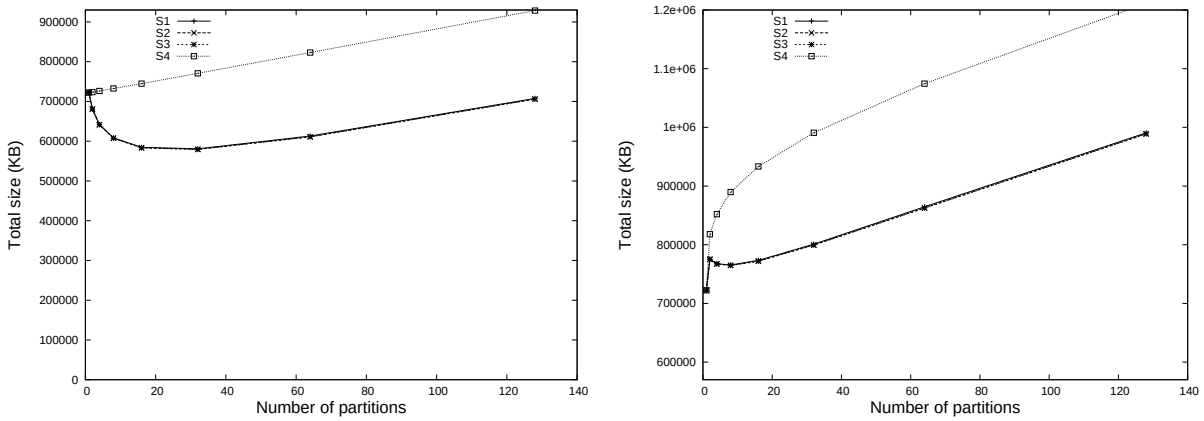

Figure 1: Total size of the term-partitioned wavelet trees, for different number of partitions, just regarding the wavelet trees (left) and adding the space of the compressed bit vectors (right).

We compared the space of the wavelet tree for the whole text versus the sum of the spaces for the term-distributed wavelet trees. See Fig. 1 for a comparison, using $P = 1$ (i.e., the original WT), $P = 2, 4, 8, 16, 32, 64$ and 128. We obtain, for strategies S1, S2, and S3, a space reduction that varies on the number of partitions used, but that is at most of 20% for $P = 32$. We conclude that the technique is fairly independent on these distribution strategies. However, there is a significant difference among these strategies and strategy S4, which does not achieve a space reduction (this is because the overhead of maintaining several WTs is greater than the space gain obtained by distributing the terms in this way). This indicates that some term distribution strategies can be less effective for zero-order compression. In general, it is not a good idea to group together terms that are very frequent (as is the case of S4). In a skewed distribution of frequencies, the most frequent terms will "compete" for a shorter codification with non-frequent terms, both of them obtaining shorter codes separately. Thus, the further compression obtained by strategies S1, S2, and S3 is a result of having a smaller local vocabulary with a skewed distribution of term frequencies. Finally, one could distribute the terms according to a clustering of correlated terms, such that terms that tend to appear together in queries of our query log will belong to the same cluster. This achieves a compression ratio comparable to that of strategies S1–S3. This method is relevant since it reduces the number of intersections of terms that reside in different processors [10].

Query Processing. At search time, queries must be sent only to the processors that store the involved query terms. We suffer from the same problems as the term-based distributed inverted indexes, e.g., the intersection of posting list for terms stored in different search nodes. As we said before, by using distribution strategies like S5 above, we can reduce the probability of inter-processor intersections. In any case, below we propose using term-partitioning along the D replicas so in this case the processor can use its secondary memory to solve the intersection.

Snippet Extraction. To extract a snippet, let us say that a term occurrence appears at position j in T_i. Hence, the corresponding position in the global text is $j' = \mathsf{select}_1(B_i, j)$. Thus, we broadcast j' to all processors. Then, every processor i' checks whether the bit sequence $B_{i'}$ has a 1 in some of the surrounding positions, and in such a case returns to the broker the corresponding terms along with the global text position (accessing a particular bit in $B_{i'}$ takes $O(\log P + \frac{\log^4 n}{\log(nP)})$ time [13]).

We carried out experiments to determine the snippet extraction capabilities of WTs in parallel. We tested with $P = 1, 2, 4$ and 8 partitions, and extract snippets of length 11 as previous experiments in this paper. For $P = 1$ (i.e., the original WT) we extract on average 131,227 snippets per second. For $P = 2$ we get 193,348 snippets per second. For $P = 4$ we get 333,258 snippets per second. Finally, for $P = 8$ we get 688,345 snippets per second on average. Thus, our method scales well with the number of partitions used, since at a given time there are more WTs generating snippets in parallel.

5. Deployment on $P \times D$ search nodes

In this section we describe an instance of use of the Wavelet Tree (WT) in the context of a large-scale Web search engine (WSE). As mentioned above, apart from its role as an efficient snippet extractor, we can regard the WT as a device capable of producing individual posting lists and pair-wise intersection lists on demand. This role resembles the three-level caching scheme proposed in [7], so we compare our proposal against that approach.

The general WSE architecture and query processing strategy is as follows. The WSE is assumed to be composed of a set of so-called search nodes and a query receptionist machine called broker. The set of search nodes is organized as a 2-dimensional array of $P \times D$ nodes, where P indicates the number of partitions and D the number of copies of each partition. For each newly arriving query, the broker looks for the query in a result cache. If found, the broker replies with the answer. If not found, the broker sends the query to all of the P partitions, where in each partition one of the D replicas is selected uniformly at random. Then the P search nodes respond with the top-K results for the query, which are then merged to produce the global top-K results. For any given query, the merge procedure is carried out by a randomly selected search node. This node sends the global top-K results to the broker which constructs the answer page for the query.

5.1. The baseline strategy

The three-level caching strategy [7] proposes keeping on disk a cache to store the intersection of pairs of inverted lists. For each intersection involving the terms a and b, they define two projection lists $I_{a \rightarrow b}$ and $I_{b \rightarrow a}$ that share the same document IDs in the intersection set, but keep data from a and b respectively that are used to score the documents. These projection lists are stored in the intersection cache. The inverted file is also assumed to be stored on disk, and the size of the intersection cache is set to consume a 40% of the space occupied by the inverted file.

In main memory, a LRU list cache is used to keep both posting lists and projection lists. The projection lists are retrieved on-demand from secondary memory if they happen to be stored in the intersection cache and were not found in the main memory LRU list cache at query processing time. This requires secondary memory accesses but [7] shows that it is worthwhile to pay this cost instead of fully calculating the intersections of much larger posting lists. There is a rule to decide whether or not storing projection lists $I_{a \rightarrow b}$ and $I_{b \rightarrow a}$ in the intersection cache depending on how frequently terms a and b take place in queries, and the cost of calculating the intersection of the respective posting lists. The three level caching scheme is completed by assuming the existence of a result cache in the broker.

To increase the efficiency of main-memory space usage, we apply term partitioning along the D replicas of each of the P partitions. The gain is that the cache entries are used more effectively since term partitioning reduces the probability of caching the same list in two or more replicas [9]. Load imbalance due to terms more popular than others is not significant since frequent queries get quickly cached in the results cache [9]. The compressed text for extracting snippets of top-K results is kept evenly distributed in the main memories of the $P \times D$ search nodes.

5.2. The WT strategy

For the sake of a fair comparison, we also assume the existence of an inverted file stored on disk. We keep in main memory the WT and the LRU list cache, which stores posting and projection lists as in the baseline strategy. In each of the $P \times D$ search nodes, the total main-memory space occupied by the WT plus the LRU list cache is the same than the space occupied by the compressed text and the LRU list cache of the baseline strategy. The space occupied by the WT tends to be larger than the space occupied by the compressed text, and thereby the LRU list cache of the WT strategy is smaller than the LRU list cache of the baseline strategy

Like in the baseline strategy, we assume term partitioning along the D replicas of the P partitions. For the WT this also means that only the terms mapped to a given replica are considered to construct the data structure. An array of bits is used to indicate how many terms t between two consecutive terms a and b in the WT of a given row are in the actual text. These terms t are indexed in the WTs located in other rows of the same column. The array of bits together with the D WTs of the column are used to build the snippets for the documents that are part of the global top-K results for a query. The same scheme of D WTs and bit arrays is constructed in the remaining columns by considering, in each case, only the documents allocated to the partition associated with the respective column.

For a query with terms a and b we define the projection lists $I_{a\to b}$ and $I_{b\to a}$ as above, and L_a and L_b as the posting lists of a and b respectively. The query processing algorithm considers the following cases that arise when posting lists and projection lists are found or not in the LRU list cache:

(i) Neither I's nor L's are in the cache, then the WT is used to generate $I_{a\to b}$ and $I_{b\to a}$.

(ii) one of the I's is in the cache, say $I_{a\to b}$, then we use $I_{a\to b}$ on the WT to generate $I_{b\to a}$ (this is a very fast operation in the WT as it only requires a few rank operations).

(iii) $I_{a\to b}$ and L_b are in the cache, then we use the items of $I_{a\to b}$ to traverse L_b in one passage and generate $I_{b\to a}$.

(iv) only L_b is in the cache, then we use L_b on the WT to generate $I_{a\to b}$ and $I_{b\to a}$.

When one of the terms is not indexed by the WT, we retrieve its posting list from secondary memory if required. Notice that similar cases can be defined for the baseline strategy but with one important difference. In the baseline strategy whenever the projection lists of terms a and b are not found either in the LRU list cache or in the intersection cache kept on disk, it is necessary to compute the intersection between the two posting lists L_a and L_b. This can be an expensive operation since one or both of those lists could have to be retrieved from disk. As the lists L_a and L_b are expected to be of a much larger size than the respective projection lists, storing L_a and L_b in the LRU cache list can unnecessarily remove from the cache a significant number of other projection lists that can be required by upcoming queries. The WT prevents from disk accesses by quickly computing the intersection without having to generate the lists L_a and L_b separately. Alternatively the baseline strategy can decide not to store the lists L_a and L_b in the LRU list cache and dispose them after computing the intersection. But this tends to increase disk accesses whereas the WT can fastly generate posting lists on-demand from main memory.

5.3. Experimental Results

The experiments were performed using a log of 36,389,567 queries submitted to the AOL Search service between March 1 and May 31, 2006. We pre-processed the query log following the rules applied in [17]. The resulting query trace has 16,900,873 queries, where 6,614,176 are unique queries and the vocabulary consists of 1,069,700 distinct query terms. The results were obtained after processing 60% of the queries by using a discrete-event simulator described in [9]. The simulator implements the actual cache strategies and is able to precisely predict query throughput by considering the different costs involved in the processing of queries. These costs were determined from the programs executed for the experiments shown in the previous sections of this paper and benchmarks from [9] for communication and considering a disk access cost of 8 ms per read block. Posting lists are kept compressed and divided in blocks for efficient caching.

We simulated query processing on three $P \times D$ search nodes configurations such that $P \cdot D = 512$. The configurations are 256×2, 128×4 and 64×8. We assume that the baseline strategy is capable of keeping in its LRU list cache, for each configuration and search node respectively, the 100%, 50% and 25% of the inverted file kept on disk. The intersection cache is kept on disk and is defined to occupy 40% of the space required by the inverted index in each partition. The inverted index was constructed using a 1.5TB sample of the UK web from 2005. We used the greedy heuristic proposed in [7] to initialize each intersection cache by using the whole set of queries that hit the respective cache.

The LRU cache of the baseline strategy is divided in three sections where 44% of the space is used to hold projection lists, 20% is used to hold posting lists of terms that do not belong to the respective row in the $P \times D$ matrix, and 36% of the remaining space is used to store posting lists of terms mapped to the row. These values were set to fit the same space assigned in the LRU cache of the WT strategy. This cache holds the projection lists (68%) and the posting lists of terms that are not mapped to the row (32%). The idea is to compare both strategies under the same space devoted to hold projection and non-local posting lists. The remaining space in baseline cache is used to hold frequently accessed posting lists assigned to the respective row. Notice that most disk accesses performed by the baseline come from accesses to retrieve posting lists of non-local terms. Namely, hit ratio did not increase for other configurations in which more space was assigned to cache local posting lists. The results cache size was set to achieve a 20% hit ratio. We have found this 20% hit setting convenient for our query log so we were able to achieve a good balance between the increased average number of terms that hit search nodes as discussed in [7] and a large enough number of queries hitting the nodes during experimentation.

The results are shown in Table 3 and they show that the WT strategy reduces accesses to secondary memory and achieves better throughput (Q/s) than the baseline strategy (values normalized to 1).

Table 3: Results on $P \times D$ search nodes.

Baseline							
$P \times D$	LRU Size	Disk Bytes	Inter. Hits	Proj. Hits	Post. Hits	Post. row	Q/s
256×2	100%	0.29	16%	34%	32%	21%	0.22
128×4	50%	0.39	13%	31%	27%	16%	0.27
64×8	25%	1.00	10%	28%	24%	12%	0.60
WT							
$P \times D$	LRU Size	Disk Bytes	WT Hits	Proj. Hits	Post. Hits		Q/s
256×2	64%	0.11	33%	35%	31%		0.28
128×4	32%	0.13	31%	31%	28%		0.53
64×8	16%	0.19	23%	27%	22%		1.00

6. Conclusions

In this paper we have presented an indexing and query processing strategy based on a novel adaptation of a self-indexed compressed text, that significantly improves performance of Web search engines implemented by using the standard array of $P \times D$ processors. The proposed optimization involves both the accesses required to get the pieces of index used to rank the documents to be included in the top-K results and the accesses required to build the answer Web pages for queries. We achieve this by combining strategies related to caching of posting lists with self-indexed compressed text.

The experimental results show the following facts considering the space A occupied by a baseline strategy based on a compressed text plus a compressed inverted file, and the space B occupied by the Wavelet tree (WT) strategy. For $P = 1$, $D = 1$ and $A = B$, the throughput achieved by the WT strategy is about 5 times better than the baseline strategy. For the case $A = 1.27 \cdot B$, the throughput achieved by the baseline strategy is about 1.74 times better than the WT strategy. However, the WT strategy can use the extra space to hold a LRU list cache. The benefits of this cache are studied in a deployment on $P \times D$ search nodes. The results show that the WT strategy improves 40% overall query throughput over a baseline strategy constructed from a state of the art three-level caching scheme. A key factor in this efficient performance is the ability of the WT to fastly produce posting list intersections and snippets.

[1] D. Arroyuelo, S. González, and M. Oyarzún. Compressed self-indices supporting conjunctive queries on document collections. In *SPIRE*, LNCS 6393, pages 43–54, 2010.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[3] S. Büttcher, C. Clarke, and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

[4] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *SPIRE*, LNCS 5280, pages 176–187. Springer, 2008.

[5] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *WWW*, pages 431–440, 2009.

[6] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, 2003.

[7] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *WWW*, pages 257–266, 2005.

[8] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Mining query logs to optimize index partitioning in parallel web search engines. In *Infoscale*, page 43, 2007.

[9] M. Marin, V. Gil-Costa, and C. Gomez-Pantoja. New caching techniques for web search engines. In *HPDC*, pages 215–226, 2010.

[10] M. Marin, C. Gomez-Pantoja, S. Gonzalez, and V. Gil-Costa. Scheduling intersection queries in term partitioned inverted files. In *Euro-Par*, pages 434–443, 2008.

[11] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR*, pages 348–355, 2006.

[12] G. Navarro and V. Mäkinen. Compressed full-text indexes. *J. of CSUR*, 39(1):article 2, 2007.

[13] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*, pages 60–70, 2007.

[14] G. Skobeltsyn, T. Luu, I. P. Zarko, M. Rajman, and K. Aberer. Query-driven indexing for scalable peer-to-peer text retrieval. *J. of FGCS*, 25(1):89–99, 2009.

[15] A. Tomasic and H. Garcia-Molina. Performance issues in distributed shared-nothing information-retrieval systems. *J. of IPM*, 32(6):647–665, 1996.

[16] H. Yan, S. Ding, and T. Suel. Compressing term positions in web indexes. In *SIGIR*, pages 147–154, 2009.

[17] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[18] J. Zobel and A. Moffat. Inverted files for text search engines. *J. of CSUR*, 38(2), 2006.