Inf Retrieval (2013) 16:747–768
DOI 10.1007/s10791-012-9217-9

Document vector representations for feature extraction in multi-stage document ranking

Nima Asadi · Jimmy Lin

Received: 29 May 2012 / Accepted: 14 December 2012 / Published online: 30 December 2012

Abstract We consider a multi-stage retrieval architecture consisting of a fast, "cheap" candidate generation stage, a feature extraction stage, and a more "expensive" reranking stage using machine-learned models. In this context, feature extraction can be accomplished using a document vector index, a mapping from document ids to document representations. We consider alternative organizations of such a data structure for efficient feature extraction: design choices include how document terms are organized, how complex term proximity features are computed, and how these structures are compressed. In particular, we propose a novel document-adaptive hashing scheme for compactly encoding term ids. The impact of alternative designs on both feature extraction speed and memory footprint is experimentally evaluated. Overall, results show that our architecture is comparable in speed to using a traditional positional inverted index but requires less memory overall, and offers additional advantages in terms of flexibility.

Keywords Learning to rank · Compression · Document stores

1 Introduction

A common architecture in web retrieval breaks document ranking into three stages: candidate generation, feature extraction, and document reranking. There are two main reasons for this multi-stage design: First, there is general consensus in the community that learning to rank provides the best solution to document ranking. In particular, ensembles of tree-based learners have proven effective, as documented in both the academic literature (Ganjisaffar et al. 2011; Li 2011) and in production commercial search engines such as Bing (Burges 2010). As it is difficult to apply learning-to-rank methods over the *entire*

N. Asadi (✉)
Department of Computer Science, University of Maryland, College Park, MD, USA
e-mail: nima@cs.umd.edu

J. Lin
The iSchool, College of Information Studies, University of Maryland, College Park, MD, USA
e-mail: jimmylin@umd.edu

collection, in practice a candidate list of potentially-relevant documents serves as the input. Thus, learning to rank is actually a *reranking* problem (hence the first and third stages). Second, separating candidate generation from feature extraction has the advantage in providing better control over cost/quality tradeoffs. For example, term proximity features are significantly more costly to compute than unigram features; therefore, by decoupling the first two stages, systems can take advantage of "cheap" features to generate candidates quickly and only compute "expensive" term proximity features when necessary—thus decreasing overall query evaluation latency.

This paper focuses on the feature extraction stage in this multi-stage architecture, assuming standard techniques for candidate generation and an arbitrary machine-learned model for the third reranking stage. Specifically, we explore index organizations where the inverted index is decoupled from the document vector index (i.e., a forward index) and experimentally evaluate the efficiency of different document vector index organizations in terms of speed (latency) and memory usage. As is the norm in web-scale search, we assume that all index structures are completely held in memory. A standard positional inverted index (PII) provides the baseline.

Of features explored in our work, term proximity features are the most interesting: previous studies have confirmed that they contribute significantly to effectiveness, yet it is also well known that such features are expensive to compute. We evaluate two alternative structures for storing document vectors, one as a mini-index of terms within the document, and another as a flat array of terms. These give rise to two algorithms for computing term proximity features: by intersecting postings of document term positions or by applying a sliding window over a flat term array representation. Finally, different index organizations are amenable to different integer compression techniques for compact encoding (both existing techniques and a novel one we propose). These various dimensions define the design space that we explore empirically.

There are two main contributions to this work. First, to our knowledge, this is the first quantitative comparison of alternative document vector index organizations for fast feature extraction. Our results show that a lightweight inverted index with an efficiently-organized document vector index compares favorably to a monolithic PII in terms of speed, but requires less memory overall and offers additional advantages in terms of flexibility. Experiments additionally reveal interesting observations on the relationship between processor and memory latency. Second, we propose a novel document-adaptive hash-based compression technique for encoding integer term ids in document vectors that is more compact than PForDelta (Yan et al. 2009; Zhang et al. 2008; Zukowski et al. 2006) (which represents current best practices in index organization) but just as fast (in terms of decoding speed).

2 Background and related work

The multi-stage retrieval architecture that provides the context for this work has been documented in the literature by many (Broder et al. 2003; Ding and Suel 2011; Matveeva et al. 2006; Tatikonda et al. 2011; Wang et al. 2011), although it remains relatively obscure[1]—partially because the emphasis of previous work has been on retrieval

[1] For example, a recent survey of web search architectures makes no mention of multi-stage retrieval (Cambazoglu and Baeza-Yates 2011); Ding and Suel (Ding and Suel 2011) characterize this approach as "folklore".

effectiveness and not architectural issues, so in general there has been a dearth of implementation details. Nevertheless, the general idea is clear: multi-stage retrieval systems typically use a “cheap” (fast) algorithm and one or perhaps more “expensive” (slower) algorithms for ranking. The “cheap” algorithm is used to generate a candidate list of potentially-relevant documents (the candidate generation phase): an example might be a linear combination of BM25 and a query-independent score. Frequently, queries are processed conjunctively, i.e., only documents that have *all* the query terms are considered. For web-scale collections, this approach leads to higher early precision and faster query evaluation (Broder et al. 2003). These candidate documents are then reranked by the “expensive” but higher quality (usually, machine-learned) models. Example implementations include gradient boosted regression trees (Burges 2010; Ganjisaffar et al. 2011), additive ensembles (Cambazoglu et al. 2010), cascades of rankers (Matveeva et al. 2006; Wang et al. 2011), just to name a few. The “expensive” algorithms typically consider features too costly to compute in the first phase (e.g., term proximity scores, which require decoding positions). Architecturally, we can view feature generation as a distinct stage between the “cheap” and “expensive” ranking algorithms.

It is widely recognized that term proximity features contribute to improvements in retrieval effectiveness over term-based (i.e., unigram) features alone. In general, term proximity features can be divided into two types: query terms occurring as a phrase (i.e., ordered sequence of consecutive terms) and query terms co-occurring within a fixed span (usually without regard to ordering). The span width is typically a parameter either hand-tuned or machine-learned. The positive benefits of term proximity features have been observed to be statistically significant across many different test collections and is regarded as a robust finding. Early studies date back to at least the 1980s (Fagan 1987), and there has been no shortage of confirmatory evidence since (Büttcher et al. 2006; Clarke et al. 1995; Croft et al. 1991; Hawking and Thistlewaite 1995; Tao and Zhai 2007).

Term proximity features, however, are “expensive”, both in terms of time and space. In order to support these features, we need to store positional information in the inverted index, which substantially increases its size.[2] Computing the features themselves requires decoding term positions and additional comparisons to determine if the terms fall within the specified window. By one estimate based on microbenchmarks, computing bigram features is 20 times slower than computing unigram features (Wang et al. 2011). Therefore, in a multi-stage architecture, it makes sense not to use term proximity features in the initial candidate generation phase. This in turn implies that positional information does not need to be stored in the inverted index. This is the route we have pursued: term proximity features are computed in the second feature generation stage, using a document vector index.

Of course, real-world search engines take advantage of caching (Baeza-Yates et al. 2007; Long and Suel 2005). In principle, expensive features in frequently-evaluated (e.g., high quality) documents might be cached to avoid recomputation. However, feature caching is a non-trivial engineering problem, since the space of cache values is the query space crossed with the document space. This challenge aside, we argue that caching is an orthogonal issue to our work: caching strategies can also be applied on top of our proposed techniques to further increase speed.

[2] Alternatively, it is possible to index phrases directly (e.g., Broschart and Schenkel 2012; Williams et al. 2004). However, given that proximity features are usually parameterized in some manner (e.g., span width), positional indexes are more flexible in being able to compute proximity features on the fly.

Finally, it is worth mentioning two other families of techniques for fast query evaluation: index pruning (Altingovde et al. 2012; Carmel et al. 2001; Ntoulas and Cho 2007), and early termination during query evaluation (Strohman and Croft 2007; Turtle and Flood 1995). However, these techniques are not directly applicable to the index organizations we explore in this paper since they are focused on increasing the efficiency of one-pass retrieval.

3 Candidates and features

We begin with a more precise specification of the three-stage architecture we consider in this paper, illustrated in Fig. 1. The input to the candidate generation stage is a query Q and the output is a list of k document ids $\{d_1, d_2, \ldots, d_k\}$. In principle, this can be considered a sorted list, but that detail is unimportant here. These document ids serve as input to the feature extraction stage, which returns a list of k feature vectors $\{\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_k\}$, each corresponding to a candidate document. These serve as input to the third document reranking stage, which returns a final ranking of document ids.

Our work primarily focuses on the feature extraction stage (input: list of document ids, output: list of feature vectors). We explore different organizations of document vector indexes and evaluate their impact on efficiency, in terms of speed (more precisely, latency) and memory usage. Throughout this paper, we assume that all data structures necessary for all three stages are held completely in main memory. This is a requirement given real-world operational constraints, and is indeed how modern web search engines work. For example, Jeff Dean has on several occasions publicly disclosed that Google serves indexes from memory. Even in the academic sphere, the capabilities of modern commodity servers suggest that working with standard IR collections in memory is not an unreasonable assumption.

Features for ranking fall into two categories: query independent and query dependent. Examples of query independent features include hyperlink-based scores [PageRank (Page et al. 1999), SALSA (Lempel and Moran 2000), HITS (Kleinberg 1999), etc.,] characteristics of the page (page length, amount of JavaScript, number of particular HTML tags, etc.), and the output of content classifiers (spam score (Cormack et al. 2010), presence of adult content, editorial "quality", etc.). In practice, all query-independent features are computed offline, and so feature extraction is simply a matter of in-memory lookup based on document id.

Query-dependent features, by their very nature, cannot be pre-computed (leaving aside caching, which as we argued in Sect. 2 is an orthogonal issue). Figure 2 provides a summary of the query-dependent features used in our work, which is similar to those used

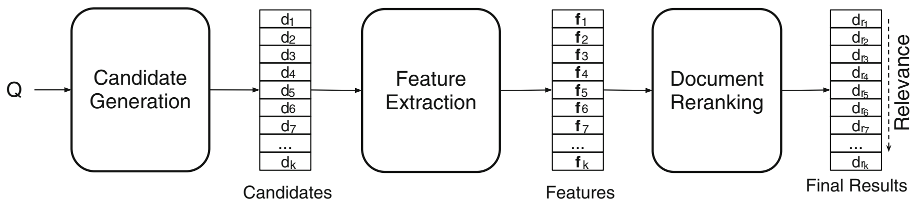

Fig. 1 Illustration of a multi-stage retrieval architecture with distinct candidate generation, feature extraction, and document reranking stages

$$
\begin{aligned}
f_{T,Dir}(q,D) &= \log\left[\frac{\mathrm{tf}(q,D)+\frac{\mu}{|C|}\mathrm{cf}(q)}{|D|+\mu}\right]\\
f_{T,BM25}(q,D) &= \frac{(k_1+1)\cdot\mathrm{tf}(q,D)}{K+\mathrm{tf}(q,D)}\log\left[\frac{N-\mathrm{df}(q)+0.5}{\mathrm{df}(q)+0.5}\right]\\
f_{O,Dir,S}(q_j,q_{j+1},D) &= \log\left[\frac{\mathrm{tf}(\textsc{od}(S,q_j,q_{j+1}),D)+\frac{\mu}{|C|}\mathrm{cf}(\textsc{od}(S,q_j,q_{j+1}))}{|D|+\mu}\right]\\
f_{O,BM25,S}(q_j,q_{j+1},D) &= \frac{(k_1+1)\cdot\mathrm{tf}(\textsc{od}(S,q_j,q_{j+1}),D)}{K+\mathrm{tf}(\textsc{od}(S,q_j,q_{j+1}),D)}\log\left[\frac{N-\mathrm{df}(\textsc{od}(S,q_j,q_{j+1}))+0.5}{\mathrm{df}(\textsc{od}(S,q_j,q_{j+1}))+0.5}\right]\\
f_{U,Dir,S'}(q_j,q_{j+1},D) &= \log\left[\frac{\mathrm{tf}(\textsc{uw}(S',q_j,q_{j+1}),D)+\frac{\mu}{|C|}\mathrm{cf}(\textsc{uw}(S',q_j,q_{j+1}))}{|D|+\mu}\right]\\
f_{U,BM25,S'}(q_j,q_{j+1},D) &= \frac{(k_1+1)\cdot\mathrm{tf}(\textsc{uw}(S',q_j,q_{j+1}),D)}{K+\mathrm{tf}(\textsc{uw}(S',q_j,q_{j+1}),D)}\log\left[\frac{N-\mathrm{df}(\textsc{uw}(S',q_j,q_{j+1}))+0.5}{\mathrm{df}(\textsc{uw}(S',q_j,q_{j+1}))+0.5}\right]
\end{aligned}
$$

Fig. 2 Definition of features used in our model. $\mathrm{tf(e,D)}$ is the count of concept e in D, $\mathrm{df(e)}$ is the document frequency of concept e, $\mathrm{cf(e)}$ is the collection frequency of concept e, where e is defined as follows: q is a query term; $\mathrm{OD(S,q_j,q_{j+1})}$ is an ordered phrase, span of S ($S \in \{0,2,4,8,16\}$); $\mathrm{UW(S',q_j,q_{j+1})}$ is an unordered phrase, span of S' ($S' \in \{2,4,8,16,32\}$). N is the number of documents in the collection; $|D|$ is the length of document D; $|D|'$ is the average document length in the collection; $|C|$ is the total length of the collection; for Dirichlet features, μ is a smoothing parameter; for BM25, $K = k_1[(1-b)-b\cdot(|D|/|D|')]$, and k_1, b are free parameters

in many previous studies (Bendersky et al. 2010; Metzler 2007) (dating back to the original INQUERY query language). We use two families of scoring functions, based on the Dirichlet score from language modeling and BM25. Each family consists of a unigram feature f_T, a bigram proximity feature f_O that takes term order into account (parameterized with a window $S \in \{1,2,4,8,16\}$), and a bigram feature score for unordered terms f_U (parameterized with a window $S' \in \{2,4,8,16,32\}$). In total, there are 22 features.

Of all the features discussed above, term proximity features are the most interesting. They are clearly important for high-quality document ranking, yet are costly to compute (as compared to unigram features). This contrast provides rich opportunities to optimize feature extraction speed. Unigram features are less interesting, since computing them boils down to a sequence of floating point operations. Query-independent features are even less interesting, since it is difficult to optimize simple memory lookups. For this reason, we ignore query-independent features in our document vector representations.[3]

We of course recognize that real-world search engines take advantage of many more features (100s, even 1,000s), but for the reasons given above, this simplified experimental setup retains the important characteristics of the problem we wish to tackle and does not detract from the generalizability of our findings.

As a baseline, we consider the standard PII, using which an algorithm can simultaneously perform candidate generation and feature extraction (input: query, output: list of feature vectors). In more detail, we assume a standard organization of postings lists into blocks, where each block is schematically illustrated as follows:

[3] Admittedly, it might be possible to mask the memory latencies of looking up query-independent features by interleaving memory dispatches with other computations, but we leave such optimizations aside for now.

$$[d_1, d_2, d_3, \ldots][tf_1, tf_2, tf_3, \ldots]$$
$$[p_{1,1}, p_{1,2}, p_{1,3}, \ldots, p_{2,1}, p_{2,2}, p_{2,3}, \ldots]$$

Each block contains n document ids (corresponding to documents containing the term), followed by n term frequencies, followed by the term positions of all the term occurrences. Following best practice, PForDelta (Yan et al. 2009; Zhang et al. 2008; Zukowski et al. 2006) is used for integer coding using 128 (32-bit integers) as the block size (i.e., $n = 128$). The document ids are converted into gaps (i.e., differences between successive values); term frequencies cannot be gap-encoded, and therefore left as is. Term positions are also gap-encoded, relative to the first term position in each document. For example, if in d_1 the term was found at positions [1, 5, 9] and in d_2 the term was found at positions [3, 16], we would code [1, 4, 4, 3, 13]. The term positions can be unambiguously reconstructed from the term frequencies, which provide offsets into the array of term positions. Since the term positions array is likely longer than 128, it is compressed in multiple blocks. Note that this design and associated "bookkeeping" overhead is necessary to take advantage of PForDelta, which is a block-based scheme. During decoding, term positions are only materialized on demand to keep a small cache footprint.

During postings traversal over a PII, it is possible to directly compute all term and term proximity features. Query-independent features would be stored in an in-memory map, keyed by document id, making those features easily accessible. Thus, a single pass suffices to generate candidate documents and extract features.

We compare this single-pass approach, illustrated in Fig. 3a, with an architecture that has distinct candidate generation and feature extraction stages, each using separate data structures. Figure 3b depicts this decoupled approach. In the candidate generation stage, it is possible to use a non-positional index, which is far more compact. In the feature extraction stage, we can rely on a document vector index: a mapping from document ids to document vectors (i.e., a forward index). Of course, this leaves open the question of how the document vectors are compressed and stored, and how features are computed. Alternative designs will be described in the following two sections.

4 Encoding document vectors

The document vector index is the main data structure used for feature generation in our decoupled architecture (see Fig. 3b). It provides low-latency random access to documents vectors by document id, which means that the data structure must be held in memory (in our current implementation, in a large hash table); on disk storage is not feasible. This section describes two approaches to representing the document vectors.

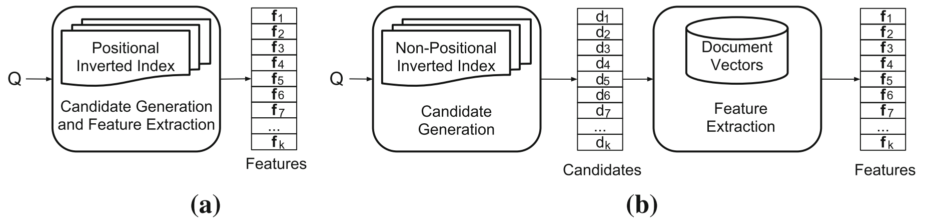
Q
Positional Inverted Index
Candidate Generation and Feature Extraction
f1 f2 f3 f4 f5 f6 f7 ... fk
Features
(a)
Q
Non-Positional Inverted Index
Candidate Generation
d1 d2 d3 d4 d5 d6 d7 ... dk
Candidates
Document Vectors
Feature Extraction
f1 f2 f3 f4 f5 f6 f7 ... fk
Features
(b)

Fig. 3 Two architectures for candidate generation and feature extraction: a single-pass approach with a positional inverted index and a two-stage decoupled approach using a non-positional index and a document vector index. **a** Single-pass architecture, **b** two-stage architecture

To begin, we construct a mapping between terms (strings) and term ids: we make the assumption that the vocabulary has been pre-processed so that term ids are assigned in increasing order of frequency, i.e., more frequent terms receive smaller ids. The first document vector representation is simply a flat array of term ids. Alternatively, we can group the terms by term id and store an indexed document vector representation—essentially a small inverted index for each document. As a working example, Fig. 4 contains a sample document as well as the dictionary that provides term to term id mappings (ties in frequency are broken by first occurrence within the document). Below, we detail both the flat array and the indexed document vector representations

Note that both representations can be extended to store an arbitrary number of query-independent scores. However, these are relatively uninteresting since "feature generation" simply involves a value lookup—i.e., constant-time operations independent of the organization of the document vector. Therefore, to simplify our experiments, we ignore query-independent scores in the document vector. Taking those into account would be equivalent to adding a constant factor in all our experiments, which would not alter the conclusions.

4.1 Flat array of term ids

In the flat array document vector representation, a document D is represented as a list of terms $\{t_{p_1}, t_{p_2}, t_{p_3}, \ldots\}$ such that t_{pi} is the term at position p_i where $1 \leq p_i \leq |D|$. For example, the sample document in Fig. 4 would be represented as follows:

$$[3, 4, 5, 6, 1, 2, 7, 8, 9, 2, 10, 1, 11, 2, 12, 1, 13, 14, 15]$$

This representation leaves open the question of how the term ids are compressed. Note that gap-compression is not appropriate in this case since there is no guarantee that term ids are in ascending order, so we must work with the unmodified term ids. We experimented with three different integer coding techniques: variable-byte coding (VByte) (Scholer et al. 2002), PFor (Zukowski et al. 2006), and our novel document-adaptive hash-based coding scheme. The first two are straightforward, and work reasonably. For VByte, because frequent terms have small term ids, they require fewer bytes to encode. For PFor, most of the term ids are small, so there are relatively few bit overflow exceptions (corresponding to rare terms). Below, we describe our novel document-adaptive hashing scheme for compactly encoding document vectors.

We begin with the observation that although the vocabulary size can be arbitrarily large, in practice each *individual* document will only have a trivial fraction of terms. Therefore, we can define a *document-adaptive* hash function $\mathcal{H}$ that maps the set of terms in a document to a smaller set of ids which take less space to encode. One candidate for $\mathcal{H}$ we considered is the family of minimal perfect hash functions (MPH) (Botelho and Ziviani

Be not afraid of greatness: some are born great, some achieve greatness, and some have greatness thrust upon them.

(a)

Be→3, not→4, afraid→5, of→6, greatness→1, some→2, are→7, born→8, great→9, achieve→10, and→11, have→12, thrust→13, upon→14, them→15

(b)

Fig. 4 A sample document along with the dictionary that provides the term to term id mapping. **a** Sample document, **b** dictionary

2007), i.e., one per document. However, MPH functions require additional space in order to store the hash table, which may not offset the space savings of using MPH to begin with.

We assume that the vocabulary has been pre-processed so that more frequent terms are assigned smaller term ids. This implies that within a particular document, there will be relatively few terms with large term ids. This yields the interesting consequence that lower order bits in the binary representation of term ids play a more important role than the higher order bits in distinguishing between term ids; a similar idea underlies a related technique known as *b*-bit Minwise Hashing (Li and König 2010). In more detail, we break our algorithm down into a few specific cases:

Case 1: On a per-document basis, we can search for a bit window starting from the least significant bit (*lsb*) that is shorter than the original 32-bit integer representation and yields zero collisions (i.e., no two unique term ids in that document will have the same value looking through this window). This window, which in practice is a mask, can be defined by its length ($1 \leq \omega_m \leq 32$). Once such ω_m is found, the algorithm computes the masked values of all term ids, $T' = \{t'_i = \textsc{Mask}(t_i; \omega_m) \mid t_i \in D\}$, where $\textsc{Mask}(x, w) = \textsc{And}(x, 2^w - 1)$.

If the length of the mask ω_m, is less than or equal to a threshold θ_ω, the algorithm terminates and returns $D' = [t'_{p_i} = \textsc{Mask}(t_{p_i}; \omega_m) \mid 1 \leq p_i \leq |D|]$ as the new representation of the document (in other words, we replace term ids with their w low order bits). Alongside the new transformed term ids, we store ω_m as the hash configuration; this fits easily in a byte (and can be viewed as overhead).

Case 2: However, if the mask is not small enough, we search for a window size $\omega_h < \omega_m$ and a function f such that $f : T' \rightarrow R = \{h \in \mathbb{N} \mid 0 < h \leq 2^{\omega_h} - 1\}$ is an injection (i.e., an injective function). To simplify the problem and to reduce the space required to store the function f, we fix f to a simple integer hash function $\mathcal{H}_{int}$ (the same for all documents) and find the smallest $\omega_h < \omega_m$ such that $\mathcal{H}_{int} : T' \rightarrow R$ is:

(a) An injection: we return $[\mathcal{H}_{int}(t'_{p_i}) \mid 1 \leq p_i \leq |D'|]$ as the new document representation. That is, after the hash transformation, the ω_h lowest order bits are unique for all terms in the document. We also store the value of ω_m and ω_h as the hash configuration.

(b) Not an injection but the number of collisions is $<\tau_{collisions}$: Let C be the smallest set of elements in T' such that $\mathcal{H}_{int} : T' - C \rightarrow R$ is an injective function. Assuming $S = \{\mathcal{H}_{int}(t'_i; \omega_h) \mid t'_i \in T', t'_i \notin C\}$, we build a manual hash table for elements in C such that each $c \in C$ is mapped to a unique h where $h \in R$ and $h \notin S$. The manual table for $e \in C$ together with the function $\mathcal{H}_{int}$ for $e \notin C$ form an injective function which can be used to construct the new representation of the document D'. That is, if $t'_{p_i} \notin C$ then use $\mathcal{H}_{int}$ to compute its new value. Otherwise, use the manual table to look up its new value. The hash configuration consists of ω_m, ω_h, as well as the manual hash table which can be compressed and stored efficiently.

Case 3: If no such $\omega_h < \omega_m$ exists, we simply return D' as the new document and ω_m as the hash configuration. This means that in the worst case, all 32 bits are necessary to distinguish the terms ids in a particular document.

In all the above cases, we use PFor to further compress data where appropriate (e.g., transformed term ids are then encoded using PFor). After preliminary experiments, we set θ_ω and $\tau_{collisions}$ to 8 and 20 respectively. Also, the integer hash function used in this work is a simple hash value modulo the length of the range (i.e., 2^ω if ω is the size of the window):

$$\mathcal{H}_{int}(x;\omega) = \text{hash}(x;\omega) \bmod 2^{\omega}$$

where

$$\text{hash}(x;\omega) = (x >> \omega)^{\wedge}(x\,\&\,(2^{\omega}-1)).$$

To process a document given a query, we transform query terms using the hash function stored along with the document vector, performing operations corresponding to the appropriate case above. Once the query terms are in the same space as the transformed document term ids, we can proceed with feature extraction. If the hash configuration has a stored value for ω_m but not ω_h, then we know the algorithm is either in case 1 or case 3; in both cases, the original term ids are masked with ω_m bits and the algorithm terminates. Otherwise, if the collision set is empty, the algorithm computes the hash values for the masked terms using the same integer hash function $\mathcal{H}_{int}$ as the one used in the document transformation phase and the variable ω_h. If the collision set is not empty, on the other hand, hash values are retrieved from the manual table if a query term exists in the collision set. Otherwise, the integer hash function $\mathcal{H}_{int}$ is used to transform the query term id. Note that in this approach, we transform query term ids into the new document term id space, and since queries are usually short, the overhead of this transformation is minimal.

Finally, a few words comparing our document-adaptive hashing scheme to other approaches: although it shares superficial similarities with PFor and PDict (Zukowski et al. 2006), there are a few substantial differences. Our general approach is to replace term ids (or hashed versions thereof) directly with some number of their low order bits—no attempt is made to patch the high order bits (as in PFor). Our goal is not to represent the original term ids directly, but to construct a new term id space that requires fewer bits to store, even with the additional overhead of storing information about the transform itself—for example, if the *b* lowest order bits of all terms in the document are unique, we only need to store the value *b* (case 1 and case 3). If the low order bits are not unique, we apply an integer hash and try again (case 2a). Finally, case 2b deals with collisions even after hashing (multiple terms with the same low order bits). We handle these by storing explicit mappings (in essence, a dictionary), but our overall approach cannot be described as dictionary-based, and hence is different from PDict.

4.2 Indexed document vector

In the alternative representation of document vectors, term occurrences are grouped by term id: this is equivalent to storing a "mini-index" of terms where the postings contain term positions. As an example, for the sample document in Fig. 4, we wish to capture the following information:

$$\begin{aligned}&1 \rightarrow 3[5,12,16]\\&2 \rightarrow 3[6,10,14]\\&3 \rightarrow 1[1]\\&4 \rightarrow 1[2]\\&\ldots\end{aligned}$$

Each line above holds the term id, the term frequency within the document and the term positions where that term is found. Note that we sort term ids in ascending order, so more common terms in the document appear earlier. However, to take advantage of integer compression techniques, the above information is reorganized in the following manner:

$$[t_1, t_2, t_3, \ldots][tf_1, tf_2, tf_3, \ldots]$$
$$[p_{1,1}, p_{1,2}, p_{1,3}, \cdots p_{2,1}, p_{2,2}, p_{2,3}, \cdots]$$

Each document vector begins with an enumeration of terms found in the document (term ids), followed by an array holding term frequencies, followed by the term positions. This structure is exactly the same as the PII, except that we are storing all terms occurrences in a document. These integer sequences are then compressed with PFor ($n = 128$) with gap-based encoding when possible (i.e., "Delta"). Since term ids are sorted in ascending order we can store differences between successive ids. Term frequencies are left unmodified, but term positions are easy to gap-encode, relative to the first occurrence of each term—we encode all positional information from all terms together and use the term frequencies to keep track of where term positions for one term ends and another begins. Putting everything together, the sample document is represented as follows:

$$[1, 1, 1, 1, \ldots]$$
$$[3, 3, 1, 1, \ldots]$$
$$[5, 7, 4|6, 4, 4|1|2, \ldots]$$

First comes the term ids, gap-encoded; next comes the term frequencies; finally, the term positions, also gap-encoded. Note that the boundaries between positions for each term (the vertical bars) are not explicitly stored, since we can reconstruct them from the term frequencies. Each of the sequences above (i.e., each line) is compressed using PFor. Of course, the vertical bars and brackets are not part of the representation, and only exist to aid in presentation.

The indexed document vector representation manifests different tradeoffs compared to the flat array representation. First, we can take advantage of gap-based compression, already detailed above. Second, the amount of decompression required to compute features depends on the query terms: as we decompress the term ids block by block, we need to decompress the term frequencies and the term positions only if we find matching query terms. For queries involving common terms (smaller term ids), this serves as an effective early termination mechanism since term ids are stored in ascending order—we can avoid having to decode the entire document vector. In contrast, for flat arrays the entire document vector needs to be uncompressed, regardless of the query. However, other issues come into play, such as the compression ratio we are able to achieve using either technique and the characteristics of the queries, so it remains an empirical question which approach is more efficient, space- and time-wise.

5 Feature extraction

As previously discussed, we ignore query-independent features in our experiments since they can be modeled as constant factors. Here, we describe how query-dependent features (e.g., those in Fig. 2) are computed. We assume that global term statistics (document length and collection frequency) are available in memory (e.g., stored in the dictionary or a separate lookup table). Depending on the organization of the document vector index, there are two possible algorithms for feature generation:

5.1 Sliding window

In the flat array representation of document vectors, we have a list of terms t_{p_i} at positions 1 to $|D|$, where $|D|$ is the length of document D. From this, generating term or proximity features can be accomplished using a sliding window that makes a single pass over the document. The sliding window starts at position p_i for every $1 \leq p_i \leq |D|$ and spans over the terms at positions $p_i \leq p < p_i + W$, where W is the width of the window. At each position, statistics required to compute the feature are collected (e.g., term frequency). At the end of the pass, the feature value can be computed. Term (unigram) features are simply computed using a sliding window of width one. For $\mathrm{OD}(\mathrm{S}, \mathrm{q_j}, \mathrm{q_{j+1}})$, we set W to $S + 2$, while for $\mathrm{UW}(\mathrm{S'}, \mathrm{q_j}, \mathrm{q_{j+1}}), \mathrm{W} = \mathrm{S'}$. In this approach, each feature requires a separate pass.

5.2 Term postings intersection

For the indexed document vector representation Sect. 4.2, unigram features are simple to compute—it is simply a matter of a few floating point operations based on term statistics. Computation of proximity features can be treated as a postings intersection problem. We use a variant of the *small adaptive* algorithm (Barbay et al. 2006; Demaine et al. 2001). In this approach, the feature extractors are provided with pairs that consist of a query term q_i and a list of positions P_i at which that term appears in the document. Let us illustrate with bigram features for two terms, q_j and q_{j+1}: to compute term frequencies for proximity features, we traverse the positions list associated with the first query term (i.e., P_j) and count the number of positions in P_{j+1} that (relative to the current position from P_j) meet the criteria of a feature's definition f_n. For example, if the current position from P_j is p_k, then we find positions $p'_k \in P_{j+1}$ where

- $p'_k - p_k \leq S + 1,\ p'_k > p_k$, to extract ordered window proximity features of the form $\mathrm{OD}(\mathrm{S}, \mathrm{q_j}, \mathrm{q_{j+1}})$, based on the definition in Metzler and Croft (2004),
- $\begin{cases} p'_k - p_k + 1 \leq S', & p'_k > p_k \\ p_k - p'_k + 1 \leq S', & p_{k-1} < p'_k < p_k \end{cases}$, to extract unordered window proximity features $\mathrm{UW}(\mathrm{S'}, \mathrm{q_j}, \mathrm{q_{j+1}})$ (Metzler and Croft 2004).

This can be generalized to phrases with more than two query terms.

Note that this term postings intersection algorithm can be directly applied to the indexed document vector representation. Alternatively, we can apply this technique to the flat array representation, provided we construct the term positions index "on the fly" (for only the query terms). As it turns out, this approach has some attractive properties, as we describe later.

6 Experimental setup

We performed experiments on the ClueWeb09 collection, which is a best-first web crawl completed by Carnegie Mellon University in 2009. It has been used in several recent TREC evaluations and is available to researchers. The collection contains approximately 1 billion web pages in 10 languages. Of those, 500 million are in English and are divided into 10 roughly equally sized segments. Our experiments focus on the first segment of the English portion of the ClueWeb09 collection (also known as "category B" in the literature). In total, the collection contains 50.2 million pages (1.53 TB uncompressed, 247 GB compressed).[4] For evaluation, we used two different sets of queries: first, the web track queries from TREC 2009–2011 (150 queries total). In addition, we used a randomly-

[4] For clarity, sizes are reported in base 10 units.

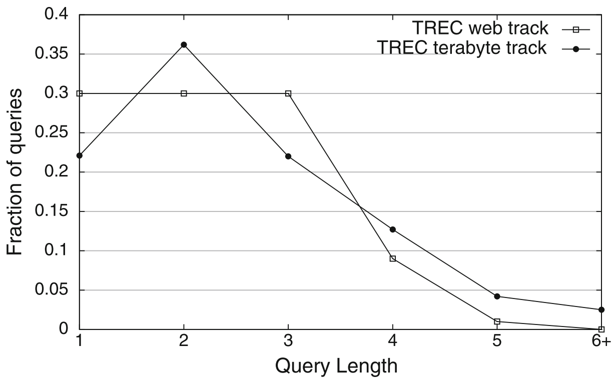

Fig. 5 Query length distribution for the 150 TREC web track queries and a subset of 5,000 queries from the TREC terabyte track

sampled subset of 5,000 queries from the "efficiency topics" of the TREC 2005 terabyte track[5] which contains 50,000 queries overall. We maintained the same length distribution of queries in the smaller sample as the original queries. Figure 5 shows the length distribution of the queries in both test sets.

For candidate generation, we selected the top 10,000 hits that contain all query terms, sorted by a query-independent score. This corresponds to conjunctive query processing (Broder et al. 2003; Matveeva et al. 2006; Tatikonda et al. 2011), which has previously been demonstrated to yield higher early precision (which is important in web retrieval). We made use of the Waterloo spam scores[6] for our query-independent score. These spam scores have been computed for every document in the English portion of the ClueWeb09 collection by Cormack et al. (Cormack et al. 2010) using a simple classifier. These (‰) scores range from 0 to 99, and documents with higher scores are estimated to have higher quality content.

We adapted the small adaptive algorithm proposed by Demaine et al. (2001) to intersect postings lists. This algorithm works as follows: an eliminator e is set to the first document id of the first postings list. It then cycles through the postings lists, searching for the current eliminator (using a one-sided binary search, also known as "galloping" search). If e exists in all postings lists the algorithm adds e to the candidate list and repeats the process for the next eliminator. If a list L does not contain e, the algorithm picks the smallest document id in L that is larger than e as the new eliminator and continues the process. The algorithm terminates when a postings list has no more elements.

There is one more optimization to increase the speed of candidate generation: Since we are only interested in the top k = 10,000 candidates, we sort the documents within a postings list by the query-independent score. Thus, the small adaptive algorithm early terminates when we accumulate k document ids in the candidate set or when a postings list has no unobserved elements. In addition, we do not actually need to keep track of the document-independent score. Both factors serve to increase postings list intersection speed. However, because document ids are no longer guaranteed to be sorted in each postings list, we cannot use gap-based compression techniques for index compression. To resolve this problem, we reassign document ids based on the query-independent score, breaking ties arbitrarily. Smaller ids are assigned to documents with higher scores, and thus sorting documents based on their scores also maintains the increasing document id order needed

[5] http://www-nlpir.nist.gov/projects/terabyte/.

[6] http://plg.uwaterloo.ca/gvcormac/clueweb09spam/.

for gap compression (cf. Silvestri 2007). The upshot of all these optimizations is that our candidate generation algorithm is fast and uses a very compact index (no need to store payloads such as term frequencies). This approach for candidate generation is similar to other setups reported in the literature (e.g., Tatikonda et al. 2011) and provides a good set of working document for downstream processing.

For our baseline comparison using PIIs, we use the same query evaluation algorithm (small adaptive postings intersection to return feature vectors for the top 10,000 hits sorted by spam score). In addition, all the same optimizations are applied: document id reassignment, early-termination, etc. This ensures a fair comparison.

Our implementation of the algorithms presented in this paper is in 100 % pure Java, built on top of the open-source Ivory[7] toolkit. We used LinkedIn's PForDelta implementation provided in the open source Kamikaze package.[8] The choice of Java puts us at a disadvantage compared to, say, implementations in C or C++. This is especially true when measuring short query latencies, since fundamental aspects of Java such as object overhead become a non-trivial fraction of the total running time. Thus, our efficiency numbers should be interpreted with this caveat in mind. Since our task is an intermediate step in an end-to-end retrieval pipeline, the ability to easily integrate our software as a component within a larger system is important. For this goal, Java holds a number of advantages in today's software ecosystem. To provide evidence for these assertions, we point out that Twitter's Earlybird real-time search engine (Busch et al. 2012), which serves over 2 billion queries a day, is written completely in Java. Other organizations such as LinkedIn have also adopted a Java platform for search. Regardless, since all implementations in our experiments are in the same language, the comparison remains fair—were we to reimplement everything in C/C++, the relative results would likely remain the same.

Experiments were performed on a server running Red Hat Linux, with dual Intel Xeon "Westmere" quad-core processors (E5620 2.4 GHz) and 128 GB RAM. This particular architecture has a 64 KB L1 cache per core, split between data and instructions; a 256 KB L2 cache per core; and a 12 MB L3 cache shared by all cores (of a single processor). As previously mentioned, all index structures are held in memory.

7 Results

Since our experiments explored many points in the design space, we begin with a brief description of our notation. The baseline PII. The indexed document vector representation is denoted IDV. The flat array document vector representation is denoted FA, but with two additional suffixes: the first denotes the approach to feature computation (W for sliding window, I for "on-the-fly" indexing); the second refers to the compression technique (H for our novel hashing technique, PFor for PFor, and VB for variable-byte encoding). As an example, an experimental condition might be labeled FA–W(PFor), which refers to sliding window approach on a flat array representation, with PFor compression.

7.1 Speed

Table 1 shows the average query latency for candidate generation and feature extraction, reported in seconds (averaged across 5 trials, with 95 % confidence intervals), for the two

[7] http://ivory.cc.

[8] http://sna-projects.com/kamikaze/.

Table 1 Average time per query in seconds to generate 10,000 candidate documents and extract features for the TREC web and terabyte queries (across 5 trials, with 95 % confidence intervals)

	Web	Terabyte
PII	0.194 (±0.001)	0.198 (±0.005)
IDV	0.173 (±0.001)	0.210 (±0.003)
FA–I(H)	0.180 (±0.001)	0.204 (±0.002)
FA–I(PFor)	0.184 (±0.002)	0.208 (±0.001)
FA–I(VB)	0.539 (±0.003)	0.539 (±0.001)
FA–W(H)	0.953 (±0.001)	0.964 (±0.002)
FA–W(PFor)	0.962 (±0.002)	0.961 (±0.001)
FA–W(VB)	1.324 (±0.017)	1.292 (±0.002)

different sets of queries (web and terabyte tracks). To be precise, query latency is defined as the elapsed time between the arrival of a query until all 22 features described in Sect. 3 are extracted for the set of 10,000 candidate documents. Note that for the PII, both candidate generation and feature extraction are performed together, while in all of the other experimental conditions, the two stages are distinct. Further note that all experimental conditions except for PII share the exact same candidate generation phase. The latency of this phase for the web track queries is 47 ms per query, and for terabyte track queries, 75 ms per query. Feature extraction latency increases linearly with the number of candidate documents and can be tuned in production systems to achieve a desired end-to-end latency. Realistically, we doubt that 10,000 candidates are actually needed to generate a good top 10 ranking for web search; however, a larger candidate pool makes our timings more accurate by reducing variance.

Although absolute values differ due to different compositions of queries in the two different test sets, the results are consistent. Considering end-to-end latency, PII, IDV, and FA–I(H) are about the same speed. For the terabyte queries, the difference between PII and FA–I(H) is not statistically significant (their 95 % confidence intervals overlap), and IDV is slightly slower (albeit significant statistically). For the web track queries, latency between the three conditions are significantly different, but the magnitude of the differences are relatively small. These results show that an architecture in which candidate generation and feature extraction are decoupled does not appear to be slower. It is interesting that IDV (storing indexed document vectors) is not any faster than FA–I(H), which performs indexing on-the-fly (obviously, more "work"). The explanation will become clear below. Overall, we see that the sliding window approach is consistently slower than the on-the-fly indexing approach. For both, it appears that PFor and our document-adaptive hashing technique are comparable in speed, but VByte encoding is substantially slower.

Figure 6 shows the average end-to-end query latency and feature extraction time broken by query length for the two sets of queries. To simplify presentation, we only show the three best techniques: PII, IDV, and FA–I(H). In terms of end-to-end query latencies, IDV and FA–I(H) behave similarly, but PII exhibits different characteristics. PII is about equal in speed with the others for three term queries, but faster on shorter queries and slower on longer queries. This behavior makes sense given the difference between the approaches. Consider single term queries: the query evaluation algorithm for PII simply traverses a single postings list for up to 10,000 postings and terminates (all the data required to compute features are stored in the postings); whereas the other two approaches require traversing two separate data structures and materializing the candidate list of documents. For queries with more terms, however, PII is slower because the index is larger, and hence

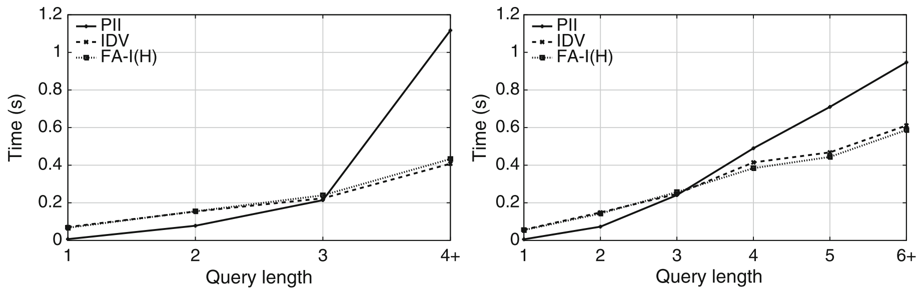

Fig. 6 Average time to generate 10,000 candidate documents and extract features for the web track queries (*left*) and a subset of the terabyte track queries (*right*), broken down by query length

Table 2 Average time to extract features for *each* document (in μs) for TREC web and terabyte queries (across 5 trials, with 95 % confidence intervals)

	Web	Terabyte
IDV	20.6 (±0.2)	40.8 (±4.7)
FA–I(H)	19.5 (±0.2)	28.1 (±0.1)
FA–I(PFor)	19.9 (±0.3)	29.0 (±0.1)
FA–I(VB)	68.6 (±0.1)	99.1 (±0.4)
FA–W(H)	138.5 (±0.2)	217.4 (±0.2)
FA–W(PFor)	139.6 (±0.2)	216.6 (±0.1)
FA–W(VB)	189.7 (±2.4)	286.4 (±0.3)

there is less cache coherence. Since modern architectures fetch entire cache lines at once, compact index structures have an advantage, because fetching a posting very likely means that the next posting is immediately ready for processing. This, however, is less likely to be the case for PII since individual postings are much larger (due to positional information) and may span cache lines. Although it is true that modern processors also perform prefetching, this is dependent on predictable memory access patterns, and with shorter postings (multi-term queries are more likely to have rare terms) there are fewer predictable access patterns for the prefetchers to observe.

In Table 2, we remove the latency from the candidate generation process and compute per-document feature extraction latency (in μs, with 95 % confidence intervals, averaged across 5 trials). To be precise, we first compute the per-document latency for each topic, and then average across all the topics—this is an important detail because not all topics return 10,000 candidate documents (since we are processing queries conjunctively).

For the most part, the per-document feature extraction latencies are consistent with the end-to-end results: the sliding window approach is much slower than the on-the-fly indexing approach. Similarly, our document-adaptive hashing technique is about as fast as PFor, and both are significantly faster than VByte. However, we make an interesting observation: whereas IDV and FA–I(H) are roughly comparable in speed on the web track queries, IDV is much slower for the terabyte queries. Why might this be so?

We believe the answer can be found in Fig. 7, which compares per document feature extraction time for IDV and FA–I(H), broken down in terms of query length. Although both approaches become slower with longer queries, feature extraction latency with IDV grows much faster. Since there are more long queries in the terabyte test set (see Fig. 5), IDV is slower overall compared to FA–I(H). But this only begs the question: why does

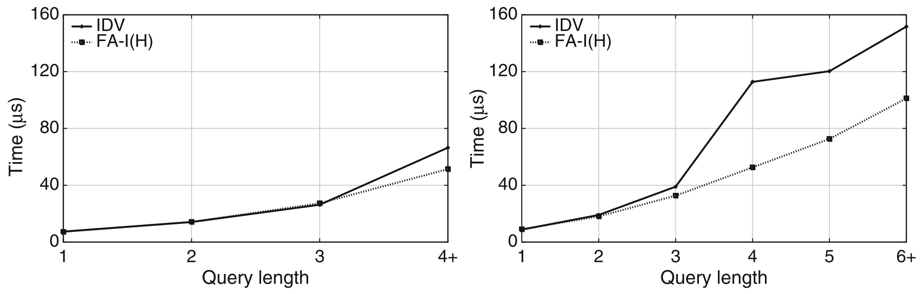

Fig. 7 Comparing IDV and FA–I(H), the per-document average time to extract features for the web track queries (*left*) and the subset of the terabyte track queries (*right*), broken down by query length

IDV's performance degrade so severely in this manner? The explanation lies in how much PFor decoding needs to be performed in each of the techniques. For the flat array representation, we need to decode the entire document, no matter what (as many PFor blocks as is necessary to hold an array of integers whose size is the document length). For the IDV technique, recall that the term ids are assigned in order of decreasing (global) frequency and encoded in ascending term id order to take advantage of gap encoding. To perform feature extraction, we decode as much of the document vector as is necessary to reconstruct the term positions for all query terms. At the minimum we need to decode three PFor blocks: one for the term id, one for the term frequencies, and one for the term positions. Since shorter queries tend to have more frequently-occurring terms, their term ids are more likely to be found in the earlier blocks, and so in many cases it is likely that no further decompression is necessary. However, longer queries tend to involve more rare terms, which require decoding additional blocks—and because the term ids, term frequencies, and term positions are stored separately (they have to be), processing each "increment" of the document vector requires at least decoding three PFor blocks (recall that positional information can take up multiple blocks).

Put another way, the relevant comparison highlighted in Fig. 7 is between the amount of time it takes to *decode* stored term positions (in multiple PFor blocks) versus the time required to *reconstruct* term positions on the fly from the flat array representation. Results show that for queries up to three terms in length, the speed of the two are comparable, but for longer queries, it is actually faster to reconstruct the postings for the query terms as needed (and pay the cost to decode the flat array of term ids every time).

The above analysis assumes extraction of all 22 f_T, f_O, and f_U features (including different window sizes). However, it may be the case that a machine-learned ranking algorithm only uses a subset of those features. What is the impact of computing different individual features? This question is answered by Table 3, which shows per-document feature extraction latency for different feature classes. In the first "block" of the table, we show latencies for computing just the BM25 unigram feature, the BM25 unigram feature and bigram features, and the BM25 unigram feature, bigram, and unordered bigram features. The second "block" of the table shows the same, expect with the Dirichlet features. In the third "block", both sets of features are computed (i.e., 2, 4, and 6 features, respectively). In all cases, we only use one window width for f_O and f_U.

Comparing Table 2 with Table 3 reveals some interesting observations. For the IDV approach and the FA–I($\cdot$) on-the-fly indexing approaches (see Table 3a), feature extraction

Table 3 Average time per document in microseconds (µs) to extract different sets of features for the TREC web track queries across 5 trials, with 95 % confidence intervals

	IDV	FA–I(H)	FA–I(PFor)	FA–I(VB)
(a) Indexed document vector versus "on the fly" indexing with flat arrays				
$f_{T,BM25}$	14.2 (±0.3)	16.4 (±0.2)	16.5 (±0.2)	66.3 (±1.5)
$f_{\{T,O^1\},BM25}$	14.3 (±0.2)	16.5 (±0.2)	17.2 (±0.1)	65.6 (±0.3)
$f_{\{T,O^1,U^1\},BM25}$	14.8 (±0.2)	16.6 (±0.1)	17.0 (±0.3)	65.9 (±0.1)
$f_{T,Dir}$	13.9 (±0.1)	16.5 (±0.2)	16.8 (±0.3)	65.6 (±0.1)
$f_{\{T,O^1\},Dir}$	14.5 (±0.3)	16.3 (±0.2)	17.0 (±0.3)	65.6 (±0.3)
$f_{\{T,O^1,U^1\},Dir}$	14.9 (±0.2)	16.8 (±0.2)	17.2 (±0.3)	66.7 (±1.2)
$f_{T,\{BM25,\ Dir\}}$	14.1 (±0.3)	16.5 (±0.1)	17.0 (±0.3)	65.7 (±0.1)
$f_{\{T,O^1\},\{BM25,Dir\}}$	14.8 (±0.3)	16.8 (±0.1)	17.3 (±0.1)	66.1 (±0.1)
$f_{\{T,O^1,U^1\},\{BM25,Dir\}}$	16.0 (±0.4)	17.2 (±0.1)	17.7 (±0.1)	66.5 (±0.2)
	IDV	FA–W(H)	FA–W(PFor)	FA–W(VB)
(b) Indexed document vector versus sliding windows with flat array				
$f_{T,\ BM25}$	14.2 (±0.3)	15.1 (±0.1)	15.9 (±0.1)	64.9 (±0.6)
$f_{\{T,O^1\},BM25}$	14.3 (±0.2)	18.7 (±0.1)	19.5 (±0.3)	68.8 (±0.5)
$f_{\{T,O^1,U^1\},BM25}$	14.8 (±0.2)	26.6 (±0.3)	27.3 (±0.3)	76.0 (±0.2)
$f_{T,\ Dir}$	13.9 (±0.1)	15.2 (±0.1)	16.0 (±0.1)	64.8 (±0.1)
$f_{\{T,O^1\},Dir}$	14.5 (±0.3)	18.8 (±0.1)	19.6 (±0.1)	69.4 (±0.1)
$f_{\{T,O^1,U^1\},Dir}$	14.9 (±0.2)	26.6 (±0.2)	27.4 (±0.4)	76.3 (±0.2)
$f_{T,\ \{BM25,\ Dir\}}$	14.1 (±0.3)	18.4 (±0.2)	19.1 (±0.2)	67.7 (±0.1)
$f_{\{T,O^1\},\{BM25,Dir\}}$	14.8 (±0.3)	27.3 (±0.1)	28.3 (±0.1)	77.3 (±0.1)
$f_{\{T,O^1,U^1\},\{BM25,Dir\}}$	16.0 (±0.4)	39.9 (±0.3)	40.7 (±0.1)	89.6 (±0.1)

time is relatively insensitive with respect to the number of features computed. For example, computing *only* BM25 unigram features takes 14.2 µs, whereas as computing *all* features takes 20.6 µs. Looking at Table 3a, we see relatively few significant differences in running times, despite the different number of features being computed. This is because the time required to access the raw term data (from memory) necessary to compute *any* feature dominates the marginal cost (in terms of floating point operations) of computing additional features—this is consistent with what we know about modern processor architectures (where memory latency is usually the biggest bottleneck). In contrast, the sliding window FA–W(·) approaches (see Table 3b) appears to become slower with more features, since it requires a pass through the document vector for each feature.

7.2 Index size

Having examined speed (latency), we now turn our attention to index size, which corresponds to memory footprint since we hold all index structures in memory. These results are shown in Fig. 8, for the first segment of ClueWeb09 (50.2 m documents). The rightmost bar shows the size of the baseline condition, the PII, which takes up 75 GB. For the other techniques, we require two separate data structures: the inverted index and the document vector index. Note that the inverted index is optimized specifically for candidate generation, and is therefore very compact (14 GB) since it stores no positional information.

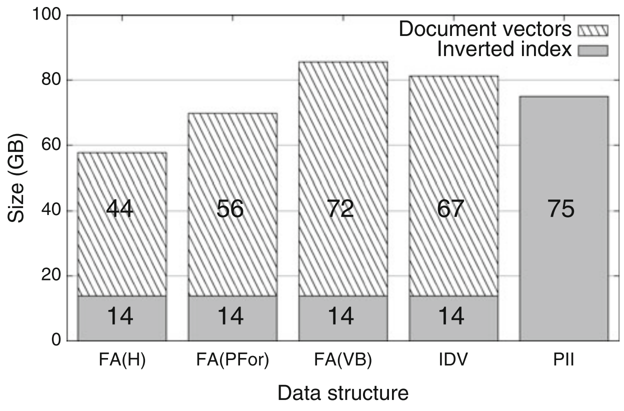

Fig. 8 Index size under various experimental conditions. Note that the decoupled architecture requires two separate data structures: a (non-positional) inverted index and a document vector index, whereas the single-pass approach requires only a positional inverted index (PII)

Table 4 For FA(H), $\theta_\omega = 8$ and $\tau_{collisions} = 20$: the percentage (p) of document vectors broken down by various hash configurations, the mean compression ratio compared to an uncompressed array representation, and the mean compression ratio compared to FA(PFor)

	p (%)	Versus raw	Versus FA(PFor)
Case 1: $\omega_m \leq \theta_\omega$	0.9	0.715	0.812
Case 2(a): $\omega_m > \theta_\omega,\ \omega_h < \omega_m,\ \lvert C\rvert = 0$	0.6	0.348	0.655
Case 2(b): $\omega_m > \theta_\omega,\ \omega_h < \omega_m,\ 0 < \lvert C\rvert \leq \tau_{collisions}$	98.2	0.371	0.741
Case 3: $\omega_m > \theta_\omega,\ \omega_h \geq \omega_m$	0.3	0.579	0.986
	100	0.374	0.742

$\lvert C\rvert$ is the number of collisions

Different organizations of the document vector index translates into substantial size differences. We see that the IDV and FA–I(VB) approaches result in the largest memory footprints (67 and 72 GB, respectively). Examining FA–I(H) and FA–I(PFor), results in the previous section show that both approaches are comparable in terms of speed; however, our novel document-adaptive hashing scheme is preferred due to a smaller index footprint (in fact, smallest overall). While IDV and FA–I(H) are comparable in terms of speed, the latter is preferred due to smaller memory requirements. The smaller index size also contributes to the speed of FA–I(H): the additional work of reconstructing term positions on the fly is balanced against smaller data structures.

As expected, the flat array representation encoded with our document-adaptive hashing scheme achieves the lowest compression ratio (i.e., compressed size over uncompressed size). Recall that the algorithm breaks down into four separate cases; for our setting ($\theta_\omega = 8$ and $\tau_{collisions} = 20$), statistics of the four different cases are shown in Table 4. Nearly all documents (98.2 %) fall under case 2(b), with hashed term ids and $\leq$20 collisions. In the third column of the table, we show compression ratio with respect to uncompressed representations, and in the fourth column, with respect to compression using PFor. In case 2(b), for example, our scheme saves 62.9 % space compared to an uncompressed encoding and 25.9 % over PFor. Surprisingly, case 1 (masking low order bits of term id) does not work as well as expected: it rarely applies, and when it does, only for short document vectors, in which case the overhead needed to store the hash configuration and bitmask information becomes significant in comparison. Note that the parameter settings reported here reflect results of preliminary explorations of the parameter space, but we made no attempt to find the optimal setting.

8 Discussion

Overall, how does the "traditional" PII, in which candidate generation and feature extraction are performed simultaneously, compare to an architecture in which the two are decoupled, e.g., FA–I(H), the best variant of the ones we examined? Empirical results give FA–I(H) a slight edge: overall, it is about as fast as PII on a realistic query load, but requires less memory. While it is true the PII is faster on shorter queries (especially single-term queries), there are two additional factors to consider: shorter queries are likely to occur more frequently, and therefore results are more likely to be cached. Shorter queries are also more likely to be navigational in nature, and often benefit from special treatment (i.e., akin to a separate vertical). In a production system, they may be handled in a completely separate module, in which the tradeoffs are different than those factors considered here.

Beyond the directly measurable metrics, we argue that the decoupled architecture has other advantages as well, the biggest of which is flexibility. Previous work (Sect. 2) has shown that candidate generation only needs to be "good enough", as modern machine-learned techniques are quite powerful; on the other hand, more and richer features improve effectiveness. In our decoupled architecture, feature engineering does not need to involve the first (candidate generation) stage, thus reducing system-level changes needed for experimentation. Incorporating richer features, for example, named-entity tags, markup attributes, multiple text fields, etc. is easier to accomplish in the document vector index than in the inverted index. Augmenting the document vector representation is an embarrassingly parallel problem, whereas indexing requires document–term inversion. Furthermore, as the size of the inverted index increases from indexing richer features, it will become slower due to decreased cache coherence, as noted in the experimental results. Note that abstractly speaking, the document vector index is basically a key-value store (if we treat the compressed document vectors as opaque blobs): although our current implementation is simply a large hash table, we have plenty of choices for scalable implementations in a production environment, ranging from well-known solutions such as memcached to a new breed of so-called "NoSQL" datastores such as HBase [an open source clone of Bigtable (Chang et al. 2006)], Cassandra [an open source clone of Dynamo (DeCandia et al. 2007)], as well as a plethora of other offerings.

In an architecture where candidate generation and feature extraction are decoupled, there is in principle no reason why both stages need to run in the same process space, or even on the same machine, for that matter. This fits well with modern web architectures, where each rendering of a page involves dozens or even hundreds of individual services. From a management point of view, fine-grained service decomposition makes monitoring, performance profiling, and load balancing easier.

Finally, we note that in production systems, there must be some service for snippet generation (i.e., the keyword-in-context summary search results that are displayed to the user). Our document vector indexes can be easily augmented to hold such information without any architectural changes (e.g., as another field). In fact, without stemming, and using PFor for compression, the flat array document vector representation can be directly used for snippet generation. With a positional inverted index, in contrast, some other document-lookup service is required to generate snippets, further increasing the overall memory footprint of the system.

9 Conclusion

Although the three-stage retrieval architecture explored in this work has been documented in the literature, it has received relatively little attention. Much of the focus of IR research in recent years has been on ranking models using machine learning techniques, and it appears that the community has gained much experience in tackling this challenge (some would go as far to say that "learning to rank" is a solved problem). However, we believe that the architectural implications of learning to rank have not been thoroughly studied: how to organize index structures and query evaluation algorithms in support of these machine-learned rankers. This paper takes a first step, focusing on the problem of fast feature extraction, but the broader design space for retrieval systems remains underexplored.

Acknowledgments This work has been supported in part by NSF under awards IIS-0916043, IIS-1144034, and IIS-1218043. Any opinions, findings, conclusions, or recommendations expressed are the authors' and do not necessarily reflect those of the sponsors. The first author's deepest gratitude goes to Katherine, for her invaluable encouragement and wholehearted support. The second author is grateful to Esther and Kiri for their loving support and dedicates this work to Joshua and Jacob.

References

Altingovde, I. S., Ozcan, R., & Ulusoy, Ö. (2012). Static index pruning in web search engines: Combining term and document popularities with query views. *ACM Transactions on Information Systems, 30*(1), 2.

Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., & Silvestri, F. (2007). The impact of caching on search engines. In *Proceedings of the 30th annual international ACM SIGIR conference on research and development in information retrieval* (*SIGIR 2007*), pp. 183–190, Amsterdam, The Netherlands.

Barbay, J., López-Ortiz, A., & Lu, T. (2006). Faster adaptive set intersections for text searching. In *Proceedings of the 5th international workshop on experimental algorithms* (*WEA 2006*), pp. 146–157.

Bendersky, M., Metzler, D., & Croft, W. B. (2010). Learning concept importance using a weighted dependence model. In *Proceedings of the 3rd ACM international conference on web search and data mining* (*WSDM 2010*), pp. 31–40, New York, NY, USA.

Botelho, F. C., & Ziviani, N. (2007). External perfect hashing for very large key sets. In *Proceedings of the 16th ACM conference on information and knowledge management* (*CIKM 2007*), pp. 653–662, Lisbon, Portugal.

Broder, A., Carmel, D., Herscovici, M., Soffer, A., & Zien, J. (2003). Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 12th international conference on information and knowledge management* (*CIKM 2003*), pp. 426–434, New Orleans, LA, USA.

Broschart, A., & Schenkel, R. (2012). High-performance processing of text queries with tunable pruned term and term pair indexes. *ACM Transactions on Information Systems, 30*(1), 5.

Burges, C. (2010). *From ranknet to lambdarank to lambdamart: An overview*. Technical report MSR-TR-2010-82, Microsoft Research.

Busch, M., Gade, K., Larson, B., Lok, P., Luckenbill, S., & Lin, J. (2012). Earlybird: Real-time search at twitter. In *Proceedings of the 28th international conference on data engineering* (*ICDE 2012*), pp. 1360–1369, Washington, DC, USA

Büttcher, S., Clarke, C., & Lushman, B. (2006). Term proximity scoring for ad-hoc retrieval on very large text collections. In *Proceedings of the 29th annual international ACM SIGIR conference on research and development in information retrieval* (*SIGIR 2006*), pp. 621–622, Seattle, Washington, DC, USA

Cambazoglu, B. B., & Baeza-Yates, R. (2011). Scalability challenges in web search engines. In M. Melucci & R. Baeza-Yates (Eds.), *Advanced topics in information retrieval* (pp. 27–50). New York: Springer.

Cambazoglu, B. B., Zaragoza, H., Chapelle, O., Chen, J., Liao, C., Zheng, Z., et al. (2010) Early exit optimizations for additive machine learned ranking systems. In *Proceedings of the 3rd ACM international conference on web search and data mining* (*WSDM 2010*), pp. 411–420, New York, NY, USA.

Carmel, D., Cohen, D., Fagin, R., Farchi, E., Herscovici, M., Maarek, Y., et al. (2001). Static index pruning for information retrieval systems. In *Proceedings of the 24th annual international ACM SIGIR conference on research and development in information retrieval* (*SIGIR 2001*), pp. 43–50, New Orleans, LA, USA.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., et al. (2006). Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX symposium on operating system design and implementation* (*OSDI 2006*), pp. 205–218, Seattle, Washington, DC, USA.

Clarke, C. L. A., Cormack, G. V., & Burkowski, F. J. (1995). Shortest substring ranking (MultiText experiments for TREC-4). In *Proceedings of 4th text retrieval conference* (*TREC 1995*), pp. 295–304, Gaithersburg, MD, USA.

Cormack, G., Smucker, M., Clarke, C. (2010). Efficient and effective spam filtering and re-ranking for large web datasets. *Journal of Information Retrieval*.

Croft, W. B., Turtle, H. R., & Lewis, D. D. (1991). The use of phrases and structured queries in information retrieval. In *Proceedings of the 14th annual international ACM SIGIR conference on research and development in information retrieval* (*SIGIR 1991*), pp. 32–45, Chicago, IL, USA.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., et al. (2007). Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM SIGOPS symposium on operating systems principles* (*SOSP 2007*), pp. 205–220, Stevenson, Washington, DC, USA.

Demaine, E., López-Ortiz, A., & Munro, J. (2001). Experiments on adaptive set intersections for text retrieval systems. In *Revised papers from the 3rd international workshop on algorithm engineering and experimentation* (*ALENEX 2001*), pp. 91–104.

Ding, S., & Suel, T. (2011). Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th annual international ACM SIGIR conference on research and development in information retrieval* (*SIGIR 2011*), pp. 993–1002, Beijing, China.

Fagan, J. (1987). *Experiments in automatic phrase indexing for document retrieval: A comparison of syntactic and non-syntactic methods*. Technical report TR87-868, Cornell.

Ganjisaffar, Y., Caruana, R., & Lopes, C. (2011). Bagging gradient-boosted trees for high precision, low variance ranking models. In *Proceedings of the 34th annual international ACM SIGIR conference on research and development in information retrieval* (*SIGIR 2011*), pp. 85–94, Beijing, China.

Hawking, D., & Thistlewaite, P. (1995). Proximity operators—So near and yet so far. In *Proceedings of the 4th text retrieval conference* (*TREC 1995*), pp. 131–144, Gaithersburg, MD, USA.

Kleinberg, J. M. (1999). Authoritative sources in a hyperlinked environment. *Journal of the ACM, 46*(5), 604–632.

Lempel, R., & Moran, S. (2000). The stochastic approach for link-structure analysis (SALSA) and the TKC effect. *Computer Networks, 33*(1–6), 387–401.

Li, H. (2011). *Learning to rank for information retrieval and natural language processing*. Morgan & Claypool.

Li, P., & König, A. C. (2010). b-bit minwise hashing. In *Proceedings of the 19th international conference on world wide web* (*WWW 2010*), pp. 671–680, Raleigh, NC, USA.

Long, X., & Suel, T. (2005). Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th international world wide web conference* (*WWW 2005*), pp. 257–266, Chiba, Japan.

Matveeva, I., Burges, C., Burkard, T., Laucius, A., & Wong, L. (2006). High accuracy retrieval with multiple nested ranker. In *Proceedings of the 29th annual international ACM SIGIR conference on research and development in information retrieval* (*SIGIR 2006*), pp. 437–444, Seattle, Washington, DC, USA.

Metzler, D. (2007). Automatic feature selection in the Markov random field model for information retrieval. In *Proceedings of the 16th acm conference on information and knowledge management* (*CIKM 2007*), pp. 253–262, Lisbon, Portugal.

Metzler, D., & Croft, W. B. (2004). Combining the language model and inference network approaches to retrieval. *Information Processing and Management, 40*(5), 735–750. doi:10.1016/j.ipm.2004.05.001.

Ntoulas, A., & Cho, J. (2007). Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th annual international ACM SIGIR conference on research and development in information retrieval* (*SIGIR 2007*), pp. 191–198, Amsterdam, The Netherlands.

Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). *The PageRank citation ranking: Bringing order to the web*. Stanford Digital Library Working Paper SIDL-WP-1999-0120, Stanford University.

Scholer, F., Williams, H. E., Yiannis, J., & Zobel, J. (2002). Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th annual international ACM SIGIR conference on research and development in information retrieval* (*SIGIR 2002*), pp. 222–229, Tampere, Finland.

Silvestri, F. (2007). Sorting out the document identifier assignment problem. In *Proceedings of the 29th European conference on information retrieval research* (*ECIR 2007*), pp. 101–112, Rome, Italy.

Strohman, T., & Croft, W. (2007). Efficient document retrieval in main memory. In *Proceedings of the 30th annual international ACM SIGIR conference on research and development in information retrieval* (*SIGIR 2007*), pp. 175–182, Amsterdam, The Netherlands.

Tao, T., & Zhai, C. (2007). An exploration of proximity measures in information retrieval. In *Proceedings of the 30th annual international ACM SIGIR conference on research and development in information retrieval* (*SIGIR 2007*), pp. 295–302, Amsterdam, The Netherlands.

Tatikonda, S., Cambazoglu, B. B., & Junqueira, F. (2011). Posting list intersection on multicore architectures. In *Proceedings of the 34th annual international ACM SIGIR conference on research and development in information retrieval* (*SIGIR 2011*), pp. 963–972, Beijing, China.

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

Wang, L., Lin, J., & Metzler, D. (2011). A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th annual international ACM SIGIR conference on research and development in information retrieval* (*SIGIR 2011*), pp. 105–114, Beijing, China.

Williams, H. E., Zobel, J., & Bahle, D. (2004). Fast phrase querying with combined indexes. *ACM Transactions on Information Systems, 22*(4), 573–594.

Yan, H., Ding, S., & Suel, T. (2009). Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on world wide web* (*WWW 2009*), pp. 401–410, Madrid, Spain.

Zhang, J., Long, X., & Suel, T. (2008). Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on world wide web* (*WWW 2008*), pp. 387–396, Beijing, China

Zukowski, M., Heman, S., Nes, N., & Boncz, P. (2006). Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd international conference on data engineering* (*ICDE 2006*).