Optimal Space-time Tradeoffs for Inverted Indexes

Giuseppe Ottaviano[1], Nicola Tonellotto[1], Rossano Venturini[1,2]
[1] National Research Council of Italy, Pisa, Italy
[2] Department of Computer Science, University of Pisa, Pisa, Italy
giuseppe.ottaviano@isti.cnr.it, nicola.tonellotto@isti.cnr.it, rossano.venturini@unipi.it

ABSTRACT

Inverted indexes are usually represented by dividing posting lists into constant-sized blocks and representing them with an encoder for sequences of integers. Different encoders yield a different point in the space-time trade-off curve, with the fastest being several times larger than the most space-efficient. An important design decision for an index is thus the choice of the fastest encoding method such that the index fits in the available memory.

However, a better usage of the space budget could be obtained by using faster encoders for frequently accessed blocks, and more space-efficient ones those that are rarely accessed. To perform this choice optimally, we introduce a linear time algorithm that, given a query distribution and a set of encoders, selects the best encoder for each index block to obtain the lowest expected query processing time respecting a given space constraint.

To demonstrate the effectiveness of this approach we perform an extensive experimental analysis, which shows that our algorithm produces indexes which are significantly faster than single-encoder indexes under several query processing strategies, while respecting the same space constraints.

Categories and Subject Descriptors

H.3.2 [**Information Storage and Retrieval**]: Information Storage; E.4 [**Coding and Information Theory**]: Data Compaction and Compression

Keywords

Compression; Knapsack Problems; Inverted Indexes

1. INTRODUCTION

Web search engines need to carefully organize billions of documents into efficient data structures to timely answer queries submitted by their users. The most central of such data structures is the inverted index, which, in its most basic and popular form, is a collection of sorted sequences of integers [5, 17, 29, 33]. An inverted index has to fulfill two goals: it must be able to efficiently process users' queries, and it must be able to compactly store large collections of documents, in order to fit in the available memory.

WSDM'15, February 2–6, 2015, Shanghai, China.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3317-7/15/02 ...$15.00.
http://dx.doi.org/10.1145/2684822.2685297.

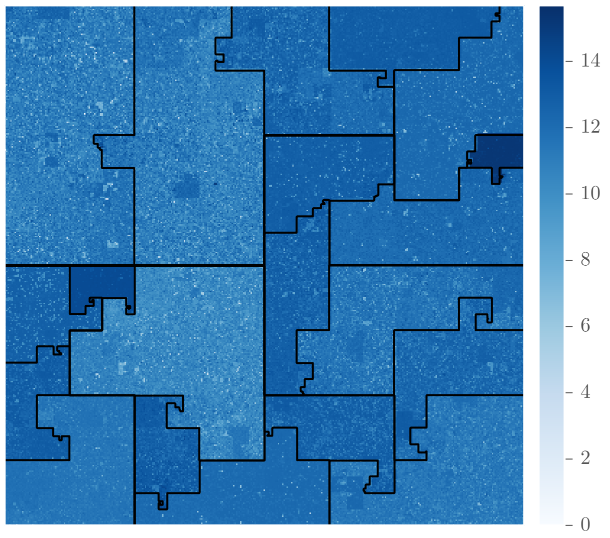

Figure 1: Access counts of 20 of the top 30 most accessed lists, plotted along a Hilbert curve (log scale).

The main ingredient to achieve the second goal is the compression of the sequences of integers that constitute the index. This is however at odds with the first goal, since more space-efficient sequence encoders have usually more complex, and hence slower, decoding. For this reason, a large amount of literature has focused on devising fast and space-efficient compression techniques, producing a multitude of encoders with different characteristics [15, 18, 20, 22, 26, 31].

While state-of-the-art encoders obtain very good space-time tradeoffs, we argue that using a single encoder for the entire index is wasteful: with typical query distributions some terms are more frequent than others, and even within the same posting list the skipping performed by most query processing algorithms produces regions of the list that are rarely accessed. We can visualize this effect in Figure 1, which shows the number of accesses for each posting in the 30 most accessed lists in an index, mapped to a square through a Hilbert curve so that list intervals are mapped to areas; the noisy appearance and the presence of darker spots indicate a large variability in access frequencies (details on how this statistics were computed can be found in Section 6).

In this scenario, it is clear that we can improve the efficiency of an index by compressing frequently accessed regions with faster but less space-efficient encoders, and rarely accessed regions with a compact but slow encoder. This can be done without incurring any overhead with the popular representation of posting lists which divides each list into constant-sized blocks (e.g., 128 postings) and encodes them independently, since we can just choose a different encoder for each block.

However, it is not obvious how to perform this choice, as it depends on the space-time characteristics of the encoders and on the distribution of the values in the block. This issue is even more severe for encoders that offer a space-time tradeoff parameter, such as PForDelta, because choosing the parameter independently for each block could be suboptimal in global sense. For these reasons, we introduce the *Space-Constrained Expected Time Minimization* problem: given a query distribution and a set of encoders, our goal is to select an encoder for each block of the index to minimize the expected query processing time, subject to a given space constraint. To perform this choice optimally we describe a linear time greedy algorithm that is guaranteed to find a solution which is at least as fast as the optimal one but may exceed the space budget by no more than few bytes, which is a negligible overhead for a typical index of several gigabytes.

An early version of this problem has been proposed in [31], where the authors describe an hybrid index organization applying different encoders to different posting lists. Our result has three fundamental differences. First, we select the best encoders for each block inside posting lists while they work at a posting list granularity, thus failing to exploit the skew in access frequencies within the same posting list. Second, we use a radically different technique to assess the decoding time for each decoder, which reduces the measurement time while improving accuracy. Finally, they describe a heuristic algorithm with no approximation guarantees; the algorithm is only sketched and lacks many important details that would allow to analyze its complexity, which is at least quadratic in the number of encoders. On the other hand, our optimization algorithm is provably optimal modulo a negligible space overhead, and its complexity is linear in both the number of blocks and encoders.

Our Contributions. We list here our main contributions.

1. We define the Space-Constrained Expected Time Minimization problem, a refined and formalized version of a problem introduced in [31]. We then propose an algorithm which produces an exact solution for this problem in $\Theta(m^2 \log m)$ time, where m is the total number of postings, thus disproving the NP-Hardness claim stated in [31].

2. Since the quadratic nature of the above algorithm makes it unfeasible in any practical scenario, we design a linear time solution derived from a greedy algorithm presented in [25] with additive approximation guarantees: it finds a solution with expected query time at least as low as the optimal one which may exceed the space budget by no more than the size of a single block. Thus, the introduced additive approximation is completely negligible in practice: a typical budget of several billions of bytes is exceeded only by a few hundreds of bytes.

3. We introduce a technique to obtain fast and accurate estimates of the expected decoding time of a block for each decoder, based on their features, circumventing the limitations in resolution and noise of CPU timers. We obtain that the overall predicted query processing times are within 4% of the actual ones.

4. We test the effectiveness of our approach on realistic datasets and queries with an extensive experimental analysis. We obtain indexes that follow a smooth space-time curve, dominating state-of-the-art methods such as OptPFD and Partitioned Elias-Fano on almost all query processing strategies. For instance, we are able to obtain indexes that are up to 2.16 times smaller than those obtained with the fastest encoder, while being no more than 14% slower.

2. BACKGROUND AND RELATED WORK

Index Organization. Given a collection $\mathcal{D}$ of documents, the *inverted index* of $\mathcal{D}$ is a collection of *posting list*, one for each term t appearing at least once in any document of the collection. Each document in $\mathcal{D}$ is represented by a natural number called document identifier, or *docid*. A posting list is the list of all the docids of documents that contain the term t. The set of such terms is usually called the *dictionary* or the *lexicon*. Posting lists often includes additional information about each document, so each docid appearing in a posting list may include, for example, the number of occurrences of the term in the document (the *term frequency* in the document), and/or the set of positions where the term occurs in the document (the *term positions* in the document) [5, 17, 33].

In this work, we consider document-sorted inverted indices, as used by at least one major search engine [8]; this enables fast query processing and efficient compression. Other efficient retrieval techniques such as frequency-sorted [27] or impact-sorted indices [1] are possible. However, there is no evidence of such index layouts in common use within commercial search engines [19].

In our experiments we focus our attention on posting lists storing docids and frequencies; we do not store positions or other additional data, since they have different nature and often require specialized compression techniques [30], thus they are outside of the scope of this paper. We also ignore additional per-document or per-term information, such as the mappings between docids and URLs, or between term identifiers and actual terms, as their space is negligible compared to the index size.

Index Compression. Inverted indexes resort to compression to reduce their space occupancy. Index compression ultimately reduces to the problem of representing sequences of integers for both docids and frequencies, specifically strictly monotone sequences for docids, and positive sequences for the frequencies. The two are equivalent: a strictly monotone sequence can be turned into a positive sequence by subtracting from each element the one that precedes it (also known as *delta encoding*), the other direction can be achieved by computing prefix sums. For this reason most of the work assumes that the posting lists are delta-encoded and focuses on the representation of sequences of positive integers.

Representing such sequences of integers in compressed space is a crucial problem, studied since the 1950s with applications going beyond inverted indexes. A classical solution is to assign to each integer an uniquely-decodable variable length binary code code; examples are *unary codes*, *Elias Gamma/Delta codes*, and *Golomb/Rice codes* [22].

Bit-aligned codes can be inefficient to decode as they require several bitwise operations, so byte-aligned or word-aligned codes are usually preferred if speed is a main concern. *Variable byte* [22] or *VByte* is the most popular byte-aligned code. In VByte the binary representation of a non-negative integer x is split into groups of bytes, whose length depends on the value of the integer to be encoded. Stepanov *et al.* [26] present a variant of variable byte (called *Varint-G8IU*) which exploits SIMD operations of modern CPUs to further speed up decoding.

A different approach is to 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. The integers in each block are all encoded with the same number of bits h, which must be large enough to accommodate all the values in the block. This technique, also called binary packing or packed binary [3, 15], is usually applied with blocks of constant size (e.g., $b = 128$ integers). There are several variants of this approach which differentiate themselves for their encoding or partitioning strategies [9, 15, 24]. For example, *Simple-9* and *Simple-16* [2, 3, 31] are two popular variants of this approach.

A major space inefficiency of binary packing is the fact that the presence of few large values in the block forces the algorithm to encode all its integers with a large h, thus affecting the overall compression performance. To address this issue, *PForDelta* (PFD) [34] introduces the concept of *patching*. In PFD, h is an arbitrary parameter; all the values that fit in h bits are binary packed, while those that do not fit are called *exceptions*, and encoded separately with a different encoder; we refer to PFD as the family of encoders $\mathsf{PFD}(h)$ depending on the value width h. Variants of PFD use different strategies to encode the exceptions and their positions; we use the variant described in Yan *et al.* [31], using Simple-16 for both. They also describe two strategies, namely NewPFD and OptPFD, to select the value of h for a given block. In particular, OptPFD chooses the one that minimizes the space occupancy.

A completely different approach is taken by *Binary Interpolative Coding* [18], which skips the delta-encoding step and directly encodes monotone sequences. This method recursively splits the sequence of integers in two halves, encoding at each split the middle element and recursively the two halves. At each recursive step the range that encloses the middle element is reduced, and so is the number of bits needed to encode it. Experiments [20, 24, 29, 31] have shown that Binary Interpolative Coding is the best encoding method for highly clustered sequence of integers. However, this space efficiency is paid at the cost of a very slow decoding algorithm.

The *Elias-Fano* representation of monotone sequences [11, 12] has been recently applied to the compression of inverted indexes [28], showing excellent query performance thanks to its efficient random access and search operations. Its space occupancy is competitive with some state-of-the-art methods such as γ-δ-Golomb codes and PForDelta. 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 [20] describe 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 show 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 $\epsilon \in (0, 1)$.

Query Processing. Given a term query as a (multi-)set of terms, the basic query operations are the boolean conjunctive (AND) and disjunctive (OR) queries, retrieving the documents that contain respectively all the terms or at least one of them. In many scenarios the query-document pairs can be associated with a *relevance score* which is usually a function of the term frequencies in the query and in the document, and other global statistics. Instead of the full set of matches, for scored queries it is often sufficient to retrieve the k highest scored documents for a given k. A widely used relevance score is BM25 [21], which we will use in our experiments.

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 query terms are processed and scored sequentially to build the result set, while, in a document-at-a-time (DAAT) strategy, the query term postings lists are processed in parallel, keeping them aligned them by docid. We will focus on the DAAT strategy as it is the most natural for docid-sorted indexes.

The alignment of the posting lists during DAAT processing can be achieved by means of the $\mathsf{NextGEQ}(d)$ operator, which returns the smallest docid in the posting list that is greater than or equal to d. A efficient implementation of the function $\mathsf{NextGEQ}(d)$ is crucial to obtain the typical subsecond response times of Web search engines. The trivial implementation that scans and checks sequentially the whole posting lists is usually too slow; typically, *skipping* strategies are employed. The basic idea is to divide the lists in small blocks that are compressed independently, and to store additional information about each block, in particular the maximum docid present in the block. This allows to find and decode only the block that may contain the sought docid by scanning the list of maximum docids, thus skipping a potentially large number of useless blocks. This basic approach can be improved also by exploiting query log information [7].

Solving scored disjunctive queries 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 $\mathsf{MaxScore}$ [27] and WAND [4]. Both strategies augments 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 impacts is smaller than the scores of the top-k documents found up to that point.

3. PROBLEM DEFINITION

In this section we introduce the Space-Constrained Expected Time Minimization problem as follows: given a query distribution $\mathcal{Q}$ and an arbitrary set $\mathcal{E}$ of k encoders, select an encoder for each index block to minimize the expected time to process queries drawn according to the distribution, subject to a given space constraint B. The problem is more formally defined in the following.

We assume that each posting list is split into blocks of size b, except the last block of each list which may contain fewer than b postings. We use n to denote the total number of blocks in the inverted index. Since each block can be encoded with any encoder in $\mathcal{E}$, an assignment of encoders to blocks can be modeled as a assignment φ from the n blocks to encoders in $\mathcal{E}$, hence $\varphi \in \mathcal{E}^n$. Given an assignment φ, φ_i is thus the encoder assigned to the ith block. We denote with $s_{i,j}$ the size of the encoding of the ith block with the jth encoder, and with $t_{i,j}$ its decoding time.

We aim at finding an assignment φ that minimizes the expected processing time of queries drawn from a probability distribution $\mathcal{Q}$, subject to a total space budget B, as follows:

$$\begin{aligned} \underset{\varphi \in \mathcal{E}^n}{\text{minimize}} \quad & \mathop{\mathbb{E}}_{q \sim \mathcal{Q}} \big[T(q, \varphi)\big] \\ \text{subject to} \quad & \sum_{i=1}^{n} s_{i,\varphi_i} \leq B, \end{aligned} \tag{1}$$

where we denote with $T(q, \varphi)$ the time to process the query q with a predetermined query processing strategy, such as Ranked AND or WAND. We assume that the processing time can be decomposed into a component T_p independent from the assignment φ, plus a component T_d which only accounts for the decoding time of the blocks, and thus depends only on the set of the blocks that the processing algorithm needs to decode, which we denote with $A(q)$. The processing time can thus be written as $T(q, \varphi) = T_p(q) + T_d(A(q), \varphi)$. Since $T_p(q)$ does not depend on the encoders choice, we can safely ignore it in the optimization.

We assume that we do not have access to the query distribution, but we can obtain a sample from it, namely a query log Q. The expected decompression time can thus be estimated with the empirical mean over Q; since $T_d(A(q), \varphi)$ is just the sum of the decoding times for all the blocks decoded during the query q, by linearity we can reformulate the problem as follows:

$$\begin{aligned} \underset{\varphi \in \mathcal{E}^n}{\text{minimize}} \quad & \sum_{i=1}^{n} f_i t_{i,\varphi_i} = \sum_{i=1}^{n} \hat{t}_{i,\varphi_i} \\ \text{subject to} \quad & \sum_{i=1}^{n} s_{i,\varphi_i} \leq B, \end{aligned} \tag{2}$$

where f_i is the number of times the block the ith block was decoded while processing the query log Q, and $\hat{t}_{i,j} = f_i t_{i,j}$ represents the overall decoding time of the ith block when encoded with the jth encoder. The counts f_i can be computed exactly just by processing all the queries in the query log and counting the number of decodings of each block. We defer to Section 5 the description of how to obtain the numbers $s_{i,j}$ and $t_{i,j}$.

4. LINEAR TIME ALGORITHM

In the following we describe a linear-time algorithm that finds an additive approximation to the solution of the Space-Constrained Expected Time Minimization problem defined in the previous section. The solution adopts a greedy strategy, which is guaranteed to find an assignment which is at least as good as the optimal one with respect to its expected processing time but exceeds the space budget constraint by no more than $S_{\max}$ bits, where $S_{\max}$ denotes the size in bits of the largest encoding of a block. Since a block can be always encoded plain within $b \log d$ bits, we have $S_{\max} \leq b \log d$ bits, where d denotes the number of documents in $\mathcal{D}$. In a typical instance, the block size b is 128 postings and $\log d \leq 32$ bits, so that $S_{\max}$ is at most 4,096 bits. Thus, the introduced additive approximation is completely negligible in practice: the budget B of several billions of bits is exceeded only by a few thousands of bits. This approximated solution is computed in linear time, namely, $\Theta(nk)$, which is linear in the number of blocks in the index.

We start by reducing our problem to the so-called *Multiple-Choice Knapsack Problem* (MCKP) [25]. In this problem, we are given a set of n classes of items. Each class contains at most k different items. Each item has its own weight and penalty. The goal is that of choosing *exactly* one item for each class so that the sum of the penalties is minimized, subject to a given knapsack capacity B.

In our setting, the classes correspond to the blocks to be encoded: each item in class i corresponds to a possible encoder j for the block i, so that the item's weight is the space occupancy $s_{i,j}$ and item's penalty is the expected decoding time $\hat{t}_{i,j}$. The goal is to minimize the expected query processing time subject to the space budget B. The integer linear programming formulation of the problem is as follows.

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} \sum_{j=1}^{k} x_{i,j} \hat{t}_{i,j} \\ \text{subject to} \quad & \sum_{i=1}^{n} \sum_{j=1}^{k} x_{i,j} s_{i,j} \leq B, \\ & \sum_{j=1}^{k} x_{i,j} = 1, \quad i \in \{1, \ldots, n\} \\ & x_{i,j} \in \{0, 1\}, \quad i \in \{1, \ldots, n\},\ j \in \{1, \ldots, k\} \end{aligned} \tag{3}$$

The optimization variable $x_{i,j}$ represents the choice of the encoder for block i, and is 1 if the chosen encoder is j and 0 otherwise (in our notation, $x_{i,j} = 1$ if and only if $\varphi_i = j$).

In its general formulation the Multiple-Choice Knapsack Problem is NP-Hard [25]. There exists, however, a pseudo-polynomial time algorithm based on a standard Dynamic Programming approach, which requires $\Theta(nB)$ time and space. Thus, this immediately implies the existence of a polynomial time algorithm that finds an optimal assignment for any instance in our index compression scenario. Indeed, the complexity of this algorithm becomes $O(m^2 \log m)$ time and space, where m is the total number of postings, since $n \leq m$ and B is at most $m \log m + O(m)$ bits, since at worst each posting can be represented with $\log m + O(1)$ bits. Due to its quadratic nature, this solution is clearly unfeasible in any practical scenario, and only serves to disprove the NP-Hardness claim stated in Yan *et al.* [30].

Our linear-time approximated algorithm proceeds by finding the exact solution of the *Continuous Multiple-Choice Knapsack Problem* (CMCKP). This is the relaxation of the original problem in which any variable $x_{i,j}$ can take any real value in $[0, 1]$ instead of being forced to be either 0 or 1. Obviously, since CMCKP is less restrictive than MCKP, an optimal solution for the former is at least as good as an optimal solution for the latter, whenever both are subject to the same space budget. It is known that an optimal solution for CMCKP has the following important property [25]. There exists an optimal solution for CMCKP in which the assignment of the variables $x_{i,j}$ is such that there are either n or $n+1$ non-zero variables. If there are n non-zero variables, then there is a non-zero variable for each of the n possible classes and these are all set to 1. Thus, this is also a feasible and optimal solution for MCKP. If there are $n+1$ non-zero variables, then $n-1$ classes have their non-zero variables set to 1. The remaining class, say i, has exactly two fractional non-zero variables, say $x_{i,p}$ and $x_{i,q}$. Thus, this solution is *almost* a feasible solution for MCKP. This can be turned into an approximated solution for MCKP by setting one of these variables to 1 and the other to 0. If we set to 1 the variable of the item with the smallest decode time, say p, the obtained solution is such that (1) it is at least as good as the original one; (2) its space budget B may be exceeded, but, in any case, by no more than $s_{i,p} \leq S_{\max}$ bits.

Interestingly, an optimal solution for CMCKP can be found with a greedy strategy which works in two phases. In the first phase, the algorithm applies two fundamental *dominance criteria* in order to discard items that are not necessary to find an optimal solution. In the second phase, the algorithm starts choosing items according to a criterion that favors items introducing a larger marginal improvement of the penalty/weight tradeoff.

The first phase discards all the items dominated by other items according to the following two dominance criteria.

Dominance criterion 1 Let p and q be two items in same class i. Item q is dominated by item p if and only if $s_{i,p} \leq s_{i,q}$ and $\hat{t}_{i,p} < \hat{t}_{i,q}$.

Dominance criterion 2 Let p, q and r be three items in the same class i such that $s_{i,p} < s_{i,q} < s_{i,r}$ and $\hat{t}_{i,p} \geq \hat{t}_{i,q} \geq \hat{t}_{i,r}$. Item q is dominated by items p and q if and only if the following condition holds $\frac{\hat{t}_{i,p} - \hat{t}_{i,q}}{s_{i,p} - s_{i,q}} \geq \frac{\hat{t}_{i,q} - \hat{t}_{i,r}}{s_{i,q} - s_{i,r}}$.

If item q in class i is dominated according to at least one of the above criteria, then there exists an optimal solution for CMCKP which does not select item q (i.e., $x_{i,q} = 0$) [25]. This implies that the algorithm can safely discard all these dominated items without affecting the optimality result. This phase is essential for the correctness of the next phase.

Once the dominated items have been discarded, the algorithm sorts items within each class increasingly by their weights. With a little abuse of notation, from now on we refer to item j in class i as the jth smallest item within this ordered sequence. By the absence of dominated items, the penalties give a non increasing sequence. Thus, if there are $\ell \leq k$ items in class i, we have $s_{i,1} < s_{i,2} < \ldots < s_{i,\ell}$ and $\hat{t}_{i,1} \geq \hat{t}_{i,2} \geq \ldots \geq \hat{t}_{i,\ell}$.

The algorithm starts the next phase by constructing a base solution which contains the first item of each class. Observe that this solution is the one with the smallest possible space occupancy. This base solution is subsequently refined in order to improve the processing time by exploiting the available space budget. This is done by employing a greedy strategy that selects items in non increasing order of the ratio $\lambda_{i,j} = \frac{\hat{t}_{i,j} - \hat{t}_{i,j-1}}{s_{i,j-1} - s_{i,j}}$. This ratio measures the marginal penalty decrease per unit weight that we can obtain by replacing item $j-1$ with item j in class i. This is done by sorting all the items according to their ratio and by scanning this sorted sequence of items. When the algorithm processes the item j in class i, it replaces item $j-1$ with j in class i within the current solution (i.e., it sets $x_{i,j-1}$ to 0 and $x_{i,j}$ to 1). The algorithm stops as soon as the space budget is exceeded. The last two modified variables $x_{i,j-1}$ and $x_{i,j}$ can be adjusted and made fractional to match the budget. This is guaranteed to be an optimal solution for CMCKP [25].

Note that a single run of this algorithm is able to find all the optimal solutions for any possible budget constraint. Indeed, any step of the algorithm finds a solution which is optimal for its space occupancy. Moreover, all the possible space budgets are found in this way.

The time complexity of the algorithm is dominated by the time required to sort the items (i.e., $\Theta(nk \log(nk))$ time). This time complexity can be improved to $\Theta(nk)$ by replacing the sorting step with a (modified) linear time median selection algorithm [32]. Due to space limitations, we omit the details of this faster solution; furthermore, in practice we can use off-the-shelf optimized sorting implementations, so the sorting step does not dominate the running time. Hence, the last improvement is only of theoretical interest.

5. DECODING TIME PREDICTION

The optimization algorithm requires the knowledge of the space occupancy $s_{i,j}$ and of the decoding time $t_{i,j}$, for each block i and each encoder j. While the space occupancy can be computed deterministically and exactly just by encoding the block, the decoding times span from tens to hundreds of nanoseconds, hence it is hard to assess them reliably, due to the low resolution and high noise of CPU timers. In order to quantify these times with sufficient accuracy, a direct solution could be to measure the time to decode the block several times and divide by the number of runs; however this would make the measuring algorithm prohibitively slow. Moreover, any attempt to make the measurements faster by running several threads in parallel would further increase the noise.

We propose to quantify the decoding times by splitting the measurement in two steps. First, a *profiling step* running in a controlled single-threaded environment is carried out, by averaging several runs of decoding on a large random sample of the index blocks, in order to cover accurately the distribution of different blocks. Second, from these measurements we learn predictors that use features extracted from the block values to predict the decoding time for each decoder. Then, the each predictor is used in the optimization algorithm to estimate $t_{i,j}$ for each block and decoder.

We model the predictor as a linear combination of features of the values in the block with non-negative weights. Being non-negative, the weight of a feature can be interpreted as the contribution of that feature to the decoding time. Given a block i we call $\mathbf{x_i}$ its feature vector; the predictor is thus $t_{i,j} = \mathbf{x_i}^T \mathbf{w_j} + b_j$ where $\mathbf{w_j}$ is the weights vector and b_j the bias learned for the decoder j.

To obtain a robust learned model we use an L_1 loss function rather than the more popular L_2 loss, and we add an L_1 regularization component which is known to make the solutions sparse [13]. Given a multiset set $\mathcal{T}_j = \{(\mathbf{x}, t)\}$ of block features and their respective measured decoding times with decoder j we thus compute $\mathbf{w_j}, b_j$ as the solution of the minimization problem:

$$\min_{\mathbf{w_j}, b_j \geq 0} \sum_{(\mathbf{x},t) \in \mathcal{T}_j} |\mathbf{x}^T \mathbf{w_j} + b_j - t| + \mu |\mathcal{T}_j| \|\mathbf{w}\|_1$$

where μ is the regularization parameter. The problem can be written as a linear program and solved with standard convex optimization algorithms; however, we found that it is faster to use a smooth approximation of the absolute value, namely the Huber loss function [14], and use a quasi-Newton solver, specifically L-BFGS [6], without compromising the accuracy of the solution.

6. EXPERIMENTAL ANALYSIS

The effectiveness of our approach in a practical setting relies on several hypotheses. First, that the different space-time characteristics of different encoders can be exploited successfully, and that their decoding times can be accurately estimated. Second, that our query processing time model fits the measured data. Finally, that block access statistics induced by a typical query log exhibit a skew that makes it convenient to use different encoders for different blocks.

In this section we validate our hypotheses with an extensive experimental evaluation in a realistic and reproducible setting, using state-of-the-art encoders, standard benchmark text collections and a large query log.

6.1 Experimental setup

Encoders. In our experiments the set $\mathcal{E}$ of encoders is composed as follows. First, we use PFD(h) for all the values of h as different encoders; this way the assignment of h to each block is optimized globally rather than on a per-block basis as in strategies like OptPFD. We then add two encoders which represent the best space occupancy and the best decoding time. For the former we choose Binary Interpolative Coding, which consistently outperforms every other encoder both in the literature and in our own experiments. For the latter we adopt Varint-G8IU, which according to the experiments in [15] is the fastest among those that support small block sizes and unaligned reads, which are needed because blocks with different alignments are concatenated together.

The encoder type chosen for each block is written by prepending one byte to the encoding of the block, which adds a negligible overhead to the overall space. The last block of each list may have fewer than b postings, in which case we call it a *partial* block and always compress it with Interpolative. Since it is accessed at most once per query, its decoding time is negligible, and the vast majority of partial blocks belongs to the single-block lists of rare terms. For these reasons we also focus our analysis only on blocks with exactly b postings, which we call *full*.

We compare the indexes obtained with our algorithm, which we call *hybrid*, against single-encoder indexes using Interpolative, OptPFD, and Varint-G8IU; again, we always compress partial blocks with Interpolative. We also compare against the recently introduced Partitioned Elias-Fano indexes [20]. We remark that Elias-Fano is not included in the set of encoders $\mathcal{E}$ because its characteristics are not suitable for our model. Indeed, Elias-Fano is faster than other encoders to decode single postings but slower to decode an entire block. However, the latter is the key operation at the basis of our model.

Datasets. We performed our experiments on the following 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.

	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

Table 1: Basic statistics for the test collections

For each document in the collection the body text was extracted using Apache Tika.[1], and the words lowercased and stemmed using the Porter2 stemmer; no stopwords were removed. The docids were assigned according to the lexicographic order of their URLs [23] Table 1 reports the basic statistics for the two collections.

The results on Gov2 and ClueWeb09 are very similar, hence in the following we focus most of our analysis on the larger dataset ClueWeb09.

Queries. The queries used in our experiments are taken from the MSN Search query log from 2006,[2] comprising 15 million queries sampled over the month of May and submitted from the US. We filter out all the queries which contain terms not present in the index, and all single-term queries, as their results can be precomputed separately.

We select the first million queries in chronological order as a training set to compute the counts f_i, and measure the query times on a sample of 5000 queries taken from the remaining queries, to simulate a realistic scenario where the index is optimized based on past queries.

Processing strategies. To test the performance on query strategies that make use of both the docids and the occurrence frequencies we perform BM25 top-10 queries using 3 different algorithms: Ranked AND, which scores the results of a conjunctive query, WAND [4], and MaxScore [27].

Estimation of the access counts. The block access counts f_i must be computed using the query processing strategies that will be used at query time; rather than building different index optimized for each query processing strategy we test, we assume a realistic scenario in which the index is optimized once and queried with different query strategies. For this reason we process each query in the training set with all the three processing strategies, and add up the counts. Figure 1 in the introduction shows the result of this counting on the

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

[2]`http://research.microsoft.com/en-us/um/people/nickcr/wscd09/`

ClueWeb09 dataset using the query training set for 20 of the top 30 most accessed posting lists.

Blocks never accessed with the training set would have zero overall time $\hat{t}_{i,j} = f_i t_{i,j}$, thus the algorithm would force them to be compressed with Interpolative regardless of the budget; to avoid this we use Laplace smoothing when estimating the counts f_i, i.e., we add 1 to each access count. In our model this is equivalent to estimating the block access probability with maximum a posteriori with a Dirichlet prior, rather than with maximum likelihood. This does not make a measurable difference in practice, since our training set is large enough to access the vast majority of the index: the fraction of unseen full blocks is less than 9%.

Testing details. All the algorithms are implemented in C++11 and compiled with GCC 4.9 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 3.13.0. 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.

The source code is available at `https://github.com/ot/ds2i/tree/WSDM15` for the reader interested in further implementation details or in replicating the experiments.

6.2 Results

Decoding time prediction. To predict the decoding time we use the following features: size of the compressed block (`size`), bit length of the largest integer (`max_h`), number of non-zero values (`nonzeros`), and sum of the logarithms of the values (`sum_of_logs`). For PFD(h) we also add the number of exceptions (`pfor_exceptions`). We found that the decoding time of PFD is fairly insensitive on the value width h, so we use the same predictor for all the PFD decoders.

To evaluate the performance of the predictors we perform an usual 80-20 split of the data into a training set and a test set, and measure the average absolute error. As a baseline, we compare against a constant predictor which returns the median of the decoding times regardless of the features.

Decoder	Time	C. Err	L. Err	HWF
Interpolative	640.8	73.3	20.0	`nonzeros`
PFD	97.8	66.6	7.7	`pfor_exceptions`
Varint-G8IU	42.4	3.5	1.8	`size`

Table 2: Median time (Time), average absolute error for the constant (C. Err) and linear (L. Err) predictors, and highest weighted feature in the linear model (HWF). All the numbers are in nanoseconds.

As shown in Table 2, we find that the linear predictor significantly reduces the error with respect to the median predictor, bringing it down to less than 8% of the median time in all cases. The highest improvement occurs with PFD, which exhibits a large variability in decoding time, as it is highly affected by the number of exceptions.

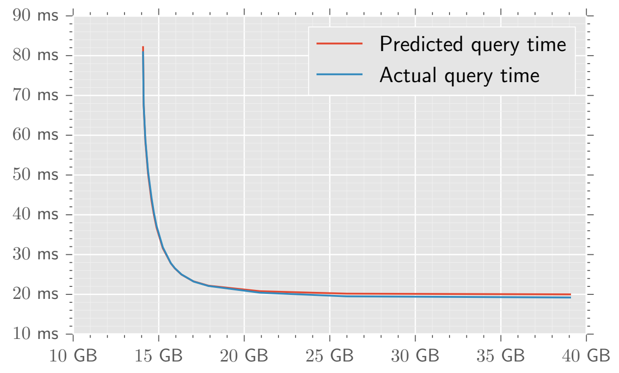

Figure 2: Predicted and actual average query times for different index spaces on the ClueWeb09 dataset.

Model validation. To test the validity of our query processing time model we compare the predicted average query time, that is $\tilde{T} = \sum_{i=1}^{n} f_i t_{i,\varphi_i}$, where the f_is are computed on the test set, with the actual average query time T using all the query processing strategies. Recall that $T = \tilde{T} + T_p(q)$, where $T_p(q)$ is assumed independent from the choice of the encoders and thus constant along the space-time curve.

We estimate $T_p(q)$ by taking the average of the differences between T and $\tilde{T}$ with different index spaces, and plot the predicted and actual query times in Figure 2. Note that the two curves are very close; in fact, the predicted times are within 4% of the actual times. Taken into account the additive factor $T_p(q)$, this error is very close to the prediction error on the single blocks, thus suggesting that an improvement on the decoding time prediction would further reduce this already small gap.

Construction time. Almost all the phases of hybrid index construction can be trivially parallelized, since each list can be handled independently. The only phase that is inherently sequential is the greedy one, which is just a scan of the sorted $\lambda_{i,j}$s and it is extremely fast. We report here the times for the ClueWeb09 dataset, using 8 threads.

The most time-consuming task is the computation of the ratios $\lambda_{i,j}$, since it involves compressing each block with each encoder to measure the space and estimate the decoding time, taking about 14 minutes. We then store the $(i, j, \lambda_{i,j}, s_{i,j}, t_{i,j})$ tuples in a file and use an external algorithm to sort them by $\lambda_{i,j}$, specifically the implementation in STXXL [10]. The reason is twofold: first, the tuples may be so many that they do not fit in memory; second, this way we can reuse the vector of sorted tuples to build indexes on several points of the space-time curve. Besides, going through disk does not affect seriously this step, taking about 6 minutes.

The next phase is the greedy algorithm that finds the space-time tradeoff by scanning the sorted vector of tuples. While this phase must be run on a single thread, it takes just a little more than a minute. The last step is using the assignments found by the greedy algorithm to construct the actual index, which takes from 5 to 8 minutes, depending on the percentages of the various chosen encoders. Note that this is roughly the same time needed to construct a single-encoder index; the overall algorithm is thus about 4 times slower than the construction of a single-encoder index. However, it can be

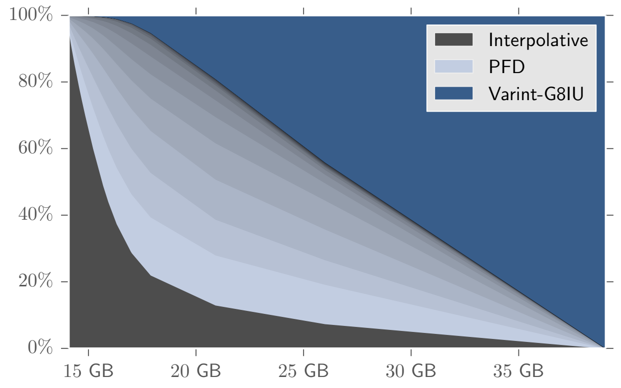

Figure 3: Percentage of full blocks compressed with each encoder for different index spaces. For PFD different shades denote different values of h.

made faster by increasing the parallelism in the first phases, while the final index construction is dominated by I/O in all cases.

Index spaces and query times. We report in Figure 4 the space-time tradeoff curves of our Hybrid and the single-encoder indexes on ClueWeb09. The different plots show the overall index size in gigabytes and the average query time, in milliseconds per query, for the three different processing strategies, namely Ranked AND, WAND and MaxScore. Interpolative and Varint-G8IU are on the extremes of the curve, being respectively the smallest and the fastest. Note that all Hybrid tradeoff curves are smooth, as they can achieve any point within their space-time tradeoff curve; any of these points is an optimal solution within a space margin of at most 512 bytes. The convexity of the curves shows no dominated points, according to both dominance criteria 1 and 2.

Partitioned EF index confirms its excellent performance with strategies that perform a large number of skips, namely Ranked AND and WAND. Conversely, its performance worsens on MaxScore, which depends on enumerating the *union* of the so-called *essential* lists, thus the decoding speed for such lists plays an important role in the overall processing time.

To get some insight on the choices made by the optimization algorithm, we show in Figure 3 the percentages of blocks that are compressed with each encoder by varying the available space budget. Not surprisingly, Hybrid chooses Interpolative for a vast majority of the blocks with a very small budget, and Varint-G8IU with a very large budget. Perhaps surprisingly, instead, the algorithm assigns PFD to about 10% of the blocks even at the smallest space budget, showing that Interpolative is dominated by PFD on those blocks. However, it is interesting to observe how the choices change in the regime where PFD becomes more prevalent. As we mention above, the value of h in PFD(h) influences both the encoding size and the decoding time. In general, a small value of h implies a small encoding size but the presence of several exceptions slows down the decoding speed. This is apparent in Figure 3: initially Hybrid prefers small values of h to favor space efficiency, and then it starts to increase h as the available budget increases to favor time efficiency.

Table 3 reports the index space, both as overall size in gigabytes, and broken down in bits per integers for docids

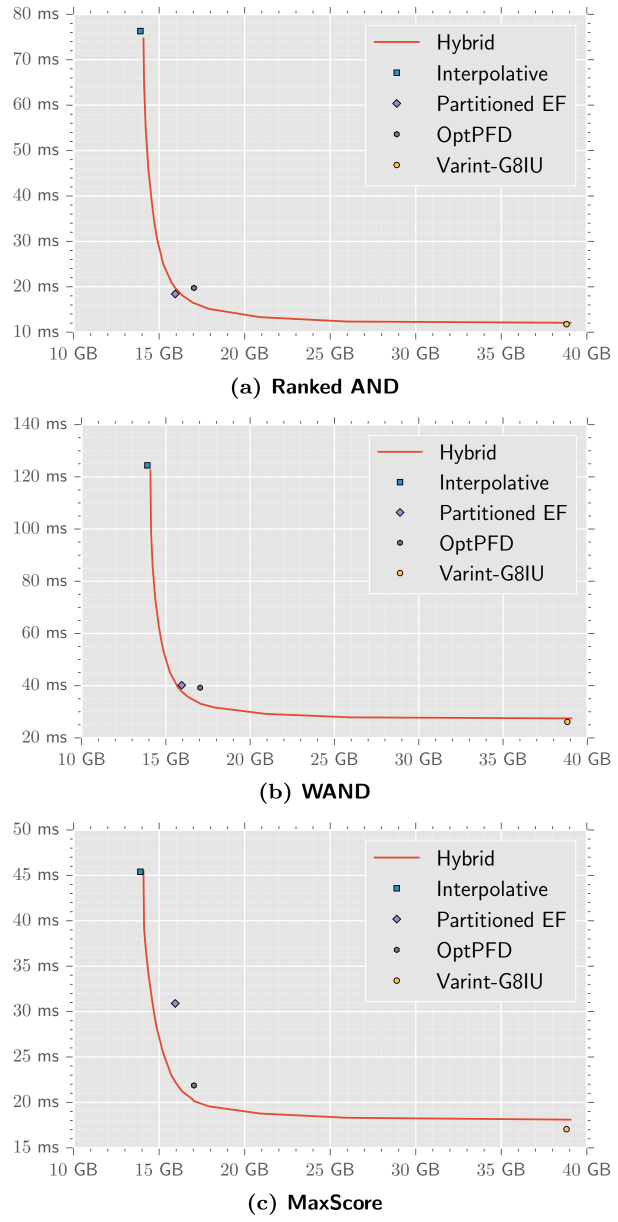

Figure 4: Average query times for different index spaces and strategies on the ClueWeb09 dataset.

and frequencies, as well as the average query time, in milliseconds per query, for the three processing strategies, for both datasets. For our Hybrid index, we select a few points from its space-time tradeoff curve, which help to more accurately compare its results to the closer single-encoder index.

On the extreme of the space-time tradeoff curve, which favors space occupancy disregarding query processing time, Hybrid@1 achieves almost the same space of Interpolative. On ClueWeb09, Hybrid@1 produces a slightly larger index (about 1%) than Interpolative. Even if in this case Hybrid@1 minimizes its space occupancy by compressing almost all the blocks with Interpolative code, it has a space overhead of 0.0625 bits per posting due to extra byte per block to write the encoder type. Interestingly, on Gov2 a large number of

	Space GB	Docids bpi	Freqs bpi	Ranked AND ms	WAND ms	MaxScore ms
ClueWeb09						
Hybrid@1	14.08	5.21	1.90	74.7	122.4	45.5
Hybrid@2	14.58	5.35	2.01	38.4	62.9	31.4
Hybrid@Partitioned EF	15.94	5.77	2.27	19.8	37.7	22.3
Hybrid@OptPFD	17.04	6.13	2.47	16.4	33.1	20.1
Hybrid@3	20.92	7.33	3.22	13.3	29.2	18.8
Hybrid@4	26.00	8.56	4.55	12.3	27.9	18.3
Hybrid@5	39.05	10.81	8.89	12.1	27.5	18.1
Interpolative	13.90	5.15	1.87	76.3	124.4	45.4
Partitioned EF	15.94	5.85	2.20	18.4	40.2	30.9
OptPFD	17.04	6.18	2.41	19.7	39.2	21.9
Varint-G8IU	38.82	10.75	8.83	11.8	26.1	17.0
Gov2						
Hybrid@1	4.21	3.85	2.01	14.1	26.2	11.4
Hybrid@2	4.54	4.12	2.21	4.2	8.7	5.9
Hybrid@Partitioned EF	4.65	4.21	2.27	3.7	7.8	5.5
Hybrid@OptPFD	4.92	4.44	2.42	3.1	6.9	5.1
Hybrid@3	6.36	5.52	3.34	2.5	5.9	4.8
Hybrid@4	8.07	6.62	4.63	2.5	5.8	4.7
Hybrid@5	13.83	10.40	8.87	2.4	5.8	4.7
Interpolative	4.26	3.80	2.14	14.5	27.0	11.5
Partitioned EF	4.65	4.10	2.38	4.0	8.6	8.0
OptPFD	4.92	4.48	2.38	3.8	8.2	5.5
Varint-G8IU	13.75	10.35	8.81	2.3	5.5	4.4

Table 3: Index spaces, both overall space in gigabytes and broken down into docids and frequencies in bits per integer, along with average query times in milliseconds for the query strategies considered.

blocks is better compressed with PFD than with Interpolative; this is exploited by Hybrid@1 which is able to improve the space of Interpolative by roughly 1%.

The ability of Hybrid to achieve any point in the tradeoff curve can be exploited to remain close to the best possible space occupancy but significantly improve the query time. For example, Hybrid@2 is obtained by imposing a space budget which is within $5 - 7\%$ the space occupancy of Interpolative. This gives an index faster than Interpolative by factors that range, depending on the query processing strategy, between 1.45 and 1.99 on ClueWeb09, and 1.95 and 3.45 on Gov2.

With Hybrid@Partitioned EF we set the budget to the space occupancy of Partitioned EF. Hybrid@Partitioned EF is faster than Partitioned EF on all the strategies but Ranked AND on ClueWeb09, where Partitioned EF is 7% faster. With WAND, Hybrid@Partitioned EFis faster by 6% on ClueWeb09 and 9% on Gov2. As expected, these percentages increase with MaxScore where they become 28% on ClueWeb09 and 31% on Gov2.

Similarly, Hybrid@OptPFD is obtained by setting the budget to the space of OptPFD. On the same conditions, Hybrid@OptPFD is always faster than OptPFD. The largest improvement is 18% on Gov2 and 17% on ClueWeb09, both with the Ranked AND strategy.

On the other extreme of the space-time tradeoff curve, which favors query processing time, Varint-G8IU index is slightly faster than Hybrid@5. Even if Hybrid@5 encodes all the blocks with the Varint-G8IU encoder, there is a small time overhead due to the dispatching of the block decoding code, which is most noticeable on MaxScore. However, if we renounce to at most 8% of Varint-G8IU overall time efficiency, we obtain an index, see Hybrid@4, which significantly reduces the space occupancy by a factor 1.49 on ClueWeb09 and 1.7 on Gov2. Finally, Hybrid@3 is slower than Varint-G8IU by no more than 14% and reduces the space occupancy by a factor 1.86 on ClueWeb09 and 2.16 on Gov2.

7. CONCLUSIONS & FUTURE WORK

In this paper we introduced and motivated the study of the Space-Constrained Expected Time Minimization problem. We presented a linear-time algorithm that computes optimal solutions with a negligible additive approximation. An extensive experimental analysis in a realistic and reproducible setting validates all the hypotheses at the basis of our model and shows that the obtained indexes dominate state-of-the-art single-encoder methods such as OptPFD and Partitioned Elias-Fano with most query processing strategies.

Since our solution takes the set of encoders to be optimized as a parameter, future experiments will investigate the consequences of optimizing with other encoders. Elias-Fano representation is one of the few encoders which cannot benefit from this approach. Indeed, compared to the other encoders, Elias-Fano is faster in providing random access to single postings but it is slower in decoding large chunks of consecutive postings (see e.g., [20] and references therein). An interesting open problem would be thus to design a solution which is able to model and to optimize the index using encoders that have these characteristics.

Finally, our solution optimizes by assuming that the posting lists are split into constant-size blocks. Allowing blocks of variable sizes relaxes the range of the achievable space-time tradeoffs, and, thus, may further improve the overall results. However, this increases the difficulty of the optimization problem, since the access statistics must be taken at the posting level. The design of an efficient algorithm in this setting is left as an interesting open problem.

Acknowledgements

This work was supported by EC-funded projects InGeoClouds (FP7-297300), MIDAS (FP7-318786) and eCloud EU Project (CIP-325091), by Italy's MIUR-funded project PRIN ARS Technomedia 2012.

8. REFERENCES

[1] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *SIGIR*, pages 35–42, 2001.

[2] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1), 2005.

[3] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Softw., Pract. Exper.*, 40(2):131–147, 2010.

[4] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Y. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, pages 426–434, 2003.

[5] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information retrieval: implementing and evaluating search engines.* MIT Press, Cambridge, Mass., 2010.

[6] R. H. Byrd, J. Nocedal, and R. B. Schnabel. Representations of quasi-newton matrices and their use in limited memory methods. *Mathematical Programming*, 63(1-3):129–156, 1994.

[7] F. Chierichetti, S. Lattanzi, F. Mari, and A. Panconesi. On placing skips optimally in expectation. In *WSDM*, pages 15–24, 2008.

[8] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM*, 2009.

[9] R. Delbru, S. Campinas, and G. Tummarello. Searching web data: An entity retrieval and high-performance indexing model. *J. Web Sem.*, 10:33–58, 2012.

[10] R. Dementiev, L. Kettner, and P. Sanders. Stxxl: standard template library for xxl data sets. *Software: Practice and Experience*, 38(6):589–637, 2008.

[11] P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.

[12] R. M. Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT, Cambridge, MA*, 1971.

[13] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning.* Springer, 2001.

[14] P. J. Huber et al. Robust estimation of a location parameter. *The Annals of Mathematical Statistics*, 35(1):73–101, 1964.

[15] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice & Experience*, 2013.

[16] C. Macdonald, N. Tonellotto, and I. Ounis. Learning to predict response times for online query scheduling. In *SIGIR*, pages 621–630, 2012.

[17] C. D. Manning, P. Raghavan, and H. Schülze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[18] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1), 2000.

[19] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 10:205–231, June 2007.

[20] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *SIGIR*, pages 273–282, 2014.

[21] S. E. Robertson and K. S. Jones. Relevance weighting of search terms. *Journal of the American Society for Information science*, 27(3):129–146, 1976.

[22] D. Salomon. *Variable-length Codes for Data Compression.* Springer, 2007.

[23] F. Silvestri. Sorting out the document identifier assignment problem. In *ECIR*, pages 101–112, 2007.

[24] F. Silvestri and R. Venturini. VSEncoding: Efficient coding and fast decoding of integer lists via dynamic programming. In *CIKM*, pages 1219–1228, 2010.

[25] P. Sinha and A. A. Zoltners. The multiple-choice knapsack problem. *Operations Research*, 27(3):503–515, 1979.

[26] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. Simd-based decoding of posting lists. In *CIKM*, pages 317–326, 2011.

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

[28] S. Vigna. Quasi-succinct indices. In *WSDM*, 2013.

[29] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images.* Morgan Kaufmann Publishers Inc., 1999.

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

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

[32] E. Zemel. An O(n) algorithm for the linear multiple choice knapsack problem and related problems. *Inf. Process. Lett.*, 18(3):123–128, 1984.

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

[34] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.