Faster BlockMax WAND with Variable-sized Blocks

Antonio Mallia
University of Pisa, Italy
a.mallia@studenti.unipi.it

Giuseppe Ottaviano
Facebook, Inc., USA
ott@fb.com

Elia Porciani
University of Pisa, Italy
e.porciani1@studenti.unipi.it

Nicola Tonellotto
ISTI-CNR, Italy
nicola.tonellotto@isti.cnr.it

Rossano Venturini
University of Pisa, Italy
rossano.venturini@unipi.it

ABSTRACT

Query processing is one of the main bottlenecks in large-scale search engines. Retrieving the top k most relevant documents for a given query can be extremely expensive, as it involves scoring large amounts of documents. Several *dynamic pruning* techniques have been introduced in the literature to tackle this problem, such as BlockMaxWAND, which splits the inverted index into constant-sized blocks and stores the maximum document-term scores per block; this information can be used during query execution to safely skip low-score documents, producing many-fold speedups over exhaustive methods.

We introduce a refinement for BlockMaxWAND that uses variable-sized blocks, rather than constant-sized. We set up the problem of deciding the block partitioning as an optimization problem which maximizes how accurately the block upper bounds represent the underlying scores, and describe an efficient algorithm to find an approximate solution, with provable approximation guarantees. Through an extensive experimental analysis we show that our method significantly outperforms the state of the art roughly by a factor 2×. We also introduce a compressed data structure to represent the additional block information, providing a compression ratio of roughly 50%, while incurring only a small speed degradation, no more than 10% with respect to its uncompressed counterpart.

ACM Reference format:
Antonio Mallia, Giuseppe Ottaviano, Elia Porciani, Nicola Tonellotto, and Rossano Venturini. 2017. Faster BlockMax WAND with Variable-sized Blocks. In *Proceedings of SIGIR '17, Shinjuku, Tokyo, Japan, August 7–11, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3077136.3080780

1 INTRODUCTION

Web Search Engines [6, 19] manage an ever-growing amount of Web documents to answer user queries as fast as possible. To keep up with such a tremendous growth, a focus on efficiency is crucial. Query processing is one of the hardest challenges a search engine has to deal with, since its workload grows with both data size and query load. Although hardware is getting less expensive and more powerful every day, the size of the Web and the number of searches is growing at an even faster rate.

Query processing in search engines is a fairly complex process; queries in a huge collection of documents may return a large set of results, but users are often interested the most relevant documents, usually a small number (historically, the *ten blue links*). The relevance of a document can be arbitrarily expensive to compute, which makes it prohibitive to evaluate all the documents that match the queried terms; query processing is thus usually divided in multiple phases. In the first phase, the query is evaluated over an inverted index data structure [3, 29] using a simple scoring function, producing a medium-sized set of candidate documents, namely the top k scored; these candidates are then re-ranked using more complex algorithms to produce the final set of documents shown to the user.

In this work we focus on improving the efficiency of the first query processing phase, which is responsible for a significant fraction of the overall work. In such phase, the scoring function is usually a weighted sum of per-term scores over the terms in the document that match the query, where the weights are a function of the query, and the scores a function of the occurrences of the term in the document. An example of such a scoring function is the widely used BM25 [24].

An obvious way to compute the top k scored documents is to retrieve all the documents that match at least one query term using the inverted index, and compute the score on all the retrieved documents. Since exhaustive methods like this can be very expensive for large collections, several *dynamic pruning* techniques have been proposed in the last few years. Dynamic pruning makes use of the inverted index, augmented with additional data structures, to skip documents during iteration that cannot reach a sufficient score to enter the top k. Thus, the final result is the same as exhaustive evaluation, but obtained with significantly less work. These techniques include MaxScore [30], WAND [4], and BlockMaxWAND (BMW) [10].

We focus our attention on the WAND family of techniques. WAND augments the posting list of each term with the *maximum score* of that term among all documents in the list. While processing the query by iterating on the posting lists of its terms, it maintains the top k scores among the documents evaluated so far; to enter the top k, a new document needs to have score larger than the current k-th score, which we call the *threshold*. WAND maintains the posting list iterators sorted by current docid; at every step, it adds up the maximum scores of the lists in increasing order, until the threshold is reached. It can be seen that the current docid of the first list that exceeds the threshold is the first docid that can

SIGIR '17, Shinjuku, Tokyo, Japan
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-5022-8/17/08…$15.00.
DOI: http://dx.doi.org/10.1145/3077136.3080780

reach a score higher than the threshold, so the other iterators can safely skip all the documents up to that docid.

The core principle is that if we can upper-bound the score of a range of docids, and that upper bound is lower than the threshold, then the whole range can be safely skipped. As such, WAND computes the upper bounds of document by using the maximum score of the terms appearing in the document. Nevertheless, it should be clear that the pruning effectiveness is highly dependent on the accuracy of the upper bound: the more precise the upper bound, the more docids we can skip, and, thus, the faster the query processing.

BMW improves the accuracy of the upper bounds by splitting the posting lists into constant-sized blocks of postings, and storing the maximum score per block, rather than per list only. This way, the upper bound of a document is the sum of the maximum score of the blocks in which it may belong to. This approach gives more precise upper bounds because the scores of the blocks are usually much smaller than the maximum in their lists. Experiments confirm this intuition, and, indeed, BMW significantly outperforms WAND [10].

However, the coarse partitioning strategy of BMW does not take into consideration regularities or variances of the scores that may occur in the posting lists and their blocks. As an example, consider a posting with a very high score surrounded by postings with much lower scores. This posting alone is responsible for a high inaccuracy in the upper bounds of all its neighbors in the same block. Our main observation is that *the use of variable-sized blocks would allow to better adapt to the distribution of the scores in the posting list.* The benefits of variable-sized blocks are apparent in the simple example above, where it is sufficient to isolate the highly-scored posting in its own block to improve the upper bounds of several other postings, stored in different blocks. More formally, for a block of postings we define the *block error* as the sum of the individual posting errors, i.e., the sum of the differences between the block maximum score and the actual score of the posting. Our goal is to find a block partitioning minimizing the sum of block errors among all blocks in the partitioning. Clearly, this corresponds to minimizing the average block error. Naïvely, the minimum cost partitioning would correspond to blocks containing only a single posting. However, if the blocks are too small, the average skip at query time will be short and, thus, this solution does not carry out any benefit. In this work we introduce the problem of finding a partition of posting lists into variable-sized blocks such that the the sum of block errors is minimized, subject to a constraint on the number of blocks of the partition. Then, we will show that an approximately optimal partition can be computed efficiently. Experiments on standard datasets show that our Variable BMW (VBMW) significantly outperforms BMW and the other state-of-the-art strategies.

Our Contributions. We list here our main contributions.

(1) We introduce the problem of optimally partitioning the posting lists into variable-sized blocks to minimize the average block error, subject to a constraint on the number of blocks. We then propose a practical optimization algorithm which produces an approximately optimal solution in almost linear time. We remark that existing solutions for this optimization problem run in at least quadratic time, and, thus, they are unfeasible in a practical setting. Experiments show that this approach is able to reduce the average score error up to 40%, confirming the importance of optimally partitioning posting list into variable-sized blocks.
(2) We propose a compression scheme for the block data structures, compressing the block boundary docids with Elias-Fano and quantizing the block max scores, obtaining a maximum reduction of space usage w.r.t. the uncompressed data structures of roughly 50%, while incurring only a small speed degradation, no more than 10% with respect to its uncompressed counterpart.
(3) We provide an extensive experimental evaluation to compare our strategy with the state of the art on standard datasets of Web pages and queries. Results show that VBMW outperforms the state-of-the-art BMW by a factor of roughly 2×.

2 BACKGROUND AND RELATED WORK

In the following we will provide some background on index organization and query processing in search engines. We will also summarize and discuss the state-of-the-art query processing strategies with a particular focus on the current most efficient strategy, namely BlockMaxWAND, leveraging block-based score upper bound approximations.

Index Organization. Given a collection $\mathcal{D}$ of documents, each document is identified by a non-negative integer called a *document identifier*, or *docid*. A *posting list* is associated to each *term* appearing in the collection, containing the list of the docids of all the documents in which the term occurs. The collection of the posting lists for all the terms is called the *inverted index* of $\mathcal{D}$, while the set of the terms is usually referred to as the *dictionary*. Posting lists typically contain additional information about each document, such as the number of occurrences of the term in the document, and the set of positions where the term occurs [5, 19, 32].

The docids in a posting list can be sorted in increasing order, which enables the use of efficient compression algorithms and document-at-a-time query processing. This is the most common approach in large-scale search engines (see for example [8]). Alternatively, the posting lists can be frequency-sorted [30] or impact-sorted [2], still providing a good compression rates as well as good query processing speed. However, there is no evidence of such index layouts in common use within commercial search engines [21].

Inverted index compression is essential to make efficient use of the memory hierarchy, thus maximizing query processing speed. Posting list compression boils down to the problem of representing sequences of integers for both docids and frequencies. Representing such sequences of integers in compressed space is a fundamental problem, studied since the 1950s with applications going beyond inverted indexes. A classical solution is to compute the differences of consecutive docids (deltas), and encode them with uniquely-decodable variable length binary codes; examples are unary codes, Elias Gamma/Delta codes, and Golomb/Rice codes [25]. More recent approaches encode simultaneously blocks of integers in order to improve both compression ratio and decoding speed. The underlying idea is to partition the sequence of integers into blocks of fixed or variable length and to encode each block separately with different strategies (see e.g., [17, 22, 28] and references therein).

More recently, the *Elias-Fano* representation of monotone sequences [11, 12] has been applied to inverted index compression [31], showing excellent query performance thanks to its efficient random access and search operations. However, it fails to exploit the local clustering that inverted lists usually exhibit, namely the presence of long subsequences of close identifiers. Recently, Ottaviano and Venturini [23] described a new representation based on partitioning the list into chunks and encoding both the chunks and their endpoints with Elias-Fano, hence forming a two-level data structure. This partitioning enables the encoding to better adapt to the local statistics of the chunk, thus exploiting clustering and improving compression. They also showed how to minimize the space occupancy of this representation by setting up the partitioning as an instance of an optimization problem, for which they present a linear time algorithm that is guaranteed to find a solution at most $(1 + \epsilon)$ times larger than the optimal one, for any given $\epsilon \in (0, 1)$. In the following we will use a variation of their algorithm.

Query Processing. In *Boolean retrieval* a query, expressed as a (multi-)set of terms, can be processed in conjunctive (AND) or disjunctive (OR) modes, retrieving the documents that contain respectively all the terms or at least one of them. Top-*k* *ranked retrieval*, instead, retrieves the *k* highest scored documents in the collection, where the *relevance score* is a function of the query-document pair. Since it can be assumed that a document which does not contain any query term has score 0, ranked retrieval can be implemented by evaluating the query in disjunctive mode, and scoring the results. We call this algorithm RankedOR.

In this work we focus on *linear* scoring functions, i.e., where the score of a query-document pair can be expressed as follows:

$$s(q, d) = \sum_{t \in q \cap d} w_t s_{t,d}$$

where the w_t are query-dependent weights for each query term, and the $s_{t,d}$ are scores for each term-document pair. Such scores are usually a monotonic function of the occurrences of the term in the document, which can be stored in the posting list alongside the docid (usually referred to as the *term frequency*).

It can be easily seen that the widely used BM25 relevance score [24] can be cast in this framework. In BM25, the weights w_t are derived from t's inverse document frequency (IDF) to distinguish between common (low value) and uncommon (high value) words, and the scores $s_{t,d}$ are a smoothly saturated function of the term frequency. In all our experiments we will use BM25 as the scoring function.

The classical query processing strategies to match documents to a query fall in two categories: in a term-at-a-time (TAAT) strategy, the posting lists of the query terms are processed one at a time, accumulating the score of each document in a separate data structure. In a document-at-a-time (DAAT) strategy, the query term postings lists are processed simultaneously keeping them aligned by docid. In DAAT processing the score of each document is fully computed considering the contributions of all query terms before moving to the next document, thus no auxiliary per-document data structures are necessary. We will focus on the DAAT strategy as it is is more amenable to dynamic pruning techniques.

Solving scored ranked queries exhaustively with DAAT can be very inefficient. Various techniques to enhance retrieval efficiency have been proposed, by dynamically pruning docids that are unlikely to be retrieved. Among them, the most popular are MaxScore [30] and WAND [4]. Both strategies augment the index by storing for each term its maximum score contribution, thus allowing to skip large segments of posting lists if they only contain terms whose sum of maximum scores is smaller than the scores of the top *k* documents found up to that point.

The alignment of the posting lists during MaxScore and WAND processing can be achieved by means of the $\text{NextGEQ}_t(d)$ operator, which returns the smallest docid in the posting list t that is greater than or equal to d. This operator can significantly improve the posting list traversal speed during query processing, by skipping large amounts of irrelevant docids. The Elias-Fano compression scheme provides an efficient implementation of the $\text{NextGEQ}_t(d)$ operator, which is crucial to obtain the typical subsecond response times of Web search engines.

Both MaxScore and WAND rely on upper-bounding the contribution that each term can give to the overall document score, allowing to skip whole ranges of docids [18].

However, both employ a *global* per-term upper bound, that is, the maximum score $s_{t,d}$ among all documents *d* which contain the term *t*. Such maximum score could be significantly larger than the typical score contribution of that term, in fact limiting the opportunities to skip large amounts of documents. For example, a single outlier for an otherwise low-score term can make it impossible to skip any document that contains that term.

To tackle this problem, Ding and Suel [10] propose to augment the inverted index data structures with additional information to store more accurate upper bounds: at indexing time each posting list is split into consecutive blocks of constant size, e.g., 128 postings per block. For each block the score upper bound is computed and stored, together with largest docid of each block.

These *local* term upper bounds can then be exploited by adapting existing algorithms such as MaxScore and WAND to make use of the additional information. The first of such algorithms is Block-MaxWAND (BMW) [10]. The authors report an average speedup of BMW against WAND of 2.78 – 3.04. Experiments in [9] report a speedup of ~3.00 and ~1.25 of BMW with respect to WAND and MaxScore, respectively. Several versions of Block-Max MaxScore (BMM), the MaxScore variant for block-max indexes, have been proposed in [7, 9, 26]. In [9], the authors implementation of BMM is 1.25 times slower than BMW on average.

3 VARIABLE BLOCK-MAX WAND

As mentioned in the previous section, BMW leverages per-block upper bound information to skip whole blocks of docids during query processing (we refer to the original paper [10] for a detailed description of the algorithm). The performance of the algorithm highly depends on the size of the blocks: if the blocks are too large, the likelihood of having at least one large value in each block increases, causing the upper bounds to be loose. If they are too small, the average skip will be short. In both cases, the pruning effectiveness may reduce significantly. A sweet spot can thus be determined experimentally.

The constant-sized block partitioning of BMW does not take into consideration regularities or variances of the scores that may

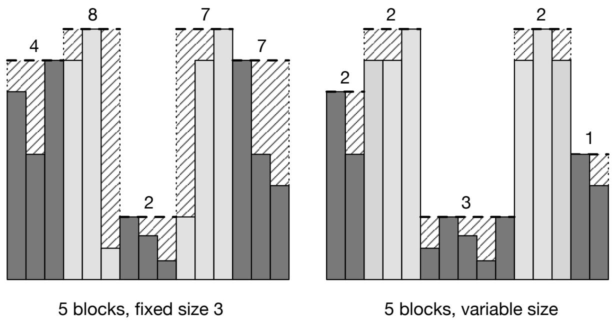

Figure 1: Block errors in constant (left) and variable (right) block partitioning.

occur in the posting lists and their blocks. The use of variable-sized blocks allows to better adapt to the distribution of the scores in the posting list.

The improvement with this kind of partitioning is apparent from the example in Figure 1. The figure shows a sequence of scores partitioned in constant-sized blocks and in variable-sized blocks. We define the *error* as the sum of the differences between each value and its block's upper bound, the shaded area in the figure. This example shows that a variable-sized partitioning can produce a much lower error, e.g., 28 in constant-sized partitioning (with blocks of length 3) versus 10 in variable-sized partitioning.

Problem definition. To give a more formal definition, for a partitioning of the sequence of scores in a posting list of n postings let $\mathcal{B}$ be the set of its blocks. Each block $B \in \mathcal{B}$ is a sequence of consecutive postings in the posting list. We use $b = |\mathcal{B}|$ and $|B|$ to denote the number of blocks of the partition and the number of postings in B, respectively. The term-document scores are defined above as $s_{t,d}$; however, since in the following we will work on one posting list at a time, we can drop the t, so s_d will denote the sequence of scores for each document d in the posting list.

We define the *error* of a partitioning $\mathcal{B}$ as follows:

$$\sum_{B \in \mathcal{B}} \left(|B| \max_{d \in B} s_d - \sum_{d \in B} s_d \right). \qquad (1)$$

Here for each block of postings we are accounting for the the sum of its individual posting errors, i.e., the sum of the differences between the block maximum score and the score of the posting.

To simplify the formula above we can notice that the right-hand side of the subtraction can be taken out of the sum, since the blocks form a partition of the list, and the resulting term does not depend on $\mathcal{B}$. Thus, minimizing the error is equivalent to minimizing the following formula, which represents the perimeter of the envelope, for a given number of blocks $b = |\mathcal{B}|$:

$$\sum_{B \in \mathcal{B}} |B| \max_{d \in B} s_d. \qquad (2)$$

Our goal is to find a block partitioning that minimizes the sum of block errors among all blocks in the partitioning. Naïvely, the minimum cost partitioning would correspond to blocks containing only a single posting. Since this solution clearly does not carry out any benefit, we fix the number of blocks in the partition to be b. As we will show in Section 5 minimizing the error can significantly improve BMW performance over constant-sized blocks.

Existing solutions. The problem of finding a partition that minimizes Equation (2) subject to a constraint b on the number of its blocks can be solved with a standard approach based on dynamic programming. The basic idea is to fill a $b \times n$ matrix M where entry $M[i][j]$ stores the minimum error to partition the posting list up to position j with i blocks. This matrix can be filled top-down from left to right. The entry $M[i][j]$ is computed by trying to place the jth posting in the optimal solutions that uses $i - 1$ blocks. Unfortunately, the time complexity of this solution is $\Theta(bn^2)$, which is $\Theta(n^3)$ since, given that the average block size n/b is small (e.g., 32–128), thus, the interesting values of b are $\Theta(n)$. This algorithm is clearly unfeasible because n can easily be in the range of millions.

This optimization problem is similar in nature to the well-studied problem of computing optimal histograms (see again Figure 1). The complexity of finding the best histogram with a given number of bars is the same as above. Several approximate solutions have been presented. Halim et al. [16] describe several solutions and introduce an algorithm that has good experimental performance but no theoretical guarantees. All such solutions are polynomial either in n or in b. Some have complexity $O(nb)$. Guha et al. [15] introduce a $(1 + \epsilon)$ approximation with $O(n + b^3 \log n + b^2/\epsilon)$ time. While these techniques can be useful in cases where b is small, in our case $b = \Theta(n)$, which makes these algorithms unfeasible for us. Furthermore, the definition of the objective function in these works is different from ours, as it minimizes the variance rather than the sum of the differences.

Our solution. We first present a practical and efficient algorithm with weaker theoretical guarantees regarding the optimal solution than what would be expected. Indeed, fixed the required number of blocks b and an approximation parameter ϵ, with $0 < \epsilon < 1$, the algorithm finds a partition with $b' \leq b$ blocks whose cost is at most a factor $1 + \epsilon$ larger than the cost of the optimal partition with b' edges. This algorithm runs in $O(n \log_{1+\epsilon} \frac{1}{\epsilon} \log(Un/b))$ time, where U is the largest cost of any block. The weakness is due to the fact that there is no guarantee on how much b' is close to the requested number of blocks b. Even with this theoretical gap, in all our experiments the algorithm identified a solution with a number of blocks very close to the desired one. In the last part of the section, we will fill this gap by showing how to refine the solution to always identify a $1 + \epsilon$ approximated optimal solution with exactly b edges.

The first solution is a variation of the approximate dynamic programming algorithm introduced by Ottaviano and Venturini [23] to optimize the partitioning of Elias-Fano indexes.

It is convenient to look at the problem as a shortest path problem over a directed acyclic graph (DAG). The nodes of the graph correspond to the postings in the list; the edges connect each ordered pair $i < j$ of nodes, and represent the possible blocks in the partition. The cost $c(i, j)$ associated to the edge is thus $(j - i) \max_{i \leq d < j} s_d$.

In this graph, denoted as G, each path represents a possible partitioning, and the cost of the path is equal to the cost of the partitioning as defined in (2). Thus, our problem reduces to an instance of *constrained shortest path* on this graph, that is, finding the shortest path with a given number of edges [13, 20].

We can compute the constrained shortest path with an approach similar to the one in [1, 13, 20]. The idea is to reduce the problem to a standard, unconstrained shortest path by using Lagrangian relaxation: adding a fixed cost $\lambda \geq 0$ to every edge. We denote the relaxed graph as G_λ. By varying λ, the shortest path in G_λ will have a different number of edges: if $\lambda = 0$, the solution is the path of $n - 1$ edges of length one; at the limit $\lambda = +\infty$, the solution is a single edge of length n. It can be shown that, for any given λ, if the shortest path in G_λ has ℓ edges, then that path is an optimal ℓ-constrained shortest path in G. Thus, our goal is to find the value of λ that give $\ell = b$ edges. However, notice that not every b can be found this way, but in practice we can get close enough. Thus, our algorithm performs a binary search to find the value of λ that gives a shortest path with b' edges, with b' close enough to b. Each step of the binary search requires a shortest-path computation.

Each of these shortest-path computations can be solved in $O(|V| + |E|)$, where V are the vertices of G_λ and E the edges; for our problem, unfortunately, this is $\Theta(n^2)$, which is still unfeasible. We can however exploit two properties of our cost function to apply the algorithm in [23] and obtain a linear-time approximate solution for a given value of λ. These properties are *monotonicity* and *quasi-subadditivity*. The monotonicity property is stated as follows.

Property 1. *(Monotonicity) A function $f : V \times V \mapsto \mathbb{R}$ is said* monotone *if for each pair of values $i, j \in V$ the following holds:*

- $f(i, j + 1) \geq f(i, j)$,
- $f(i - 1, j) \geq f(i, j)$.

It is easy to verify that our cost function $c(i, j)$ satisfies Property 1, because if a block B is contained in a block B', then it follows immediately from the definition that the cost of B' is greater than the cost of B. Monotonicity allows us to perform a first pruning of G_λ: for any given approximation parameter $\alpha \in (0, 1]$, we define G^1_λ as the graph with the same nodes as G_λ, and all the edges (i, j) of G_λ that satisfy at least one of the following conditions.

(1) There exists an integer h such that

$$c(i, j) \leq \lambda(1 + \alpha)^h < c(i, j + 1)$$

(2) (i, j) is the last outgoing edge from i.

The number of edges in G^1_λ is $n \log_{1+\alpha}(\frac{U}{\lambda})$ where U is the maximum cost of an edge (which is equal to $n \max_d s_d$).

We denote as π_{G_λ} the shortest path of the graph G_λ and extend c to denote the cost of a path. It can be shown that $c(\pi_{G^1_\lambda}) \leq (1 + \alpha)c(\pi_{G_\lambda})$, that is, the optimal solution in G^1_λ is a $(1 + \alpha)$ approximation of the optimal solution in G_λ; see [14] for the proof. The complexity to find the shortest path decreases from $O(n^2)$ to $O(n \log_{1+\alpha}(\frac{U}{\lambda}))$. This would be already applicable in many practical scenarios, but it depends on the value U of the maximum score. We can further refine the algorithm in order to decrease the complexity and drop the dependency on U by adding an extra approximation function $(1 + \beta)$ for any given approximation parameter $\beta \in (0, 1]$, by leveraging the quasi-subadditivity property.

Property 2. *(Quasi-subadditivity) A function $f : V \times V \mapsto \mathbb{R}$ is said* λ-quasi-subadditive *if for any i, k and $j \in V$, with $0 \leq i < l < j < |V|$ the following holds:*

$$f(i, k) + f(k, j) \leq f(i, j) + \lambda.$$

It is again immediate to show that $c(i, j)$ satisfies Property 2: splitting a block at any point can only lower the upper bound in the two resulting sub-blocks, so the only extra cost is the additional λ of the new edge.

This property allows us to prune from G^1_λ all the edges with cost higher than $L = \lambda + \frac{2\lambda}{\beta}$; we call the resulting graph G^2_λ. The new graph has $O(n \log_{1+\alpha} \frac{1}{\beta}) = \Theta(n)$ edges, thus shortest paths can be computed in linear time. It can be shown (see [23]) that this pruning incurs an extra $(1 + \beta)$ approximation; the overall approximation factor is thus $(1 + \alpha)(1 + \beta)$, which is $1 + \epsilon$ for any $\epsilon \in (0, 1]$ by appropriately fixing $\alpha = \beta = \frac{\epsilon}{3}$.

Clearly it is not feasible to materialize the graph G_λ and prune it to obtain G^2_λ, since the dominating cost would still be the initial quadratic phase. It is however possible to visit the graph G^2_λ without constructing it explicitly, as described in [23].

By using the above algorithm, every shortest path computation requires $O(n \log_{1+\epsilon} \frac{1}{\epsilon}) = \Theta(n)$ time and linear space.

Since we are binary searching on λ, the number of required shortest path computations depends on the range of possible values of λ. It is easy to see that $\lambda \geq 0$. Indeed, the shortest path in G_0 has the largest possible number of edges, $n - 1$ and the smallest possible cost. We now prove that the shortest path in G_λ with $\lambda > Un/(b-1)$ has less than b edges, where U is the largest cost on G. Thus, in the binary search we can restrict our attention to integer values of λ in $[0, Un/(b - 1)]$. The proof is as follows. Consider the optimal path with one edge in G, and let O_1 be its cost. By monotonicity, we know that $O_1 = U$. Let O_b be the cost of the best path with b edges in G. For any λ, the cost of these two paths in G_λ are $O_1 + \lambda$ and $O_b + b\lambda$. Observe that if $\lambda > Un/b$, the former path has a cost which is smaller than the cost of the latter. This means that we do not need to explore values of λ larger than $Un/(b - 1)$ when we are looking for a path with b edges. Thus, the first phase of the algorithm needs $O(\log(Un/b))$ shortest path computations to find the target value of λ. Thus, if we restrict our search to integer values of λ, the number of shortest path computations is $O(\log(Un/b))$.

We can refine the above solution to find a provable good approximation of the shortest path with *exactly* b edges. The refinement uses the result in [13]. Theorem 4.1 in [13] states that, given a DAG G with integer costs which satisfy the monotonicity property, we can compute an additive approximation of the constrained shortest path of G. More precisely, we can compute a path with b edges such that its cost is at most $O_b + U$, where O_b is the cost of an optimal path with b edges and U is the largest cost on G. The algorithms works in two phases. In the first phase, it reduces the problem to a standard, unconstrained shortest path by using Lagrangian relaxation as we have done in our first solution. Thus, the first phase binary searches for the value of λ for which the shortest path on G_λ with the least number of edges has at most b edges, while the one with the most edges has at least b edges. If one of these two paths has exactly b edges, this is guaranteed to be an optimal solution and we are done. Otherwise, we start the second phase of the algorithm. The second phase is called *path-swapping* and its goal is to combine these two paths to find a path with b edges whose cost is worse than the optimal one by at most an additive term A, which equals the largest cost in the graph. We refer to [1] and [13] for more details.

We cannot immediately apply the above optimization algorithm because of two important issues. In the following we will introduce and solve both of them.

The first issue is that the above optimization algorithm assumes that the costs in G are integers, while in our case are not. The idea is to obtain a new graph with integer costs by rescaling and rounding the original costs of G. More precisely, we can obtain a new graph by replacing any cost $c(i,j)$ with $\lceil c(i,j)/\delta \rceil$, where $\delta \in (0,1]$ is an approximation parameter. We can prove that this operation slightly affects the cost of the optimal path. Indeed, let O_b the cost of the shortest path with b edges in G, the shortest path on the new graph as cost $\tilde{O}_b$ which is $O_b \leq \tilde{O}_b \leq O_b + \delta b$. Due to space limitations, we defer the proof of this inequality to the journal version of the paper. Even if in general we cannot bound the additive approximation δb in terms of O_b, in practice the approximation is negligible because O_b is much larger that δb. Notice that this approximation increases U to $\frac{U}{\delta}$.

The second issue to address is the fact that additive approximation term A in the result of [13] is the largest edge cost U. In our problem this additive approximation term is the cost of the edge from 1 to n, which equals the cost of the worst possible path. This means that the obtained approximation would be trivial. However, we observe that, due to the approach of the previous paragraph, the largest cost on the approximated graph G^2_λ is $L = \lambda + \frac{2\lambda}{\beta}$ and we know that $\lambda \leq Un/b$. Thus, the additive approximation term A is $O(\frac{Un}{b\beta})$, which is negligible in practice.

Thus, we obtained the following theorem.

Theorem 3.1. *Given a sequence of scores $S[1,n]$ and a fixed number of blocks b, we can compute a partition of S into b blocks whose cost is at most $(1+\epsilon)O_b + O(\frac{Un}{b\epsilon}) + \delta b$ in $O(n \log_{1+\epsilon} \frac{1}{\epsilon} \log(\frac{Un}{b\delta\epsilon}))$ time and linear space, where O_b is the cost of the optimal partition with b blocks, $U = \sum_{i=1}^{n} S[i]$, and $\epsilon, \delta \in (0,1]$ are the two approximation parameters.*

4 REPRESENTING THE UPPER BOUNDS

BlockMaxWAND is required to store additional information about the block upper bounds. This additional information must be stored together with the traditional inverted index data structures, and while these upper bounds can improve the time efficiency of query processing, they introduce a serious space overhead problem.

The additional information required by BlockMaxWAND can be seen as two aligned sequences: the sequence of block boundaries, that is, the largest docid in each block, and the score upper bound for each block.

In the original implementation, the sequences are stored uncompressed, using constant-width encodings (for example, 32-bit integers for the boundaries and 32-bit floats for the upper bounds), and are usually interleaved to favor cache locality. We can however use more efficient encodings to reduce the space overhead.

First, we observe that the sequence of block boundaries is monotonic, so it can be efficiently represented with Elias-Fano. In addition to saving space, Elias-Fano provides an efficient NextGEQ operation that can be used to quickly locate the block containing the current docid at query execution time.

Second, as far as the upper bounds are concerned, we can reduce space use by *approximating* their value. The only requirement to preserve the correctness of the algorithm is that each approximate value is an upper bound for all the scores in its block. Thus, we can use the following quantization. First, we partition the score space into fixed size buckets. Any score is represented with the identifier of its bucket. Let us assume that the score space is $[0,U]$ and that we partition it into w buckets. Then, instead of storing a block upper bound with value $s \in [0,U]$, we store the identifier i such that $\frac{iU}{w} < s \leq \frac{(i+1)U}{w}$. At query time, the actual score s will be approximated with the largest possible value in its bucket, i.e., $\frac{(i+1)U}{w}$. Clearly, the representation of any score requires $\lfloor \log w + 1 \rfloor$ bits, a large space saving with respect to the 32 bits of the float representation. Obviously, the value of w can be chosen to trade off the space usage and the quality of the approximation.

A simple optimization to speed up access is to interleave the two sequences, by modifying of the Elias-Fano data structure. Elias-Fano stores a monotonic sequence by splitting each value into its ℓ *low bits*, and the remaining *high bits*. The value of ℓ a constant for the sequence. While the high bits are encoded with variable-length, the low bits are encoded verbatim in exactly ℓ bits per element, thus the low bits of the i-th element are at the position $i\ell$ of the low bitvector. We can then interleave the low bits and the quantized score by using a bitvector of $(\ell + w)$-bit entries, so that when the block is located, its quantized upper bound is already in cache.

5 EXPERIMENTAL RESULTS

In this section we analyze the performance of VBMW with an extensive experimental evaluation in a realistic and reproducible setting, using state-of-the-art baselines, standard benchmark text collections, and a large query log.

Testing details. All the algorithms are implemented in C++11 and compiled with GCC 5.4.0 with the highest optimization settings. The tests are performed on a machine with 8 Intel Core i7-4770K Haswell cores clocked at 3.50GHz, with 32GiB RAM, running Linux 4.4.0. The indexes are saved to disk after construction, and memory-mapped to be queried, so that there are no hidden space costs due to loading of additional data structures in memory. Before timing the queries we ensure that the required posting lists are fully loaded in memory. All timings are measured taking the results with minimum value of five independent runs. All times are reported in milliseconds.

The source code is available at https://github.com/rossanoventurini/Variable-BMW for the reader interested in further implementation details or in replicating the experiments.

Datasets. We performed our experiments on the following standard datasets.

- ClueWeb09 is the ClueWeb 2009 TREC Category B collection, consisting of 50 million English web pages crawled between January and February 2009.
- Gov2 is the TREC 2004 Terabyte Track test collection, consisting of 25 million .gov sites crawled in early 2004; the documents are truncated to 256 kB.

For each document in the collection the body text was extracted using Apache Tika[1], the words lowercased and stemmed using the

[1] http://tika.apache.org

Table 1: Basic statistics for the test collections

	ClueWeb09	Gov2
Documents	50,131,015	24,622,347
Terms	92,094,694	35,636,425
Postings	15,857,983,641	5,742,630,292

Porter2 stemmer; no stopwords were removed. The docids were assigned according to the lexicographic order of their URLs [27]. Table 1 reports the basic statistics for the two collections. If not differently specified, the inverted index is compressed by using partitioned Elias-Fano (PEF) [23] in the ds2i library[2].

Queries. To evaluate the speed of query processing we use Trec05 and Trec06 Efficiency Track topics, drawing only queries whose terms are all in the collection dictionary and having more than 128 postings. These queries are, respectively, the 90% and 96% of the total Trec05 and Trec06 queries for the Gov2 collection and the 96% and 98% of the total Trec05 and Trec06 queries for the ClueWeb09 collection. From those sets of queries we randomly select 1 000 queries for each length.

Processing strategies. To test the performance on query strategies that make use of the docids and the occurrence frequencies we perform BM25 top 10 queries using 5 different algorithms: RankedOR, which scores the results of a disjunctive query, WAND [4], MaxScore [30], BlockMaxWAND (BMW) [10], and the proposed Variable BMW (VBMW) in its uncompressed and compressed variants.

We use BMW_x to indicate that the *fixed* block size in BMW is x postings, while we use VBMW_x to indicate that the *average* block size in VBMW is x postings. The compressed version of VBMW as described in Section 4 is denoted as $\mathrm{C\text{-}VBMW}_x$.

Validating our BMW implementation. We implemented our version of BMW because the source code of the original implementation was not available. To test the validity of our implementation we compared its average query time with the ones reported in [10]. We replicated their original setting by using the same dataset (Gov2), by compressing postings with the same algorithm (PForDelta), by using queries from the same collections (Trec05 and Trec06), and by using BMW_{64}. However, since we are using a different faster machine, we cannot directly compare query times, but, instead, we compare the improving factors with respect to RankedOR, which is an easy-to-implement baseline.

Table 2 shows the query times reported in the original paper (top) and the ones obtained with our implementation (bottom). Results show that the two implementations are comparable, with ours which is generally faster. For example, it is faster by a factor larger than 2.4 on queries with more than three terms in Trec06.

The effect of the block size in BMW. Although the most commonly used block sizes for BMW are 64 and 128, a more careful experimental evaluation shows that the best performance in terms of query time is obtained with a block size of 40 postings.

Table 3 shows the average query time of BMW with respect to Trec05 and Trec06 on both Gov2 and ClueWeb09, by varying the block size. We select the block size in the set $\{32, 40, 48, 64, 96, 128\}$. It is clear that in all cases, the best average query time is achieved with blocks size 40. BMW_{40} is 10% faster, on average, than BMW_{128}.

Table 2: Query times (in ms) of RankedOR and BMW_{64} on Gov2 with queries in Trec05 and Trec06 as reported by Ding and Suel [10] (top) and the ones obtained with our implementation (bottom), for different query lengths.

	Number of query terms				
	2	3	4	5	6+
Trec05 (from [10])					
RankedOR	62.1 (x17.7)	238.9 (x18.8)	515.2 (x20.4)	778.3 (x25.9)	1,501.4 (x14.4)
BMW_{64}	3.5	12.7	25.2	30.0	104.0
Trec06 (from [10])					
RankedOR	60.0 (x14.7)	159.2 (x13.8)	261.4 (x7.8)	376.0 (x6.9)	646.4 (x5.7)
BMW_{64}	4.1	11.5	33.6	54.5	114.2
Trec05					
RankedOR	15.5 (x13.2)	51.3 (x17.3)	100.3 (x22.6)	158.0 (x22.7)	275.1 (x17.3)
BMW_{64}	1.2	3.0	4.5	7.0	15.9
Trec06					
RankedOR	15.5 (x14.7)	57.6 (x16.9)	117.6 (x19.7)	178.0 (x18.5)	311.2 (x13.8)
BMW_{64}	1.1	3.4	6.0	9.6	22.5

Table 3 also reports the space usage of the (uncompressed) additional information stored by BMW, namely the largest score in the block (as float) and the last posting in the block (as unsigned int). Posting lists with fewer postings than the block size do not store any additional information. The size of the inverted index of the Gov2 and ClueWeb09 collections (compressed with PEF) is 4.32 GiB and 14.84 GiB respectively. Thus, the space of the additional information required by BMW is not negligible, since it ranges between 15% and 42% of the compressed inverted index space on both Gov2 and ClueWeb09. As we will see later, this space usage can be reduced significantly by compressing the additional information.

The effect of the block size in VBMW. Now, we proceed by analyzing the behavior of VBMW. Instead of adopting the more sophisticated approximation approach detailed in Section 3, we use the simpler optimization algorithm which has no theoretical guarantees on the final number of blocks. Thus, we cannot choose an exact block size for our partitioning but we binary search for the λ in the parameter space that gives an average block size close to the values in $\{32, 40, 48, 64, 96, 128\}$.

Table 4 reports the average block sizes and score errors for different block sizes w.r.t. BMW and VBMW on Gov2 and ClueWeb09, and optimal values for the Lagrangian relaxation parameter λ. Note that for BMW, the average block size is not perfectly identical to the desired block size due to the length of the last block in the posting lists, which may be smaller than the desired block size. Our optimization algorithm is able to find an average block size for VBMW within 3% of the average block size for BMW. Thus, the weaker optimization algorithm of Section 3 suffices in practice to obtain the desired average block sizes. More importantly, the

[2]https://github.com/ot/ds2i

Table 3: Space usage of the additional data required by BMW and average query times with queries in Trec05 and Trec06 on Gov2 and ClueWeb09, by varying the block size.

	Block size					
	32	40	48	64	96	128
Additional space (GiB)						
Gov2	1.83	1.55	1.38	1.15	0.92	**0.85**
ClueWeb09	5.04	4.14	3.62	3.04	2.40	**2.24**
Query time (ms) on Trec05						
Gov2	**3.6**	**3.6**	3.7	3.8	3.9	4.2
ClueWeb09	12.8	**12.6**	**12.6**	12.8	13.3	13.9
Query time (ms) on Trec06						
Gov2	8.3	**8.2**	8.3	8.5	8.9	9.2
ClueWeb09	**26.4**	**26.3**	26.5	27.0	28.0	29.4

Table 4: Average block sizes and score errors for different block sizes w.r.t. BMW and VBMW on Gov2 and ClueWeb09, and optimal values for the Lagrangian relaxation parameter.

		Block Size					
		32	40	48	64	96	128
Gov2							
Average Block Size	BMW	31.94	39.90	47.87	63.74	95.35	127.14
	VBMW	31.32	39.63	47.09	63.60	98.40	126.30
Average Score Error	BMW	1.47	1.55	1.61	1.70	1.83	1.92
	VBMW	0.82	0.91	0.98	1.09	1.26	1.35
λ	VBMW	12.0	15.2	18.0	24.0	35.1	45.9
ClueWeb09							
Average Block Size	BMW	31.96	39.94	47.91	63.83	95.65	127.29
	VBMW	30.24	39.54	48.03	63.29	97.43	127.72
Average Score Error	BMW	1.94	2.05	2.15	2.29	2.49	2.63
	VBMW	1.20	1.34	1.45	1.60	1.83	1.98
λ	VBMW	16.0	21.0	25.5	33.4	50.3	64.5

average score error for VBMW is sensibly smaller than the average score error for BMW, with a reduction ranging from 40% for small blocks up to 25% for large blocks. This confirms the importance of partitioning the posting lists with variable-sized blocks.

In Table 5 we can see that VBMW reaches the best average query times with approximatively 32 – 40 elements per block, similar to the best block size for BMW reported in Table 3, i.e., 40 postings per block. As shown in Figure 2, the trade-off in choosing this block size w.r.t. average query time is that we use more space to store block information, as reported in Table 3.

The effect of compression in VBMW. Figure 2 shows how the choice of w affects both query time and space usage of C-VBMW when the average number of blocks is fixed to 40 elements. We fixed the number of buckets w to quantize the scores to the powers of two from 32 to 512 and we reported the query time and the space of the additional information on both datasets with both set of queries. For comparison, we also plot the results of the plain version of VBMW by varying the average size of the blocks.

Table 5: Average query times of VBMW with queries in Trec05 and Trec06 on Gov2 and ClueWeb09, by varying the block size.

	Block size					
	32	40	48	64	96	128
Query time (ms) on Trec05						
Gov2	**2.1**	**2.1**	**2.1**	2.2	2.5	2.8
ClueWeb09	**7.2**	**7.2**	7.4	8.1	9.7	11.0
Query time (ms) on Trec06						
Gov2	**4.6**	4.7	4.8	5.3	6.1	6.9
ClueWeb09	**14.7**	15.2	16.1	17.8	21.2	23.7

The first conclusion is that the compression approach is very effective. Indeed, C-VBMW improves space usage by roughly a factor 2 with respect to VBMW$_{40}$. We also notice that the compression approach is more effective than simply increasing the block size in the uncompressed VBMW. Indeed, for example, C-VBMW with $w = 32$ uses almost the same space as VBMW$_{128}$ but is faster by 20% – 40%.

The second conclusion is that compression does not decrease query time which actually sometimes even improves. For example, C-VBMW with $w = 512$ and $w = 256$ is faster that its uncompressed version (VBMW$_{40}$) on both datasets with Trec05. This effect may be the results to a better cache usage resulting from the smaller size of additional information in C-VBMW.

We observe that there are small differences (less than 10%) in efficiency between the different values of w. Thus, for the next experiments we will fix w to 512 to obtain the best time efficiency.

Overall comparison. To carefully evaluate the performance of C-VBMW w.r.t. other processing strategies, we measured the query times of different query processing algorithms for different query lengths, from 2 terms queries to more than 5 terms queries, as well as the overall average processing times and the space use of any required additional data structure with respect the whole inverted indexes represented with PEF.

In Table 6, next to each timing is reported in parenthesis the relative speedup of C-VBMW$_{40}$ with respect to this strategy. Table 6 also reports, in GiB, the additional space usage required by the different query processing strategies. Next to each size measure is reported in parenthesis the relative percentage against the data structures used to compress posting lists storing docids and frequencies only, as used by RankedOR.

Not very surprisingly, RankedOR is always at least 34 times slower than C-VBMW$_{40}$, while both MaxScore and WAND are from 1.4 to 11 times slower than C-VBMW$_{40}$. The maximum speedup of C-VBMW$_{40}$ is achieved with queries of two terms where it ranges from 6.5 to 11. Space usage of MaxScore and WAND plainly store the score upper bounds for each term using the 4% – 5% of the inverted index.

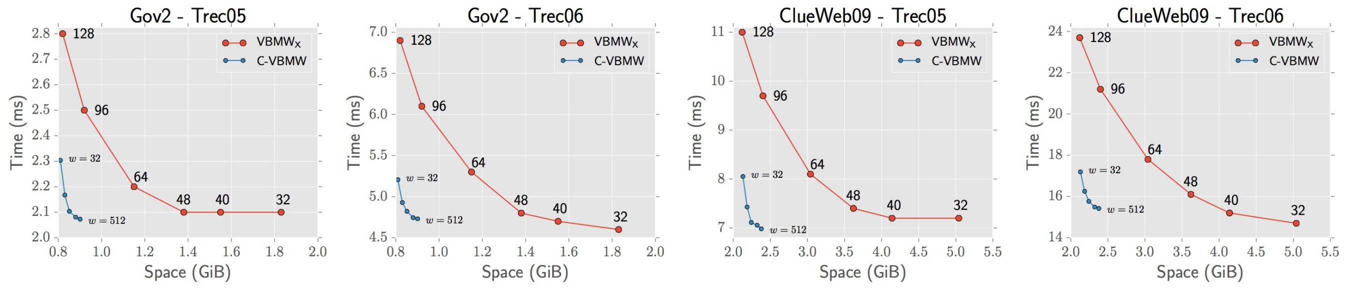

Figure 2: Space consumed vs. average query times of VBMW with different block sizes and C-VBMW with block size 40 by varying w for 32 to 512 with queries in Trec05 and Trec06 on Gov2 and ClueWeb09.

All block-based strategies report a minimal variance of query times among different query lengths. For both the most common block size (128 postings per block) and the most efficient one (40 postings per block), VBMW strategies process queries faster than BMW strategies, with the same space occupancies. The corresponding compressed versions, C-VBMW$_{128}$ and C-VBMW$_{40}$, sensibly reduce the space occupancies (by 6% and 17% respectively) but while C-VBMW$_{128}$ never processes queries faster than the corresponding uncompressed VBMW$_{128}$, C-VBMW$_{40}$ does not show relevant performance losses with respect to VBMW$_{128}$, but exhibits some cache-dependent benefits for short queries.

With respect to the current state-of-the-art processing strategy BMW$_{128}$, our best strategy in terms of query times is C-VBMW$_{40}$, able to improve the average query time by a factor of roughly 2×, effectively halving the query processing times for all query lengths, with a relative $3\% - 5\%$ gain in space occupancy. If space occupancy is the main concern, our best strategy is C-VBMW$_{128}$, able to reduce the space by a relative 30% against BMW$_{128}$, while still boosting the query times by a factor of roughly 1.5×.

6 CONCLUSIONS

We introduced Variable BMW, a new query processing strategy built on top of BlockMaxWAND. Our strategy uses variable-sized blocks, rather than constant-sized. We formulated the problem of partitioning the posting lists of a inverted index into variable-sized blocks to minimize the average block error, subject to a constraint on the number of blocks, and described an efficient algorithm to find an approximate solution, with provable approximation guarantees. We also introduced a compressed data structure to represent the additional block information. Variable BMW significantly improves the query processing times, by a factor of roughly 2× w.r.t. the best state-of-the-art competitor. Our new compression scheme for the block data structures, compressing the block boundary docids with Elias-Fano and quantizing the block max score, provides a maximum reduction of space usage w.r.t. the uncompressed data structures of roughly 50%, while incurring only a small speed degradation, no more than 10% with respect to its uncompressed counterpart.

Future work will focus on exploring the different space-time trade-offs that can be obtained by varying the quantization scheme exploited in the compression of the additional data structures.

ACKNOWLEDGMENTS

This work was partially supported by the EU H2020 Program under the scheme INFRAIA-1-2014-2015: *Research Infrastructures*, grant agreement #654024 *SoBigData: Social Mining & Big Data Ecosystem*.

REFERENCES

[1] Alok Aggarwal, Baruch Schieber, and Takeshi Tokuyama. 1994. Finding a Minimum-Weight k-Link Path Graphs with the Concae Monge Property and Applications. *Discrete & Computational Geometry* 12 (1994), 263–280.
[2] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. 2001. Vector-space ranking with effective early termination. In *SIGIR*. 35–42.
[3] Nima Asadi and Jimmy Lin. 2013. Effectiveness/Efficiency Tradeoffs for Candidate Generation in Multi-stage Retrieval Architectures. In *SIGIR*. 997–1000.
[4] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Y. Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *CIKM*. 426–434.
[5] Stefan Büttcher, Charles L.A. Clarke, and Gordon V. Cormack. 2010. *Information retrieval: implementing and evaluating search engines.* MIT Press.
[6] Stefan Büttcher and Charles L. A. Clarke. 2007. Index compression is good, especially for random access. In *CIKM*. 761–770.
[7] Kaushik Chakrabarti, Surajit Chaudhuri, and Venkatesh Ganti. 2011. Interval-based Pruning for Top-k Processing over Compressed Lists. In *ICDE*. 709–720.
[8] Jeffrey Dean. 2009. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM*.
[9] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. 2013. Optimizing Top-k Document Retrieval Strategies for Block-max Indexes. In *WSDM*. 113–122.
[10] Shuai Ding and Torsten Suel. 2011. Faster top-k document retrieval using block-max indexes. In *SIGIR*. 993–1002.
[11] Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. *J. ACM* 21, 2 (1974), 246–260.
[12] Robert M. Fano. 1971. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT, Cambridge, MA* (1971).
[13] Andrea Farruggia, Paolo Ferragina, Antonio Frangioni, and Rossano Venturini. 2014. Bicriteria data compression. In *SODA*. 1582–1595.
[14] Paolo Ferragina, Igor Nitto, and Rossano Venturini. 2011. On Optimally Partitioning a Text to Improve Its Compression. *Algorithmica* 61, 1 (2011), 51–74.
[15] Sudipto Guha, Nick Koudas, and Kyuseok Shim. 2006. Approximation and Streaming Algorithms for Histogram Construction Problems. *ACM Trans. Database Syst.* 31, 1 (2006), 396–438.
[16] Felix Halim, Panagiotis Karras, and Roland H.C. Yap. 2009. Fast and Effective Histogram Construction. In *CIKM*. 1167–1176.
[17] Daniel Lemire and Leonid Boytsov. 2015. Decoding Billions of Integers Per Second Through Vectorization. *Softw. Pract. Exper.* 45, 1 (2015), 1–29.
[18] Craig Macdonald, Iadh Ounis, and Nicola Tonellotto. 2011. Upper-bound approximations for dynamic pruning. *ACM Trans. Inf. Syst.* 29, 4 (2011), 17.
[19] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schülze. 2008. *Introduction to Information Retrieval.* Cambridge University Press.
[20] Kurt Mehlhorn and Mark Ziegelmann. 2000. Resource Constrained Shortest Paths. In *ESA*. 326–337.
[21] Alistair Moffat, William Webber, Justin Zobel, and Ricardo Baeza-Yates. 2007. A pipelined architecture for distributed text query evaluation. *Inf. Retr.* 10, 3 (2007), 205–231.
[22] Giuseppe Ottaviano, Nicola Tonellotto, and Rossano Venturini. 2015. Optimal Space-time Tradeoffs for Inverted Indexes. In *WSDM*. 47–56.
[23] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano Indexes. In *SIGIR*. 273–282.

Table 6: Query times (in ms) of different query processing strategies for different query lengths, average query times (Avg, in ms) and additional space (Space, in GiB) w.r.t. Trec05 and Trec06 on Gov2 and ClueWeb09.

	Number of query terms					Avg	Space
	2	3	4	5	6+		
			Gov2 Trec05				
RankedOR	23.6 (x32.89)	76.5 (x44.39)	147.9 (x59.74)	235.4 (x60.47)	418.7 (x50.18)	106.7 (x50.88)	0.00
WAND	5.1 (x7.11)	5.9 (x3.43)	7.0 (x2.82)	8.8 (x2.27)	17.8 (x2.13)	7.0 (x3.36)	0.22 (5%)
MaxScore	4.7 (x6.61)	6.0 (x3.45)	7.1 (x2.86)	9.2 (x2.37)	14.2 (x1.70)	6.6 (x3.14)	0.22 (5%)
BMW$_{40}$	1.2 (x1.62)	2.9 (x1.65)	4.3 (x1.72)	6.7 (x1.72)	14.8 (x1.78)	3.6 (x1.74)	1.55 (36%)
VBMW$_{40}$	0.8 (x1.10)	1.7 (x1.01)	2.4 (x0.97)	3.8 (x0.97)	8.1 (x0.97)	2.1 (x1.00)	1.55 (36%)
C-VBMW$_{40}$	0.7	1.7	2.5	3.9	8.3	2.1	0.82 (19%)
BMW$_{128}$	1.4 (x1.99)	3.5 (x2.01)	4.8 (x1.93)	7.2 (x1.85)	15.9 (x1.90)	4.2 (x1.98)	0.85 (20%)
VBMW$_{128}$	1.0 (x1.42)	2.4 (x1.40)	3.2 (x1.30)	4.9 (x1.26)	10.7 (x1.28)	2.8 (x1.34)	0.85 (20%)
C-VBMW$_{128}$	1.1 (x1.53)	2.5 (x1.47)	3.4 (x1.37)	5.1 (x1.32)	11.3 (x1.36)	3.0 (x1.42)	0.58 (13%)
			Gov2 Trec06				
RankedOR	23.1 (x34.72)	83.3 (x42.20)	169.1 (x52.34)	261.3 (x52.20)	470.7 (x37.56)	212.0 (x44.41)	0.00
WAND	4.7 (x7.12)	8.0 (x4.06)	9.3 (x2.86)	12.4 (x2.47)	26.3 (x2.10)	12.9 (x2.69)	0.22 (5%)
MaxScore	4.5 (x6.78)	7.8 (x3.97)	9.2 (x2.83)	11.7 (x2.34)	19.4 (x1.55)	11.3 (x2.37)	0.22 (5%)
BMW$_{40}$	1.1 (x1.58)	3.2 (x1.62)	5.5 (x1.70)	8.7 (x1.74)	22.1 (x1.76)	8.2 (x1.73)	1.55 (36%)
VBMW$_{40}$	0.8 (x1.12)	2.0 (x1.02)	3.2 (x0.98)	4.9 (x0.98)	12.3 (x0.98)	4.7 (x0.98)	1.55 (36%)
C-VBMW$_{40}$	0.7	2.0	3.2	5.0	12.5	4.8	0.82 (19%)
BMW$_{128}$	1.2 (x1.88)	3.9 (x1.98)	6.5 (x2.01)	10.1 (x2.03)	23.8 (x1.90)	9.2 (x1.93)	0.85 (20%)
VBMW$_{128}$	0.9 (x1.39)	3.1 (x1.56)	4.7 (x1.45)	7.3 (x1.46)	17.3 (x1.38)	6.9 (x1.43)	0.85 (20%)
C-VBMW$_{128}$	1.0 (x1.48)	3.2 (x1.64)	4.8 (x1.50)	7.6 (x1.52)	18.4 (x1.47)	7.2 (x1.51)	0.58 (13%)
			ClueWeb09 Trec05				
RankedOR	77.9 (x36.01)	228.3 (x42.15)	429.3 (x55.20)	659.7 (x50.14)	1,214.0 (x41.72)	312.6 (x43.76)	0.00
WAND	23.8 (x10.98)	29.2 (x5.40)	25.7 (x3.31)	29.1 (x2.21)	57.1 (x1.96)	28.7 (x4.01)	0.53 (4%)
MaxScore	19.3 (x8.91)	22.9 (x4.23)	22.7 (x2.92)	28.1 (x2.14)	42.2 (x1.45)	23.4 (x3.28)	0.53 (4%)
BMW$_{40}$	4.2 (x1.93)	10.2 (x1.89)	14.7 (x1.89)	22.5 (x1.71)	49.7 (x1.71)	12.6 (x1.76)	4.14 (28%)
VBMW$_{40}$	2.7 (x1.23)	5.7 (x1.06)	7.8 (x1.01)	12.7 (x0.96)	27.8 (x0.96)	7.2 (x1.01)	4.14 (28%)
C-VBMW$_{40}$	2.2	5.4	7.8	13.2	29.1	7.1	2.12 (14%)
BMW$_{128}$	3.9 (x1.80)	11.2 (x2.06)	16.3 (x2.10)	25.6 (x1.94)	54.0 (x1.85)	13.9 (x1.94)	2.24 (15%)
VBMW$_{128}$	3.1 (x1.43)	8.9 (x1.63)	12.0 (x1.55)	19.2 (x1.46)	42.2 (x1.45)	11.0 (x1.54)	2.24 (15%)
C-VBMW$_{128}$	3.3 (x1.53)	9.6 (x1.77)	12.8 (x1.65)	20.4 (x1.55)	45.4 (x1.56)	12.0 (x1.67)	1.48 (10%)
			ClueWeb09 Trec06				
RankedOR	60.6 (x33.63)	215.9 (x37.04)	439.1 (x41.46)	686.5 (x40.57)	1,270.5 (x32.81)	542.5 (x34.56)	0.00
WAND	14.2 (x7.86)	23.1 (x3.96)	27.3 (x2.58)	37.3 (x2.20)	73.8 (x1.91)	37.2 (x2.37)	0.53 (4%)
MaxScore	12.7 (x7.04)	21.3 (x3.66)	27.1 (x2.56)	33.9 (x2.00)	55.0 (x1.42)	32.3 (x2.06)	0.53 (4%)
BMW$_{40}$	3.2 (x1.77)	10.0 (x1.72)	17.5 (x1.65)	28.1 (x1.66)	65.9 (x1.70)	26.3 (x1.68)	4.14 (28%)
VBMW$_{40}$	2.1 (x1.15)	6.0 (x1.02)	10.3 (x0.97)	16.2 (x0.96)	37.0 (x0.96)	15.2 (x0.96)	4.14 (28%)
C-VBMW$_{40}$	1.8	5.8	10.6	16.9	38.7	15.7	2.12 (14%)
BMW$_{128}$	3.6 (x1.99)	12.0 (x2.06)	20.9 (x1.97)	32.5 (x1.92)	71.0 (x1.83)	29.4 (x1.87)	2.24 (15%)
VBMW$_{128}$	2.7 (x1.49)	10.0 (x1.71)	16.8 (x1.58)	25.9 (x1.53)	56.6 (x1.46)	23.6 (x1.50)	2.24 (15%)
C-VBMW$_{128}$	2.9 (x1.59)	10.6 (x1.83)	18.0 (x1.69)	28.0 (x1.65)	61.0 (x1.58)	25.2 (x1.60)	1.48 (10%)

[24] Stephen E. Robertson and Karen S. Jones. 1976. Relevance weighting of search terms. *Journal of the Am. Soc. for Information science* 27, 3 (1976), 129–146.

[25] David Salomon. 2007. *Variable-length Codes for Data Compression.* Springer.

[26] Dongdong Shan, Shuai Ding, Jing He, Hongfei Yan, and Xiaoming Li. 2012. Optimized Top-k Processing with Global Page Scores on Block-max Indexes. In *WSDM*. 423–432.

[27] Fabrizio Silvestri. 2007. Sorting Out the Document Identifier Assignment Problem. In *ECIR*. 101–112.

[28] Fabrizio Silvestri and Rossano Venturini. 2010. VSEncoding: Efficient Coding and Fast Decoding of Integer Lists via Dynamic Programming. In *CIKM*. 35–42.

[29] Nicola Tonellotto, Craig Macdonald, and Iadh Ounis. 2013. Efficient and Effective Retrieval Using Selective Pruning. In *WSDM*. 63–72.

[30] Howard Turtle and James Flood. 1995. Query evaluation: Strategies and optimizations. *Information Processing & Management* 31, 6 (1995), 831–850.

[31] Sebastiano Vigna. 2013. Quasi-succinct indices. In *WSDM*. 83–92.

[32] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM Comput. Surv.* 38, 2 (2006).