2019

A Flexible Space-Time Tradeoff on Hybrid Index with Bicriteria Optimization

Xingshen Song  
*School of Advanced Interdisciplinary Studies, National University of Defense Technology, Changsha 410000, China.*

Yuexiang Yang  
*College of Computer, National University of Defense Technology, Changsha 410000, China.*

Yu Jiang  
*Northwest Institute of Nuclear Technology, Xi’an 710000, China.*

Follow this and additional works at: [https://tsinghuauniversitypress.researchcommons.org/tsinghua-science-and-technology](https://tsinghuauniversitypress.researchcommons.org/tsinghua-science-and-technology)

Part of the Computer Sciences Commons, and the Electrical and Computer Engineering Commons

Recommended Citation

Xingshen Song, Yuexiang Yang, Yu Jiang. A Flexible Space-Time Tradeoff on Hybrid Index with Bicriteria Optimization. Tsinghua Science and Technology 2019, 24(1): 106-122.

This Research Article is brought to you for free and open access by Tsinghua University Press: Journals Publishing. It has been accepted for inclusion in Tsinghua Science and Technology by an authorized editor of Tsinghua University Press: Journals Publishing.
A Flexible Space-Time Tradeoff on Hybrid Index with Bicriteria Optimization

Xingshen Song*, Yuexiang Yang, and Yu Jiang

Abstract: Inverted indexes are widely adopted in the vast majority of information systems. Growing requirements for efficient query processing have motivated the development of various compression techniques with different space-time characteristics. Although a single encoder yields a relatively stable point in the space-time tradeoff curve, flexibly transforming its characteristic along the curve to fit different information retrieval tasks can be a better way to prepare the index. Recent research comes out with an idea of integrating different encoders within the same index, namely, exploiting access skewness by compressing frequently accessed regions with faster encoders and rarely accessed regions with succinct encoders, thereby improving the efficiency while minimizing the compressed size. However, these methods are either inefficient or result in coarse granularity. To address these issues, we introduce the concept of bicriteria compression, which aims to formalize the problem of optimally trading the compressed size and query processing time for inverted index. We also adopt a Lagrangian relaxation algorithm to solve this problem by reducing it to a knapsack-type problem, which works in $O(n \log n)$ time and $O(n)$ space, with a negligible additive approximation. Furthermore, this algorithm can be extended via dynamic programming pursuing improved query efficiency. We perform an extensive experiment to show that, given a bounded time/space budget, our method can optimally trade one for another with more efficient indexing and query performance.

Key words: inverted index; bicriteria compression; Lagrangian relaxation

1 Introduction

Indexing is essential to efficiently manage a growing data and perform fast queries in many Information Retrieval (IR) systems. The use of inverted indexes on Web search engines is probably one of the most successful examples of IR. Although an inverted index is a relatively old and simple data structure, it is extensible with supplementary information to answer different queries within reduced space. Compression has been one of the most active fields of inverted index research in the last decades, because it not only reduces space occupancy but also accelerates data transfer in input/output (I/O). References [1–3] provide a detailed survey.

A typical inverted index is composed of two parts: lexicon and posting lists. The inverted index compression discussed in this paper usually focuses on the compression of posting lists. However, divergence exists between compressed size and decompression time because pursuing size minimization will undoubtedly impair decoding. Encoders with different space-time tradeoffs keep coming out to meet various compression requirements [4–11]. Among
these techniques, splitting posting list into blocks (or chunks) is an efficient way that is widely adopted by encoders because faster decoding speed and hierarchical structure can be achieved by keeping blocks separately accessible. In addition, variable-sized blocks can utilize the clustering property of list to beat entropy\cite{11,12}. Nonetheless, designers of information systems have to carefully choose one of these encoders to satisfy different IR tasks: whether to achieve optimal compression ratio or decoding speed by sacrificing one of them, or to balance them by adopting a suitable encoder with finely tuned space-time tradeoff\cite{13}.

As a single encoder has its own stable performance in the space-time tradeoff curve, a natural approach is to ask for an encoder that can trade the space occupancy and decoding time flexibly to fit in a different system requirement and device context. An intuitive way to solve this problem is by combining different encoders into one index, specifically, compressing the frequently accessed parts using a faster but less space-efficient encoder and other parts using a slower encoder that provides improved compression. This idea was first adopted by Yan et al.\cite{14} However, the procedure of choosing encoders works at a posting list granularity, which is too coarse to obtain a flexible space-time tradeoff; these encoders also fail to explore the complexity of the algorithm as it works on an empirical foundation. Ottaviano et al.\cite{8} explicitly defined the space-constrained expected time minimization problem as follows: given a dataset $D$ and an upper bound $S$ on its compressed size, the goal is to select one proper encoder from an encoder set $\mathcal{E}$ for each block of the index. This method minimizes the expected query processing time provided that it can be compressed within $S$ space. Furthermore, Ottaviano et al.\cite{8} proposed a linear time solution for this problem, which is straightforward. Under two predefined dominance criteria, their algorithm works by enumerating all the advisable compression methods, and then sorting them in a non-increasing order by the time/space gain ratio of each advisable encoder on each block. Thereafter, a sequential scan-and-replace method finds all the optional solutions for any possible budget constraint $S$. As each posting list is divided into fixed-sized blocks with a total number of $n$ and a set of $k$ encoders, the algorithm finishes in $\Theta(nk \log (nk))$ time and $O(nk)$ space. By refining the granularity to block level and the carefully designed algorithm, Ottaviano et al.\cite{8} found a solution with minimal query time. The approximation is also additive, which is at most a block size larger than $S$. However, the first concern is that the algorithm takes too much space to sort the compression options that it can be hardly fit into the memory; second, as the sorting and sequential scan cannot be accelerated in parallel, the entire procedure is delayed by them somehow; third, the solution is based on the assumption that the posting lists are split into fixed-sized blocks; otherwise, the compression options are difficult to sort and might take much more space.

We argue that global ordering of the compression options for all the blocks is not a good idea, because it leads to the aforementioned three shortcomings. In this paper, we try to solve this problem from another perspective. The main contributions are listed as follows:

1. We introduce the concept of bicriteria compression from Farruggia et al.\cite{15} to extend the space-constrained expected time minimization problem and its dual problem (exchange the role of space and time by asking for the minimal compressed size, provided an upper bound $T$ in its average query processing time or decompression time). By solving the bicriteria compression problem, we can expect the compression method to be more universal for different IR tasks. Moreover, we also explore performance enhancement by introducing variable-sized partitions into index compression, which is complex to implement for the method in previous work\cite{8}.

2. We reduce the problem into a resource constrained shortest path problem\cite{16} over a weighted DAG $G$, whose edges contain two types of penalties: space cost and time weight. Thus, the problem can be formalized using Lagrangian relaxation, and we use the algorithm introduced by Farruggia et al.\cite{15} and Handler and Zang\cite{17} to settle in $O(n \log n)$ time and $O(n)$ space while keeping the same additive approximation guarantee.

3. Different from the method used by Ottaviano et al.\cite{8}, which is a greedy algorithm, our method amortizes the expected budget over each posting list in a more balanced way. To guarantee a lower bound of the query processing time (or decoding time), each posting list is treated indiscriminately rather than focusing on a few frequently accessed lists. Thus, we can expect a better average performance. While keeping the same space and time complexity, our method is more efficient and needs less runtime memory in practice without global sorting.
(4) We execute a detailed experimental evaluation on two realistic datasets, GOV2 and Common Crawl, with AOL query log. Analysis shows that our method is able to obtain flexible tradeoffs under bicriteria compression. Our method’s compressed size and query processing time are competitive with the hybrid index by Ottaviano et al.\cite{8}, but with a more detailed description and more strict theoretical foundation.

2 Preliminaries

2.1 Index structure and compression

Given a collection of $D$ documents, an inverted index can be considered as a large table that maps each unique term to a posting list, which contains all the document identifiers (called `docid`), the number of occurrences in the document (called `frequency`), and possibly other information such as the positions of each occurrence within the documents (to support phrasal and proximity matching). The set of terms is called `lexicon`, which is smaller than postings. To rank the documents in response to a query, the posting list for the terms of a query must be traversed. Numerous studies have investigated structures and query processing strategies; see Refs.\cite{18–20} for a detailed survey.

To facilitate compression and query processing, posting lists from the index are always considered separate data streams, components in each posting list (docid, frequency, and position) are stored in a non-interleaved way, and the lists of docid and position are usually sorted and transformed to the corresponding sequence of differences (or `gaps`) between adjacent values. Thus, the task of index compression is best viewed as coding sequences of integers of the form $x_1, x_2, x_3, \ldots, x_m$, where $x_i \geq 0$ for all $1 \leq i \leq m$, and $m$ is the number of postings.

Index compression is a relatively old and established topic in the IR literature, and researchers have summarized their previous work explicitly\cite{1,2,4,11}. The variety of compression techniques can be roughly divided into two classes, namely, integer-oriented encoders and list-oriented encoders. The integer-oriented encoders assign a unique codeword to each integer of the input list, and then the compression procedure turns into a mapping or substitution from the integer space to code space. As they compress integers without considering their neighbors, the integer-oriented encoders are also called oblivious encoders\cite{41}, such as `unary code`, `Elias Gamma/Delta codes`, and `Golomb/Rice codes`. Most integer-oriented encoders are difficult to decode because they need bitwise operations to cross computer word boundaries; thus, byte/word-aligned encoders are proposed to solve this problem, such as `Variable Byte` and `Group Varint`. More importantly, they can be further improved by SIMD instructions of modern CPUs\cite{2,7,21}. List-oriented encoders are designed to exploit the cluster of neighboring integers, each time a fixed-sized or variable-sized group of integers is binary packed with a uniform bit width, thereby providing equivalent compression ratio and faster decoding speed; the technique used by these encoders is called `Frame Of Reference` (FOR) or `binary packing`\cite{22,23}. Basically, their compression ratios are inferior to those of the first category as a batch of integers are encoded indiscriminately, and useless zeros are filled in the codeword to maintain word alignment. However, when decoded, list-oriented encoders can obtain an entire block while integer-oriented encoders only decode one integer at a time. More importantly, with the help of `skip pointers` or `skip list`, it is possible to step along the codewords compressed by list-oriented encoders and stop when the required number of blocks has been bypassed. Examples of these encoders are `Simple-X`\cite{24–27}, `AFOR`\cite{28}, and `Patched FOR` (PFOR, OptPFOR, and FastPFOR)\cite{4,6,14,29}.

Our scenario follows the settings used by Ottaviano et al.\cite{8}, where encoders in $E$ are selected from list-oriented encoders and posting lists are uniformly split into fixed-sized blocks of 128 postings. These posting lists are virtually concatenated into one large list to facilitate our computation under a bounded space/time budget. As we only store some statistics about each block, this list will not occupy a large space.

2.2 Pareto-optimal compression

Recently, researchers have begun to use breakthroughs in succinct data structures to solve problems in inverted index compression. Ottaviano and Venturini\cite{9} used pruned DAG\cite{30} to optimally partition a posting list for minimal compression and convenient access. Petri et al.\cite{10} described a data structure that combines a pruned suffix tree and inverted index together to facilitate phrase-based ranking. Our work is similar to these efforts, namely, applying bicriteria compression to the inverted index.

In this study, we are interested in optimally trading space occupancy with query processing time
under a specified space/time budget. With respect to optimality, we introduce the concept of Pareto-optimal compression to define it in a principled way: encoders achieve different points in the space-time tradeoff curve to set the space occupancy and query processing time as two dimensions of the coordinate system. Optimal ones are those extreme points whose performances are not worse than others in one dimension, and if to be optimized in any dimension, performances in another dimension will be impaired accordingly. Obviously, a set of Pareto-optimal compressions with different considerations exist. Figure 1 shows the performances of different compression techniques. The installation of these implementations is same as that in the Experiments section. Although the results are different from those of other works, they are directly comparable. From the fitting curve and marginal rugs, we can observe that points that approach the limit of one dimension begin to cluster and the performances of the other two dimensions drop considerably. Points beyond the curve are either superior or inferior to the average (i.e., VB and AFOR). Since encoding speed is less urgent than the other two dimensions, we only focus on the tradeoff between decoding speed and compressed size. As the figure shows, the points span a broad spectrum of time and sizes, and if properly assigned, they can satisfy any budget target of required space and time.

As an extreme example, two solutions are considered: one is space-optimal compressed by Binary Interpolative Coding \[12\] and the other one is time-optimal compressed by SIMD encoders. That is, for any block of the posting list, we can always select either the space-efficient encoder or the time-efficient one. However, between these two extremes, a plethora of encoders such as PFOR and Simple-X exists. Our goal is to move from one extreme to another by trading decompression speed for space occupancy or vice versa. As there are \(k\) encoders for \(n\) blocks, the key is to automatically and efficiently select any of them for optimal tradeoffs.

2.3 Directed acyclic graph

By modeling index compression as a directed acyclic graph \(\mathcal{G}\), we can reduce the optimal solution to the single source shortest path problem over it.

\(\mathcal{G}\) is built as follows: a posting list can be treated as a list containing only positive integers \(S[0, m]\), where \(m\) is the number of postings. Then, each integer is represented by a vertex, plus a dummy vertex marking the end of the sequence. \(\mathcal{G}\) is complete with \(O(m^2)\) edges, and each edge is an exact correspondence of a partition in \(S\). The weight of an edge in \(\mathcal{G}\) is equal to the cost in bits consumed by the partition. The problem of fully partitioning \(S\) is converted to finding a path \(\pi \in \Pi\) in \(\mathcal{G}\), with its total edge weights \(\omega(\pi)\) minimized. It has been commonly used in many list-oriented encoders\[31\]. These encoders usually partition a posting list into chunks of different lengths aligned to its local clusters. Earlier encoders such as Fixed Binary\[24\] and AFOR use a simple greedy search without backtracking. Silvestri and Venturini\[11\] introduced dynamic programming recurrence for VEncoding while Ottaviano and Venturini\[9\] resorted to a more efficient approximation algorithm.

However, our version of DAG \(\mathcal{G}\) has different definitions. The aforementioned methods are designed to pursue space-optimal partitioning at all costs, without considering the decompression speed. Our goal is to find a Pareto-optimal solution under predefined space/time budget. In the new graph, each edge is attached with two weights: a time weight that accounts for the time to decompress a block, and a space cost that accounts for the number of bits needed to store a block associated to that edge. The most notable contrast in our \(\mathcal{G}\) lies in the blocks of constant size (for example, 128 elements), which determine edges that have fixed start and end points. Between these points, multiple edges are available to choose from due to the diversity of encoders. Thus, the total number of edges is exactly \(k \cdot \left[\frac{m}{128}\right]\) rather than \(O(m^2)\). For every path \(\pi\) from node 1 to node \(m\), we use \(s(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{k} x_{i,j} s_{i,j}\)

\[\text{Fig. 1 Performance of different encoders under tri-criteria. The x-axis (bpi) is arranged in a reverse order and the color of each point indicates the encoding speed in million integers per second (mis). A darker color represents faster speed.}\]
to represent the entire compressed size, where \( n = \left\lceil \frac{m}{128} \right\rceil \), \( x_{i,j} \) represents the choice of the encoder for block \( i \), and is 1 if the chosen encoder is \( j \) and 0 otherwise; and \( s_{i,j} \) denotes the space cost of block \( i \) using encoder \( e_j \). Accordingly, the total decompression time is set to be \( t(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{k} x_{i,j} t_{i,j} \). Thus, we are able to rephrase the bicriteria compression problem into a Resource Constrained Shortest Path problem (RCSP) over the double weighted graph \( G \), namely, finding a path \( \pi \) whose compressed size \( s(\pi) < S \) while its decompression time \( t(\pi) \) is minimized. The role of space/time resources can also be exchanged symmetrically, which requires minimal \( s(\pi) \) with \( t(\pi) < T \). For convenience, we will consider the first formulation only in the rest of the paper. An example of two different DAGs is shown in Fig. 2.

3 Modeling the Problem

To formulate the bicriteria compression problem, we first need to model the space occupancy and decompression time of each block. As stated in Section 2, we assume that each posting list is split into fix-sized blocks with a total number of \( n \). Its space cost \( s_{i,j} \) can be obtained by compressing blocks using all the encoders before processing. We follow the previous work by Ottaviano et al.\,[8], namely, we set \( T \) as the expected time to process queries from a given query set \( \mathcal{Q} \), and adopt the same model to predict the decompression time \( t_{i,j} \). Thus, \( T \) is closely related to query processing strategy used and the distribution in \( \mathcal{Q} \). We set \( t(\pi) = \sum_{i=1}^{n} t_{i,j} \), where \( \hat{t}_{i,j} = f_{i} t_{i,j} \), \( f_{i} \) is the number of times the \( i \)-th block was decoded while processing \( \mathcal{Q} \). Also, \( f_{i} \) can be set to 1 to measure the thorough decompression time of the index.

Now, we can formulate our problem using Lagrangian relaxation into the famous RCSP as follows:
\[
\begin{align*}
\min_{\pi \in \mathcal{P}} & \ t(\pi), \\
\text{s.t.} & \ s(\pi) \leq S
\end{align*}
\]
By setting \( f(\pi) = t(\pi) \) and \( g(\pi) = s(\pi) - S \), we may rewrite Formula (1) as
\[
\begin{align*}
\min_{\pi \in \mathcal{P}} & \ f(\pi), \\
\text{s.t.} & \ g(\pi) \leq 0.
\end{align*}
\]
A brute force solution to this problem would cost \( O(n^5) \) time and \( O(nk) \) space. By adopting a dynamic programming approach\,[32], we can reduce both time and space to \( \Theta(nS) \), which is \( O(m^2 \log m) \) at its worst. Unfortunately, this bound is still unacceptable in practice.

To efficiently solve this problem, we introduce the Lagrangian function to relax the constraint \( g(\pi) \leq 0 \):
\[
L(\mu, \pi) = f(\pi) + \mu g(\pi)
\]
and let
\[
L(\mu) = \min_{\pi \in \mathcal{P}} L(\mu, \pi)
\]
\( \mu \) is the Lagrangian multiplier following \( \mu \geq 0 \). It has been shown that this problem can be solved in linear time by solving this dual problem\,[17]. Let \( \pi \) be the path we find, and \( \pi^* \) be the optimal path of the RCSP problem. At the end of Section 4, we will prove the following theorem:

**Theorem 1** \( \pi \) can be computed in \( O(\log n) \) time and \( O(n) \) time with its time cost \( t(\pi) \leq f(\pi^*) + t_{\text{max}} \) and space cost \( s(\pi) \leq S + s_{\text{max}} \).

Here, we use \( s_{\text{max}} \) and \( t_{\text{max}} \) to denote the maximum space and time cost of one block.

4 Our Approximation Algorithm

In this section, we turn to the algorithm that solves the Lagrangian relaxation problem. Recall the algorithm used by Ottaviano et al.\,[8], which first sorts all the possible encodings of each block across the entire index, and then adopts a greedy search to scan the sorted list and replace a space-optimal solution with faster encodings until the space budget \( S \) is reached, thereby meeting its additive guarantee.

Our algorithm works in a different way, and its procedure can be divided into two phases. In the first phase, we adopt the cutting-plane algorithm proposed by Handler and Zang\,[17] to solve the Lagrangian dual problem. This algorithm starts from two extreme paths.
which are optimal for $L$. Thus we obtain an instantiation of $\pi$ that represents the lower envelope of all the paths generated so far, so it consists of two lower segments of the corresponding paths and the point of intersection maximizes $L$. If another path $L(\mu)$ is generated using $\mu$ exists, which further tightens the lower envelope, then we can obtain the maximum value of $L$ that is closer to $L^*$ from above. Thus, the entire procedure works to find a minimal maximum of $L$ because $L(\mu) > L(\mu^*)$. The iteration stops once $L(\mu')$ cannot be reduced or we find a path $\pi^0 \in \Pi$ with $g(\pi) = 0$ and $L(\mu') = L(\mu', \pi^*) < L'$, then the point $(\mu', L(\mu'))$ represents the optimal solution for the dual problem.

For a given $\mu$, a path is called $\mu$-optimal if its Lagrangian cost $L(\pi, \mu)$ is equal to the value of $L(\mu)$ in Eq. (3). We consider the initial step, which is to find two extreme paths with respect to function $f(\pi)$ and $g(\pi)$, respectively. Since $\mu \in [0, +\infty)$ and for each path, $L(\mu)$ represents a monotonic line in $(\mu, L)$ space. Naturally, the extreme path exists at $\mu = 0$ and $\mu = \infty$. Let $\pi(\mu)$ denote a $\mu$-optimal path associated with $\mu$, and $\pi(0)$ corresponds to $L(0) = \min\{f(\pi) | \pi \in \Pi\}$ which is the minimum of $f(\pi)$, if $g(\pi) < 0$ then $\pi(0)$ is also optimal for RCSP. And $\pi(\infty)$ corresponds to the minimum of $g(\pi)$, if $g(\infty) > 0$, then no solution can be obtained for RCSP.

We can easily prove that $\pi(0)$ and $\pi(\infty)$ are exactly time-optimal and space-optimal paths in our scenario. We start by setting $(\pi(0), \pi(\infty))$ as $(\pi_l, \pi_r)$, each time we intersect them, we get the point $\mu' = (f_l - f_r)/(g_r - g_l)$ and $L' = f_l + \mu' g_l$. Then a new path $\pi'$ is generated as $L(\mu')$, if $g(\pi') < 0$, $\pi_r$ will be replaced by $\pi'$ and vice versa. The procedure terminates when $L(\mu') = L'$ or $g(\pi') = 0$. It has been shown the number of iterations is $O(n s_{\text{costs}} t_{\text{costs}})$ [16], where $s_{\text{costs}}$ and $t_{\text{costs}}$ represent the block space/time cost in integers, in our scenario these two are negligible constants since block size is fixed to 128. This procedure is illustrated in Fig. 3 (quoted from Ref. [15]).

It remains to describe how to generate a $\mu$-optimal path $\pi'$ given the point of intersection $(\mu', L')$. A brute force search along the whole graph for $\pi'$ by evaluating its Lagrangian cost is impossible, even if our version of $G$ has already reduced its number of edges from (for example, $\pi(0)$ and $\pi(\infty)$) and fuses them to obtain a new path $\pi'$, then recursively fuses $\pi'$ with one of the former path until a lower bound of $L(\mu, \pi)$ is reached.

Unfortunately, this condition rarely occurs in practice and in most cases we have to address the duality gap between $\pi_l$ and $\pi_r$. In the second phase, we will execute a sequential exchange of edges from both paths. Thereafter, we will be able to obtain an optimal path whose space cost effectively approximates the budget.

### 4.1 Solving the dual problem

This phase mainly contributes to the fusion of two extreme paths, one of which is space-optimal while the other is time-optimal, thereby making the fused path converge gradually to the given range through a finite iteration. The key step lies in generating the optimal path $\pi'$ for $\mu$ in each iteration.

Note that in Eq. (2) we have $\mu \geq 0$ and $g(\pi) \leq 0$. Therefore, the best lower bound for the optimal path in RCSP can be determined by solving its dual problem defined as

\[
L^*(\mu) = L(\mu^*) = \max_{\mu \geq 0} L(\mu) = \max_{\mu \geq 0} \min_{\mu \geq 0} \{f(\pi) + \mu g(\pi)\}
\]

Now the problem has been converted from searching for an optimal path to a linear programming that calculates maximal $L(\mu)$, where each path defines a feasible region. Our goal is to determine the vertex which maximizes $L(\mu)$ from the convex hull defined by the intersection of paths.

Figuring out the convex hull using the traditional simplex method is impossible because $O(n^k)$ paths may exist. However, we only need a small part of these paths to heuristically prune out unfeasible ones and determine the maximal $L(\mu)$.

The procedure can be best explained geometrically. Each path in $\Pi$ may be associated in $(\mu, L)$ space with a line in the form $L = f(\pi) + \mu g(\pi)$, where $f(\pi)$ is the intercept along the L-axis and $g(\pi)$ is the slope of the line. Feasible paths have a non-positive slope (because $g(\pi) \leq 0$) and unfeasible paths have a positive slope. As mentioned, we only keep a pair of paths in hand and recursively intersect them to determine the point $(\mu', L')$. Note that $L(\mu)$ represents the lower envelope of all the paths generated so far, so it consists of two lower segments of the corresponding paths and the point of intersection maximizes $L(\mu)$. If another path $L(\mu')$ is generated using $\mu'$ exists, which further tightens the lower envelope, then we can obtain the maximum value of $L$ that is closer to $L^*$ from above. Thus, the entire procedure works to find a minimal maximum of $L$ because $L(\mu) \geq L(\mu^*)$. The iteration stops once $L(\mu')$ cannot be reduced or we find a path $\pi^0 \in \Pi$ with $g(\pi) = 0$ and $L(\mu') = L(\mu', \pi^*) < L'$, then the point $(\mu', L(\mu'))$ represents the optimal solution for the dual problem.

For a given $\mu$, a path is called $\mu$-optimal if its Lagrangian cost $L(\pi, \mu)$ is equal to the value of $L(\mu)$ in Eq. (3). We consider the initial step, which is to find two extreme paths with respect to function $f(\pi)$ and $g(\pi)$, respectively. Since $\mu \in [0, +\infty)$ and for each path, $L(\mu)$ represents a monotonic line in $(\mu, L)$ space. Naturally, the extreme path exists at $\mu = 0$ and $\mu = \infty$. Let $\pi(\mu)$ denote a $\mu$-optimal path associated with $\mu$, and $\pi(0)$ corresponds to $L(0) = \min\{f(\pi) | \pi \in \Pi\}$ which is the minimum of $f(\pi)$, if $g(\pi) < 0$ then $\pi(0)$ is also optimal for RCSP. And $\pi(\infty)$ corresponds to the minimum of $g(\pi)$, if $g(\infty) > 0$, then no solution can be obtained for RCSP.

We can easily prove that $\pi(0)$ and $\pi(\infty)$ are exactly time-optimal and space-optimal paths in our scenario. We start by setting $(\pi(0), \pi(\infty))$ as $(\pi_l, \pi_r)$, each time we intersect them, we get the point $\mu' = (f_l - f_r)/(g_r - g_l)$ and $L' = f_l + \mu' g_l$. Then a new path $\pi'$ is generated as $L(\mu')$, if $g(\pi') < 0$, $\pi_r$ will be replaced by $\pi'$ and vice versa. The procedure terminates when $L(\mu') = L'$ or $g(\pi') = 0$. It has been shown the number of iterations is $O(n s_{\text{costs}} t_{\text{costs}})$ [16], where $s_{\text{costs}}$ and $t_{\text{costs}}$ represent the block space/time cost in integers, in our scenario these two are negligible constants since block size is fixed to 128. This procedure is illustrated in Fig. 3 (quoted from Ref. [15]).

It remains to describe how to generate a $\mu$-optimal path $\pi'$ given the point of intersection $(\mu', L')$. A brute force search along the whole graph for $\pi'$ by evaluating its Lagrangian cost is impossible, even if our version of $G$ has already reduced its number of edges from
O(n^2) to O(nk) by fixing the edge length. However, fixing the size of blocks indeed brings one significant advantage where blocks are kept isolated from each other. Thus we can break this problem into numerous subproblems of selecting proper encodings for each block, thereby minimizing its Lagrangian cost given any \( \mu \geq 0 \). Finally, we can sum their space/time costs together without collision. Specifically,

\[
L(\mu, \pi) = L(\mu, \sum_{i=1}^{n} e_{i}) = \sum_{i=1}^{n} L(\mu, e_{i}),
\]

here \( e_{i} \) denotes edges in path \( \pi \).

To find the appropriate encodings, we need first to recall the dominance criteria proposed in Ref. [8]:

1. Let \( p \) and \( q \) be two items in the same class \( i \). Item \( q \) is dominated by item \( p \) if \( s_{i,p} \leq s_{i,q} \) and \( \hat{t}_{i,p} < \hat{t}_{i,q} \).
2. Let \( p, q, \) and \( r \) be three items in the same class such that \( s_{i,p} < s_{i,q} < s_{i,r} \) and \( \hat{t}_{i,p} > \hat{t}_{i,q} > \hat{t}_{i,r} \). Item \( q \) is dominated by \( p \) and \( r \) if the following condition holds:

\[
\frac{\hat{t}_{i,p} - \hat{t}_{i,q}}{s_{i,p} - s_{i,q}} \geq \frac{\hat{t}_{i,q} - \hat{t}_{i,r}}{s_{i,q} - s_{i,r}}.
\]

After applying these criteria, for each block we have the dominated items pruned from \( \mathcal{G} \) and the rest items sorted in ascending order by their space cost. Also a time/space gain ratio between adjacent items is defined as \( \lambda_{i,j} = \frac{\hat{t}_{i,j} - t_{i,j-1}}{s_{i,j-1} - s_{i,j}} \).

Let \( L_{i,j} \) denote the Lagrangian cost of the \( i \)-th block encoded by the \( j \)-th encoder. Our key observation is that the minimal Lagrangian cost of each block can be found by making the smallest \( \lambda_{i,j} \) that approximates \( \mu \) from above, as shown in the following equation:

\[
L_{i,j} - L_{i,j-1} = f(e_{i,j}) - f(e_{i,j-1}) + \mu(g(e_{i,j}) - g(e_{i,j-1})) = \hat{t}_{i,j} - \hat{t}_{i,j-1} + \mu(s_{i,j} - s_{i,j-1}) = \lambda_{i,j}(s_{i,j} - s_{i,j-1}) + \mu(s_{i,j} - s_{i,j-1}) = (s_{i,j} - s_{i,j-1})(\mu - \lambda_{i,j})
\]

Since \( s_{i,j} > s_{i,j-1} \), encodings with \( \lambda_{i,j} \geq \mu \) will reduce the cost of current block, and the minimal \( L_{i} \) is exactly \( \min \{ \lambda_{i,j} | \lambda_{i,j} \geq \mu \} \).

The set of \( \lambda \)s is sorted once and accessed many times, which can be done in constant time with simple operations because only \( k \) encoders are available to choose from. Here, we emphasize that the above operations are conducted within a block level rather than the entire index. Generating a full path would obviously cost \( O(nk) \) time and space. Recall the number of iterations previously mentioned, the dual problem can be solved in \( O(\bar{n} \log(n_{cost}k_{cost})) \) time and \( O(\bar{n}) \) space, where \( n = nk \) is the size of \( \mathcal{G} \).

### 4.2 Closing the gap

Upon termination of the iteration, we have two paths \((\pi_{l}, \pi_{r})\) which specify a lower bound and an upper bound of the minimal value \( f(\pi^{*}) \). From Eq. (4), we know the lower bound is the intersection point, namely \( LB = L(\mu^{*}) \); also, any path \( \pi \) with \( g(\pi) \leq 0 \) generated during the iteration is a feasible solution for RCSP; and the best value for upper bound is \( f(\pi_{l}) \) from the last path \( \pi_{l} \). Thus we have the bound \( L(\mu^{*}) \leq f(\pi^{*}) \leq f(\pi_{r}) \), which is also shown in Fig. 3.

Obviously, if \( L(\mu^{*}) = f(\pi_{r}) \) or any one path in \((\pi_{l}, \pi_{r})\) satisfies \( g(\pi) = 0 \), no duality gap would exist between solutions to RCSP and its dual problem. However, this rarely happens in practice. Therefore, we have to deal with duality gap until an optimal path is found for RCSP.

Following the solution used in Ref. [15], we reuse the path-swapping algorithm introduced for this task. The notion of path swapping can be explained as follows: given a pair of paths \((\pi_{l}, \pi_{r})\), we first choose one point \( v \) along the paths arbitrarily as a pivot, then we create two new paths \((\pi_{1}, \pi_{2})\) by exchanging edges before the pivot of \((\pi_{l}, \pi_{r})\). In our context, the algorithm is greatly simplified since edges of all paths have fixed start and end points, we do not need to consider cases where edges are split or merged for shared pivot. It has been shown that any path generated by path-swap is off at most \( t_{max} \) in time and \( s_{max} \) in space from being a \( \mu \)-optimal path, \( t_{max} \) and \( s_{max} \) denote the maximum time and space cost of one block, respectively. Indeed, the operation is executed on two closely resembled \( \mu \)-optimal paths edgewise. By moving the pivot from left to right, we are actually substituting path \( \pi_{l} \) for path \( \pi_{r} \) edge by edge. These intermediate paths are naturally...
bounded by $\pi_l$ and $\pi_r$, and there definitely exists a $\mu$-optimal path whose cost is at most one block size smaller than these paths.

The block-wise operation also guarantees that paths generated using two adjacent pivots differ from each other by at most $t_{max}$ and $s_{max}$. A left-to-right scanning is sufficient to change the Lagrangian cost from $L(\mu^*)$ to $f(\pi_r)$ in an appropriate granularity, and find an optimal solution for $f(\pi^*)$ with an additive approximation. During the procedure, we only need an accumulator to store the current pivot and time/space cost; thus, the path-swap algorithm can be conducted in $O(n)$ time and $O(1)$ space.

5 Engineering the Block Size Issue

In the previous section, we have addressed an algorithm to find the optimal assignment of encoders to build the inverted index under given budget, however, using only fixed-sized blocks. In this section, we take things one step further by exploring the feasibility of extending our algorithms with blocks of variable sizes. There is a common view that splitting posting list into fixed-sized blocks tend to be suboptimal, since integers are not expected to be uniformly distributed along the list, a mandatory split makes the compressed size vulnerable to outliers inside the block\[9, 11\]. Using variable-sized blocks can effectively improve the situation, for it not only reduces the space occupancy, but also the number of blocks in the whole index, thus, we can expect a more efficient skipping when processing user queries. However, arbitrarily dividing a posting list can be prohibitive due to the following reasons. First, the edge complexities of DAG grow larger by introducing variable-sized blocks, which can be $O(n^2k)$; second, encoders may have additional requirements on the block size, for example, binary packing (PFOR, AFOR) needs a 32-fold block size to keep the codeword boundary aligned with computer words, also the compression and decompression procedure can benefit from hard coded function; third, the optimization algorithm determines the block size and encoder based on the knowledge of the space occupancy and decoding time for each block. Given many possible blocks, a large overhead for runtime memory is needed to store the whole information, and the decoding time becomes unpredictable with existing method.

To use blocks of variable sizes, we have to make a compromise by adding several specifications. The base block size is set as 128, and can be multiplied up to $h$-fold, where $h$ is set as 8. Specifically, we compress the posting list at a block level rather than posting level, and only merge base blocks rather than splitting them, so the number of edges is reduced to $O(nhk)$ when building a path $\pi$.

We emphasize that the original algorithm of Ref. [8] is unable to address this problem, since it relies on a large sorted queue to choose proper encoders for every block, once merged blocks are mixed in, elements in the queue are no longer globally unique. However, our method adopts linear programming to asymptotically calculate a path with minimal $L$ under a given $\mu$, the path can be found by a sequential scan and compare all the possible blocks. Although inefficient in practice, our method is certainly a feasible way to integrate blocks of variable sizes.

Next we discuss the details of our implementation to optimally choose the block sizes and encoders. The algorithm stated in Section 4 mainly stays unchanged, the only difference lies in how we generate the new path $\pi'$. In the previous section, blocks are kept isolated from each other, we only sequentially pick the encodings that offer optimal time/space gain ratio $\lambda_{i,j}$ for each block under given $\mu$. Here, we first generate a path using only base blocks by selecting encoders with $\min\{\lambda_{i,j} | \lambda_{i,j} \geq \mu\}$, then a recursive function is invoked to merge blocks to find a smaller Lagrangian cost. Note the function traverses the list at a block level rather than posting level, we refer to the block of 128 elements as our base operand in the following section.

This function can be solved via dynamic programming paradigm with the following recurrence:

$$A[i] = \min_{l \in \{\max(0, i-h), i-1\}} (A[j] + c(l, i))$$

(7)

here, $A[i]$ denotes the accumulated optimal Lagrangian cost of the previous blocks, and $c(l, i)$ accounts for the Lagrangian cost of the merged base blocks, which start from $l$ to $i$, as a single one. Namely, $c(l, i) = f(l, i) + \mu g(l, i)$.

Generally, the recurrence starts from block 0 to $n$, with $k$ encoders and at most $h$ blocks to merge, the whole procedure can be solved in $O(nhk)$. However, further stipulations are needed to reduce its time complexity. We have retained the calculation for single blocks in the first step, not only because it can be extremely fast to perform, but also for the reason that it provides a baseline for comparison. According to
the encoders assigned to each base block, we can skip over many invalid trials for merged blocks because the initial threshold is set larger than 0. Also, if two adjacent base blocks differ considerably in the assigned encoders or access counts, they are less likely to be merged, otherwise the accumulated Lagrangian cost changes dramatically. Last, although we compute the overall Lagrangian cost for the virtually concatenated list, blocks cannot be merged across the boundary of the actual posting list. Since the majority of the posting lists have only a few blocks, the recurrence can quickly skip to another posting list once it encounters an endpoint.

In the recurrence (see Algorithm 1), we start by setting each \(A[i]\) as the accumulated Lagrangian cost of single blocks (Steps 1–3), then we refine entries of \(A\) from left to right (Steps 2–13). Each time we compute \(A[i]\) by identifying an index \(l^*\) with minimum 
\[ A[j] + c(l,i), \]
where \(c(l,i)\) is found by enumerating all the possible combinations backward from \(i - 1\) to \(i - h\). To further skip improbable combinations, we postulate that access counts between two adjacent single blocks cannot differ by a 10% margin, and the access count for the merged block is aligned to the maximal one inside it; second, the assigned encoders for the two adjacent single blocks must be the same, as for binary packing, the bit widths cannot differ by 2, or we can immediately mark an endpoint at the position of current base block (Steps 6–13). Thus, we are able to obtain the optimal combination of these single blocks after the execution.

The key to computing the Lagrangian cost of merged blocks is to efficiently predict their space and time costs. Formerly, we obtain the space cost for each single block exactly by encoding it and the time cost by a predictor using feature statistics. However, variable-sized blocks will take much more time to prepare these auxiliary information. If not limited by the compression time, we can calculate the costs for all the possible blocks offline before compression. Despite its inefficiency, plenty of these cost information also take up much memory to store. We prefer to calculate the costs of merged blocks on-the-fly, as most of the variable-sized blocks are barely to be taken into consideration. Based on the assigned encoders for these single blocks, their features are reused to predict their merged blocks rather than recomputation. Also, we can cache the last computed value and use it at the next call. While quite simple, this idea is essential in alleviating the delay caused by additional calculation in practice. Recall that the time cost of a base block is actually the decoding time multiplied by its access count, \(t_{l,j} = f_{l} t_{l,j}\). For merged blocks, we choose the maximal access count \((\max\{f_{l} | i \in \text{merged blocks}\})\) inside it as its final one. Thus, the merged block will always have larger Lagrangian cost than the sum of single ones. It is consistent with the fact that a longer block takes longer time to decode and to locate one specified element when processing user queries. An excessive number of merged blocks may lead to a decrease in query processing performance. Since the procedure is invoked only when building a path, it adds a multiplicative factor \(h\) to the whole time and space complexity, namely \(O(n h k \log(n s_{cost} ^l c_{cost} ^l))\) time and \(O(n h k)\) space.

### 6 Experiments

In order to provide an experimental evaluation of the proposed compression technique, we have implemented our algorithms, which we call bicriteria, in C++. More precisely, the one that uses fixed-sized blocks is called bicrtF, and that uses variable-sized blocks is called bicrtV. All the

---

**Algorithm 1: Algorithm used to generate a path using variable-sized blocks via dynamic programming**

**Input:** \(A[n], h, \mu,\)

base blocks \(e[n],\)

functions to calculate Lagrangian cost \(f(), g(),\)

endpoints of every actual posting list \(ep[f_i]\)

**Output:** Vector of optimal Partition \(P\)

1. \(A[1] = 0; p[1] = 1; q = 1;\)

2. for \(i = 2: i <= n + 1; i = i + 1\) do

3. \(A[i] = A[i - 1] + f(e[i - 1]) + \mu g(e[i - 1]);\)

4. if \(i == ep[q]\) then

5. \(q++:\)

6. for \(l = i - 1: l >= \max(ep[q], i - h); l = l - 1\) do

7. if \(e[l] \) and \(e[l + 1]\) have similar access counts and encoders then

8. \(c[l,i] = f(e[l,i]) + \mu g(e[l,i]);\)

9. if \(A[l] + c[l,i] < A[i]\) then

10. \(A[i] = A[l] + c[l,i];\)

11. \(p[i] = l - 1;\)

12. else

13. \(p[i] = l; break;\)

14. \(P = \{p[n + 1], p[p[n + 1]] . p[p[p[n + 1]]] . \ldots\};\)

15. reverse the order of \(P;\)

16. return \(P;\)
codes related to experiments are available at GitHub (https://github.com/Sparklexs/ds2i/), based on the implementation of Ottaviano[34]. In particular, we would like to compare its efficiency and compression effectiveness with the original method called hybrid in Ref. [8], so we retain most configurations of their experiments to avoid interference from other factors.

6.1 Experimental setup

We choose the same encoder set $E$ used in Ref. [8], namely Interpolative, PFD($h$), and Varint-G8IU. Thus our method can be compared with theirs in a straightforward manner without introducing additional errors. The objective of this experiment is to validate the efficiency of our encoder-allocating algorithm, it is decoupled from any specified encoders, as long as they offer a wide span of space-time efficiency. It is still worth mentioning that SIMD-BP128 is much preferable to Varint-G8IU as shown in Fig. 1. We also note that $h$ stands for exception bit length, and PFDs with different $h$ are treated as different encoders. Blocks with less than 128 postings are always compressed by Interpolative.

For each base block and encoder, we determine the space occupancy by encoding it in real world and its decoding time through a linear predictor. This method works well in the preliminary experiment. For blocks using variable sizes, we adopt the same manner to avoid excessive additional computation during the optimization. To be more specific, within the three encoders, Interpolative always bisects a block and compresses each half independently, Varint-G8IU is an integer-oriented encoder whose compressed size is irrelevant to the block length, compressed size of merged blocks using these two encoders can be summed up directly. Compressed size of encoder PFD($h$) has something to do with the exceptions inside it and we have to recompress the merged block rather than simply adding up. The same goes for time prediction of the merged blocks, we only recompute the decoding time for PFD($h$) due to the variation of features.

We use the posting lists extracted from the following two collections: TREC GOV2 and Common Crawl. TREC GOV2 is a crawl of the .gov sites used in TREC 2004 Terabyte Track, which consists of 25.2 million documents and 15.3 million terms. Common Crawl is a corpus of Web crawl data composed of over 5 billion web pages over last 7 years, and it continues to grow. The crawl data is stored using WARC 1.0 format with size of approximately 541 TB in size. This whole collection is freely available on Amazon S3 (http://commoncrawl.org/).

Since the Common Crawl corpus is too large to fit in one machine, we only extract a small part of the data crawled in February 2016. Documents from these two collections are prepared by applying Porter stemmer after removing stopwords, then the docids are reordered by the lexicographic order of URLs. Table 1 compares these two collections using some basic statistics.

In Table 1, we can find the trend that document length grows with age, making documents involved in more posting lists. However, Common Crawl is less organized than GOV2, since the former is collected from the whole web while the latter is limited in .gov domain, then terms in GOV2 are more repetitive and posting list length is larger. When split into blocks of size 128, each list is likely to contain a fraction that is less than 128. These fractions are called partial blocks and can only be compressed using Interpolative. Partial blocks account for a large proportion of the total number of blocks; specifically, they are 30% on GOV2 and 43% on Common Crawl. Next we will reveal how these differences affect performances of index.

All the implementations are conducted on a PC server with an 8 core Intel(r) Xeon(r) E5620 processor running at 2.40 GHz, with 32 GB RAM and 12 288 KB cache. Our algorithm is compiled with GCC 4.8.1 with -O3 optimizations. In all our runs, executions are reported as the mean of 4 consecutive runs.

The query set $Q$ we used is a realistic query log released by AOL[33]. It contains about 20 million Web queries collected from 650 000 users over three months. We first execute all the queries using processing strategies that will be used in experiment to count up the access statistics $f_i$ of each block, then add 1 to each access count to avoid blocks that are never accessed being overlooked. Then we randomly pick 10 000 queries, which contains at least 2 terms present in the posting lists, to examine query efficiency.

| Table 1 Collection statistics for GOV2 and Common Crawl. |
|-----------------------------------------------------------|
| Number of documents | Number of terms | Number of pointers | Document length (avg.) | Posting list length (avg.) |
|---------------------|-----------------|--------------------|------------------------|---------------------------|
| GOV2                | 25 203 921      | 15 324 160         | 4 655 778 182          | 645.25                    | 303.82                    |
| Common Crawl        | 17 157 948      | 51 990 893         | 9 118 361 737          | 2151.83                   | 173.38                    |
of the built index. Before performing queries, the whole inverted index is completely loaded into main memory, in order to avoid the confusion caused by disk I/O.

6.2 Indexing performance

Construction time We have analyzed our algorithms in detail in the previous section. For bicrtF, its complexity is $O(\bar{n} \log(n s_{costs} t_{costs}))$ in time and $O(\bar{n})$ in space, almost equalled by hybrid which is $O(\bar{n} \log(\bar{n}))$ in time and $O(\bar{n})$ in space. bicrtV adds a multiplicative factor $h$ to time and space complexity. Both our methods and hybrid concatenate all the posting lists into one to arrange encodings of each block, and are delayed by one time-consuming phase that cannot be paralleled, for bicrtF and bicrtV is the iteration when solving the dual problem, for hybrid is the sorting of $\lambda_{i,j}$.

However, in practice, bicrtF compresses more efficiently than hybrid. The average time used on GOV2 is 22 minutes, and on Common Crawl is 50 minutes. In contrast, hybrid costs 30 and 71 minutes, respectively. For bicrtV, the time increases considerably to 80 and 152 minutes due to the inefficient dynamic programming. Note that time complexity of the iteration is measured in the worst case scenario. In our experiment, the algorithm used to determine the optimal path converges rapidly after few iterations. Actually, we can further improve the compression efficiency by amortizing the expected budget over each posting list rather than concatenating all the posting lists together, at the cost of losing the approximation guarantee. Then we can calculate the optimal path and compress blocks for each posting list separately, and the whole phases of bicriteria can be accelerated by parallel.

Trading off space and time To demonstrate that bicriteria is able to flexibly trade off compressed size and query processing time, we perform two series of experiments by specifying space and time as budget. The space budget is obviously the compressed size, as for the time budget, we set it to be the sum of expected query processing time for all the queries in $Q$. The budget is represented using a compression level $l = \frac{B - B_0}{B_{\infty} - B_0}$, where $(B_0, B_{\infty})$ is the minimum and maximum space/time that the index can achieve. For example, if $B = S$, then $B_0$ is the most succinct space optimal compression with $l = 0$, while $B_{\infty}$ is the fastest compression with $l = 1$.

First we discuss the percentages of encoder used by bicrtF under different budgets. As shown in Fig. 4a, by linearly relaxing the budget in space, bicrtF trends to choose more fast encoders. At first more than 90% blocks are compressed using Interpolative, then more PFD-compressed blocks are involved and nearly all of them are compressed using Varint-G8IU. The result agrees with that of Ottaviano et al. [8], however, their space-time curve is more smooth because when the space budget changes, hybrid executes a sequential scan to replace encoders right starting from the last block it stopped. bicrtF treats all the posting list indiscriminately, the budget is amortized to all the blocks since it relies on path finding rather than sorting, while hybrid is a greedy algorithm, it always prefers blocks which are more frequently accessed.

Figure 4b shows how space changes when we linearly tighten time budget ($T$ varies from 1 to 0). It is quite different from Fig. 4a, as Interpolative is the preferred choice in most cases, even when $l = 0.5$, there are still nearly 50% blocks using Interpolative. This is caused by the fact that the average length of posting list in Common Crawl is 173.38, a large number of
partial blocks leads to the use of Interpolative. Also, Interpolative is much slower than other encoders, a small decrease of its proportion leads to a much reduce in expected query processing time. The changed blocks are scattered around the whole index, resulting in the shortened average time for all the lists. Thus, the improvement is quite implicit in real scenario. However, we can use it to find the proper point where the average decoding time is sharply reduced while its space occupancy keeps low. For example, we can choose \( l = 0.5 \) in expectation of a 50% drop in query time with only an 11% increase in size.

Then we use a heat map to exhibit the density distribution of block sizes and encoders when using \( \text{bicrtV} \), as shown in Fig. 5. Due to the limited space and similarity of neighboring results, we only show part of them with an interval of 0.2. Areas with gray color are blocks which never appear in the index, otherwise, the darker the area, the more blocks choose the corresponding size and encoder. Note the number is represented in logarithmic scales. The tendency nearly stays the same with \( \text{bicrtF} \) under both space and time as budget, when the budget is biased toward a smaller space, we can see the dark area shift left to the encoders which offer better compression; on the other hand, when we loose the budget to faster query speed, the dark area begins to shift right. Longer block sizes are less likely to appear for low possibility of similar consecutive base blocks. Also, when setting time as budget, Interpolative barely loses its popularity due to its highly-rated space-time ratio. However, we can see proportions of other encoders growing evidently as the budget decreases from 1 to 0.

At each end of the subfigure, \( \text{bicrtV} \) gets the same results as they represent the space-optimal and time-optimal solutions. The most noticeable difference with \( \text{bicrtF} \) is that Varint-G8IU occupies a larger proportion in all the cases. This condition can be attributed to the emergence of too many exceptions inside one block, where all the encoders may perform poorly and encoders that offer faster decoding speed are preferable. And blocks of docid and their corresponding blocks of frequency are interactive, namely, a merging of docid blocks will lead to a mandatory merging of frequency blocks and further affect other blocks in dynamic programming. Thus, distribution of encoders in \( \text{bicrtV} \) diverges from that in \( \text{bicrtF} \). Another difference is when varying space budget \( S \) from 0 to 1, PFD occupies a greater proportion than the other two encoders. As we set the access count of a merged block to be the maximal one inside it, the merged blocks are likely to have larger querying time than single ones. And among these three encoders, PFD can better adjust its parameters to maintain high efficiency while other two only perform linear combination. We can also find \( \text{bicrtV} \) is discreet because it tends to choose blocks with smaller sizes for all the encoders.

Tables 2 and 3 report the indexing performances of \( \text{bicriteria} \) under various budgets in comparison with \( \text{hybrid} \). Results are classified by “method@ [compression level]@[budget type]”. Space denotes the index size in gigabytes and time denotes the expected average time cost in nanoseconds of decoding one 128-sized block. The expected decoding time cost is obtained using \( \sum_{i=1}^{n} f_i t_i / \sum_{i=1}^{n} f_i \), taking into account the impact of frequently accessed blocks.

Generally, \( \text{bicriteria} \) is able to find the exact solution under different budgets. As shown in the space column...
### Table 2: Comparison of index size in gigabytes and expected average decoding time per block in nanoseconds of bicriteria and hybrid under space budgets on GOV2 and Common Crawl.

| Dataset         | Method | Space (GB) | Time (ns) |
|-----------------|--------|------------|-----------|
| GOV2            | bicrtF@0S | 3.05       | 3026      |
|                 | bicrtF@0.1S | 3.29       | 405       |
|                 | bicrtF@0.2S | 3.52       | 222       |
|                 | bicrtF@0.3S | 3.76       | 173       |
|                 | bicrtF@0.4S | 4.00       | 156       |
|                 | bicrtF@0.5S | 4.24       | 148       |
|                 | bicrtF@0.6S | 4.48       | 145       |
|                 | bicrtF@0.8S | 4.95       | 144       |
|                 | bicrtF@0.9S | 5.20       | 143       |
|                 | bicrtF@1S   | 5.43       | 124       |
|                 | bicrtV@0S   | 3.05       | 3026      |
|                 | bicrtV@0.2S | 3.52       | 200       |
|                 | bicrtV@0.3S | 3.76       | 152       |
|                 | bicrtV@0.4S | 4.00       | 136       |
|                 | bicrtV@0.5S | 4.24       | 129       |
|                 | bicrtV@0.6S | 4.48       | 126       |
|                 | bicrtV@0.7S | 4.72       | 125       |
|                 | bicrtV@0.8S | 4.95       | 125       |
|                 | bicrtV@0.9S | 5.20       | 124       |
|                 | bicrtV@1S   | 5.43       | 124       |
| Common Crawl    | bicrtF@0S   | 9.38       | 4092      |
|                 | bicrtF@0.1S | 10.06      | 411       |
|                 | bicrtF@0.2S | 10.73      | 225       |
|                 | bicrtF@0.3S | 11.41      | 180       |
|                 | bicrtF@0.4S | 12.09      | 158       |
|                 | bicrtF@0.5S | 12.76      | 149       |
|                 | bicrtF@0.6S | 13.44      | 148       |
|                 | bicrtF@0.7S | 14.11      | 147       |
|                 | bicrtF@0.8S | 14.79      | 146       |
|                 | bicrtF@0.9S | 15.46      | 146       |
|                 | bicrtF@1S   | 16.14      | 146       |
|                 | bicrtV@0S   | 13.44      | 147       |
|                 | bicrtV@0.2S | 14.11      | 146       |
|                 | bicrtV@0.3S | 14.79      | 146       |
|                 | bicrtV@0.4S | 15.46      | 146       |
|                 | bicrtV@0.5S | 16.14      | 146       |
|                 | bicrtV@0.6S | 16.14      | 146       |
|                 | bicrtV@0.7S | 16.14      | 146       |
|                 | bicrtV@0.8S | 16.14      | 146       |
|                 | bicrtV@0.9S | 16.14      | 146       |
|                 | bicrtV@1S   | 16.14      | 146       |

### Table 3: Comparison of index size in gigabytes and expected average decoding time per block in nanoseconds of bicriteria and hybrid under time budgets on GOV2 and Common Crawl.

| Dataset         | Method | Space (GB) | Time (ns) |
|-----------------|--------|------------|-----------|
| GOV2            | bicrtF@0T | 5.43       | 124       |
|                 | bicrtF@0.1T | 5.36       | 415       |
|                 | bicrtF@0.2T | 3.15       | 705       |
|                 | bicrtF@0.3T | 3.11       | 905       |
|                 | bicrtF@0.4T | 3.08       | 1285      |
|                 | bicrtF@0.5T | 3.06       | 1575      |
|                 | bicrtF@0.6T | 3.05       | 1865      |
|                 | bicrtF@0.8T | 3.05       | 2445      |
|                 | bicrtF@0.9T | 3.05       | 2735      |
|                 | bicrtF@1T   | 3.05       | 3026      |
|                 | bicrtV@0T   | 5.43       | 124       |
|                 | bicrtV@0.1T | 5.36       | 415       |
|                 | bicrtV@0.2T | 3.15       | 705       |
|                 | bicrtV@0.3T | 3.11       | 905       |
|                 | bicrtV@0.4T | 3.08       | 1285      |
|                 | bicrtV@0.5T | 3.06       | 1575      |
|                 | bicrtV@0.6T | 3.05       | 1865      |
|                 | bicrtV@0.8T | 3.05       | 2445      |
|                 | bicrtV@0.9T | 3.05       | 2735      |
|                 | bicrtV@1T   | 3.05       | 3026      |
| Common Crawl    | bicrtF@0T   | 16.14      | 146       |
|                 | bicrtF@0.1T | 9.85       | 541       |
|                 | bicrtF@0.2T | 9.56       | 942       |
|                 | bicrtF@0.3T | 9.45       | 1332      |
|                 | bicrtF@0.4T | 9.40       | 1734      |
|                 | bicrtF@0.5T | 9.39       | 2119      |
|                 | bicrtF@0.6T | 9.39       | 2520      |
|                 | bicrtF@0.7T | 9.39       | 2968      |
|                 | bicrtF@0.8T | 9.38       | 3316      |
|                 | bicrtF@0.9T | 9.38       | 3727      |
|                 | bicrtF@1T   | 9.38       | 4092      |
|                 | bicrtV@0T   | 16.14      | 146       |
|                 | bicrtV@0.1T | 9.82       | 541       |
|                 | bicrtV@0.2T | 9.55       | 2182      |
|                 | bicrtV@0.3T | 9.44       | 1332      |
|                 | bicrtV@0.4T | 9.39       | 1734      |
|                 | bicrtV@0.5T | 9.39       | 2119      |
|                 | bicrtV@0.6T | 9.39       | 2520      |
|                 | bicrtV@0.7T | 9.39       | 2968      |
|                 | bicrtV@0.8T | 9.38       | 3316      |
|                 | bicrtV@0.9T | 9.38       | 3727      |
|                 | bicrtV@1T   | 9.38       | 4092      |
|                 | bicrtV@0T   | 16.14      | 146       |
|                 | bicrtV@0.1T | 9.82       | 541       |
|                 | bicrtV@0.2T | 9.55       | 2182      |
|                 | bicrtV@0.3T | 9.44       | 1332      |
|                 | bicrtV@0.4T | 9.39       | 1734      |
|                 | bicrtV@0.5T | 9.39       | 2119      |
|                 | bicrtV@0.6T | 9.39       | 2520      |
|                 | bicrtV@0.7T | 9.39       | 2968      |
|                 | bicrtV@0.8T | 9.38       | 3316      |
|                 | bicrtV@0.9T | 9.38       | 3727      |
|                 | bicrtV@1T   | 9.38       | 4092      |

(GB) (ns)
of Table 2 and the time column of Table 3, bicrtF and bicrtV achieve the same space occupancy and decoding time as that of hybrid on both GOV2 and Common Crawl. However, there exist some differences in Table 2. Given the same space budget, the expected time costs of each methods are quite different from each other. We can see that hybrid always has a smaller cost than bicrtF, except under two extreme cases where they all adopt the same optimal solution. Note this phenomenon is caused by the internal mechanism of two different methods. Given extra space, hybrid prefers to update the most frequently accessed blocks of the remaining ones, this will have a direct effect on the decrease of expected time. However, bicrtF tries to share the space with as more blocks as possible, because when we generate a path in the first phase, the edge-selection for each block is independent. Thus resulting a larger average cost than hybrid. As expected, bicrtV achieves the smallest expected decoding time. The reason can be two fold, first, the time-consuming dynamic programming grants bicrtV the ability to find a better combination of block sizes and encoders; second, a larger block size reduces the total number of blocks to be decoded and decoding the merged blocks in one time is naturally faster than decoding them separately.

The result is also consistent with the percentage shown in Figs. 4a and 5a. Compared with hybrid, bicrtF and bicrtV initially use more Interpolative. And in the middle of the figure, PFD accounts for a larger proportion than that of hybrid. In a word, our method focuses on improving the average performance of query processing while hybrid focuses on guaranteeing performance of the most frequently used queries.

Table 3 reports the result of bicriteria and hybrid using time as budget. Different from Table 2, where decoding times differ from each other obviously, here the spaces of the three methods are almost the same except bicrtV is a little smaller at first. However, as we request a longer decoding time, space occupancies of all the methods quickly shrink to their lowest levels.

We can observe the space slowly grows as we tighten the time budget from 1 to 0.1, because 10% growth in space can sharply reduce the expected time cost according to Table 2. Also, the expected time cost can hardly be satisfied since it is an averaged value. From Fig. 4b, we can see most of the blocks are compressed using Interpolative. A subdivision between 0T and 0.1T can be used to find more practical tradeoffs, but there needs a careful measurement of the connection between the expected time budget and actual query processing time.

### 6.3 Query efficiency

In order to explore the efficiency differences between bicriteria and hybrid, we adopt three widely-used methods of ranked query to find the top-20 results, Ranked AND, WAND, and Maxscore. Due to the limited space, we only report partial results (where the space budget grows from 0 to 1 by 0.2) on Common Crawl, and the omitted part and result on GOV2 exhibit similar performance except that query time on GOV2 gets an overall decline because of smaller index size. We also omit the performances of single encoders, as our goal is only to prove that bicriteria obtains a better average processing time in practice than hybrid by equally treating all the posting lists rather than the few frequently accessed ones.

As shown in Fig. 6, we can see that all the methods get similar query efficiency under various scenarios: except two extreme cases where these three methods can hardly differ from each other, bicriteria always

![Figure 6 Query time distribution of bicriteria and hybrid using different processing strategies on Common Crawl, the budget varies from 0 to 1. The average query time is represented by a + sign.](image)

Fig. 6 Query time distribution of bicriteria and hybrid using different processing strategies on Common Crawl, the budget varies from 0 to 1. The average query time is represented by a + sign.
outperforms hybrid and bicrV always outperforms bicrF. Even though Table 2 reports hybrid has an expected processing time between bicrF and bicrV, practical experiment shows a quite different result, bicriteria works better in most cases, which proves bicriteria works better than prediction. From Tables 2 and 3, we know that a slight relaxation of space can sharply reduce the decoding time. When varying time budget from 0.1 to 0.9, the space range can be included in the space budget from 0 to 0.1. Accordingly, the result in Fig. 6b from 0.2T to 1T can be seen as a detailed change from 0.1S to 0S of that in Fig. 6a. Another noticeable phenomenon is that performance gaps get blurred when the budget increases. The reasons are twofold. First, both methods adopt more proportion of fast encoders, the different blocks begin to reduce. Second, limited timing accuracy makes it harder to compare these slight differences.

Among all three processing strategies, the most evident performance gap occurs in Ranked AND, and the other two query methods stay relatively stable. Ranked AND involves more skip operations to locate the documents that contain all the query terms, while WAND and Maxscore work more like disjunctive query and evaluate more documents than Ranked AND. The reason that bicriteria outperforms hybrid can be attributed to the fact that hybrid prefers optimizing those frequently accessed blocks inside the whole index, if one posting list has few such blocks or the entire list is rarely accessed, then the processing time will be extended because of too many “slow” blocks. However, encoders assigned to blocks using bicriteria is more evenly distributed. As Ranked AND processes queries by traversing all the blocks inside the posting lists, the difference is more apparent. We can also find gaps caused by WAND and MaxScore are less vulnerable to this problem because these two strategies are more likely to decode those frequently accessed blocks.

The expected query times given in Tables 2 and 3 are reported under a priori query set and a given processing strategy. They are calculated after we obtain exact statistics on the access times of each blocks. However, the random queries used in this section are impossible to predict which part of the index would be fetched. When using time (specifically, the predicted average query time) as a budget, a performance gap naturally exists between the expected time cost and the actual one. Namely, the practical performance is always less effective than expected since its goal is to find the minimal index size that satisfies time budget. The result relies heavily on the accuracy of query prediction algorithms. However, this would be another complicated subject beyond the topic of this essay to predict query time of unknown queries. In the future work, we would like to tune the time budget to a more fine-grained level to bridge the gap between predicted times and actual ones.

7 Conclusion and Future Work

In this paper we have introduced a bicriteria optimization on building hybrid index. Compared with its original version, our method is also able to yield a flexible space-time tradeoff under a given budget, while preserving the same additive approximation guarantees. Moreover, it supports regarding either time or space as budget and is able to be extended using variable-sized blocks. Our method is more theoretically well-founded, and the experiment has demonstrated bicriteria gains smaller construction time and average query processing time.

Since our solution only focuses on modeling methodology rather than implementation, we do not consider integrating other encoders into the method. Furthermore, a better tuned measurement is necessary to connect the expected budget and practical performance when using time as budget. Finally, with respect to variable-sized blocks, we only considered merging in our paper, it would be worthwhile to split blocks into more fine-grained ones, even randomized-sized ones for better compression.

Acknowledgment

Our implementation is based on the codes provided by Giuseppe Ottaviano. We are grateful to the anonymous reviewers for their valuable comments. We thank the Natural Science Foundation of Hunan Province (No. 2016JJ2007) for their financial support.

References

[1] M. Catena, C. Macdonald, and I. Ounis, On inverted index compression for search engine efficiency, in *Advances in Information Retrieval*, M. de Rijke, T. Kenter, A. P. de Vries, C. X. Zhai, F. de Jong, K. Radinsky, and K. Hofmann, eds. Springer, 2014, pp. 359–371.

[2] A. Trotman, Compression, SIMD, and postings lists, in *Proc. 2014 Australasian Document Computing Symp.*, Melbourne, Australia, 2014, p. 50.

[3] J. Zobel and A. Moffat, Inverted files for text search
[4] D. Lemire and L. Boytsov, Decoding billions of integers per second through vectorization. *Softw.: Pract. Exp.*, vol. 45, no. 1, pp. 1–29, 2015.

[5] D. Lemire, L. Boytsov, and N. Kurz, SIMD compression and the intersection of sorted integers. *Softw.: Pract. Exp.*, vol. 46, no. 6, pp. 723–749, 2015.

[6] W. X. Zhao, X. D. Zhang, D. Lemire, D. D. Shan, J. Y. Nie, H. F. Yan, and J. R. Wen. A general SIMD-based approach to accelerating compression algorithms. *ACM Trans. Inf. Syst. (TOIS)*, vol. 33, no. 3, p. 15, 2015.

[7] D. Lemire, N. Kurz, and C. Rupp. Stream VByte: Faster byte-oriented integer compression. *Inf. Process. Lett.*, vol. 130, pp. 1–6, 2018.

[8] G. Ottaviano, N. Tonellotto, and R. Venturini. Optimal space-time tradeoffs for inverted indexes. in *Proc. 8th ACM Int. Conf. Web Search and Data Mining*, Shanghai, China, 2015, pp. 47–56.

[9] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. in *Proc. 37th Int. ACM SIGIR Conf. Research & Development in Information Retrieval*, Gold Coast, Australia, 2014, pp. 273–282.

[10] M. Petri, A. Moffat, and J. S. Culpepper. Score-safe term-dependency processing with hybrid indexes. in *Proc. 37th Int. ACM SIGIR Conf. Research & Development in Information Retrieval*, Gold Coast, Australia, 2014, pp. 899–902.

[11] F. Silvestri and R. Venturini. VSEncoding: Efficient coding and fast decoding of integer lists via dynamic programming. in *Proc. 19th ACM Int. Conf. Information and Knowledge Management*, Toronto, Canada, 2010, pp. 1219–1228.

[12] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, vol. 3, no. 1, pp. 25–47, 2000.

[13] J. Lin, M. Crane, A. Trotman, J. Callan, I. Chattopadhyaya, J. Foley, G. Ingersoll, C. Macdonald, and S. Vigna. Toward reproducible baselines: The open-source IR reproducibility challenge. in *Proc. 38th European Conf. IR Research*, Padua, Italy, 2016, pp. 408–420.

[14] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. in *Proc. 18th Int. Conf. World Wide Web*, Madrid, Spain, 2009, p. 401.

[15] A. Furruglia, P. Ferragina, A. Frangioni, and R. Venturini. Bicriteria data compression. in *Proc. 25th Ann. ACM-SIAM Symp. Discrete Algorithms*, Portland, OR, USA, 2014, pp. 1582–1595.

[16] K. Mehlhorn and M. Ziegelmann. Resource constrained shortest paths. in *Proc. 8th Ann. European Symp. Algorithms*, Saarbrücken, Germany, 2000, pp. 326–337.

[17] G. Y. Handler and I. Zang. A dual algorithm for the constrained shortest path problem. *Networks*, vol. 10, no. 4, pp. 293–309, 1980.

[18] S. Ding and T. Suel. Faster top-K document retrieval using block-max indexes. in *Proc. 34th Int. ACM SIGIR Conf.* Research and Development in Information Retrieval, Beijing, China, 2011, pp. 993–1002.

[19] R. Konow, G. Navarro, C. L. A. Clarke, and A. López-Ortiz. Faster and smaller inverted indices with treaps. in *Proc. 36th Int. ACM SIGIR Conf. Research and Development in Information Retrieval*, Dublin, Ireland, 2013, pp. 193–202.

[20] G. Navarro and S. J. Puglisi. Dual-sorted inverted lists. in *String Processing and Information Retrieval*, E. Chavez and S. Lonardi, eds. Springer, 2010, pp. 309–321.

[21] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. O’beroi. SIMD-based decoding of posting lists. in *Proc. 20th ACM Int. Conf. Information and Knowledge Management*, Glasgow, UK, 2011, pp. 317–326.

[22] R. Debru, S. Campinas, K. Samp, and G. Tummarrello. Adaptive frame of reference for compressing inverted lists. Technical Report, DERI-Digital Enterprise Research Institute, Ireland, 2010.

[23] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. in *Proc. 14th Int. Conf. Data Engineering*, Orlando, FL, USA, 1998, pp. 370–379.

[24] V. N. Anh and A. Moffat. Index compression using fixed binary codewords. in *Proc. 13th Australasian Database Conf.-Volume 27*, Dunedin, New Zealand, 2004, pp. 61–67.

[25] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, vol. 8, no. 1, pp. 151–166, 2005.

[26] V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 6, pp. 857–861, 2006.

[27] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Softw.: Pract. Exp.*, vol. 40, no. 2, pp. 131–147, 2010.

[28] R. Debru, S. Campinas, and G. Tummarrello. Searching web data: An entity retrieval and high-performance indexing model. *Web Semant.*, vol. 10, pp. 33–58, 2012.

[29] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. in *Proc. 22nd Int. Conf. Data Engineering*, Atlanta, GA, USA, 2006, p. 59.

[30] P. Ferragina, I. Nito, and R. Venturini. On optimally partitioning a text to improve its compression. *Algorithmica*, vol. 61, no. 1, pp. 51–74, 2011.

[31] X. S. Song, K. Jiang, Y. Jiang, and Y. X. Yang. On optimizing partitioning strategies for faster inverted index compression. in *Proc. 16th Int. Conf. Computational Science and Its Applications*, Beijing, China, 2016, pp. 246–260.

[32] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. New York, NY, USA: Courier Corporation, 2001.

[33] G. Pass, A. Chowdhury, and C. Torgeson. A picture of search. in *Proc. 1st Int. Conf. Scalable Information Systems*, Hong Kong, China, 2006, p. 1.

[34] Giuseppe Ottaviano. https://github.com/ot/ds2i.git, 2017.
Xingshen Song received the PhD degree in computer science from National University of Defense Technology (NUDT), China in 2018. Currently, he is an assistant research fellow of the School of Advanced Interdisciplinary Studies, NUDT. His research interests include data structures for search engine, inverted index compression, and query processing optimization.

Yuexiang Yang received the PhD degree in computer science from National University of Defense Technology (NUDT), China in 2008. Currently, he is a professor of the College of Computer, NUDT. His main research fields include information retrieval, information security, and cloud computing.

Yu Jiang is currently a master student at the College of Computer, NUDT. She received the BE degree from Xi’an Jiaotong University, China in 2010. Her research interests include query processing optimization and data structures for search engine.