Efficient In-Memory Top-k Document Retrieval

J. Shane Culpepper
School of CS & IT
RMIT University
Melbourne, VIC, 3001, Australia
shane.culpepper@rmit.edu.au

Matthias Petri
School of CS & IT
RMIT University
Melbourne, VIC, 3001, Australia
matthias.petri@rmit.edu.au

Falk Scholer
School of CS & IT
RMIT University
Melbourne, VIC, 3001, Australia
falk.scholer@rmit.edu.au

ABSTRACT

For over forty years the dominant data structure for ranked document retrieval has been the inverted index. Inverted indexes are effective for a variety of document retrieval tasks, and particularly efficient for large data collection scenarios that require disk access and storage. However, many efficiency-bound search tasks can now easily be supported entirely in-memory as a result of recent hardware advances.

In this paper we present a hybrid algorithmic framework for in-memory bag-of-words ranked document retrieval using a self-index derived from the FM-Index, wavelet tree, and the compressed suffix tree data structures, and evaluate the various algorithmic trade-offs for performing efficient queries entirely in-memory. We compare our approach with two classic approaches to bag-of-words queries using inverted indexes, *term-at-a-time* (**TAAT**) and *document-at-a-time* (**DAAT**) query processing. We show that our framework is competitive with state-of-the-art indexing structures, and describe new capabilities provided by our algorithms that can be leveraged by future systems to improve effectiveness and efficiency for a variety of fundamental search operations.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*indexing methods*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*file organization*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*query formulation, retrieval models, search process*; I.7.3 [**Document and Text Processing**]: Text Processing—*index generation*

Keywords

Text Indexing, Text Compression, Data Storage Representations, Experimentation, Measurement, Performance

1. INTRODUCTION

Top-k retrieval algorithms are important for a variety of real world applications, including web search, on-line advertising, relational databases, and data mining. Efficiently ranking answers to queries in large data collections continues to challenge researchers as the collection sizes grow, and the ranking metrics become more intricate. Despite recent hardware advances, inverted indexes remain the tool of choice for processing efficiency-bound search tasks [17]. However, large memory systems also provide new opportunities to explore another class of indexing algorithms derived from the *suffix array* to potentially improve the efficiency of various in-memory ranked document retrieval tasks [25, 27].

In this paper we present a hybrid algorithmic framework for in-memory bag-of-words ranked document retrieval using a self-index derived from the *FM-Index*, *wavelet tree*, and the *compressed suffix tree* data structures [12, 20, 27, 22], and evaluate the various algorithmic trade-offs for performing efficient in-memory ranked querying. We compare our approach with two classic approaches to bag-of-words queries using inverted indexes, *term-at-a-time* (**TAAT**) and *document-at-a-time* (**DAAT**) query processing. We show that our framework is competitive with state-of-the-art indexing structures, and describe new capabilities provided by our algorithms that can be leveraged by future systems to improve efficiency and effectiveness for various document retrieval tasks.

Our contributions. Firstly, we propose a hybrid approach to solving a subset of important top-k document retrieval problems – *bag-of-words* queries. Secondly, we present a comprehensive efficiency analysis comparing in-memory inverted indexes with top-k self-indexing algorithms for bag-of-words queries on text collections an order of magnitude larger than any other prior experimental study. To our knowledge, this is the first comparison of this new algorithmic framework for realistic sized text collections using a standard similarity metric – **BM25**. Finally, we describe how our algorithmic framework can be extended to efficiently and effectively support other fundamental document retrieval tasks.

2. PROBLEM OVERVIEW

In this paper, we investigate the use of self-indexing algorithms to solve the **top-k document search problem**. A document collection $\mathcal{T}$ is a contiguous string drawn from an alphabet Σ, where $\sigma = |\Sigma|$ is the number of distinct "terms" or strings. In practice, Σ can be characters (UTF8 or ASCII), bytes, integers, or even phrases. Each document in $\mathcal{T}$ is separated by a unique end of document symbol defined to be lexicographically smaller than any $s \in \Sigma$.

DEFINITION 1. *A* **top-k document search** *takes a query $q \in \Sigma$, an integer $0 < k \leq d$, and a text $\mathcal{T} \in \Sigma$ partitioned into d documents $\{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_d\}$, and returns the top-$k$ documents ordered by a similarity measure $S(q, \mathcal{D}_i)$.*

In this work, we focus primarily on bag-of-words queries, so our baseline $S(q, \mathcal{D}_i)$ ranking function is **BM25**. Our $S(q, \mathcal{D}_i)$ ranking function has the following formulation:

$$\textbf{BM25} = \sum_{t \in q} \log \left(\frac{N - f_t + 0.5}{f_t + 0.5} \right) \cdot \text{TF}_{\textbf{BM25}}$$

$$\text{TF}_{\textbf{BM25}} = \frac{f_{d,t} \cdot (k_1 + 1)}{f_{d,t} + k_1 \cdot ((1 - b) + (b \cdot \ell_d / \ell_{avg}))}$$

Here, N is the number of documents in the collection, f_t is the number of distinct document appearances of t, $f_{d,t}$ is the number of occurrences of term t in document d, $k_1 = 1.2$, $b = 0.75$, ℓ_d is the number of symbols in the dth document, and ℓ_{avg} is the average of ℓ_d over the whole collection. The free parameters k_1 and b can be tuned for specific collections to improve effectiveness, but we use the standard Okapi parameters suggested by Robertson et al. [36].

3. ALGORITHMS

We now present an overview of the key data structures and algorithms used in our framework. Here, we only outline the key properties and features of wavelet trees and suffix arrays used in our search engine; for a more in-depth tutorial, see, for example Navarro and Mäkinen [30] and the references therein. Efficient operations in succinct data structures depend on two fundamental operations over a bitvector $\mathcal{B}[0, n-1]$:

$\text{RANK}_{0/1}(\mathcal{B}, i)$:	Return the number of 0's/1's in $\mathcal{B}[0, i]$.
$\text{SELECT}_{0/1}(\mathcal{B}, i)$:	Return the position of the i^{th} of 0's/1's in $\mathcal{B}$.

Both operations can be performed in constant time. A simple constant time $\text{RANK}_{0/1}$ solution uses $o(n)$ space in addition to storing $\mathcal{B}$ [23]. More space efficient $\text{RANK}_{0/1}$ algorithms are possible [35].

3.1 Wavelet Trees

Efficient RANK_s and SELECT_s over an alphabet of size $\sigma > 2$ can be performed using a wavelet tree [20]. A wavelet tree decomposes the RANK_s and SELECT_s operations over $[0, \sigma - 1]$ into $\text{RANK}_{0/1}$ and $\text{SELECT}_{0/1}$ operations on a binary alphabet using a binary tree. The root of the tree represents the whole alphabet. Its children represent each half of the alphabet of the parent node. Each leaf node in the tree represents one symbol in $[0, \sigma - 1]$. When answering the RANK_s query for a specific symbol s, we perform $\text{RANK}_{0/1}$ operations at each level in the tree until we arrive at the leaf node representing s. The overall $\text{RANK}_s(\mathcal{T}, i)$ can be computed by combining the $\text{RANK}_{0/1}$ results at each tree level in $\mathcal{O}(\log \sigma)$ time. Any symbol $s = \mathcal{T}[i]$ is also computed in time $\mathcal{O}(\log \sigma)$ with a similar algorithm; we call this operation $\text{ACCESS}(\mathcal{T}, i)$. Using a succinct representation of $\text{RANK}_{0/1}$ and $\text{SELECT}_{0/1}$ [35], a wavelet tree requires $nH_0 + o(n \log \sigma)$ bits of space, where $H_0 \leq \log \sigma$ is the zero-order entropy of $\mathcal{T}$.[1]

Wavelet trees are a surprisingly versatile data structure, and have attractive time and space bounds for many primitive operations in self-indexing algorithms [14]. As a result, many top-k document retrieval approaches rely heavily on wavelet trees. A subset of important wavelet tree operations include:

$\text{RANK}_s(\mathcal{T}, s, sp, ep)$:	Return the number of occurrences of symbol s in a range $\mathcal{T}[sp, ep]$.
$\text{SELECT}_s(\mathcal{T}, s, j, sp, ep)$:	Return the position of the jth occurrence of symbol s in a range $\mathcal{T}[sp, ep]$.
$\text{ACCESS}(\mathcal{T}, i)$:	Return symbol $\mathcal{T}[i]$.
$\text{RMQ}(\mathcal{T}, sp, ep)$:	Return the smallest symbol s in a range $\mathcal{T}[sp, ep]$.
$\text{RQQ}(\mathcal{T}, k, sp, ep)$:	Return the kth smallest symbol s in a range $\mathcal{T}[sp, ep]$.

[1]We assume logarithms are in base 2.

3.2 Self-indexing

A suffix array $\text{SA}[0, n-1]$ over $\mathcal{T}$ stores the offsets to all suffixes in $\mathcal{T}$ in lexicographical order. Any pattern $\mathcal{P}$ of length m occurring in $\mathcal{T}$ is a prefix of one or more suffixes in SA. These suffixes, due to the lexicographical order within SA, are grouped together in a range $\text{SA}[sp, ep]$. To determine $\text{SA}[sp, ep]$, we perform two binary searches over SA and $\mathcal{T}$. Each binary search comparison requires up to m symbol comparisons in $\mathcal{T}$, for a total of $\mathcal{O}(m \log n)$ time. Using additional auxiliary data structures this cost can be reduced to $\mathcal{O}(m + \log n)$ [25]. Suffix array construction is a well studied problem, and many solutions with various time and space trade-offs exist in the literature [34]. However, searching for a pattern $\mathcal{P}$ in $\mathcal{T}$ using only a suffix array requires $\mathcal{O}(n \log n)$ bits to store both $\mathcal{T}$ and SA, which in practice is at least 5 times the text size.

By replacing $\mathcal{T}$ with the Burrows-Wheeler Transform (BWT) permuted text, the key operations of a basic SA can be emulated with much less space, close to the size of $\mathcal{T}$ in compressed form. The Burrows-Wheeler Transform [7] – also known as the block-sorting transform – produces a permutation of a string T, denoted T^{BWT}, by sorting the n cyclic rotations of T into full lexicographical order, and taking the last column of the resulting $n \times n$ matrix. The resulting string T^{BWT} tends to be more compressible as symbols are grouped together based on their context in $\mathcal{T}$, which makes the BWT an important part in many state of the art compression systems [26]. To produce T^{BWT} for a given text $\mathcal{T}$, it is not necessary to construct $\mathcal{M}$ as there is a duality between T^{BWT} and the SA over a text $\mathcal{T}$: $T^{\text{BWT}}[i] = \mathcal{T}[\text{SA}[i] - 1 \bmod n]$.

The original text $\mathcal{T}$ can be recovered from T^{BWT} in linear time without the need for any additional information. To recover $\mathcal{T}$ from only T^{BWT} we first recover the first column, F, in $\mathcal{M}$ by sorting the last column ($L = T^{\text{BWT}}$), in lexicographical order. By mapping the symbols in L to their respective positions in F so $L[i] = F[j]$ (usually referred to as the LF mapping, $j = \text{LF}(i)$) we can recover $\mathcal{T}$ backwards as $\mathcal{T}[n-1] = T^{\text{BWT}}[0] = \$$ and $\mathcal{T}[j-1] = T^{\text{BWT}}[\text{LF}(i)]$ if and only if $\mathcal{T}[j] = T^{\text{BWT}}[i]$. Since F is simply a sort of the n characters of the string in lexicographical order, it can be represented succinctly as a lookup table of alphabet characters along with the count of all symbols that appear before the current character c. The LF mapping is computed using the equation

$$\text{LF}(i) = \text{LF}(i, c) = C[c] + \text{RANK}_s(T^{\text{BWT}}, i) \qquad (1)$$

where c is the symbol $T^{\text{BWT}}[i]$, and $C[c]$ stores the number of symbols in T^{BWT} smaller than c.

Performing a search in $\mathcal{T}$ using the BWT permuted text is straightforward. Recall that all rows are sorted in lexicographical order in $\mathcal{M}$. Therefore, for a pattern $\mathcal{P}$, all occurrences of $\mathcal{P}$ in $\mathcal{T}$ must have a corresponding row in $\mathcal{M}$ within a range $\langle sp, ep \rangle$. To determine the range within $\mathcal{M}$, we first determine the range $\langle sp_m, ep_m \rangle$ within $\mathcal{M}$ that corresponds to $\mathcal{P}_m$ using $C[\,]$. Then, for each symbol $j = m-1 \ldots 0$ in $\mathcal{P}$, we iteratively find $\langle sp_j, ep_j \rangle$ by calculating the number of rows within $\langle sp_{j+1}, ep_{j+1} \rangle$ that are preceded by the symbol $\mathcal{P}_j$ in $\mathcal{T}$. For a given row j, the LF mapping can be used to determine the row in $\mathcal{M}$ representing the symbol preceding j in $\mathcal{T}$. The preceding row is determined by counting the number of occurrences of $c = T^{\text{BWT}}[j]$ *before* the current row and ranking these occurrences within $C[s]$. Assume we have located $\langle sp_{j+1}, ep_{j+1} \rangle$, which corresponds to the rows prefixed by $\mathcal{P}[j+1,m]$. Then

$$sp_j \;=\; \text{LF}(sp_{j+1}-1, p_j) \tag{2}$$

will calculate the position in F of the first occurrence of $\mathcal{P}_j$ within $\langle sp_{j+1}, ep_{j+1}\rangle$, and thus compute the start of our range of rows within $\mathcal{M}$ that correspond to $\mathcal{P}[j,m]$. Similarly, we compute

$$ep_j \;=\; \text{LF}(ep_{j+1}, p_j) - 1. \tag{3}$$

Once the area $\langle sp, ep\rangle$ is determined, self-indexes offer a way to find any occurrence position $\text{SA}[j]$, for $sp \le j \le ep$. This is accomplished by sampling $\mathcal{T}$ at regular intervals, and marking positions of SA that point to sampled text positions in a bitmap $E[0, n-1]$. Sampled suffix array positions are stored in an array $G[\text{RANK}_1(E,j)] = \text{SA}[j]$ if $E[j] = 1$. Given a target value $\text{SA}[j]$, the successive values $i = 0, 1, \ldots$ are evaluated until $E[\text{LF}^i(j)] = 1$, producing the desired answer of $\text{SA}[j] = \text{SA}[\text{LF}^i(j)] + i$. If every κth text position is sampled, we guarantee i can be found for every $0 \le i < \kappa$, and sampling requires $\mathcal{O}((n/\kappa)\log n)$ extra bits for G (and for E in compressed form [35]), and computes any entry of SA within κ applications of LF.

Similarly, in order to recover any text substring $\mathcal{T}[l, r-1]$ (including the whole $\mathcal{T}$), we can use the same sampling of text position multiples of κ, and store $H[i] = \text{SA}^{-1}[i\cdot\kappa]$. Thus, we extend the range to $T[l, r'-1]$, for $r' = \kappa \cdot \lceil r/\kappa \rceil$ and display from the suffix array position $j = \text{SA}^{-1}[r']$. Then, we can display the area backwards as $T^{\text{BWT}}[j], T^{\text{BWT}}[\text{LF}(j)], T^{\text{BWT}}[\text{LF}^2(j)], \ldots$. Each step requires one RANK_s and one ACCESS operation, which has the same cost as LF. Therefore, we can display $\mathcal{T}[l, r-1]$ within $\mathcal{O}(r-l+\kappa)$ probes of LF.

In practice self-indexes can be reduced to a wavelet tree over T^{BWT} with auxiliary information to emulate F (the C array) and the sampling information. This representation of a self-index is referred to as an *FM-Index* [12, 13]. A wavelet tree built over T^{BWT} uses $nH_k(\mathcal{T}) + o(n\log\sigma)$ bits [24] for any $k \le \alpha\log_\sigma(n) - 1$ and constant $\alpha < 1$, so the space requirements are reasonable. Here $H_k(\mathcal{T}) \le H_{k-1}(\mathcal{T}) \le \ldots \le H_0(\mathcal{T}) \le \log\sigma$ is the k-th order entropy of $\mathcal{T}$ [26], a measure of the performance of any compressor using k-th order statistical modeling on $\mathcal{T}$. Many other self-indexing variations exist with different time / space trade-offs [30, 15]. In principle, any of these approaches are compatible with the framework we present here, as long as the method returns a $\langle sp, ep\rangle$ range of matching suffixes.

4. TOP-K DOCUMENT RETRIEVAL USING SELF-INDEXES

In order to efficiently solve the **top-k document search problem**, unadorned self-indexing algorithms are not sufficient. Two approaches to enhance the self-index have been proposed. The first is to use a *document array*, that is, a mapping between every suffix in $\mathcal{T}$ to its corresponding document identifier [10, 18, 27, 41]. The second is to store, in addition to the global self-index, one self-index of each individual document in the collection [22, 37]. These alternatives offer different theoretical frameworks that are not directly comparable, but experimental studies [10, 31] have consistently shown that the first approach offers better space and time performance in practice.

Representing the document array with a single wavelet tree can provide additional important advantages. For example, the list of distinct documents where a substring $\mathcal{P}$ appears, with the corresponding term frequencies, can be obtained without any additional structure [18], in $\mathcal{O}(\log d)$ time per document retrieved, once the self-index has given the suffix array range of $\mathcal{P}$. This information

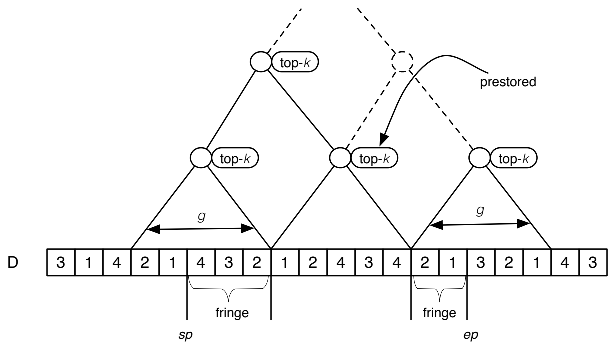

Figure 1: Precomputed top-k results over fixed intervals g stored in a skeleton succinct suffix tree using the HSV approach. Only the fringe leaves are processed for a given $\langle sp, ep\rangle$ range.

can then be used to calculate simple TF×IDF based $S(q, \mathcal{D}_i)$ metrics at query time [37]. In addition, several other operations such as Boolean intersection can be performed efficiently using only the wavelet tree over the document array [19].

Culpepper et al. [10] showed how to use the same wavelet tree to find the top-k documents (with raw term frequency weights) for a string $\mathcal{P}$. Among all of the strategies proposed, the heuristic algorithm **GREEDY** worked best in practice. Despite the lack of worst-case theoretical guarantees, they show unadorned wavelet trees are efficient in time and space for this task. Hon et al. [22] presented a technique with worst-case performance guarantees. The **HSV** approach builds on the same document listing strategy originally proposed by Sadakane [37]. While **HSV** was originally described using individual self-indexes for each document as in Sadakane's approach [37], the method can be applied on top of either document-listing solution in practice. The key insight of the **HSV** method is to precompute the top-k results for the lowest suffix tree nodes in a predetermined sampling range. Figure 1 shows a **HSV** tree over D. In this example, a $\langle sp', ep'\rangle$ range of size g is precalculated and stored in a succinct suffix tree. An arbitrary query $\langle sp, ep\rangle$ is received. The bulk of the query result is already precomputed as $\langle sp', ep'\rangle$. The remainder of the query can then be processed using RQQ queries over the *fringe* ranges to generate the final top-k counts.

Using g samples guarantees that any suffix array interval $\langle sp, ep\rangle$ for a given $\mathcal{P}$ falls into one of three categories: (1) The range is completely covered by the sampled interval, and the top-k answer is precomputed; (2) The range is partially covered, and at most $2g$ fringe leaves must to be processed at query time and merged with the sample; or (3) The range is too small to be covered. For Case (3), the complete $\langle sp, ep\rangle$ range must be processed at runtime, but is guaranteed to be smaller than $2g$.

Navarro and Valenzuela [31] demonstrated that implementing **HSV** over a document array, and using the **GREEDY** approach of Culpepper et al. [10] to speed up document listing, is more efficient than using either **GREEDY** or **HSV** in isolation. The hybrid approach requires additional space to support **HSV** on top of **GREEDY**, but efficiency is significantly improved by limiting the number of rank queries required at query time. This approach as well as other trade-offs are explored more fully in this paper.

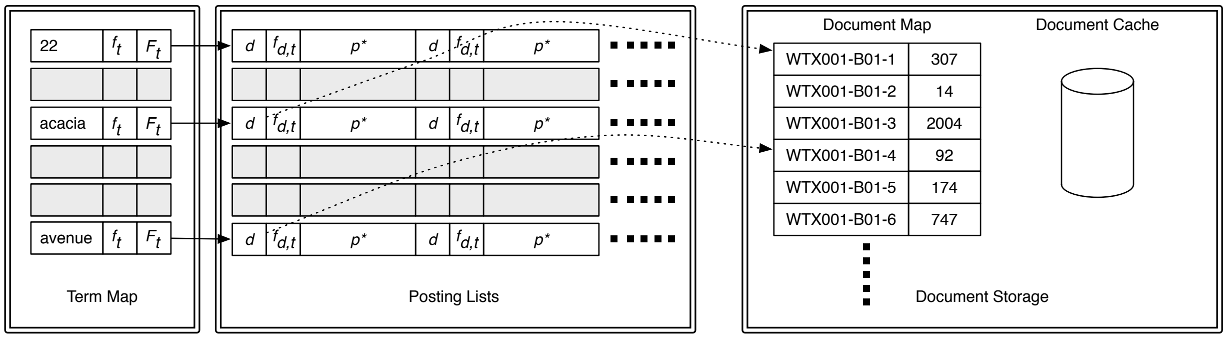

Figure 2: The three fundamental components of an inverted index. Each term in the vocabulary is mapped to a posting list of $\langle d, f_{d,t} \rangle$ tuples. For each tuple, the position offsets p^* are also stored to support phrase or term proximity queries.

5. INVERTED INDEXES

Traditional approaches to the **top-k document search problem** rely on *inverted indexes*. Inverted indexes have been the dominant data structure for a variety of ranked document retrieval tasks for more than four decades [44]. Despite various attempts to displace inverted indexes from their dominant position for document ranking tasks over the years, no alternative has been able to consistently produce the same level of efficiency, effectiveness, and time / space trade-offs that inverted indexes can provide (see, for instance Zobel et al. [45]).

Figure 2 shows a typical inverted indexing system. The system contains three key components: (1) *Term Map* - The vocabulary of terms, along with the number of documents containing one or more occurrence of the term (f_t), the number of occurrences of the term in the collection (F_t), and a pointer to the corresponding posting list. (2) *Posting Lists* - An ordered list of tuples, $\langle d, f_{d,t} \rangle$, containing the document identifier and the frequency of the term in document d. For each tuple, the ordered position offsets, p^* are also maintained in order to support phrase queries. For indexes that do not require phase queries, p^* can be omitted. (3) *Document Storage* - A document map to match d to the document name, and a pointer to the document in a document cache.

Ranked document retrieval requires that only the top-k documents are returned, and, as a result, researchers have proposed many heuristic approaches to improve top-k efficiency [1, 4, 5, 6, 32, 38]. These approaches can be classified in two general categories: *term-at-a-time* (**TAAT**) and *document-at-a-time* (**DAAT**). Each of these approaches have various advantages and disadvantages.

5.1 Term-at-a-Time Processing (TAAT)

For **TAAT** processing, a fixed number of accumulators are allocated, and the rank contribution incrementally calculated for each query term in increasing document order. When inverted files are stored on disk, the advantages of this method are clear. The inverted file for each term can be read into memory, and processed sequentially. However, when k is small relative to the total number of matching documents in collection, **TAAT** can be inefficient, particularly when the number of terms in the query increases, since all of the inverted lists must be processed before knowing the full rank score of each document. In early work, Buckley and Lewit [6] proposed using a heap of size k to allow posting lists to be evaluated in **TAAT** order. Processing is terminated when the sum of the contributions of the remaining lists cannot displace the minimum score in the heap.

Moffat and Zobel [29] improved on this pruning approach with two heuristics: **STOP** and **CONTINUE**. The **STOP** strategy is somewhat similar to the method of Buckley and Lewit, but the terms are processed in order of document frequency from least frequent to most frequent. When the threshold of k accumulators is reached, processing stops. In contrast, the **CONTINUE** method allows the current accumulators to be updated, but new accumulators cannot be added. These accumulator pruning strategies only approximate the true top-k result list.

If approximate results are acceptable, the **TAAT** approach can be made even more efficient using *impact ordering* [1, 2, 33]. The key idea of impact ordering is to precompute the TF for each document a term appears in. Next, quantize the TF values into a variable number of buckets, and sort the buckets (or blocks) for each term in decreasing impact order. Now, the top-k representative can be generated by sequentially processing each of the highest ranking term contribution blocks until a termination threshold is reached. The authors refer to this blockwise processing method as *score-at-a-time* processing. Despite not using the full TF contribution for each term, Anh and Moffat [1] demonstrate that the effectiveness of impact ordered indexes is not significantly reduced, but efficiency is dramatically improved.

5.2 Document-at-a-Time Processing (DAAT)

The alternative approach is to process all of the terms simultaneously, one document at a time [8]. The advantage of this approach is that the final rank score is known as each document is processed, so it is relatively easy to maintain a heap containing exactly k scores. The disadvantage is that all of the term posting lists are cycled through for each iteration of the algorithm requiring non-sequential disk reads for multi-word queries. However, our focus in this paper is *in-memory* ranked retrieval, so **DAAT** tends to work very well in practice.

Pruning strategies to further increase efficiency also exist for **DAAT** processing. The most widely used pruning strategy for **DAAT** is **MAXSCORE**. Turtle and Flood [40] observed that the **BM25** TF component can never exceed $k_1 + 1 = 2.2$. So, the total score contribution for any term is at most $2.2 \cdot \log(N/N_t)$. Using this observation, Turtle and Flood present an algorithm that allows posting values below the threshold to be skipped. As the minimum bounding score in the heap slowly increases, more and more postings can be omitted. Enhanced **DAAT** pruning strategies similar in spirit to **MAXSCORE** have been shown to further increase efficiency [4, 38].

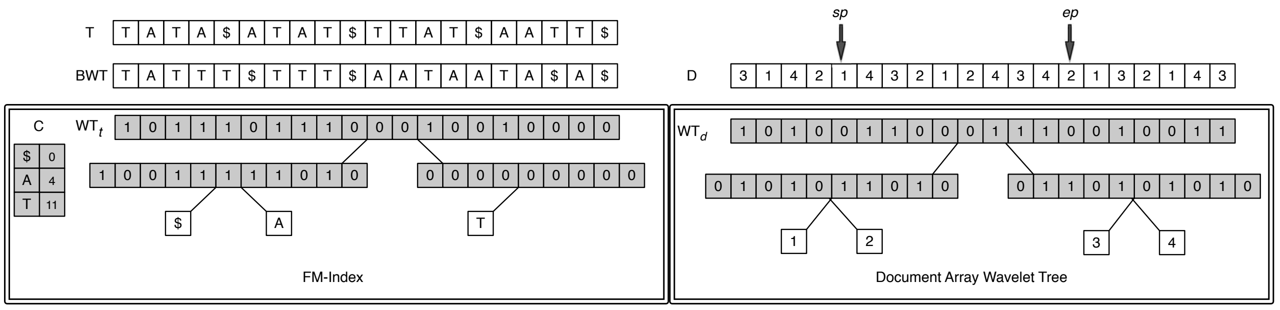

Figure 3: Given the text collection $\mathcal{T}$ = TATA\$ATAT\$TTAT\$AATT\$ of four documents, a self-indexing system requires two wavelet trees. The first wavelet tree supports backwards search over the BWT **permuted text, and the second supports statistical calculations over the document array. Note that only the items in grey are stored and used for query operations.**

Turtle and Flood also describe a similar approach to improve the efficiency of **TAAT** strategies. However, the **TAAT** variant is more complex than the **DAAT** approach as it requires an ordered candidate list of k documents to be maintained. The candidate list is used to skip document postings in each term list which could not possibly displace the current top-k documents once the heap contains k items.

Fontoura et al. [17] compare several **TAAT** and **DAAT** based in-memory inverted indexing strategies. The authors present novel adaptations of **MAXSCORE** and **WAND** [4] to significantly improve query efficiency of in-memory inverted indexes. The authors go on to show further efficiency gains in **DAAT** style processing by splitting query terms into two groups: rare terms and common terms. The exact split is based on a fixed threshold selected at query time. For our baselines, we use **WAND** for **DAAT** query processing, and **MAXSCORE** for **TAAT** query processing.

6. SELF-INDEXING APPROACHES

We now describe our general approach to in-memory indexing and retrieval. Figure 3 shows the key components of our retrieval system: an FM-Index and the document array wavelet tree, WT_d. In addition, our system requires a *Document Map* to map document identifiers to human readable document names (or URLs). No document cache is required and the original documents or snippets around each match can be recreated directly from the FM-Index by extracting the required text positions using the suffix array sampling. Only the items in grey are stored and used for character-based top-k document retrieval. All other components are shown for illustration purposes only.

ALGORITHM **GREEDY-TAAT**

INPUT	A sorted list t of q terms.
OUTPUT	A list of k documents in rank order.

```
1: Initialize a max-heap $R \leftarrow \{\}$
2: for $i \leftarrow 1$ to $q$ do
3:     Determine $\langle sp, ep \rangle$ for term $t_i$
4:     $A_i \leftarrow$ GREEDY$(sp, ep, k')$
5: end for
6: for $i \leftarrow 1$ to $q$ do
7:     for $j \leftarrow 1$ to $k'$ do
8:         if $A_i[j] \in R$ then
9:             UPDATE$(R, A_i[j], score)$
10:        else
11:            ADD$(R, A_i[j], score)$
12:        end if
13:    end for
14: end for
15: return $R[1 \ldots k]$
FUNCTION GREEDY $(sp, ep, k)$
1: $\ell \leftarrow \text{WT}_d$.root
2: A max-heap, sorted by $ep - sp$, $h \leftarrow$ PUSH$(\ell, [sp, ep])$
3: A priority queue PQ $\leftarrow \{\}$.
4: $i \leftarrow 0$
5: while $h \neq \emptyset$ and $i < k$ do
6:     $\ell, [sp', ep'] \leftarrow$ POP$(h)$
7:     if $\ell$ is leaf then
8:         PQ $\leftarrow$ ENQUEUE$(\ell.\text{docid}, ep' - sp' + 1)$
9:         $i \leftarrow i + 1$
10:    else
11:        $[s_0, e_0] \leftarrow [\text{RANK}_0(\mathcal{B}_\ell, sp'), \text{RANK}_0(\mathcal{B}_\ell, ep')]$
12:        $[s_1, e_1] \leftarrow [\text{RANK}_1(\mathcal{B}_\ell, sp'), \text{RANK}_1(\mathcal{B}_\ell, ep')]$
13:        if $e_0 - s_0 > 0$ then $h \leftarrow$ PUSH$(\ell.\text{left}, [s_0, e_0])$
14:        end if
15:        if $e_1 - s_1 > 0$ then $h \leftarrow$ PUSH$(\ell.\text{right}, [s_1, e_1])$
16:        end if
17:    end if
18: end while
19: return PQ
```

A simple bag-of-words search using a self-index retrieval system is outlined in Algorithm **GREEDY-TAAT**. Recall that the sp and ep range for any string can be found using a backwards search in the BWT permuted text using only a wavelet tree over T^{BWT} and C. So, the $\langle sp, ep \rangle$ for each query term in Line (3) can be calculated in $\mathcal{O}(|t_i| \log \sigma)$ time using an FM-Index. Now, a wavelet tree over the document array WT_d can be used to retrieve exactly k documents in frequency order for each term using **GREEDY** or **QUANTILE** [10]. This algorithm is analogous to **TAAT** processing, and is referred to as **GREEDY-TAAT**. Note that Function **GREEDY** can also be augmented with **HSV** as described in Section 4 to further increase the efficiency of constructing A_i for each query term.

We also present several variations on this general strategy. First, we consider the addition of **HSV** style precomputations over WT_d as described by Navarro and Valenzuela [31]. Instead of storing the top-k most frequent symbols in the skeleton suffix tree, we store

Query length	TREC 7 & 8 queries			TREC WT10G queries		
	Total	Matches	Average n_i	Total	Matches	Average n_i
$\lvert q \rvert$	Queries	('000)	('000)	Queries	('000)	('000)
1	100	9.9	9.9	100	3.3	4.7
2	100	24.8	12.5	100	68.6	42.5
3	100	104.5	38.5	100	292.8	123.2
4	100	238.1	69.0	100	601.1	166.6
5	100	351.2	95.1	100	866.5	228.5
6	100	408.7	107.8	100	1041.9	280.4
7	100	463.8	126.2	100	1149.7	319.6
8	100	489.8	148.3	100	1171.8	339.2
random sample	800	234.9	70.0	800	621.5	181.2

Table 1: Statistics of the queries used in experiments (sampled based on query length, or sampled from the filtered MSN query log), reporting the number of queries run, the mean number of documents that contained one or more of the query terms, and the mean length of the inverted lists processed.

the top-k most important symbols sorted by term impact for each interval g to improve effectiveness. In order to capture k-values commonly used in IR systems ($k = 10, 100, 1000$), we prestore values of any k that is a power of 2 up to 8192 in term contribution order. Note that we go higher than 1024 since the values of k' necessary to ensure good effectiveness can be greater than the desired k.

Observe that in typical bag-of-words query processing over English text, the size of the vocabulary is often small relative to the total size of the collection. As such, we also present a new hybrid approach to top-k bag-of-words retrieval using a *Term Map* and WT$_d$. If we assume the vocabulary is fixed for each collection, then the $\langle sp, ep \rangle$ range for each term can be precalculated and retrieved using a term map, as in the inverted indexing solution. This means that the FM-Index component is no longer necessary when processing bag-of-words queries. We refer to these hybrid approaches as **SEM-GREEDY** and **SEM-HSV**. These methods reduce the overall space requirements of our approach, but also limit the full functionality of some auxiliary operations. For example, the text can no longer be reproduced directly from the index, so snippets cannot be generated on the fly, and phrase queries are no longer natively supported. However, for classic bag-of-words queries, our hybrid approach provides an interesting trade-off to consider. Our final variation is to support a term-based self-index. We refer to this approach as **FM-TERM**.

It is also possible to support a **DAAT** query processing strategy in our retrieval system, but this would require efficiently supporting RMQ over the document array. Our approach currently supports a generalization of RMQ – RQQ. But, the cost of RQQ is $\mathcal{O}(\log d)$ per k value extracted, while constant time solutions for RMQ currently exist [16]. However, an RMQ style approach as presented by Fischer and Heun [16] incurs an additional $2n$ bits of space, and so we do not explore the possibility further in this work.

Also note the current top-k bag-of-words approach shown in **GREEDY-TAAT** is based entirely on the frequency counts of each item. This means that our current implementation only approximates the top-k items. This is a well-known problem in the inverted indexing domain. This limitation holds for any character-based bag-of-words self-indexing system that does frequency counting at query time since we can not guarantee that item $k + 1$ in any of the term lists does not have a higher score contribution than any item currently in the top-k intermediate list. A method of term contribution precalculation is required in order to support **BM25** or language-model ranking. Without the term contribution scoring, **WAND** and **MAXSCORE** enhancements are not possible, and therefore every document in the $\langle sp, ep \rangle$ must be evaluated in order to guarantee the final top-k ordering. However, this limitation can be mitigated by using **HSV** since we can precalculate the impact contribution for each sample position and store this value instead of storing only the frequency ordering. Top-k guarantees are also possible using a term-based self-indexing system where each distinct term is mapped to an integer using **HSV** or other succinct representations of term contribution preprocessing. In future work, we intend to fully examine all of the possibilities for top-k guarantees using self-indexes in various bag-of-words querying scenarios.

When using character-based self-indexing approaches for bag-of-words queries, there is another disadvantage worth noting. For self-indexes, there is an efficiency trade-off between locating the top-k $f_{d,t}$ values and accurately determining f_t since the index can extract exactly k $f_{d,t}$ values without processing every document. For a fixed vocabulary, f_t is easily precomputed, and can be stored in the term map with the $\langle sp, ep \rangle$ pairs. But, in general it is not straightforward to determine f_t for arbitrary strings over WT$_d$ without auxiliary algorithms and data structures to support calculating the value on the fly. The **FM-HSV** approach allows us to prestore f_t for each sampled interval which can be be used to calculate f_t over $\langle sp, ep \rangle$ more efficiently by only processing potential fringe leaves. Calculating f_t using only WT$_d$ for arbitrary strings in near constant time using no additional space remains an open problem.

7. EXPERIMENTS

In order to test the efficiency of our approach, two experimental collections were used. For a small collection, we used the TREC 7 and 8 *ad hoc* datasets. This collection is composed of 1.86 GB of newswire data from the *Financial Times*, *Federal Register*, *LA Times*, and *Foreign Broadcast Information Service*, and consists of around $528{,}000$ total documents [42]. For a larger in-memory collection, we used the TREC WT10G collection. This collection consists of 10.2 GB of markup text crawled from the internet, totalling $1{,}692{,}096$ documents [21].

All of the algorithms described in this paper were implemented using C/C++ and compiled with gcc 4.6.1 with -O3 optimizations. For our baselines, we have implemented the in-memory variant of **WAND** as described by Fontoura et al. [17] for **DAAT**, and an in-memory variant of **MAXSCORE** for **TAAT**. Experiments were run on a single system with $2\times$ Intel Xeon E5640 Processors with a 12 MB smart cache, 144 GB of DDR3 DRAM, and running Ubuntu Linux 11.10. Times are reported in milliseconds unless otherwise

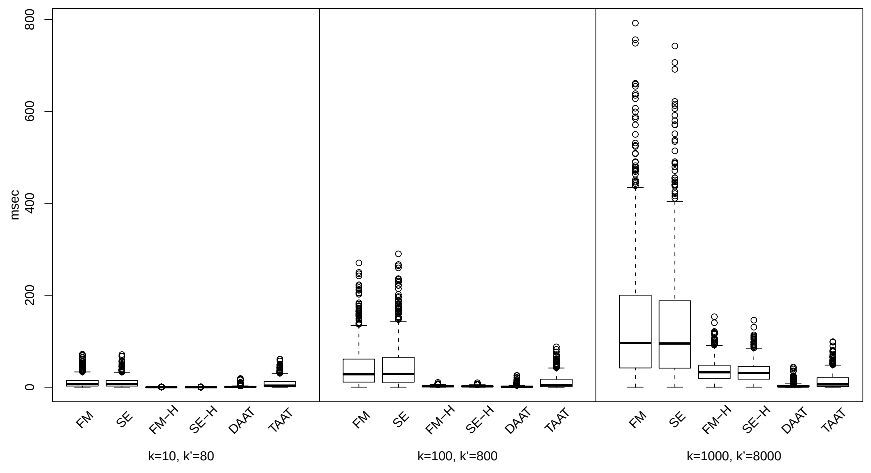

Figure 4: Efficiency for 1,000 randomly sampled MSN queries against the TREC 7 & 8 collection.

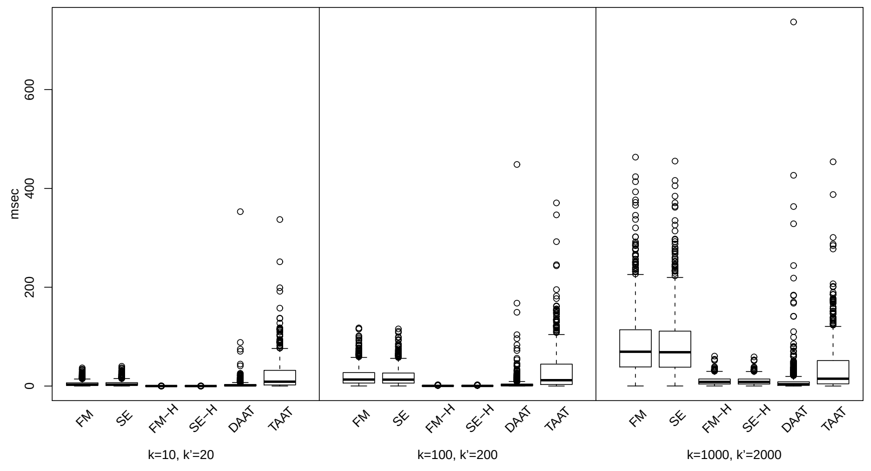

Figure 5: Efficiency for 1,000 randomly sampled MSN queries against the the TREC WT10G collection.

noted. All efficiency runs are reported as the mean and median of 10 consecutive runs of a query, and all necessary information is preloaded into memory.

Note that we do not carry out an full evaluation of the effectiveness of the algorithms presented in this paper. In previous work, we showed that the **BM25** ranking and query evaluation framework used in our approach can be as effective as other state-of-the-art open source search engines when using $k' > k$, and do not repeat those experiments here [11]. In these experiments, we use the minimum k' values that result in retrieval performance that is comparable to the effectiveness obtained through exhaustive processing. In all experiments we use $k' = 8 * k$ for the TREC 7 & 8 dataset, and $k' = 2 * k$ for the TREC WT10G dataset. These values for k' give results for the MAP and P@10 effectiveness measures that are not statistically significantly different compared to exhaustive processing, for both collections (paired t-test, $p > 0.05$). We intend to pursue additional efficiency and effectiveness trade-offs in future work.

7.1 Experimental Setup

In order to test the efficiency of our algorithms, queries of varying length were extracted from a query log supplied by Microsoft. Each query was tested against both TREC collections, and the filtering criteria used was that every word in the query had to appear in at least 10 distinct documents, resulting in a total of $656{,}172$ unique queries for the TREC 7 & 8 collection, and a total of $793{,}334$ unique queries for the TREC WT10G collection. From the resulting filtered query sets, two different query samples were derived.

First, 1000 queries of any length were randomly sampled from each set, to represent a generic query log run. The $1{,}000$ sampled queries for TREC 7 & 8 have an average query length of 4.224, and the average query length of the WT10G sample set is 4.265 words per query. For the second set of experiments, 100 queries for each query length 1 to 8 were randomly sampled from the same MSN query sets. Table 1 shows the statistical properties of the sampled queries that were used in the second experimental setup, including the average number of documents returned for each query for each query length, and the average length of postings lists processed for each query, computed as $(\sum_{i=1}^{|q|} n_i)/|q|$.

7.2 Average Query Efficiency

In order to test the efficiency of our algorithms, two experiments were performed on each of the collections. The first experiment is designed to measure the average efficiency for each algorithm,

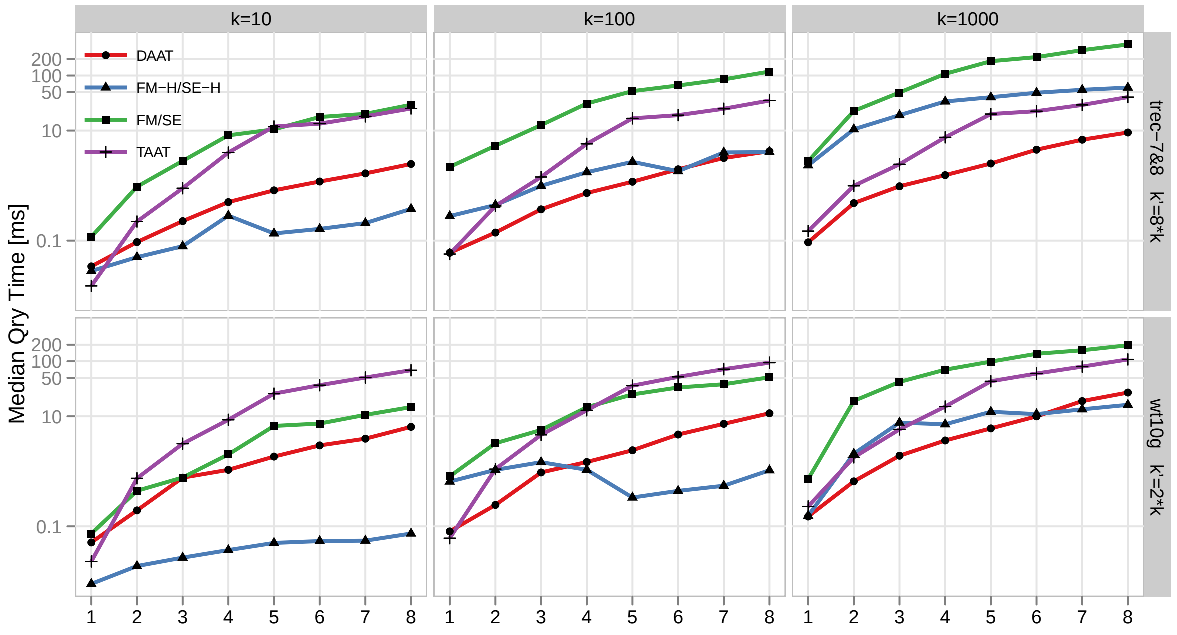

Figure 6: Efficiency of query length of 1 to 8 on the TREC 7 & 8 collection (top row) and TREC WT10G collection (bottom row) for *k*=10,100 and 1000. For each query length, 100 randomly sampled queries are used from the MSN query log set.

given a sampling of normal queries. For this experiment, the length of the queries was not bounded during sampling, and had an average query length of just over 4 words per query as mentioned in Section 7.1.

Figures 4 and 5 show the relative efficiency of each method averaged over $1{,}000$ randomly sampled MSN queries for TREC 7 & 8, and TREC WT10G. Each boxplot summarizes the time values as follows: the solid line indicates the median; the box shows the 25th and 75th percentiles; and the whiskers show the range, up to a maximum of 1.5 times the interquartile range, with outliers beyond this shown as separate points. In both figures, the following abbreviations are used for the algorithms: **FM-GREEDY** (**FM**), **SEM-GREEDY** (SE), **FM-HSV** (**FM-H**), **SEM-HSV** (**SE-H**), **DAAT** , and **TAAT**.

We report the timings for all of the self-indexing methods using the *character-based* indexes. We also ran the same experiments using our *term-based* indexes, but the performance was identical. This result is not surprising since the dominant cost in the self-indexing method is traversing the wavelet tree over the document array, and is dictated by the depth of the wavelet tree and not the overall length. Since the depth depends only on the number of documents, both approaches consistently produce similar running times. So, the only efficiency difference between character-based and term-based indexes is in space-usage which is discussed in Section 7.

We see that the self-indexing methods which must calculate all frequency scores (**FM** and **SE**) incur the most overhead as k increases. This is largely due to the multiplicative effect of collating many consecutive k values. For example, when collating frequency values from WT_d, the number of rank operations is proportional to the depth of the wavelet tree. In the case of the TREC WT10G collection, which contains around 1.6 million documents, the depth of the wavelet tree is 24. So, the number of random rank probes in wavelet tree begins to significantly degrade the performance for larger k.

This effect can be marginalized by augmenting WT_d with **HSV**. Since an **HSV** style index has a portion of each $\langle sp, ep\rangle$ ranges, only the *fringe* positions for each range need to be calculated at runtime, reducing the total number of page faults. For all k, the **HSV** indexes are efficient and remarkably resilient to outliers. In general, the **WAND** variant of **DAAT** is more efficient for large values of k, but can perform poorly for certain queries. For example, the query "`point out the steps to make the world free of pollution`" on the WT10G collection consistently performed poorly in our **DAAT** framework.

7.3 Efficiency based on Query Length

We now break down the efficiency of each of our algorithms relative to two parameters: k and q, where q is the number of terms in a query. Figure 6 shows the average of 10 runs of 100 queries per query length, q. For one-word queries, for all values of k, the inverted indexing approaches **DAAT** and **TAAT** are superior. This is not surprising since only a single term posting must be traversed to calculate **BM25**, and the algorithms have excellent locality of access. Still, the **HSV** variant is the most efficient for small k.

For $|q| > 1$, the results also depend on k. For $k = 10$ and $k = 100$, the self-indexing methods are more efficient than **TAAT** since the methods can extract exactly k values. Since the sample rates in the lower regions of the **HSV** methods are close to k, very little work needs to be done by the indexes. The **WAND**-based **DAAT** method remains remarkably efficient for all values of k. As k increases, the performance of the **HSV**-based approaches begins to degrade since the sample size for the precalculated top-k orderings grows exponentially. The performance degradation at large k is equivalent to Case (3) as described in Section 4. In essence, most of the $\langle sp, ep\rangle$ ranges turn out to be much smaller than any of the samples, so the complete $\langle sp, ep\rangle$ range must be computed at runtime, reducing the performance to **FM-GREEDY** when an appropriate sample is not available. Note that the performance of **HSV** for TREC 7 & 8 is worse than for WT10G for two reasons. First, k'

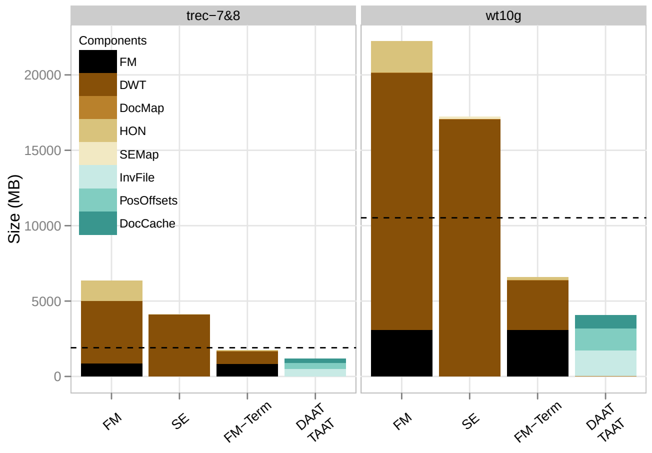

Figure 7: Space usage for each component in the three indexing approaches presented in this paper for the TREC 7 & 8 collection (left) and the TREC WT10G collection (right). The dashed line in both graphs represents the total space usage of the original uncompressed text for the collection.

is four times larger in TREC 7 & 8 resulting in fewer sample points. Secondly, if only a partial match is found, the self-index approach must retrieve 8 times more intermediate documents for scoring than in the inverted indexing approaches.

Note that none of our self-indexing approaches currently employ **MAXSCORE** or other methods to guide scoring. In principle our approach could also benefit from similar enhancements. We intend to explore the benefits and drawbacks of various early termination and impact scoring approaches for self-indexes in future work.

7.4 Space Usage

We now address the issue of space usage for the different algorithmic approaches. Inverted indexes are designed to take advantage of a myriad of different compression techniques. As such, our baselines also support several state-of-the-art byte and word aligned compression algorithms [3, 9, 28, 39, 43]. So, when we report the space usage for an inverted index, the numbers are reported using compressed inverted indexes and compressed document collections.

Figure 7 presents a break down of space usage for each component of the inverted indexing and self-indexing approaches. From a functionality perspective, there are several different componentization schemes to consider. First, consider the comparison of an inverted index method (including the term map, the postings list with p^* offsets, the document map, and the compressed document cache) with an FM-Index (including WT_d, the document map, and any other precomputed values – for instance the **HSV** enhancement). We consider these two in-memory indexes as functionally equivalent, as both can support bag-of-words or phrase queries, and can recreate snippets or even the original uncompressed document. The character based variant **FM** is significantly larger, but able to support a range of special character and arbitrary substring queries that term-based indexes do not support. Therefore, the term-based self-indexing variant **FM-TERM** is much closer to the inverted indexing variant in space usage and functionality.

The second alternative are indexes that support *only* bag-of-words queries. Now, an inverted index method requires only the term map, the postings list *without* p^* offsets, and the document map. The character-based self-indexes are essentially the same, but the FM-Index component is replaced with a term map component. Note that the **FM** component of **FM-TERM** is only required for phrase queries, and can also be dropped if only bag-of-words queries are required. When considering all of the current self-indexing options presented in this paper, using an FM-Index component instead of a term map appears to offer the most flexible configuration for character-based self-indexes, while the term-based variant is competitive in both time, space, and functionality with an inverted index.

8. CONCLUSION

We have presented an algorithmic framework for in-memory bag-of-words query processing that is efficient in practice. We have compared and contrasted our framework with industry and academic standard inverted indexing algorithms. Our approach shows great promise for advancing the state-of-the-art in exciting new directions.

However, several challenges must be overcome before these algorithms can reach widespread acceptance. For instance, recent work has dramatically reduced the space required for self-indexing algorithms, there are still opportunities to further reduce space usage in self-indexes. Another shortcoming of bag-of-words querying with self-indexing algorithms is providing top-k guarantees. While good solutions exist for providing top-k guarantees on singleton pattern queries, optimally merging multiple queries remains problematic.

However, self-indexing algorithms can also efficiently provide functionality that is notoriously inefficient, and sometimes even impossible, using inverted indexes. In addition to basic bag-of-words queries, our approach has the capability to perform phrase queries of any length, as well as the ability to support complex statistical calculations at query time, with no additional indexing costs. In fact, phrase queries were shown to be significantly faster using **FM-GREEDY** than when using inverted indexing approaches in prior work [10]. Self-indexes also inherently preserve term proximity. So, not only can each term be found quickly, but the n-terms surrounding the keyword can quickly be extracted, tabulated, and used for on-the-fly statistical calculations. Applications of this functionality include more efficient relevance feedback algorithms, construction of higher order language models in ranking metrics, or term dependency extraction and query expansion. In summary, all of the disadvantages outlined in this paper for self-indexing paper warrant further research, as the potential benefits of this new approach are compelling indeed.

In future work, we will explore new algorithmic approaches to reduce space usage, and to further improve efficiency for larger values of k. We also intend to investigate the combination of efficient phrase querying and proximity calculations to produce and evaluate novel ranking metrics. Finally, we will design and evaluate new approaches to support distributed in-memory query processing in order to scale our system to terabyte size collections.

9. ACKNOWLEDGMENTS

This work was supported in part by the Australian Research Council. We thank Gonzalo Navarro and Daniel Valenzuela for insightful discussions on the **HSV** method. We also thank Alistair Moffat for valuable feedback on a draft of this paper.

References

[1] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR*, pages 372–379, 2006.

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

[3] N. R. Brisaboa, A. Fariña, G. Navarro, and M. F. Esteller. (S, C)-dense coding: An optimized compression code for natural language text databases. In *SPIRE*, volume 2857 of *LNCS*, pages 122–136, 2003.

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

[5] E. W. Brown. Fast evaluation of structured queries for information retrieval. In *SIGIR*, pages 30–38, 1995.

[6] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *SIGIR*, pages 97–110, 1985.

[7] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, May 1994.

[8] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and evaluating search engines.* MIT Press, Cambridge, Massachusetts, 2010.

[9] J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In *SPIRE*, volume 3772 of *LNCS*, pages 1–12, November 2005.

[10] J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *ESA, Part II*, LNCS 6347, pages 194–205, 2010.

[11] J. S. Culpepper, M. Yasukawa, and F. Scholer. Language independent ranked retrieval with NeWT. In *ADCS*, pages 18–25, December 2011. See `http://goanna.cs.rmit.edu.au/~e76763/publications/cys11-adcs.pdf`.

[12] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398, 2000.

[13] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005. A preliminary version appeared in FOCS 2000.

[14] P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207:849–866, 2009.

[15] P. Ferragina, R. Gonzaález, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics*, 13:12.1–12.31, 2009.

[16] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[17] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proceedings of the VLDB Endowment*, 4(12):1213–1224, 2011.

[18] T. Gagie, S. J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *SPIRE*, pages 1–6, 2009.

[19] T. Gagie, G. Navarro, and S. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426:25–41, 2012.

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

[21] D. Hawking. Overview of the TREC-9 web track. In *TREC-8*, pages 87–102, 1999.

[22] W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *FOCS*, pages 713–722, 2009.

[23] G. Jacobson. *Succinct static data structures.* PhD thesis, Carnegie Mellon University, 1988.

[24] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *SPIRE*, LNCS 4726, pages 229–241, 2007.

[25] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comp*, 22(5):935–948, 1993.

[26] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, May 2001.

[27] S. Mithukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666, 2002.

[28] A. Moffat and V. N. Anh. Binary codes for non-uniform sources. In *DCC*, pages 133–142, 2005.

[29] A. Moffat and J. Zobel. Self indexing inverted files for fast text retrieval. *ACM TOIS*, 14(4):349–379, 1996.

[30] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2–1 – 2–61, 2007.

[31] G. Navarro and D. Valenzuela. Space-efficient top-k document retrieval. In *SEA*, LNCS 7276, pages 307–319, 2012.

[32] M. Persin. Document filtering for fast ranking. In *SIGIR*, pages 339–348, 1994.

[33] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency sorted indexes. *JASIST*, 47(10):749–764, 1996.

[34] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comp. Surv.*, 39(2):4.1–4.31, 2007.

[35] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, pages 233–242, 2002.

[36] S. E. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *TREC-3*, 1994.

[37] K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discr. Alg.*, 5(1):12–22, 2007.

[38] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *SIGIR*, pages 219–225, 2005.

[39] A. Trotman. Compressing inverted files. *Information Retrieval*, 6(1): 5–19, 2003.

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

[41] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *CPM*, LNCS 4580, pages 205–215, 2007.

[42] E. M. Voorhees and D. K. Harman. Overview of the Eighth Text REtrieval Conference (TREC-8). In *TREC-8*, pages 1–24, 1999.

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

[44] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comp. Surv.*, 38(2):6–1 – 6–56, 2006.

[45] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM TODS*, 23(4):453–490, 1998.