Fast Candidate Generation for Real-Time Tweet Search with Bloom Filter Chains

NIMA ASADI and JIMMY LIN, University of Maryland at College Park

The rise of social media and other forms of user-generated content have created the demand for real-time search: against a high-velocity stream of incoming documents, users desire a list of relevant results at the time the query is issued. In the context of real-time search on tweets, this work explores candidate generation in a two-stage retrieval architecture where an initial list of results is processed by a second-stage rescorer to produce the final output. We introduce Bloom filter chains, a novel extension of Bloom filters that can dynamically expand to efficiently represent an arbitrarily long and growing list of monotonically-increasing integers with a constant false positive rate. Using a collection of Bloom filter chains, a novel approximate candidate generation algorithm called BWAND is able to perform both conjunctive and disjunctive retrieval. Experiments show that our algorithm is many times faster than competitive baselines and that this increased performance does not require sacrificing end-to-end effectiveness. Our results empirically characterize the trade-off space defined by output quality, query evaluation speed, and memory footprint for this particular search architecture.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Scalability, efficiency, top-*k* retrieval, tweet search, bloom filters

ACM Reference Format:
Asadi, N. and Lin, J. 2013. Fast candidate generation for real-time tweet search with bloom filter chains. ACM Trans. Inf. Syst. 31, 3, Article 13 (July 2013), 36 pages.
DOI: http://dx.doi.org/10.1145/2493175.2493178

1. INTRODUCTION

This article focuses on real-time search in the context of Twitter, a communications platform on which users can send short, 140-character messages, called *tweets*, to their *followers*, which are other users who subscribe to those messages. Conversely, users can receive tweets from people they follow via a number of mechanisms, including Web clients, mobile clients, and SMS. As of Winter 2012, Twitter had over 200 million active users worldwide who collectively post over 400 million tweets per day—translating into an average of several thousand tweets per second, with short bursts of activity at even higher velocities. One salient aspect of Twitter is that users demand to know what is happening right now, especially in response to breaking news stories around the world, such as natural disasters, celebrity deaths, or mass protests. For this, they desire real-time search capabilities.

This work has been supported by NSF under awards IIS-0916043, IIS-1144034, and IIS-1218043.
Authors' addresses: N. Asadi, Department of Computer Science, University of Maryland at College Park; email: nima@cs.umd.edu; J. Lin, The iSchool, College of Information Studies, University of Maryland at College Park; email: jimmylin@umd.edu.

DOI: http://dx.doi.org/10.1145/2493175.2493178

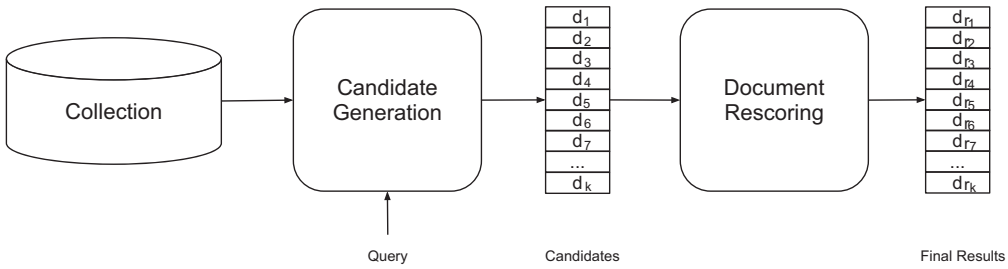


Fig. 1. A two-stage retrieval architecture for real-time search: given query Q, the first stage generates candidates; the second stage rescores the candidates to compose the final results.

Let us begin by more precisely defining the real-time search task. We assume a stream of documents being created at a high velocity, which in our case consists of tweets.[1] Against this stream of incoming documents, the user issues a query representing an information need and desires documents that meet the need at that time, taking into account both traditional relevance criteria, such as topicality, as well as temporality—in general, more recent documents are preferred over older documents. In terms of result presentation, whether tweets are displayed in reverse chronological order (as in Twitter's current search interface) or arranged in relevance order (as in a traditional Web search engine) is primarily an interface issue that can be decoupled from the underlying scoring algorithms. This formulation matches the microblog search task in recent TREC evaluations [Ounis et al. 2011, Soboroff et al. 2012a, 2012b; McCreadie et al. 2012] and represents an initial if not emerging consensus on how the real-time search problem should be framed.

We envision a real-time search architecture comprising two distinct stages: candidate generation and document rescoring (see Figure 1). In the first stage, a list of candidate documents is generated, sorted using a simple scoring function. In the second stage, these candidate documents are then rescored by a separate component. Mirroring the basic architecture of Web search engines today, we assume that candidate generation is cheap and fast and that the second stage uses more sophisticated algorithms (e.g., machine-learned models operating over a panoply of features).

Within this real-time search architecture, our work focuses on the candidate generation stage, where we believe that speed is the most important consideration, even more so than output quality. In a two-stage retrieval architecture, candidate generation only needs to be "good enough", since the second-stage rescorer is able to identify relevant documents by exploiting effective machine-learned models and rich features. Furthermore, due to an emphasis on early precision in most search tasks today—we simply need to make sure there are enough relevant documents for the second-stage rescorer to identify. As such, our goal is to generate candidate documents as quickly as possible, even at the cost of introducing approximations and sacrificing some (component-level) effectiveness. However, as we experimentally show, we can achieve a multiple-fold increase in speed without sacrificing end-to-end effectiveness.

Building on a previous short paper [Asadi and Lin 2012b], we achieve fast candidate generation by introducing a novel variant of Bloom filters, which are probabilistic data structures that support approximate membership tests. In our indexing algorithm, postings are inserted into dynamically-allocated in-memory inverted lists in chronological order with increasing document ids. In addition, postings are stored in Bloom

[1]Following standard IR parlance, we refer to the unit of indexing generically as a "document", even though in actuality it may be a tweet, a webpage, a blog post, etc.

filter chains, a novel data structure we propose. We introduce a variant of the WAND query evaluation algorithm [Broder et al. 2003; Ding and Suel 2011], which we call BWAND, that takes advantage of the Bloom filter chains to either perform postings intersection (conjunctive query processing) or top k retrieval (disjunctive query processing). Experiments show that our candidate generation algorithm is several times faster than competitive baselines, and that in terms of end-to-end effectiveness with a second-stage rescorer, our algorithm produces NDCG and precision scores that are statistically indistinguishable from the much slower exact algorithms. The trade-off is that our approach requires additional memory to hold the Bloom filter auxiliary data structures, but we show that the requirements are reasonable given modern server configurations.

Contributions. We view this work as having three contributions.

(1) We propose a novel variant of Bloom filters called Bloom filter chains that can dynamically expand to efficiently represent an arbitrarily long and growing list of monotonically-increasing integers with a constant false positive rate. We describe how to organize a large collection of Bloom filter chains in a compact data structure capable of supporting two primary operations: INSERT for element insertion and PROBE for membership tests.
(2) We describe a novel algorithm for candidate generation that takes advantage of Bloom filter chains in the context of real-time search on tweets. Our algorithm, called BWAND (short for Bloom WAND) introduces a number of optimizations that take advantage of characteristics specific to tweets. Experiments show that our algorithm is several times faster than competitive baselines.
(3) In the context of candidate generation in a two-stage retrieval architecture for real-time search on tweets, we empirically characterize the trade-off space defined by output quality, query evaluation speed, and memory footprint. Our experimental results help us better understand the various contributions to important design aspects of retrieval engines.

2. BACKGROUND AND RELATED WORK

This section sets the stage for our work in several steps, since it builds on and draws from diverse threads of research. We begin by discussing the standard two-stage ranking architecture for Web search that follows from modern learning-to-rank techniques. Within this architecture, one fundamental design choice is whether queries are processed conjunctively or disjunctively: we overview both approaches.

Next, we discuss how the problem of real-time search differs from traditional Web search. These differences hold important implications for the design of retrieval engines, the most significant of which is the need to make documents immediately searchable. As a reference design, we describe the salient aspects of Earlybird, Twitter's production real-time retrieval engine, which forms the starting point of this work. We then outline our previous idea of how Bloom filters can be exploited for efficient postings intersection, which we extend into our BWAND algorithm.

2.1. The Two-Stage Architecture for Web Search

There is consensus in the information retrieval community that the challenge of Web ranking is best addressed using machine-learning techniques, known as *learning to rank*. In particular, ensembles of tree-based learners have proven effective, as documented in both the academic literature [Liu 2009; Li 2011; Ganjisaffar et al. 2011] and in production in commercial search engines such as Bing [Burges 2010]. This approach generally assumes that a candidate list of potentially-relevant documents has already

been gathered by other means.[2] Thus, learning to rank is actually a *reranking* (or equivalently, rescoring) problem [Liu 2009]. Hence, modern Web search can be viewed as a two-stage process: candidate generation followed by reranking [Tatikonda et al. 2011; Ding and Suel 2011; Asadi and Lin 2012b; Macdonald et al. 2012; Tonellotto et al. 2013]. In the first phase, a fast, cheap algorithm is applied to generate a candidate list of potentially-relevant documents. A popular choice is to use BM25 [Liu 2009; Cambazoglu et al. 2010; Tatikonda et al. 2011; Macdonald et al. 2012], possibly in combination with a query-independent score, for example, PageRank [Page et al. 1999], HITS [Kleinberg 1999], SALSA [Lempel and Moran 2000], a page quality score [Cormack et al. 2011], etc. These candidate documents are then reranked by a slower, expensive but higher-quality (usually, machine-learned) algorithm, which typically considers features that would be too costly to compute in the first phase (e.g., term proximity features). A few examples include gradient-boosted regression trees [Burges 2010; Ganjisaffar et al. 2011], additive ensembles [Cambazoglu et al. 2010], and cascades of rankers [Matveeva et al. 2006; Wang et al. 2011]. Macdonald et al. [2012] recently studied this two-stage architecture in detail, examining issues such as the size of the candidate list and metrics for training learning-to-rank models in the Web context. Our work also adopts this architecture, focusing on candidate generation for real-time tweet search.

Candidate generation is usually accomplished with document-sorted inverted indexes [Zobel and Moffat 2006]. A fundamental distinction of retrieval algorithms is whether queries are processed conjunctively or disjunctively. In conjunctive query processing, only documents that have all the query terms are considered—in other words, the query terms are AND'ed together. For Web-scale collections, there is evidence that this approach leads to higher early precision [Broder et al. 2003] (although this work predates much of the modern work on learning to rank). Conjunctive query processing is equivalent to intersection of postings lists [Demaine et al. 2001; Barbay et al. 2006; Tsirogiannis et al. 2009; Culpepper and Moffat 2010; Tatikonda et al. 2011], with well-known solutions such as the small adaptive and SvS algorithms. Conjunctive query processing is very fast for a few reasons. The algorithms generally eschew a scoring model, so there is no need to compute document scores or to maintain a heap of scored documents. Many researchers have studied the problem, and thus the algorithms have received much attention and refinement. Finally, index structures for conjunctive query processing are smaller since there is no need to store payloads such as term frequencies; this decreases the amount of data that need to be processed during query evaluation.

The alternative to conjunctive query processing is disjunctive query processing, often referred to as top k retrieval. In this approach, a document that contains any query term is potentially a candidate for retrieval (i.e., the query terms are OR'ed together). Disjunctive query processing only makes sense in the context of a scoring model, where the algorithm returns the top k results in terms of document scores. The primary weakness of disjunctive query processing is much slower query evaluation speed compared to conjunctive query processing, since any document that contains a query term may need to be considered. Over the years, researchers have developed clever algorithms and optimizations to increase the speed of disjunctive query processing, for example, by arranging postings so that promising documents are considered early [Anh et al. 2001; Anh and Moffat 2005; Strohman and Croft 2007] or by skipping documents that cannot possibly make it into the top k ranking [Brown 1995; Turtle and Flood 1995; Broder et al. 2003; Strohman et al. 2005; Ding and Suel 2011]. The two state-of-the-art

[2]Liu [2009] calls this the document sampling process and refers to candidate documents as "samples". We prefer the more neutral term "candidate documents" because sampling evokes other connotations that may be misleading.

disjunctive query processing algorithms are Block-Max WAND [Ding and Suel 2011] and the technique of Stroham and Croft [2007]. However, the latter requires impact-sorted indexes, which are inappropriate for our problem formulation (see next section).

The literature also discusses a wide range of systems engineering techniques for controlling efficiency-related factors, such as query latency and throughput. Examples include inter-datacenter query routing [Kayaaslan et al. 2011], different partition strategies within a single datacenter [Baeza-Yates et al. 2007], and caching techniques [Skobeltsyn et al. 2008]. Overall, these designs are not specific to information retrieval, but represent general principles for building large-scale distributed systems [Hamilton 2007; Barroso and Hölzle 2009]. We consider systems engineering orthogonal to our work in the sense that we focus on an individual retrieval engine, which serves as a building block in large-scale search services.

2.2. Web vs. Real-Time Search

There is, of course, more to information retrieval than searching (relatively) static Web content. Our problem of interest—real-time search on a high velocity stream of documents—shares two similarities with traditional Web search.

—*Low-latency, high-throuughput query evaluation.* Users are impatient and demand results quickly. Google reports that as little as an additional 100 milliseconds of search latency causes a measurable drop in search usage [Brutlag 2009].
—*In-memory indexes.* The only practical way to achieve performance requirements is to maintain index structures in memory so that query evaluation never (or very rarely) involves disk. This is the approach adopted by commercial Web search engines today, for example, Google's Jeff Dean has on multiple occasions disclosed that Google's Web indexes are served from memory.

Despite these similarities, there are important differences as well.

—*High ingestion rate and immediate data searchability.* In real-time search, documents arrive at a high velocity, often with sudden spikes corresponding to "flash mobs". Users expect content to be searchable within a short amount of time—on the order of a few seconds. In other words, the indexer must operate incrementally and achieve both low latency and high throughput. This requirement departs from common assumptions in typical search environments, where indexing is considered a batch operation. Although modern Web crawlers achieve high throughput, it is not expected that crawled content be available for searching immediately. Depending on the type of content, an indexing delay of minutes, hours, or even days may be acceptable. This allows engineers to trade off latency for throughput in running indexing jobs on batch systems, such as MapReduce [Dean and Ghemawat 2004]. Alternatively, substantially more machine resources can be brought to bear to reduce indexing latency using the Percolator architecture [Peng and Dabek 2010], but it is unclear if this alternative achieves the speed required for real-time search. Although there is work in the information retrieval literature on low-latency indexing with a focus on explicitly making documents rapidly searchable [Büttcher and Clarke 2005; Strohman and Croft 2006; Guo et al. 2007], the research deals with disk-resident indexes and therefore is not applicable to our environment (see preceding).
—*Importance of the temporal signal.* The nature of real-time search means that temporal signals are important for document scoring, with more recent documents generally favored over older documents. This stands in contrast to Web search, where the time stamp or creation date of a webpage has a relatively minor role in relevance ranking (news search being the obvious exception).

These two distinguishing characteristics of the real-time search problem create an important design requirement. Indexing must be online and incremental—documents must be searchable within a short time (seconds) after they are created. These characteristics also yield a natural architectural constraint: inverted indexes should be document-sorted. While there is much research on efficient query evaluation with impact-sorted indexes [Anh et al. 2001; Anh and Moffat 2005; Strohman and Croft 2007], such an index organization does not appear to be practical for real-time search. Assuming that ingested documents are assigned sequentially-increasing document ids, in a document-sorted index, the postings will be sorted in chronological order. Given the importance of the temporal signal and the bias toward retrieving more recent documents, it is natural to traverse postings lists backwards during query evaluation (i.e., from their ends), so that newer documents are considered before older documents.

2.3. Postings Allocation for In-Memory Incremental Indexing

The problem of incremental indexing, of course, is not new [Cutting and Pedersen 1990; Tomasic et al. 1994; Büttcher and Clarke 2005; Lester et al. 2006, 2008]. However, previous work focused on a different point in the design space, making the assumption that the inverted lists are too large to fit in memory. Most algorithms operate by performing in-memory inversion within a bounded and relatively small buffer (e.g., [Heinz and Zobel 2003]), which inevitably must be "flushed" to disk once the buffer is full. The core challenge is controlling how this happens and how to best merge inverted lists in memory with those that are already on disk. Lester et al. [2008] outlined three basic options: to rebuild the index on disk from scratch, to modify postings in place on disk, or to selectively merge in-memory and on-disk segments and rewrite to another region on disk. In particular, they explored a geometric partitioning and hierarchical merging strategy that limits the number of outstanding partitions, thereby controlling query costs (cf. [Büttcher and Clarke 2005]). Since these approaches all involve disk operations, they are not appropriate for our application, which demands in-memory indexes.

Allocating memory for postings lists to support rapid incremental indexing boils down to a *Goldilocks problem*. As memory remains relatively scarce (at least compared to disk), we would like the solution to be fast yet parsimonious with respect to memory usage. This process needs to be dynamic because postings lists grow as the collection increases in size. Furthermore, postings lists vary significantly in size, since term occurrences are roughly Zipfian. As a result, it is tricky to a priori choose the optimal amount of memory to allocate for each term's postings. Selecting a value that is too large leads to inefficient memory utilization, because most of the allocated space for storing postings will be empty. On the other hand, selecting a value that is too small leads to waste: time, obviously, for memory allocation (which is a relatively costly operation), but also space because noncontiguous postings require pointers to chain together (in the limit, allocating one posting at a time is akin to a linked list).[3] Furthermore, during postings traversal, blocks that are too small also result in suboptimal memory access patterns (e.g., due to cache misses, lack of memory prefetching, etc.). Ideally, we would like to strike a balance and find a "sweet spot".

As a reference solution, we describe Twitter's Earlybird retrieval engine, which forms the starting point of our work. The Java-based system, which comprises the core of Twitter's production search service, is detailed in a previous paper [Busch et al. 2012], but here we discuss the salient aspects. To our knowledge, this is the only detailed exposition of a production solution to the incremental in-memory indexing problem that we are aware of, and thus serves as a good foundation to build on. Earlybird implements

[3]The alternative index organization, retaining contiguous inverted lists, requires copying around data and thus is too slow for production requirements.

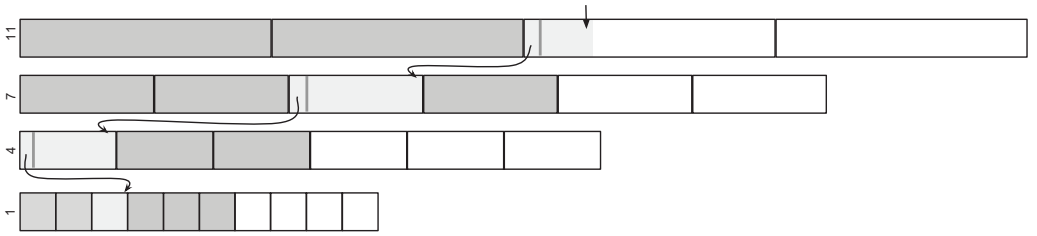


Fig. 2. Organization of an index segment in Earlybird. Increasingly larger slices are allocated in large pools to hold postings lists. Except for slices in pool 1 (the bottom pool), the first 32 bits of each slice are used for storing the backpointer that links the slices together. Pool 4 (the top pool) can hold multiple slices for a term. In this figure, dark gray slices are allocated, white slices are free, and the light gray slices marked with the arrow (i.e., tail pointer) comprise the postings list for a particular term.

a dynamic postings allocation policy that allocates increasingly-larger "slices" from a small number of large, fixed pools of memory.

In Earlybird, each posting is simply a 32-bit integer: 24 bits are devoted to storing the document ID and 8 bits for the term position. Since tweets are limited to 140 characters, 8 bits are sufficient to hold term positions.[4] A postings list is simply an integer array, and indexing new documents involves inserting elements into a pre-allocated array. Postings traversal in reverse chronological order corresponds to iterating through the array backwards. This organization allows every array position to be a possible entry point for postings traversal. In addition, it allows for efficient binary search (to find a particular document ID)[5] and does not require any additional skip-list data structures to enable faster traversal through the postings lists. Finally, this organization is cache friendly, since array traversal involves linear memory strides and this predictable access pattern provides reliable prefetch cues to the hardware.

Note, critically, that postings lists are not compressed. To understand this design decision, we must first realize that an important aspect of real-time search in production is managing concurrent (i.e., multithreaded) access to index structures—query evaluation is interleaved with indexing, which means that index structures may be mutated while the postings lists are simultaneously being traversed for retrieval. Maintaining proper invariants to guarantee search correctness is difficult and something that Busch et al. [2012] discuss. In the Earlybird design, each instance of the retrieval engine holds roughly a dozen index segments, and each segment holds a relatively small number of tweets (since the docid is limited by a 24-bit integer space). Ingested tweets first fill up a segment before proceeding to the next one. Therefore, at any given time, there is at most one index segment actively being modified, whereas the remaining segments are read-only. Once an index segment stops accepting new tweets, it can be converted from a write-friendly structure into an optimized, compressed, read-only structure. Separating the active index segment from the read-only index segments has the advantage of isolating the scope of concurrent index read/write operations, which simplifies concurrency management. Due to the rapid rate at which tweets arrive, an index segment does not spend too long in the uncompressed state, and therefore working with uncompressed integer arrays is acceptable from a memory usage perspective. Concurrency management is not a focus of this work, but we return to discuss this issue later.

The Earlybird solution to the Goldilocks problem is to create four separate *pools* for holding postings, shown in Figure 2. Conceptually, each pool can be treated as an

[4]If a term appears in the tweet multiple times, it will be represented with multiple postings.

[5]In contrast, block-based compression techniques, such as PForDelta, are not friendly to binary search, since an element probe requires decoding an entire block of postings.

unbounded integer array, but in practice, pools are large integer arrays allocated in blocks. *Slices* are allocated from the pools to hold individual postings belonging to a term. In each pool, the slice sizes are fixed: they are 2^1, 2^4, 2^7, and 2^{11}, respectively. For convenience, we refer to these as pools 1 through 4, respectively. When a term is first encountered, a 2^1 integer slice is allocated in the first pool, which is sufficient to hold postings for the first two term occurrences. When the first slice runs out of space, another slice of 2^4 integers is allocated in pool 2 to hold the next $2^4 - 1$ term occurrences (32 bits are used to serve as the backpointer, discussed next). After running out of space, a slice is allocated in pool 3 to store the next $2^7 - 1$ term occurrences, and finally a slice in pool 4 to store the following $2^{11} - 1$ term occurrences. Additional space is allocated in pool 4 in slices of 2^{11} integers as needed.

One advantage of this strategy is that no array copies are required as postings lists grow in length, which means that there is no garbage to collect (important for memory-managed languages such as Java). Furthermore, the data structure is compact in the sense that there is very little memory overhead (e.g., in terms of object headers and alignment padding), since we are mostly manipulating primitive integer arrays (once again, important in a language such as Java). However, the trade-off is that postings are noncontiguous and we need a mechanism to link the slices together. Addressing slice positions is accomplished using 32-bit pointers: 2 bits are used to address the pool, 19–29 bits are used to address the slice index, and 1–11 bits are used to address the offset within the slice. This creates a symmetry in that postings and addressing pointers both fit in a standard 32-bit integer. The dictionary maintains pointers to the current *tail* of the postings list using this addressing scheme (thereby marking where the next posting should be inserted and where query evaluation should begin). Pointers in the same format are used to link the slices in different pools together and possibly multiple slices in pool 4. In all but the first pool, the first 32 bits of each slice are used to store this backpointer.

Note that Earlybird's design can be extended to an arbitrary number of pools and pool sizes; we can create more memory pools or use different slice sizes—this is further explored in Asadi et al. [2013] A particular instantiation of this general strategy can be described by $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$, where m is the number of pools and where pool n has slices of size 2^{P_n}. For example, in the default setup, $\mathcal{P} = \{1, 4, 7, 11\}$.

2.4. Efficient Postings List Intersection with Bloom Filters

Finally, we arrive at the last piece of background: efficient postings list intersection with Bloom filters, described in our recent short paper [Asadi and Lin 2012b]. The intuition is that in a two-stage retrieval architecture, the second-stage rescorer is relatively insensitive to the quality of the candidate documents. As long as this intermediate product is of reasonable quality, the final output will remain high quality—particularly in the Web context due to an emphasis on early precision. Thus, our goal is to generate candidate documents as quickly as possible, even at the cost of introducing approximations. Recently, this approach was echoed by the work of Tonellotto et al. [2013], who selectively applied aggressive and unsafe pruning in the candidate generation stage to control end-to-end effectiveness/efficiency trade-offs in the context of a similar two-stage retrieval architecture.

In Asadi and Lin [2012b], we introduced a novel algorithm for postings list intersection using Bloom filters, which are probabilistic data structures that support $O(1)$ approximate membership tests [Bloom 1970]. Previously, Bloom filters have been used in information retrieval applications for P2P retrieval [Li et al. 2003] and signature files [Shepherd et al. 1989; Tirdad et al. 2011]. In our approach, each postings list is stored both as a compressed sequence of integers and as a Bloom filter, that is, the

Bloom filter representation of a postings list allows us to quickly answer the question, is this document ID contained in the postings list?

A Bloom filter is comprised of a bit array of size m and κ hash functions that generate values in the range $[0, m-1]$. To insert an element into the filter, κ hash values are computed, and the corresponding bit positions in the underlying bit array are set to one. Hash collisions are ignored, that is, if a bit has already been set to one previously, it is left untouched. To probe the filter (i.e., determine if an element is a member of the Bloom filter), κ hash values of the test element are computed using the same hash functions used in the insertion process. The corresponding bit positions are probed. The membership test passes if all the bit positions are set (i.e., 1), and it fails when any of the bit positions are unset (i.e., 0). A common lookup optimization is to compute the hash values one at a time and terminate when the probed bit position is not set (to save unnecessary hash value computations).

Bloom filters only support approximate membership tests, where false positives are possible, that is, a Bloom filter may assert true on a membership test when in fact the element was never inserted. However, false negatives are impossible, that is, if a membership test fails, the test element is guaranteed never to have been inserted. False positives occur when there are hash collisions and increase as more elements are inserted into the Bloom filter. Fortunately, the false positive rate can be theoretically predicted and controlled given the values m and κ. Assuming the hash functions select a bit position in the array of m bits with equal probability, the probability that a bit is not set by one hash function during the insertion is equal to $1 - \frac{1}{m}$. Therefore, with κ hash functions and n inserted elements, the probability that a bit is set to one is

$$p = 1 - \left(1 - \frac{1}{m}\right)^{\kappa n}. \tag{1}$$

During a membership test, a false positive is possible when all κ bit positions (computed by the hash functions) are set to one. The error thus should be

$$p^{\kappa} = \left(1 - \left(1 - \frac{1}{m}\right)^{\kappa n}\right)^{\kappa} \approx \left(1 - e^{\frac{-\kappa n}{m}}\right)^{\kappa}. \tag{2}$$

This commonly-cited bound is actually incorrect, since it assumes independence for the probabilities of each bit being set. Bose et al. [2008] showed that the preceding expression is the lower bound on false positive error, with the actual upper bound is

$$p^{\kappa} \times \left(1 + \frac{\frac{\kappa}{p}\sqrt{\frac{\ln m - 2\kappa \ln p}{m}}}{1 - \frac{\kappa}{p}\sqrt{\frac{\ln m - 2\kappa \ln p}{m}}} + \frac{2}{\sqrt{m}}\right). \tag{3}$$

Nevertheless, for large enough values of m and small values of κ, the difference between p^{κ} and the actual false positive rate is negligible. In our work, we specify m in terms of r, the number of bits that we devote to an element in the Bloom filter (i.e., $r = m/n$); as we shall see, this is a more natural formulation in our application.

Our postings list intersection algorithm proceeds as follows: for a query Q with $|Q|$ terms, we find the term q with the smallest document frequency (i.e., rarest query term) and look up its standard postings list. We refer to this as the *base* postings list. The algorithm traverses this postings list and probes the Bloom filter representation of the other query terms to compute the set intersection. A document is added to the candidate list if all membership tests pass. The approximation aspect of our algorithm lies in the fact that Bloom filters can produce false positives, that is, a filter can assert that an element is contained within it, even when in reality that element was never

inserted. In other words, the test can erroneously assert that a document contains a query term when in fact it does not.

Although this exposition captures the gist of our idea, we refer the reader to Asadi and Lin [2012b] for additional details. In particular, the short paper describes experiments that empirically characterize the trade-offs between effectiveness (result quality), time (retrieval speed), and space (index size). We show that for a range of parameter settings, our approximate postings list intersection algorithm with Bloom filters achieves very high relative recall with respect to an exact postings list intersection baseline. Furthermore, in end-to-end experiments with a simple second-stage machine-learned ranking model, the approximations introduced by Bloom filters do not yield significant differences in NDCG@{1,3,5,10,20} compared to a baseline using exact postings intersection. However, postings lists intersection with Bloom filters is substantially faster but at the cost of additional memory necessary for storing the auxiliary data structures.

3. BLOOM FILTER CHAINS

Bloom filter chains are extensions of standard Bloom filters that can dynamically expand to efficiently represent an arbitrarily long and growing list of monotonically-increasing integers with a constant false positive rate that is specified a priori. In this section, we present a compact data structure for managing an arbitrarily-large collection of Bloom filter chains. This data structure is compact in the sense that it requires minimal memory overhead in terms of space not directly devoted to storing the data (e.g., object headers and alignment padding). At a high level, this data structure can be thought of as adapting Earlybird's memory allocation policy to create chains of Bloom filters on demand.

We define two primitives that our collection of Bloom filter chains supports.

—INSERT(t, e) inserts an element e into the list of monotonically-increasing integers associated with t.
—PROBE(t, e) performs an approximate membership test to determine if e is an element of the list of monotonically-increasing integers associated with t.

Importantly, our proposed data structure does not support the following operation.

—GET(t) returns the list of monotonically-increasing integers associated with t.

That is, we do not support an operation for enumerating the list of integers associated with a particular key, only an element membership test via the PROBE operation. Finally, we define one additional optional operation.

—DELETE(t, e) removes element e from the list of sorted integers associated with t.

Although this data structure was designed to support our approximate candidate generation algorithm (where the lists of monotonically-increasing integers represent document ids), it may be useful for other applications as well.

3.1. Expandable Bloom Filters

It is clear, based on the analysis in Section 2.4, that standard Bloom filters are not appropriate for representing sets whose cardinalities are not known a priori. Since Bloom filters are constructed with a fixed number of bits m, as we insert more elements, the false positive rate increases. When trying to pre-size Bloom filters, we encounter exactly the Goldilocks problem discussed in Section 2.3. If we allocate too much space to begin with, we waste memory. If we allocate too little space, the false positive rate may grow too high to be acceptable for the application.

The scalability of Bloom filters is not a new problem. Almeida et al. [2007] addressed this issue by using not one, but a series of Bloom filters of geometrically-increasing sizes.

When a Bloom filter reaches capacity, a new one is created and added to the list. For a membership test, all Bloom filters in the list are probed; the membership test passes if any of the Bloom filters assert a positive result. There are a few drawbacks to this approach. First, membership tests become progressively slower as the number of Bloom filters increases. Second, false positive rates compound with multiple filters, which must be counteracted by increasing their size according to a geometric progression to maintain a desired false positive rate. Finally, Almeida et al. do not discuss how memory would be efficiently allocated for a large number of these Bloom filters. In a naïve implementation, each Bloom filter would be represented as an object, which leads to a proliferation of objects of varying sizes if we have a large collection of such lists of Bloom filters.

Our proposed data structure addresses these issues: for each key, the associated Bloom filter chain can accommodate a growing, arbitrarily-sized list of monotonically-increasing integers and support efficient probing by exploiting the sorted behavior of the inserted elements. In addition, we describe an efficient memory allocation policy for managing a large collection of Bloom filter chains.

3.2. Element Insertions and Probes

Bloom filter chains grow in a manner similar to how space for postings are allocated in Earlybird (see Section 2.3). The first time we encounter a term, we construct a Bloom filter with a fixed length. Since we are interested in maintaining a constant false positive rate, from this fixed length we can calculate the maximum number of elements that the Bloom filter can safely accommodate. Once a Bloom filter reaches its capacity, we allocate another Bloom filter with a larger size and link it to the last filter. In the end, we have a linked list of Bloom filters, each storing a portion of the sorted list (hence, a Bloom filter chain).

One important difference that distinguishes our Bloom filter chains from the work of Almeida et al. [2007] is in deciding which Bloom filter to probe. They require probing all the Bloom filters, but in our case, since elements are monotonically increasing, each Bloom filter in the chain contains a non-overlapping range of elements. That is, if the nth Bloom filter contains elements in the range $[b_n, l_n]$, where b_n and l_n indicate the first and last element inserted into the Bloom filter, respectively, then $l_n < b_{n+1}$. This means that we can compare the query element with the range associated with each Bloom filter to determine whether the queried element might have been inserted into that filter; these range checks tell us the right Bloom filter to probe, and thus we can save unnecessary membership tests. To enable this comparison, we must keep track of b_n and l_n for every Bloom filter in the chain; however, since $b_n < l_n$ and we know that $l_n < b_{n+1}$, storing b_n's alone suffices.

Abstractly, each Bloom filter chain can be viewed as a linked list of Bloom filters. However, a naïve linked list implementation would be inefficient in terms of memory use, since each individual Bloom filter would be represented by an object, which translates into substantial overhead for object headers and other associated bookkeeping metadata. To address this problem, we allocate memory for each Bloom filter in the chain using the approach implemented in Earlybird; this structure is illustrated in Figure 3. Let us assume we have a pool structure consisting of m memory pools with slice sizes $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$. We can view each memory pool as an unbounded array of 32-bit integers, although in practice, we dynamically grow each pool as necessary. In each pool, the slices are numbered numerically starting from one, such that a tuple $\langle n, s \rangle$, consisting of a pool number and a slice number, uniquely identifies a particular region in memory (we can pack both values in a 32-bit integer for compact storage). For example, $\langle 1, 1 \rangle$ starts at index position 0 in pool 1 and has length 2^{P_1}; $\langle 1, 2 \rangle$ starts at index position 2^{P_1} in pool 1 and has the same length. In general, $\langle n, s \rangle$ refers to a

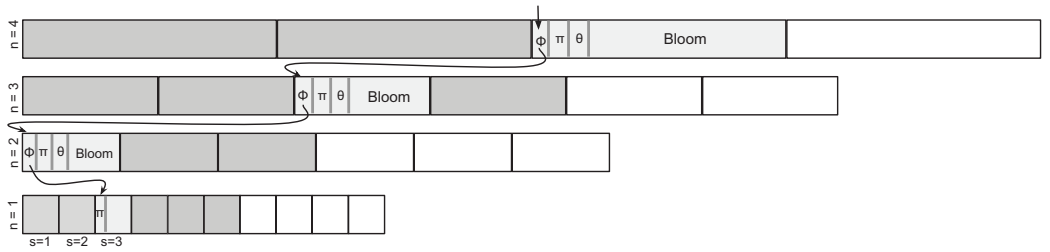


Fig. 3. Organization of a collection of Bloom filter chains: ϕ is the backpointer, π is the counter, and θ is the first document ID inserted into the Bloom filter. In this figure, dark gray slices are allocated, white slices are free, and the light gray slices comprise the Bloom filter chain for a term.

region in memory in pool n, starting at index position $(s-1)2^{P_n}$. It is in these regions that we allocate space for the Bloom filters that comprise the chains. Next, we explain in detail element insertions and the membership tests.

The implementation of the INSERT operation is as follows (see Algorithm 1). Before indexing begins, we must specify a target false positive rate by setting r, the number of bits per element in the Bloom filters, and κ, the number of hash functions. The impact of these parameters is explored in our experiments. The first time we encounter a term, we allocate a slice from pool 1 (the next available s). A slice from the first pool, if used entirely for a Bloom filter, can accept a maximum of $32 \times 2^{P_1}/r$ elements without compromising the false positive rate. Recall that slice sizes are denoted in units of 32-bit integers so that they can be efficiently implemented as integer arrays and manipulated via standard bitwise operators. However, we must keep track of the number of elements that has been inserted into a Bloom filter in order to trigger the expansion mechanism. To achieve this, we use one integer as a counter and treat the rest of the slice as a Bloom filter. As a result, the actual capacity of a slice from pool 1 is $C_1 = 32 \times (2^{P_1} - 1)/r$. We

ALGORITHM 1: INSERT(t, e)

Input: t – key
Input: e – element to be inserted into the Bloom filter chain
Given: $\mathcal{P}$ – memory pools $\{P_1, P_2, \ldots, P_m\}$

```
⟨n, s⟩ ← DICTIONARY.LOOKUP(t)
if ⟨n, s⟩ = ∅ then
    n ← 1
    s ← ALLOCATESLICEFROMPOOL(1)
    π⟨n,s⟩ ← 1
    F⟨n,s⟩.INSERT(e)
    DICTIONARY.SET(t, ⟨n, s⟩)
else if π⟨n,s⟩ < CAPACITY_n then
    F⟨n,s⟩.INSERT(e)
    π⟨n,s⟩ ← π⟨n,s⟩ + 1
else
    n′ ← MIN(n + 1, |P|)
    s′ ← ALLOCATESLICEFROMPOOL(n′)
    π⟨n′,s′⟩ ← 1
    φ⟨n′,s′⟩ ← ⟨n, s⟩
    θ⟨n′,s′⟩ ← e
    F⟨n′,s′⟩.INSERT(e)
    DICTIONARY.SET(t, ⟨n′, s′⟩)
end if
```

denote the Bloom filter at $\langle n, s\rangle$ as $F_{\langle n,s\rangle}$, and use $\pi_{\langle n,s\rangle}$ to denote the value of the associated counter, which keeps track of the number of elements that has been inserted. We stop inserting new elements into the first Bloom filter once $\pi_{\langle n,s\rangle}$ reaches its capacity, C_1.

Insertion of elements into a Bloom filter requires computing κ hashes: the hash computations must be fast, and the hash values must distribute the keys as evenly as possible. In our implementation, we construct an arbitrary number of hash functions using Jenkin's integer hash, of the form $h(x, S) \mod L$, where S is a seed and L is the length of the Bloom filter. For $\kappa = 1$, we simply use a large prime number as the seed. For $\kappa > 1$, we compute the nth hash value by setting the seed to $h^{n-1}(x, S)$.

Every time an element is inserted into a Bloom filter $F_{\langle n,s\rangle}$, we increment $\pi_{\langle n,s\rangle}$. Once we have reached the Bloom filter's maximum capacity, the algorithm allocates another slice from pool $n + 1$ (unless we have reached the last pool, in which case we continue allocating slices from the same pool). For $n > 1$, we reserve three integers to hold metadata and use the rest of the slice for the Bloom filter itself. The three metadata elements, each 32 bits, are the following.

(1) $\pi_{\langle n,s\rangle}$. A counter storing the number of inserted elements (same as in pool 1).
(2) $\phi_{\langle n,s\rangle}$. A backpointer which links the current slice to the previous slice.
(3) $\theta_{\langle n,s\rangle}$. The first document ID inserted into the Bloom filter.

The capacity of the Bloom filter in pool $n > 1$ is $C_n = 32 \times (2^{P_n} - 3)/r$. Thus, the final capacity function used in Algorithm 1 is defined as follows.

$$\textsc{Capacity}_{\langle n,s\rangle} = \begin{cases} \frac{32\times(2^{P_1}-1)}{r}, & n = 1; \\ \frac{32\times(2^{P_n}-3)}{r}, & n > 1, \end{cases}$$

where $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ are the slice sizes, and r is the number of bits per element in a Bloom filter.

To support the insertion operation, we assume the existence of a dictionary, which holds the mapping from keys to the *tail pointers* to the last slices of the chains. As previously described, this tail pointer is represented as a tuple $\langle n, s\rangle$ and stored in a 32-bit integer. Overall, it is easy to see that INSERT is an $O(1)$ operation.

Next, we describe the implementation of the PROBE operation (see pseudo code in Algorithm 2). We wish to quickly determine if element e is contained in the Bloom filter chain associated with key t. The Bloom filter chain is identified by the tail pointer in the dictionary, which contains a unique tuple $\langle n, s\rangle$ that identifies the current end of the chain. Testing for membership begins by comparing the test element e with $\theta_{\langle n,s\rangle}$. There are three possible outcomes.

(1) $e = \theta_{\langle n,s\rangle}$. The membership test passes.
(2) $e > \theta_{\langle n,s\rangle}$. This means that e must have been inserted into $F_{\langle n,s\rangle}$. In this case, we probe the current Bloom filter to determine if e is a member.
(3) $e < \theta_{\langle n,s\rangle}$. It is clear that if e had been inserted, it must be contained in an earlier Bloom filter in the chain. In this case, we follow the backpointer $\phi_{\langle n,s\rangle}$ to the previous Bloom filter and repeat this comparison. Note that when we reach the first Bloom filter, we probe the filter regardless of e.

Technically, PROBE is an $O(n)$ operation where n is the length of the list associated with t, but in practice only a few range checks are necessary to find the correct Bloom filter to probe, an $O(1)$ operation. In Section 4, we introduce a simple optimization that reduces the complexity of PROBE operations to $O(1)$ in the context of our candidate generation algorithm.

ALGORITHM 2: PROBE(t, e)

Input: t – key
Input: e – test element
Given: $\mathcal{P}$ – memory pools $\{P_1, P_2, \ldots, P_m\}$
Output: true if the Bloom filter chain contains the test element, false otherwise

```
⟨n, s⟩ ← DICTIONARY.LOOKUP(t)
while e < θ⟨n,s⟩ AND n > 1 do
  ⟨n, s⟩ ← φ⟨n,s⟩
end while
return F⟨n,s⟩.CONTAINS(e)
```

3.3. Element Deletions

On the collection of Bloom filter chains, we define a third optional operation, DELETE(t, e), which removes element e from the list of sorted integers associated with t. We describe how deletes can be handled, although since there are no deleted documents in our experiments, this operation is not implemented.

The fundamental challenge we face is that Bloom filters do not support deletes. There is a simple solution, however: we can separately record the deletes, and before returning results of the PROBE operation, check against this record to make sure that the element still exists. There are a number of choices for storing deletes: an obvious approach would be a hash table, but we could be clever and store the deletes themselves in another Bloom filter. In this case, the potential false positive behavior of Bloom filters matches user expectations: our data structure would never assert the presence of a deleted document via the PROBE operation, but might falsely deny the existence of a document—this serves to compound the error associated with membership tests but can be analytically modeled and thus controlled.

There is, of course, the question of how to size the Bloom filters used for storing deletes. We could use the technique of Almeida et al. [2007] (note that deletes may be arbitrarily ordered), but an alternative solution would be to rebuild the entire collection of Bloom filter chains from scratch periodically. This is similar to how deletes are often applied in standard inverted indexes [Chiueh and Huang 1999; Guo et al. 2007]: since random accesses to postings are expensive, deletes are first stored separately, and the cost of deletions are later amortized across index rebuilds (which might happen for independent reasons). Deletes in the log-structured merge-tree data structures that underlie Bigtable [Chang et al. 2006] and many NoSQL stores are handled in a similar fashion—by first separately recording deletes and then actually purging records during a compaction cycle.

4. EFFICIENT CANDIDATE GENERATION WITH BLOOM FILTERS

In this section, we introduce an efficient candidate generation algorithm that takes advantage of the collection of Bloom filter chains described in the previous section. Using this data structure to complement traditional inverted lists (i.e., sorted lists of integers), we describe a novel variant of the WAND algorithm, called BWAND (short for Bloom WAND), that is many times faster. Like WAND, BWAND is capable of both conjunctive query processing as well as disjunctive query processing. In end-to-end experiments with a second-stage rescorer, BWAND produces precision and NDCG (at various cut-offs) that are not significantly different from candidates generated by either the SvS postings intersection algorithm in the conjunctive case or exhaustive OR evaluation in the disjunctive case.

Before delving into the algorithm particulars, we begin with a general discussion of pertinent issues. Conjunctive query processing is equivalent to postings intersection and does not take advantage of a scoring model—in this sense, the output is an unsorted set. However, for practical reasons, to limit the number of candidates that are considered by the second stage, we wish to return only k results. In the context of the real-time search task, it makes sense to return the k most recent results, corresponding to a reverse chronological sort order. Since documents are inserted into inverted lists in chronological order, if we traverse postings lists from their ends, we can early terminate as soon as we find k candidate documents.

On the other hand, disjunctive query processing (i.e., top k retrieval) necessitates a scoring model. We use BM25 [Robertson et al. 1995] as the reference scoring model, which has been shown to produce effective rankings in a wide variety of settings. The score of a document D with respect to a query is defined as follows.

$$\textsc{Score}_{\text{BM25}}(Q, D) = \sum_{q \in Q} \frac{(k_1 + 1) \cdot \text{tf}(q, D)}{K + \text{tf}(q, D)} \log \left[\frac{N - \text{df}(q) + 0.5}{\text{df}(q) + 0.5} \right], \tag{4}$$

where $\text{tf}(q, D)$ is the term frequency of query term q in D, $\text{df}(q)$ is the document frequency of q, and N is the number of documents in the collection. K is defined as $k_1[(1 - b) - b \cdot (|D|/|D|')]$, where $|D|$ is the length of document D and $|D|'$ is the average document length. Finally, k_1 and b are free parameters.

Abstractly speaking, BM25 comprises three components: a score component based on the term frequency (the first term in the summation), a score component based on the document frequency (the second term in the summation), and a length normalization factor (K). Applying these three score components to tweets, we make two observations. First, tweets are short and limited to 140 characters, which suggests that length normalization should not have much of a scoring impact. Second, also due to the short length of tweets, term frequency is one almost all the time (we empirically verify this later). Thus, when scoring tweets, it seems the only statistic that really matters is document frequency of the matching query terms.

Let us operationalize these observations by simplifying BM25 to the sum of the document frequency score component of matching query terms—in other words, if a query term is contained in a document, we simply assume that its term frequency is one and ignore document length normalization. More precisely, we define the IDF scoring model as follows.

$$\textsc{Score}_{\text{IDF}}(Q, D) = \sum_{q \in Q \cap D} \log \left[\frac{N - \text{df}(q) + 0.5}{\text{df}(q) + 0.5} \right] \equiv \sum_{q \in Q \cap D} \textsc{IDF}(q). \tag{5}$$

These simplifications allow us to exploit a number of optimizations that substantially increase the speed of candidate generation. We experimentally examine the effectiveness of this simplified scoring model later.

Next, we provide an overview of WAND: this algorithm for top k retrieval uses a *pivot*-based pointer-movement strategy which enables the algorithm to skip over postings of documents that cannot possibly be in the top k results. In this approach, each postings list has a *current* pointer that moves forward as the algorithm proceeds. Postings to the left of the pointer have already been examined, while postings to the right have yet to be considered. The algorithm keeps postings lists sorted in increasing order of the current document ID. In addition, each postings list is associated with a score upper bound, indicating the maximum score contribution of the term. At first, the algorithm creates an empty heap of size k. At each step, the algorithm finds a pivot term: it does so by adding up the score upper bounds of each term (in sorted order). The pivot term is the first term where the sum exceeds a given threshold, typically set to the lowest

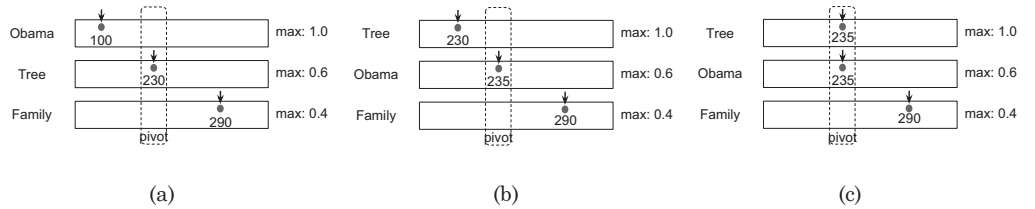


Fig. 4. Illustration of the WAND algorithm on a three-term query in which the current pointers point to documents 100, 230, and 290. With a score threshold of 1.4, WAND selects "tree" as the pivot term (a). Since the current pointer for "Obama" points to a document ID that is less than 230, the algorithm moves its current pointer to the posting whose document ID is at least 230, and resorts the lists (b). Current pointers now point to 230, 235, and 290, and document 235 becomes the pivot. Due to the same reason as in (a), the first term's (now, "tree") current pointer is moved to the posting whose document ID is at least 235, resulting in (c). Finally, since the pivot term's document ID is equal to that of its preceding term, the algorithm computes the score for document 235, and adds it to the heap if its actual score is greater than the threshold.

score in the heap, that is, a document's score must be higher than this value to appear in the top k results. If no such term exists, then the algorithm terminates. Otherwise, there are two cases. If all the preceding terms' current document ids are equal to that of the pivot term, the candidate document pointed to by the pivot term is scored, and the document is added to the heap if the actual score exceeds the threshold. Otherwise, the algorithm moves the current pointer for one of the preceding terms to the posting whose document ID is greater than or equal to that of the pivot term, and the algorithm repeats.

As an example, assume that we are given the query "Obama family tree", where the score upper bounds for "Obama", "family", and "tree" are 1.0, 0.4, and 0.6 respectively. Further assume that, as illustrated in Figure 4(a), the current pointers point to documents 100, 290, and 230 for the query terms. WAND sorts the postings lists such that the current document IDs are in ascending order. Suppose the current threshold is set to 1.4 (the current lowest score in the heap). Therefore, "tree" is selected as the pivot term ($1.0 + 0.6 > 1.4$). The algorithm can now confidently claim that the smallest document ID that can make it into the top k is 230. Because the current document id for "Obama" is not equal to 230, the algorithm moves its pointer to the posting whose document ID is greater than or equal to 230 and resorts the lists, resulting in Figure 4(b). Current pointers now point to documents 230 for "tree", 235 for "Obama", and 290 for "family". Note that document 230 is never fully scored because it does not contain "Obama". The term "Obama" is selected as the pivot next, and similar to the case in Figure 4(a), the pointer for "tree" is moved to the posting whose document ID is at least 235, resulting in Figure 4(c). Since the current document ID for the pivot term is equal to that of its preceding term, the algorithm computes the score for document 235, and if its actual score is greater than the threshold 1.4, it adds the document to the heap.

Recently, Ding and Suel [2011] introduced an optimization on top of WAND, called Block-Max WAND, that substantially increases query evaluation speed. The idea is that instead of using the global maximum score of each term to compute the pivots, the algorithm uses a piecewise upper-bound approximation of the scores for each postings list. Note what happens with WAND if we switch to the simplified IDF scoring model described in Equation (5). We see that the Block-Max optimization becomes irrelevant! Because we assume that all term frequencies are one and ignore length normalization, the score contribution for the matching query term in all documents is identical.

A critical limitation of our proposed Bloom filter data structure is the inability to support a GET operation, to enumerate the list of documents that contain a term. In this sense, it is impossible to traverse a Bloom filter chain. We get around this by using

ALGORITHM 3: BWAND(Q, k, ω)

Input: Q – query consisting of $|Q|$ terms
Input: k – number of documents to retrieve
Input: ω – conjunctive/disjunctive tuning parameter
Output: top k documents ranked by IDF scoring model

```
H ← new Heap(k)
θ ← ω × Σ_{q_i ∈ Q} IDF(q_i)
q_b ← argmin_i [DICTIONARY.DF(q_i), q_i ∈ Q]
P_b ← GETPOSTINGS(q_b)
while P_b.HASNEXT() AND θ < Σ_{q_i ∈ Q} IDF(q_i) do
  d ← P_b.GETDOCID()
  s ← IDF(q_b)
  for q_i ∈ Q − q_b do
    if PROBE(q_i, d) = TRUE then
      s ← s + IDF(q_i)
    end if
  end for
  if s > θ then
    H.INSERT(⟨d, s⟩)
    if H.ISFULL() then
      θ ← H.MINSCORE()
    end if
  end if
end while
return H
```

the Bloom filter chains as an auxiliary data structure, only for its PROBE operation. In our BWAND algorithm, we traverse the normal inverted list of the rarest query term and probe the Bloom filter chains corresponding to the other query terms. This has the effect that candidate documents must contain the rarest query term. We experimentally explore the impact of this heuristic on end-to-end effectiveness later.

Having laid out the preliminaries, we are now ready to more formally describe our BWAND candidate generation algorithm, whose pseudocode is shown in Algorithm 3. Note that the operation of our algorithm is "flipped" with respect to the standard left-to-right operation of WAND, where "left" refers to the beginnings of postings lists and "right" refers to the ends. Since we traverse postings lists from their ends in order to consider documents in reverse chronological order, our algorithm conceptually operates from right to left.

The input to our BWAND algorithm consists of a query Q, the number of hits to retrieve k, and a tuning parameter ω that controls whether the algorithm operates in conjunctive or disjunctive query processing mode. The algorithm begins by creating a heap of size k and computing an initial score threshold θ, which is equal to ω times the sum of the IDF scores of all the query terms. The algorithm then identifies the rarest query term and fetches its postings list—we refer to this as the *base* postings list. Query evaluation proceeds by traversing this base postings list backwards from the end, as previously discussed. The initial score of the document under consideration is the rarest term's IDF score. The algorithm then probes the Bloom filter chains associated with the other query terms. For every membership test that passes, it adds the IDF score of the corresponding term to the document score. If the final score is strictly greater than θ, the document is placed in the heap. When the heap is full, every time a document is added to the heap, θ is adjusted to the minimum score in the heap (i.e., that of the k_{th} document).

Initializing ω to 0 yields disjunctive query processing (although our algorithm still enforces that the rarest query term must be present), whereas setting ω to $1 - \epsilon$ yields conjunctive query processing. Values between the extremes control the extent to which query evaluation is more disjunctive-like or conjunctive-like (similar to an equivalent parameter in WAND). A straightforward optimization in pure conjunctive query processing mode is to short circuit the membership tests once a PROBE fails and proceed to the next document in the base postings list. We also note that the strict comparison with θ guarantees that ties are broken in favor of more recent documents—this also means that conjunctive evaluation can early-terminate as soon as the heap is full.

As another optimization, in the PROBE operation we can take advantage of the fact that test document ids are monotonically decreasing (since we are traversing the base postings list backwards). Assuming we probed slice s in pool n to perform a membership test for d_i, we can begin Algorithm 2 from the same location to perform the membership test for d_{i+1}. In other words, instead of starting from the last slice in the Bloom filter chain each time, we can begin with the last Bloom filter probed. This optimization improves performance by reducing pointer chasing to find the right slice, and makes the PROBE operation $O(1)$ in the context of the candidate generation algorithm.

5. EXPERIMENTAL SETUP

We performed experiments on a server running Red Hat Linux, with dual Intel Xeon Westmere quad-core processors (E5620 2.4GHz) and 128GB RAM. This particular architecture has a 64KB L1 cache per core, split between data and instructions; a 256KB L2 cache per core; and a 12MB L3 cache shared by all cores of a single processor. We used Java 1.6.0u37, with a heap size of 100GB given to the Java Virtual Machine, of which only a small fraction was actually used in our experiments.

In what follows, we describe the evaluation data, implementation details of the candidate generation algorithms, the second-stage rescorer in our end-to-end search pipeline, and metrics used to evaluate our approach.

5.1. Test Collections and Evaluation Methodology

Our experiments used the Tweets2011 collection, created for the TREC 2011 and TREC 2012 microblog tracks [Ounis et al. 2011; Soboroff et al. 2012a; McCreadie et al. 2012; Soboroff et al. 2012b].[6] The collection includes approximately 16 million tweets over a period of two weeks (24 January 2011 until 8 February, inclusive), which covers both the time period of the Egyptian revolution and the U.S. Superbowl. Different types of tweets are present, including replies and retweets, in English as well as other languages.

Analysis of the collection reveals that term frequency is one for 96% of all term instances and is two for 3.5% of the instances. Most of the terms that appear more than once in a tweet do not appear to be important from a retrieval perspective. Examples include “rt”, “http”, “haha”, “like”, “da”, “youtub”, “video”, top level domains (e.g., “com”, “edu”). Often, spam tweets have terms with high term frequencies due to attempts at keyword stuffing. However, cursory examination does reveal that, occasionally, terms of interest (such as “world”, “TSA”) appear multiple times in tweets.

For evaluation, we used three different sets of queries.

—The TREC 2005 terabyte track “efficiency” queries (50,000 queries total).[7] Since there are no relevance judgments for these queries, they were used solely for efficiency experiments.

[6] http://trec.nist.gov/data/tweets/.

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

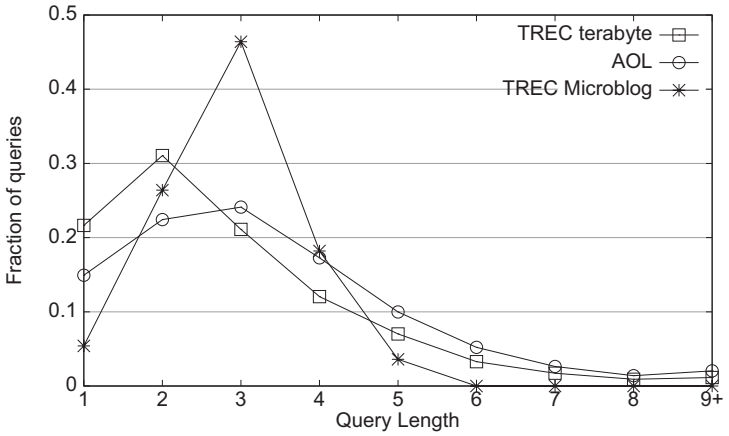


Fig. 5. Query-length distribution for the AOL, TREC terabyte, and TREC microblog queries.

—A set of 100,000 queries sampled randomly from the AOL query log [Pass et al. 2006], which contains approximately 10 million queries in total. Our sample retains the query length distribution of the original dataset. Similar to the TREC 2005 terabyte track queries, we used these queries only to evaluate efficiency.
—The TREC microblog track topics from 2011 and 2012, 110 in total. These queries comprise a complete test collection in that we have relevance judgments, but there are too few queries for meaningful efficiency experiments.

Figure 5 shows the query-length distribution for these query sets.

One final detail about our evaluation methodology: in the efficiency experiments, we performed retrieval after all tweets in the collection have been added to the index, even though in a production setting, queries are issued at different times, interleaved with arriving documents. This simplification was made so that all queries operated over the same exact index—otherwise, some queries might execute faster by the mere fact that they were issued earlier over a smaller index. Furthermore, the TREC efficiency queries are not associated with time stamps, and the AOL query time stamps are not meaningful in our search context, so we do not have a realistic model of query arrival rates. Our evaluation methodology allowed us to measure variance and analyze latency by query length in a meaningful way. To be consistent, for the effectiveness experiments on the microblog queries, we adopted the same methodology, but post-filtered the results to discard tweets after the query time associated with each topic, prior to evaluation. Following the evaluation guidelines from the TREC 2012 microblog track, we constructed the final ranked list by simply sorting documents by score (more details later).

We realize that this evaluation methodology does not fully exercise the real-time aspects of our search problem, where documents and queries arrive in arbitrarily-interleaved order, but defend our choice as follows. First, in a high-fidelity simulation of real-time query and tweet arrival, our implementations would need to address the concurrency challenges discussed in Section 2.3, which would require a substantial software engineering effort. We see concurrency management primarily as a system's issue orthogonal to the trade-offs we examine here. Any production implementation must deal with program correctness in a multithreaded execution environment, but there is nothing about our candidate generation algorithm that makes concurrency management more difficult. We return to discuss this issue as part of future work in Section 7 but note here that since the concurrency challenges affect all implementations

equally, the fact that we set aside the issue will not affect our experimental results. Second, even if we wished to evaluate the concurrency aspects of the real-time search problem, it is not completely clear how we should design the experiments and what exactly to measure. For example, how do we quantify mean query latency if each query was issued against a different index? What are the arrival rates of queries relative to documents? A proper evaluation methodology that accounts for these and other complexities remains an open question, worthy of a separate study in its own right. For these two reasons we have simplified the query-time aspects in our experiments, but do not believe that this choice alters our findings.

5.2. Implementation of Candidate Generation Algorithms

We implemented the algorithms presented in this article in Java on top of the open-source Ivory toolkit.[8] The choice of Java puts us at a disadvantage compared to implementations in C or C++. This is especially true when measuring short query latencies, because fundamental aspects of Java such as object overhead become a nontrivial fraction of the total running time. Thus, our efficiency numbers should be interpreted with this caveat in mind. However, 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. As an example, Twitter's production Earlybird retrieval engine (described in Section 2.3) serves over two billion queries per day and is entirely written in Java. Other organizations such as LinkedIn have also adopted the Java Virtual Machine as a platform for search and data processing. Although some may find our language choice to be unusual given our focus on efficiency, we believe our design is well supported by existing commercial services in production.

We organized our index using the same parameters reported in Busch et al. [2012]. Our index size is capped at 2^{24} (about 16 million) tweets, and coincidentally, the entire Tweets2011 collection fits in one segment. To build the inverted index, we used four memory pools for the base postings lists, $\mathcal{P} = \{1, 4, 7, 11\}$, as prescribed by Busch et al. For the collection of Bloom filter chains $\mathcal{P}'$, recall that we require one integer in the first slice to store a counter—thus, a choice of $P'_0 = 1$ would make little sense, since there would only be 32 bits left over for the Bloom filter. Therefore, we set P'_0 to 2 in the collection of Bloom filter chains. Otherwise, the configuration is the same as with the base postings lists, with $\mathcal{P}' = \{2, 4, 7, 11\}$. Following standard practice, terms in tweets were stemmed and stopwords were removed prior to indexing.

In order to evaluate the performance of our proposed candidate generation algorithm, we implemented four baselines. For conjunctive retrieval (i.e., postings list intersection), we implemented the small adaptive and the SvS algorithms [Demaine et al. 2001; Culpepper and Moffat 2010]. For disjunctive retrieval, we used WAND [Broder et al. 2003] with the simplified IDF scoring model (see Section 4) and exhaustive OR. Because these algorithms were not designed to work on our linked lists of memory slices, we had to adapt them, as described next. In all cases, the candidate generation stage returned 1,000 hits.

To ensure a fair comparison, we have put in a best-faith effort to optimize our implementation of small adaptive, SvS, and WAND. We are confident that observed differences are not caused by neglect or an underperforming baseline. Finally, we note that all implementations are presently single-threaded.

5.2.1. Small Adaptive. The small adaptive algorithm for postings list intersection works on standard document-sorted inverted lists as follows: it sets an eliminator e to the

[8]http://ivory.cc.

first document ID of the first postings list. It then cycles through the postings lists, searching for the current eliminator. The search is carried out using a one-sided binary search, or *galloping* search. If e exists in all postings lists, the algorithm adds e to the candidate list and repeats this process for the next document ID. If a list L does not contain e, then the algorithm picks the smallest element in L that is larger than e as the eliminator and repeats the cycle. In the standard implementation, the algorithm starts at the beginnings of the postings lists and searches for increasing document IDs. In our implementation, everything is flipped around since we start at the ends of the postings lists, for example, the search for eliminators proceeds backwards. The other implementation difference concerns the fact that our postings lists are not contiguous: when the search for an eliminator on a postings list L overshoots a slice, we follow the backpointer to retrieve its previous slice. The algorithm terminates when all elements of any postings list are examined or when it accumulates k elements in the candidate list. Since we are traversing the postings in reverse chronological order, the results are also sorted in this manner.

5.2.2. SvS. The SvS algorithm for postings list intersection first sorts the postings lists in increasing length. It begins by intersecting the two smallest lists. Then, at each step, the algorithm intersects the current result set with the next postings list, until all lists have been consumed. In our implementation, we use galloping search to intersect postings lists. Unlike small adaptive, the SvS algorithm has no mechanism for early termination, and thus must exhaustively compute the entire intersection. Once the intersection set is constructed, the algorithm returns the most recent k documents.

5.2.3. WAND. The WAND algorithm was discussed in Section 4. We used the simplified IDF scoring model, which obviates the need for the Block-Max optimization of Ding and Suel [2011]. Note that our implementation is also flipped around with respect to the standard implementation, since we traverse postings from their ends. As with the other algorithms, since our inverted lists are stored in discontiguous slices, we adapted how the pointers move—but other than these differences, our implementation of WAND is what one would expect.

5.2.4. OR. Finally, we implemented exhaustive OR, which considers all documents that have at least one query term. Note that unlike in the WAND algorithm, with exhaustive OR, we compute the full BM25 document scores. We include this condition as a reference to assess the impact of approximations introduced by the other algorithms.

5.3. Second-Stage Rescorer

Although the focus of this work is on fast candidate generation, to illustrate end-to-end retrieval effectiveness, we implemented a learning-to-rank model to rescore the candidate documents. We used the simple yet effective greedy feature selection algorithm described by Metzler [2007]. In this approach, we trained a linear ranking function where features are iteratively added to the model, one at a time, according to a greedy selection criterion. At each iteration, the feature that yields the largest marginal increase in effectiveness is selected. This algorithm continues until the difference in effectiveness between successive iterations drops below a given threshold.

Our model used the standard set of features described by Metzler, detailed in Asadi and Lin [2012a]. We use two families of scoring functions, based on the Dirichlet score from language modeling and BM25. Each family consists of a unigram feature, a bigram proximity feature that takes term order into account (parameterized with a window $w \in \{1, 2, 4, 8, 16\}$), and a bigram feature score for unordered terms (parameterized with a window $w \in \{2, 4, 8, 16, 32\}$). In total, there are 22 features. Similar features were used in related work on Web retrieval [Wang et al. 2011; Tonellotto et al. 2013], as

well as the highly-ranked run of Metzler and Cai [2011] in the TREC 2011 microblog track. Two models were trained separately on the TREC 2011 and 2012 microblog queries using NDCG [Järvelin and Kekäläinen 2002] and evaluated on the other set (i.e., two-fold cross validation).

We readily concede that there is nothing novel about our choice of features and that our feature set neither captures tweet-specific characteristics nor temporal signals [Li and Croft 2003; Jones and Diaz 2007; Dakka et al. 2008; Efron 2010; Elsas and Dumais 2010]. However, we stress that the focus of this work is not on learning-to-rank for tweet search, but rather candidate generation in a two-stage retrieval architecture. Our rescoring model is only meant to provide context for end-to-end experiments and thus only needs to be reasonably competitive with existing approaches.

As an aside, we also experimented with gradient-boosted decision trees (GBRTs), which represent the state of the art in learning to rank, using the jforests open-source implementation [Ganjisaffar et al. 2011].[9] However, due to the relatively small number of features, it was very easy to overfit and required significant parameter tuning. Even in the best configuration, GBRTs on our particular feature set performed only about as well as our linear model. Hence, we opted for the simpler method.

5.4. Evaluation Metrics

There are three important considerations in the design of search engines: the quality of the results, query evaluation speed (time), and memory footprint (space). Each of the algorithms explored in this article represents a trade-off point in this design space, which we seek to better understand. For the BWAND algorithm, these three considerations are influenced by the parameters used in the Bloom filters: r (number of bits per element) and κ (number of hash functions). Based on preliminary calculations from theoretical bounds (see Section 2.4), we restricted our consideration to $r = \{8, 16, 24\}$, and $\kappa = \{1, 2, 3\}$. Note that these values are fixed for all Bloom filters.

Speed is measured in terms of query latency: the amount of time it takes to perform candidate generation and return a list of top $k = 1{,}000$ documents. In conjunctive retrieval, the returned documents contain all query terms and are sorted in reverse chronological order, whereas in disjunctive retrieval, documents are sorted with respect to the scoring model. Elapsed time is measured using Java's `System.nanoTime()` method and reported in microseconds. We compute average latency across five trials on each query set, and also break down the results by query length. Since our current implementation is single-threaded, throughput (i.e., queries per second) is simply the inverse of latency—there is presently no mechanism to trade off those two measures.

We define index size as the number of 32-bit integers that have been allocated to ingest a collection of documents (recall that postings lists and Bloom filters are simply integer arrays). To measure index size, we index the entire document collection and then compute the total number of allocated 32-bit integers. For the base postings lists, this includes slices that have been allocated but may contain few postings (i.e., the last allocated slice which has not filled up yet). For example, let us suppose that for term t, we just allocated a 2^{11} slice in pool 4 and then the indexing process finished. The entire slice would count toward memory usage, even though most of the slice is "wasted" and will never be filled. For the Bloom filter chains, we include space occupied by all allocated Bloom filters, regardless of whether or not they have reached capacity. Note that the Bloom filter chains are auxiliary data structures required for BWAND in addition to the standard base postings lists.

Finally, let us discuss effectiveness. This work focuses on candidate generation in a two-stage retrieval architecture, but since we introduce approximations that trade

[9]http://code.google.com/p/jforests/.

Table I. End-to-End Effectiveness in Terms of Precision and NDCG at Various Cut-offs for Different Candidate Zeneration Algorithms

(a) Conjunctive Query Processing

	Precision			NDCG			
	@5	@10	@30	@1	@3	@5	@10
SvS	0.36	0.30	0.19	0.30	0.28	0.27	0.25
BWAND (8,1)	0.39*	0.34*	0.22*	0.30	0.29	0.29*	0.27*
BWAND (24,3)	0.35	0.30	0.19	0.30	0.28	0.27	0.25

(b) Disjunctive Query Processing

	Precision			NDCG			
	@5	@10	@30	@1	@3	@5	@10
OR	0.49	0.45	0.36	0.37	0.37	0.38	0.38
WAND	0.49	0.46	0.37	0.38	0.37	0.38	0.38
BWAND (8,1)	0.48	0.43	0.34	0.38	0.37	0.38	0.37
BWAND (24,3)	0.48	0.44	0.35	0.38	0.37	0.38	0.37

Note: *indicates statistical significance wrt. SvS and OR.

effectiveness for speed, it makes sense to report both component-level and end-to-end metrics. At the component level, we measured effectiveness in terms of relative recall with respect to the exact baseline: SvS in the case of conjunctive query processing and exhaustive OR in the case of disjunctive query processing. That is, of all documents retrieved by the baseline, what fraction is retrieved by our candidate generation algorithm? In the conjunctive case, this metric captures the false positive errors introduced by the Bloom filters, and in the disjunctive case, this metric additionally captures a number of simplifications and approximations in the scoring model compared to BM25. Relative recall is computed via macro-averaging (i.e., computed per topic, then averaged across topics), which has the effect of disproportionately weighting topics that have fewer matching documents. Note that since we are concerned with generating a candidate list that will be passed to a second-stage rescorer, precision or any metric that incorporates precision such as average precision or NDCG is not an appropriate component-level measure.

The other aspect of quality is end-to-end effectiveness. Given the emphasis on early precision in the web context, we measured precision and NDCG at various cut-offs [Järvelin and Kekäläinen 2002]. The ubiquity of Twitter mobile usage also suggests that these metrics are appropriate—the limited screen size of mobile devices means that we should emphasize early precision. Following the evaluation guidelines from the TREC 2012 microblog track, we constructed the final ranked list by sorting documents by score, without taking into account the tweet creation time.

6. RESULTS

6.1. Effectiveness: End-to-End Results

We begin with the most important question about our proposed BWAND algorithm: in terms of end-to-end effectiveness, how does it compare to the exact baselines?

Precision and NDCG at various cut-offs for all 110 microblog topics from TREC 2011 and 2012 are shown in Table I. In all cases, the candidate generation algorithms returned 1,000 hits that are then rescored by the second stage, as described in Section 5.3. For conjunctive query processing, SvS served as the baseline and was compared against BWAND in conjunctive query processing mode. For disjunctive query processing, exhaustive OR served as the baseline and was compared against WAND and BWAND in disjunctive query processing mode. For BWAND, the relevant Bloom filter parameter settings

Table II. Statistics for Retrieved Results at Rank Ten Cut-off for Different Candidate Generation Algorithms

(a) Conjunctive Query Processing

	#topics	unjudged	0	1	2
SvS	81	43	207	168	155
BWAND (8,1)	101	120	350	200	165
BWAND (24,3)	82	50	210	168	155

(b) Disjunctive Query Processing

	#topics	unjudged	0	1	2
OR	110	97	510	281	212
WAND	110	91	511	284	214
BWAND (8,1)	110	97	531	261	211
BWAND (24,3)	110	94	527	267	212

Note: Last three columns indicate relevance grades: nonrelevant (0), relevant (1), highly-relevant (2).

are r (number of bits per element) and κ (number of hash functions): in these experiments, we report results for $r = 8, \kappa = 1$, denoted (8,1) in the tables, and $r = 24, \kappa = 3$, denoted (24,3) in the tables; these represent the extremes of the parameter value ranges we explored. A paired t-test ($p < 0.05$) was used to determine the statistical significance of differences observed with respect to the baselines. Table II provides statistics for retrieved results at rank ten to offer more context: it shows the number of topics for which the algorithm retrieved results, the number of unjudged tweets, and the number of retrieved tweets that were nonrelevant (0), relevant (1), and highly-relevant (2).

Let us first examine the conjunctive query processing results: the SvS algorithm returned results for only 81 of 110 topics, whereas the BWAND (8,1) condition returned results for 101 topics; the BWAND (24,3) condition returned results for 82 topics. Note that in all cases the effectiveness metrics in Table I were computed over all 110 topics. Some examples of queries for which SvS did not return any results include "Pakistan diplomat arrest murder" and "Dog Whisperer Cesar Millans techniques". For the most part, they are long queries that appear to over-specify the information need. For the BWAND (8,1) condition, the false positive errors associated with Bloom filters are advantageous in returning results that do not actually contain all the query terms. In cases where exact postings list intersection (SvS) returns an empty set, something is always better than nothing. On the other hand, in cases where exact postings list intersection returns a non-empty set, the false positives introduce noise, but it appears that the second-stage rescorer is not adversely affected by these spurious documents. Overall, BWAND (8,1) achieves significantly higher precision at all cut-offs and NDCG at cut-offs five and ten. For BWAND (24,3), we observe no significant differences compared to SvS. There is a simple explanation of the effect of different Bloom filter settings, which will become apparent in the next section.

Turning our attention to the disjunctive query processing results, let us first review what the experimental conditions were designed to reveal: the exhaustive OR condition scores candidates with the full BM25 model and serves as the reference to assess the impact of approximations and simplifications introduced by the other approaches. The WAND condition adopts the simplified IDF scoring model, which ignores term frequencies and length normalization, but is otherwise exact in the sense that it produces a precise ranking (without any errors). In the BWAND conditions, we introduce the additional constraint that retrieved documents must contain the rarest query term and the possibility of false positives from the Bloom filters. The

Table III. Relative Recall for Different Bloom Filter Chain Settings with Respect to SvS for Conjunctive Query Processing and Exhaustive OR for Disjunctive Query Processing

(a) Conjunctive			
$r\backslash\kappa$	1	2	3
8	0.981	0.993	0.997
16	0.991	0.998	0.999
24	0.994	0.998	0.999

(b) Disjunctive			
$r\backslash\kappa$	1	2	3
8	0.354	0.365	0.368
16	0.364	0.369	0.370
24	0.367	0.370	0.370

results in Table I show no significant differences between exhaustive OR and WAND in disjunctive query processing mode. There are also no significant differences between exhaustive OR and BWAND in disjunctive query processing mode, with either Bloom filter parameter setting. Looking at Table II, we can confirm that these results do not appear to be the product of unjudged documents or idiosyncrasies in the way that judgment pools were constructed during the TREC evaluations.

In absolute terms, how do these figures compare to official runs submitted for the TREC microblog evaluations? Recall that since the focus of our work is candidate generation and not learning-to-rank for tweet search per se, our goal is to build a reasonable second-stage rescorer to illustrate characteristics of our candidate generation algorithm. In this respect, our results compare favorably. According to the TREC 2012 microblog track overview paper (notebook version) [Soboroff et al. 2012b], the best performing run achieved a P30 of 0.27. In TREC 2011, we observed higher scores using query expansion, for example, Metzler and Cai [2011] achieved a P30 score of 0.45 by using Latent Concept Expansion [Metzler and Croft 2007], but this does not represent a fair comparison with our runs, since expansion techniques are substantially slower. Regardless, the results reported here are well above the median scores from TREC 2011 and 2012, and thus we believe that the conclusions drawn here are trustworthy.

Overall, we see that disjunctive query processing is more effective than conjunctive query processing. The small size of the document collection, we believe, is a major cause; for approximately one quarter of the topics in our test collection, SvS returned zero results. We will return to discuss this issue in Section 7 but note here that our tweet collection is only a small sample of the entire tweet stream—it is perhaps possible that we would obtain different results using a larger collection.

All considered, here is the bottom line: experiments confirm that our BWAND candidate generation algorithm with Bloom filter chains yields end-to-end effectiveness that is statistically indistinguishable from exact algorithms, in either conjunctive query processing mode or disjunctive query processing mode.

6.2. Effectiveness: Bloom Filter Parameters

Our next set of experiments examined the impact of Bloom filter parameter settings on component-level and end-to-end effectiveness. Table III shows the relative recall of BWAND with different Bloom filter configuration parameters, r (bits per element) and κ (number of hash functions), for both conjunctive and disjunctive query processing. That is, of the documents retrieved by the baselines (SvS and exhaustive OR), what fraction is retrieved by BWAND? Table IV shows relative recall figures, but only with respect to relevant documents.

In the conjunctive query processing case, with BWAND we achieve nearly perfect recall in all cases. This means that nearly all documents retrieved by the exact algorithm are also retrieved by our approximate algorithm and available to the second-stage rescorer. Documents are lost in that the false positive errors associated with Bloom filters cause

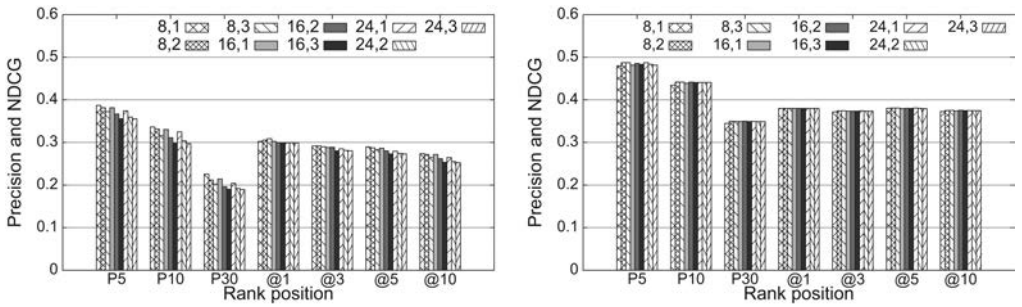
Table IV. Relative Recall for Different Bloom Filter Chain Settings Considering only Relevant Documents with Respect to SvS for Conjunctive Query Processing and Exhaustive OR for Disjunctive Query Processing

(a) Conjunctive

$r\backslash\kappa$	1	2	3
8	0.972	0.992	0.997
16	0.989	0.999	1.0
24	0.994	1.0	1.0

(b) Disjunctive

$r\backslash\kappa$	1	2	3
8	0.684	0.707	0.711
16	0.704	0.714	0.715
24	0.708	0.714	0.714



(a) Conjunctive (b) Disjunctive

Fig. 6. End-to-end effectiveness of BWAND with various Bloom filter settings (κ, r) in both conjunctive and disjunctive query processing mode.

BWAND to erroneously include documents in the top k that do not contain all query terms, which crowds out documents that actually do, since we only return 1,000 hits in the candidate generation stage. If we performed full postings list intersection without limiting the size of the result set, the Bloom filter errors would not lower relative recall, but just create a bigger results set. Focusing on only the relevant documents, relative recall decreases but is still well above 97%. In the disjunctive query processing case, relative recall with respect to exhaustive OR is low (as expected), but when focused only on relevant documents, relative recall increases substantially. That is, although with BWAND we lose documents that would have been retrieved under OR (with full BM25 scoring), the lost documents have a greater tendency to be nonrelevant.

Tables III and IV together show that relative recall can be controlled by appropriately tuning Bloom filter parameters, and the trade-offs are exactly what we would expect. Increasing the number of hash functions κ reduces the false positive rate and thus improves relative recall. Similarly, increasing the number of bits per element r reduces hash collisions, which also reduces the false positive rate and increases relative recall. However, larger values of κ reduce speed and larger values of r increase memory requirements. We further explore these trade-offs in Sections 6.3 and 6.4.

What is the impact of various Bloom filter settings on end-to-end effectiveness? The answer is shown in Figure 6, where we plot precision and NDCG at various cut-offs for all 110 TREC microblog topics from 2011 and 2012. We find that different settings of r and κ do not yield any significant differences; in fact, in many cases, the absolute values do not change at all. However, we do notice a trend for conjunctive query processing: increasing r or κ results in a slight drop in effectiveness, although not statistically significant. We explain this as follows: smaller values of r and κ generate more false positives, where the candidate generation algorithm retrieves documents that may not actually contain all query terms. This makes query evaluation more "disjunctive-like", and these documents are often relevant, as identified by the

Table V. Average Query Latency (in microseconds) for the TREC 2005 Terabyte Track Queries and the AOL Queries to Retrieve 1,000 Candidate Documents, Averaged Across Five Trials with 95% Confidence Intervals

(a) Conjunctive Query Processing

	S. A.	SvS	$r\backslash\kappa$	1	2	3
TREC05	223 (±2)	175 (±2)	8	61 (±1)	75 (±1)	89 (±1)
			16	73 (±2)	78 (±3)	83 (±1)
			24	70 (±1)	82 (±1)	83 (±2)
AOL	292 (±1)	193 (±1)	8	65 (±1)	82 (±1)	88 (±1)
			16	71 (±1)	72 (±1)	84 (±3)
			24	73 (±1)	76 (±1)	89 (±1)

(b) Disjunctive Query Processing

	OR	WAND	$r\backslash\kappa$	1	2	3
TREC05	42,079 (±418)	1,527 (±7)	8	347 (±44)	376 (±47)	396 (±52)
			16	367 (±49)	386 (±50)	426 (±47)
			24	359 (±49)	387 (±48)	413 (±46)
AOL	156,291 (±3928)	2,777 (±12)	8	475 (±22)	516 (±24)	529 (±21)
			16	463 (±21)	497 (±23)	527 (±23)
			24	477 (±22)	509 (±23)	586 (±24)

second-stage rescorer. Increasing r or κ makes the candidate results closer to exact postings intersection, which is actually undesirable because disjunctive query processing yields higher effectiveness overall. In other words, the errors introduced by the Bloom filters serendipitously increase end-to-end effectiveness.

6.3. Query Evaluation Speed

Having established that our BWAND candidate generation algorithm is statistically indistinguishable from exact baselines in terms of end-to-end effectiveness, we now turn our attention to query evaluation speed. The question is how much speed can we gain by "cutting corners"?

Table V shows the average query latency of BWAND for different values of r (bits per element) and κ (number of hash functions). For these efficiency experiments, there are not enough queries from the microblog track to produce meaningful results, and so we used queries from the TREC 2005 terabyte track and the AOL query logs. The table also includes query latencies of the baseline algorithms for reference. Reported values represent the average across five trials for each parameter setting and include the 95% confidence intervals.

For conjunctive query processing, we find that the SvS algorithm is faster than small adaptive (abbreviated "S. A." in the table), even though it does not have the ability to early-terminate upon finding k hits (unlike with small adaptive). This finding appears to be consistent with results reported by Culpepper and Moffat [2010]. With the fastest setting of BWAND ($r = 8, \kappa = 1$), we observe nearly a threefold increase in speed over SvS. At the other extreme of parameter settings we explored ($r = 24, \kappa = 3$), BWAND is still more than twice as fast as SvS.

For disjunctive query processing, we observe that BWAND with $r = 8, \kappa = 1$ is more than four times faster than WAND on the TREC queries and is more than five times faster on the AOL queries. With $r = 24, \kappa = 3$, BWAND is still more than three times faster than WAND on the TREC queries and more than four times faster on the AOL queries. Exhaustive OR is included here only for reference; unlike the other algorithms, we did not make a significant effort to optimize its implementation.

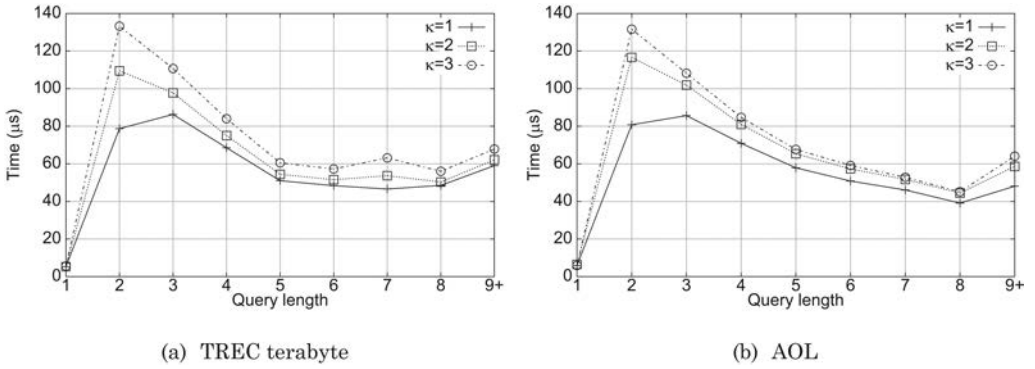


Fig. 7. Effect of query length on latency (μs) for BWAND performing conjunctive retrieval ($r = 8$).

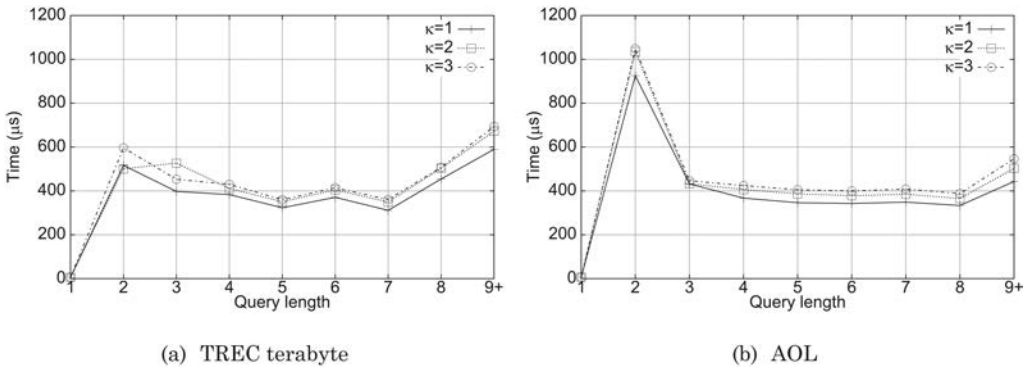
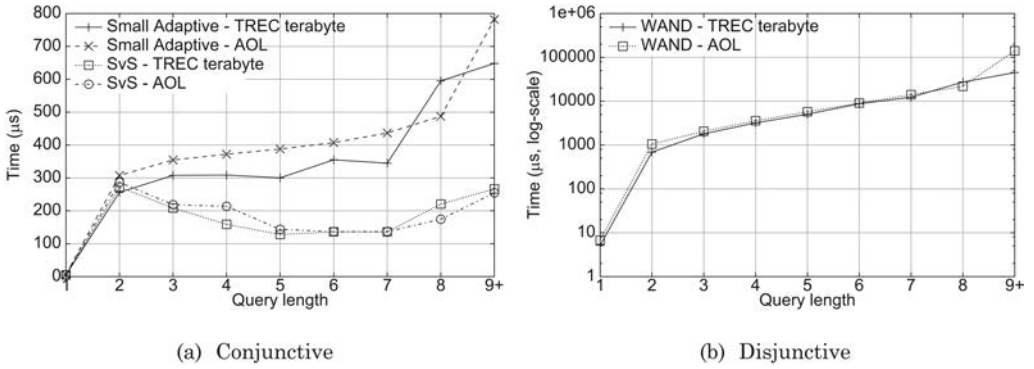


Fig. 8. Effect of query length on latency (μs) for BWAND performing disjunctive retrieval ($r = 8$).

How do Bloom filter settings impact query latency? From first principles, we can reason that increasing the number of hash functions κ will increase average query latency because the algorithm needs to compute more hash values and probe additional bit positions for membership tests. From Table V, we do see that this prediction is borne out: query evaluation becomes slower with increasing values of κ. However, larger values of κ have the advantage of reducing Bloom filter false positives, thus yielding higher component-level recall, per the results in Tables III and IV.

Increasing the bits per element parameter r brings into play two counteracting factors. On the one hand, increasing r leads to more individual Bloom filters in the chain (since the capacity of each is reduced), hence more pointer chasing and less locality. This translates into more cache misses and longer memory latencies. On the other hand, as the size of the Bloom filters increases, the false positive rate drops. Therefore, fewer hash computations are needed to reject a nonexistent document ID. From the results in Table V, we see that query latency is relatively insensitive to r; using three times more bits per element only increases query latency slightly. In some cases, $r = 24$ is actually faster than $r = 16$. From Tables III and IV, we see the benefit of increasing r in terms of higher relative recall.

Figures 7 and 8 show the average query latency for the TREC terabyte queries and the AOL queries broken down by query length for conjunctive and disjunctive query processing using $r = 8$. For $r = 16$ and $r = 24$, the plots appear similar, so they are not included for brevity. Due to different query characteristics, results differ slightly from one query set to another, but the trends are consistent. For single-term queries,



(a) Conjunctive (b) Disjunctive

Fig. 9. Effect of query length on latency (μs) for baseline algorithms.

both conjunctive query processing and disjunctive query processing are very fast; this is because BWAND can early terminate as soon as it reads k documents from the posting list associated with the query term. For multitoken queries, longer queries are not necessarily slower, for example, with conjunctive query processing, seven-term queries are actually faster than three-term queries (this is possible because longer queries tend to contain rarer query terms). A similar, though weaker, effect is observed for disjunctive query processing—query latency appears to be relatively insensitive to the length of queries (except for very long queries).

As a reference, query latencies for the baseline algorithms, broken down by query length, are shown in Figure 9. Interestingly, for conjunctive query processing, the behavior of SvS and small adaptive diverge for longer queries. Small adaptive steadily becomes slower as query length increases, but SvS actually becomes faster in some cases (once again, because longer queries tend to contain rarer query terms). For disjunctive query processing, WAND latency grows with the number of query terms; note that the y axis of Figure 9(b) is in log scale.

Finally, what is the impact of these observed candidate generation speedups on end-to-end query latency? This is a tough question to answer, which will obviously depend on the features and the implementations of the feature extraction algorithm and the second-stage rescorer. Nevertheless, it may be helpful to present the reader with some rough figures to contextualize our results. Note that we have made no effort to optimize our feature extraction and rescorer implementations, so our results should be interpreted with this caveat in mind. With 1,000 candidates, it takes roughly two milliseconds per query for feature extraction in Java and roughly 0.3 milliseconds per query for rescoring using a Python script. Note that the feature extraction time is for computing all features that are presented to the learner—only a subset are actually selected in the final scoring model. Applying these figures to the AOL queries, with BWAND ($r = 8, \kappa = 1$) we can reduce end-to-end latency by roughly 5% in the conjunctive query processing case (compared to SvS) and roughly 45% in the disjunctive query processing case (compared to WAND). Since conjunctive query processing is much faster, it occupies a smaller fraction of the overall retrieval time, so improving candidate generation speed has a smaller overall effect. These figures should be interpreted as lower bounds, since with more optimized implementations of the feature extractor and the second-stage rescorer, a greater fraction of the overall time budget will be occupied by candidate generation (but of course, if we introduce more computationally-intensive features, feature extraction will take longer). Although the end-to-end latency improvements will depend on the exact composition of the search pipeline, these rough

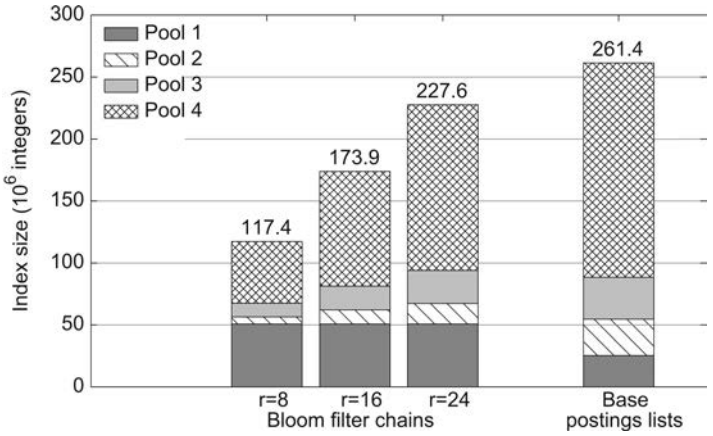


Fig. 10. Effect of r (bits per element) on the size of the Bloom filter chains for the Tweets2011 collection, with the size of the base postings lists as reference. Index size is measured in the number of allocated 32-bit integers.

calculations show that the optimizations introduced in BWAND can have a real and noticeable effect.

6.4. Index Size

The final dimension of evaluation is index size, which is the amount of space required by the Bloom filter chains and the base postings lists. Index size is illustrated in Figure 10 as a function of r, number of bits per element; numbers are reported in units of 32-bit integers. These figures include all allocated space, including slices in the base postings lists that have not yet been filled and slices for Bloom filters that have not yet reached capacity. For each case we have broken down memory usage by the different pools. Note that the uneven memory usage in each pool is not a cause for concern. Conceptually, the pools represent unbounded integer arrays from which slices are allocated, but they are implemented as arrays that grow in fixed but large blocks.

Note that for our BWAND candidate generation algorithm to work, we need the Bloom filter chains in addition to the standard postings lists. For example, with $r = 8$, we require about 45% more memory than we would otherwise need for the exact algorithms. Therein lies the true cost of our approach: results from the previous sections show that we can substantially reduce query latency without giving up end-to-end effectiveness, but the extra memory requirements are unavoidable. In essence, we are trading space for time by building auxiliary data structures that increase the speed of candidate generation. However, for modern retrieval environments where query latency has a direct impact on site usage [Brutlag 2009], we believe this is a justifiable trade-off.

The final takeaway message from these experiments is that with our Bloom filter chains, there is no free lunch. Although different values of r and κ do not have an impact on end-to-end effectiveness, different settings affect component-level effectiveness in candidate generation. This metric may be important in cases where BWAND is used to analyze tweet collections for other purposes beyond search, for example, as the first step in an event summarization system (where recall would be more important). It is possible to achieve higher relative recall, but at the cost of longer query latencies or a larger memory footprint. If we wish to make the Bloom filter chains more compact, we must either sacrifice speed or quality, and so on. The balance between these various trade-offs is shown by our experiments, but it is up to a developer to select the

parameter setting that is most appropriate for a particular hardware configuration and application context.

7. LIMITATIONS AND FUTURE WORK

The starting point of our work is the observation that in a two-stage retrieval architecture, candidate generation only needs to be good enough, since the second-stage rescorer can take advantage of machine-learned models and rich features to provide high-quality results. Thus, we hypothesized that it is possible to cut corners in candidate generation by introducing approximations that do not adversely impact end-to-end effectiveness. Experiments do indeed confirm this hypothesis. We are well aware, however, that many of our techniques are tweet-specific and not applicable to the general case, for example, we depend on the 140-character length limitation as justification to ignore term frequencies and length normalization. These tricks are applicable to few other retrieval domains (SMS, perhaps), and may leave the reader somewhat unsatisfied. However, this work should be viewed as a study of *domain-specific* retrieval: there are plenty of other retrieval problems (e.g., patent search, enterprise search, genomics search, etc.) that benefit from specialized techniques without general applicability to IR problems. We believe that Twitter, or at least the idea of broadcasting short messages to interested parties, has acquired sufficient critical mass that it cannot be dismissed as a passing fad—thus, it is worthwhile to explore retrieval techniques optimized for such types of documents.

We readily concede that a limitation of this work is that our conclusions are arrived at using test collections from recent TREC microblog tracks, which represent only one particular view of the types of information needs that tweet collections might be useful for. Most NIST assessors are retired intelligence analysts and do not match the demographic characteristics of the typical Twitter user today—so how do they know what Twitter queries should look like? It is likely that the TREC microblog topics are representative of some class of information needs, but we have no idea which and, more importantly, the relationship of these topics to the broader range of information needs on tweets. In the Web context, researchers are familiar with Broder's taxonomy [2002], and the field has come to understand that TREC ad hoc topics are representative of so-called "torso" queries—this allows researchers to better contextualize results without overstating their claims. Since the real-time search task on tweets has only come to the attention of researchers recently, we have no comparable framework on which to hang our results. There is also the important element of time: the current tweet collection spans a short two-week window in January 2011, which constrains the types of topics that are practical. Would the types of information needs change qualitatively if assessors were given a much larger collection for topic development? Or a more recent collection? To what extent would different topics alter the conclusions presented here? The answer is that we do not know for sure and that more work in understanding information needs is needed.

There has been some work that begin to answer these questions: Efron [2011] discussed some of the issues presented here and defined different types of searches on microblogs. Teevan et al. [2011] attempted to formulate a taxonomy of Twitter search, but their log data came from Twitter queries gathered via the Bing search toolbar, not Twitter search logs. Lin and Mishne [2012] studied the temporal characteristics of actual Twitter search queries but did not attempt to categorize the queries topically or develop a taxonomy. Clearly, there is much more work here to be done.

Of the findings presented in this article, the one most potentially affected by idiosyncrasies in the test collection is the relative effectiveness difference between conjunctive and disjunctive query processing. Conventional wisdom suggests that with sufficiently large collections, conjunctive query processing is adequate because it will return enough

candidates for the second-stage rescorer to compose a final ranked list with high early precision. This, however, does not appear to be true in our experiments—for nearly a quarter of the topics, postings list intersection returned zero results. A closer examination shows that these are queries such as "Dog Whisperer Cesar Millans techniques", which appear to over-specify the information need. To what extent are these queries "natural" or representative of "real" needs? We do not know the answer, and this is exactly the research we question discussed that demands more attention. The effectiveness difference between conjunctive and disjunctive retrieval is perhaps the direct result of the small size of our tweet collection, which is only a sample of the full Twitter stream. Would the same queries have returned zero results on a larger collection? Once again, we do not know, and answering this question is difficult because of data collection issues.

Another limitation of this work is the heavy influence of the Earlybird architecture, which of course represents only a single point in the design space of real-time retrieval engines. Earlybird makes a number of assumptions, such as relatively small uncompressed index segments and delayed compression of postings, and employs a specific strategy for postings allocation. While these design choices are carefully justified in Busch et al. [2012], Earlybird nevertheless represents only one possible architecture. Since our work extends this design, we inherit all its assumptions. On the other hand, work in the space of real-time search architectures is so sparsely populated that we are not aware of similarly-detailed expositions of production systems that can serve as the basis of further academic research. This is an underexplored topic in the literature, and we hope to see more contributions from industry and academia in the future.

In addition to many open questions about information needs in the context of tweets previously discussed, there are many interesting directions for future work. One important issue to address is the current single-threaded implementation of our candidate generation algorithm, which under-utilizes multicore processors. Tatikonda et al. [2011] recently examined the problem of postings intersection on multicore architectures, but in the real-time search context, we additionally face the concurrency challenges discussed earlier, where the retrieval engine needs to coordinate concurrent reads and writes to shared index structures as queries and documents arrive in an interleaved order. Busch et al. [2012] described the use of memory barriers to maintain search correctness, eschewing heavyweight mechanisms such as locks to ensure atomic index updates. For our Bloom filter chains, we face similar issues: indexing operations on one thread might be setting Bloom filter bits while another is probing the same filter. Since both indexing and multiterm queries involve accesses to multiple Bloom filters, interleaved execution orders might yield incorrect search results—although without more data (i.e., execution traces from production servers) it is difficult to know the extent to which this is actually an issue. Similar to Earlybird, locking multiple Bloom filters to guarantee atomic index updates will result in very poor performance because there will be significant lock contention on frequently-occurring terms. Perhaps one solution to this challenge is to not explicitly guarantee search correctness, but model inconsistencies as part of the Bloom filter errors—since we are dealing with approximate data structures anyway, this can be viewed as yet another source of approximation error. Nevertheless, to explore this issue in more detail requires access to larger tweet collections and actual query logs, both of which are currently not available to academic researchers.

Other future directions include taking advantage of improved variants of Bloom filters that have been developed since the original formulation, for example, Tirdad et al. [2011] recently proposed a type of Bloom filter for representing documents that are aware of co-occurring terms, which has the effect of substantially reducing false positive error rates. It may be possible to adapt this idea to our usage scenario, for example,

by making our Bloom filters aware of document similarity, that is, documents that are likely to be retrieved by the same query. Another possible extension is to relax our assumption in the IDF scoring model that all term frequencies are one by using *counting* Bloom filters [Fan et al. 2000], which, as the name suggests, allow us to store counts associated with keys. These data structures could be used to store term frequencies—the advantage is that BWAND would now have applicability beyond tweets to retrieval in other domains, where term frequencies are important. Another interesting extension to our work would be to apply Bloom filters to computing proximity-based features. Determining whether two query terms are part of a phrase or within a window of w terms also boils down to traversals of sorted integer sequences (of term positions, in this case). Since term proximity is an important feature used by many ranking models, faster algorithms for computing such features are worth exploring.

8. CONCLUSION

This work tackles the problem of candidate generation in a two-stage retrieval architecture for real-time tweet search. We bring together two previously unrelated threads in the literature: a dynamic postings allocation policy for incremental indexing and a retrieval algorithm based on Bloom filters. These two ideas come together in a data structure for holding a collection of Bloom filter chains, which are novel extensions of standard Bloom filters that can dynamically expand to efficiently represent a growing list of monotonically-increasing integers with a constant false positive rate. By storing postings in both sorted integer lists and Bloom filter chains, we are able to devise a novel candidate generation algorithm capable of performing both conjunctive and disjunctive query processing. In end-to-end experiments with a second-stage rescorer, our algorithm yields precision and NDCG scores that are statistically indistinguishable from candidates generated by exact algorithms, but is multiple times faster.

ACKNOWLEDGMENTS

N. Asadi's deepest gratitude goes to Katherine for her invaluable encouragement and wholehearted support. J. Lin is grateful to Esther and Kiri for their loving support and he dedicates this work to Joshua and Jacob. Finally, we wish to thank three anonymous reviewers whose feedback has helped us significantly improve this work.

REFERENCES

ALMEIDA, P. S., BAQUERO, C., PREGUIÇA, N., AND HUTCHISON, D. 2007. Scalable Bloom filters. *Inform. Process. Lett. 101,* 6, 255–261.

ANH, V., DE KRETSER, O., AND MOFFAT, A. 2001. Vector-space ranking with effective early termination. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'01)*. 35–42.

ANH, V. AND MOFFAT, A. 2005. Simplified similarity scoring using term ranks. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'05)*. 226–233.

ASADI, N. AND LIN, J. 2012a. Document vector representations for feature extraction in multi-stage document ranking. *Inform. Retriev.* To appear.

ASADI, N. AND LIN, J. 2012b. Fast candidate generation for two-phase document ranking: Postings list intersection with Bloom filters. In *Proceedings of the 21st Annual International ACM Conference on Information and Knowledge Management (CIKM'12)*. 2419–2422.

ASADI, N., LIN, J., AND BUSCH, M. 2013. Dynamic memory allocation policies for postings in real-time twitter search. arXiv:1302.5302. http://arvix.org/abs/1302.5302.

BAEZA-YATES, R., CASTILLO, C., JUNQUEIRA, F., PLACHOURAS, V., AND SILVESTRI, F. 2007. Challenges on distributed web retrieval. In *Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE'07)*. 6–20.

BARBAY, J., LÓPEZ-ORTIZ, A., AND LU, T. 2006. Faster adaptive set intersections for text searching. In *Proceedings of the 5th International Workshop on Experimental Algorithms (WEA'06)*. 146–157.

Barroso, L. and Hölzle, U. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool.

Bloom, B. 1970. Space/time trade-offs in hash coding with allowable errors. *Comm. ACM 13,* 7, 422–426.

Bose, P., Guo, H., Kranakis, E., Maheshwari, A., Morin, P., Morrison, J., Smid, M., and Tang, Y. 2008. On the false-positive rate of Bloom filters. *Inform. Process. Lett. 108*, 210–213.

Broder, A. 2002. A taxonomy of Web search. *SIGIR Forum 36,* 2, 3–10.

Broder, A., Carmel, D., Herscovici, M., Soffer, A., and 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'03)*. 426–434.

Brown, E. W. 1995. Fast evaluation of structured queries for information retrieval. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'95)*. 30–38.

Brutlag, J. 2009. Speed matters for Google web search. Tech. rep. Google, Mountain View, CA.

Burges, C. 2010. From RankNet to LambdaRank to LambdaMART: An overview. Tech. rep. MSR-TR-2010-82, Microsoft Research, Redmond, WA.

Busch, M., Gade, K., Larson, B., Lok, P., Luckenbill, S., and Lin, J. 2012. Earlybird: Real-time search at Twitter. In *Proceedings of the 28th International Conference on Data Engineering (ICDE'12)*. 1360–1369.

Büttcher, S. and Clarke, C. L. A. 2005. Indexing time vs. query time: Trade-offs in dynamic information retrieval systems. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM'05)*. 317–318.

Cambazoglu, B. B., Zaragoza, H., Chapelle, O., Chen, J., Liao, C., Zheng, Z., and Degenhardt, J. 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'10)*. 411–420.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation (OSDI'06)*. 205–218.

Chiueh, T. and Huang, L. 1999. Efficient real-time index updates in text retrieval systems. Tech. rep. State University of New York and Stony Brook, Stony Brook, NY.

Cormack, G. V., Smucker, M. D., and Clarke, C. L. A. 2011. Efficient and effective spam filtering and re-ranking for large web datasets. *Inform. Retriev. 14,* 5, 441–465.

Culpepper, J. S. and Moffat, A. 2010. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst. 29,* 1, Article 1.

Cutting, D. and Pedersen, J. 1990. Optimization for dynamic inverted index maintenance. In *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'90)*. 405–411.

Dakka, W., Gravano, L., and Ipeirotis, P. G. 2008. Answering general time-sensitive queries. In *Proceedings of the 17th International Conference on Information and Knowledge Management (CIKM'08)*. 1437–1438.

Dean, J. and Ghemawat, S. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI'04)*. 137–150.

Demaine, E. D., López-Ortiz, A., and Munro, J. I. 2001. Experiments on adaptive set intersections for text retrieval systems. In *Revised Papers from the 3rd International Workshop on Algorithm Engineering and Experimentation (ALENEX'01),* Lecture Notes in Computer Science, vol. 2153, Springer Verlag, Berlin Heidelberg, 91–104.

Ding, S. and 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'11)*. 993–1002.

Efron, M. 2010. Linear time series models for term weighting in information retrieval. *J. Amer. Soc. Inf. Sci. Technol. 61,* 7, 1299–1312.

Efron, M. 2011. Information search and retrieval in microblogs. *J. Amer. Soc. Inf. Sci. Technol. 62,* 6, 996–1008.

Elsas, J. L. and Dumais, S. T. 2010. Leveraging temporal dynamics of document content in relevance ranking. In *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining (WSDM'10)*. 1–10.

Fan, L., Cao, P., Almeida, J., and Broder, A. Z. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw. 8,* 3, 281–293.

GANJISAFFAR, Y., CARUANA, R., AND 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'11)*. 85–94.

GUO, R., CHENG, X., XU, H., AND WANG, B. 2007. Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In *Proceedings of the 16th International Conference on Information and Knowledge Management (CIKM'07)*. 751–759.

HAMILTON, J. 2007. On designing and deploying Internet-scale services. In *Proceedings of the 21st Conference on Large Installation System Administration (LISA'07)*. 18:1–18:12.

HEINZ, S. AND ZOBEL, J. 2003. Efficient single-pass index construction for text databases. *J. Amer. Soc. Inf. Sci. Technol. 54,* 8, 713–729.

JÄRVELIN, K. AND KEKÄLÄINEN, J. 2002. Cumulative gain-based evaluation of IR techniques. *ACM Trans. Inf. Syst. 20,* 4, 422–446.

JONES, R. AND DIAZ, F. 2007. Temporal profiles of queries. *ACM Trans. Inf. Syst. 25,* 3, Article 14.

KAYAASLAN, E., CAMBAZOGLU, B. B., BLANCO, R., JUNQUEIRA, F., AND AYKANAT, C. 2011. Energy-price-driven query processing in multi-center Web search engines. In *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'11)*. 983–992.

KLEINBERG, J. M. 1999. Authoritative sources in a hyperlinked environment. *J. ACM 46,* 5, 604–632.

LEMPEL, R. AND MORAN, S. 2000. The stochastic approach for link-structure analysis (SALSA) and the TKC effect. *Comput. Netw. 33*, 387–401.

LESTER, N., MOFFAT, A., AND ZOBEL, J. 2008. Efficient online index construction for text databases. *ACM Trans. Datab. Syst. 33,* 3, 19:1–19:33.

LESTER, N., ZOBEL, J., AND WILLIAMS, H. E. 2006. Efficient online index maintenance for contiguous inverted lists. *Inf. Proces. Manag. 42,* 4, 916–933.

LI, H. 2011. *Learning to Rank for Information Retrieval and Natural Language Processing*. Morgan & Claypool Publishers.

LI, J., LOO, B. T., HELLERSTEIN, J. M., KAASHOEK, M. F., KARGER, D. R., AND MORRIS, R. 2003. On the feasibility of peer-to-peer Web indexing and search. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*. 207–215.

LI, X. AND CROFT, W. B. 2003. Time-based language models. In *Proceedings of the 12th International Conference on Information and Knowledge Management (CIKM'03)*. 469–475.

LIN, J. AND MISHNE, G. 2012. A study of "churn" in tweets and real-time search queries. In *Proceedings of the 6th International AAAI Conference on Weblogs and Social Media (ICWSM'12)*. 503–506.

LIU, T.-Y. 2009. Learning to rank for information retrieval. *Found. Trends Inf. Retriev. 3,* 3, 225–331.

MACDONALD, C., SANTOS, R. L., AND OUNIS, I. 2012. The whens and hows of learning to rank for Web search. *Inf. Retriev.* To appear.

MATVEEVA, I., BURGES, C., BURKARD, T., LAUCIUS, A., AND 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'06)*. 437–444.

MCCREADIE, R., SOBOROFF, I., LIN, J., MACDONALD, C., OUNIS, I., AND MCCULLOUGH, D. 2012. On building a reusable Twitter corpus. In *Proceedings of the 35th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'12)*. 1113–1114.

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'07)*. 253–262.

METZLER, D. AND CAI, C. 2011. USC/ISI at TREC 2011: Microblog track. In *Proceedings of the 20th Text REtrieval Conference (TREC'11)*.

METZLER, D. AND CROFT, W. B. 2007. Latent concept expansion using Markov random fields. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'07)*. 311–318.

OUNIS, I., MACDONALD, C., LIN, J., AND SOBOROFF, I. 2011. Overview of the TREC-2011 Microblog Track. In *Proceedings of the 20th Text REtrieval Conference (TREC 2011)*.

PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. 1999. The PageRank citation ranking: Bringing order to the Web. Stanford Digital Library Working Paper SIDL-WP-1999-0120, Stanford University, Standford, CA.

PASS, G., CHOWDHURY, A., AND TORGESON, C. 2006. A picture of search. In *Proceedings of the 1st International Conference on Scalable Information Systems (InfoScale'06)*.

Peng, D. and Dabek, F. 2010. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI'10)*. 251–264.

Robertson, S. E., Walker, S., Hancock-Beaulieu, M., Gatford, M., and Payne, A. 1995. Okapi at TREC-4. In *Proceedings of the 4th Text REtrieval Conference (TREC-4)*. 73–96.

Shepherd, M. A., Phillips, W. J., and Chu, C.-K. 1989. A fixed-size Bloom filter for searching textual documents. *Comput. J. 32,* 3, 212–219.

Skobeltsyn, G., Junqueira, F. P., Plachouras, V., and Baeza-Yates, R. 2008. ResIn: A combination of results caching and index pruning for high-performance Web search engines. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'08)*. 131–138.

Soboroff, I., McCullough, D., Lin, J., Macdonald, C., Ounis, I., and McCreadie, R. 2012a. Evaluating real-time search over tweets. In *Proceedings of the 6th International AAAI Conference on Weblogs and Social Media (ICWSM'12)*. 579–582.

Soboroff, I., Ounis, I., Macdonald, C., and Lin, J. 2012b. Overview of the TREC 2012b Microblog Track. In *Proceedings of the 21st Text REtrieval Conference (TREC'12)*.

Strohman, T. and Croft, W. B. 2006. Low latency index maintenance in Indri. In *Proceedings of the Open Source Information Retrieval Workshop (OSIR'06)*. 7–11.

Strohman, T. and Croft, W. B. 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'07)*. 175–182.

Strohman, T., Turtle, H., and Croft, W. B. 2005. Optimization strategies for complex queries. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'05)*. 219–225.

Tatikonda, S., Cambazoglu, B. B., and 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'11)*. 963–972.

Teevan, J., Ramage, D., and Morris, M. R. 2011. #TwitterSearch: A comparison of microblog search and Web search. In *Proceedings of the 4th ACM International Conference on Web Search and Data Mining (WSDM'11)*. 35–44.

Tirdad, K., Ghodsnia, P., Munro, J. I., and López-Ortiz, A. 2011. COCA filters: Co-occurrence aware Bloom filters. In *Proceedings of the 18th International Symposium on String Processing and Information Retrieval (SPIRE'11)*. 313–325.

Tomasic, A., García-Molina, H., and Shoens, K. 1994. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*. 289–300.

Tonellotto, N., Macdonald, C., and Ounis, I. 2013. Efficient and effective retrieval using selective pruning. In *Proceedings of the 6th ACM International Conference on Web Search and Data Mining (WSDM'13)*.

Tsirogiannis, D., Guha, S., and Koudas, N. 2009. Improving the performance of list intersection. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*. 838–849.

Turtle, H. and Flood, J. 1995. Query evaluation: Strategies and optimizations. *Inf. Proces. Manage. 31,* 6, 831–850.

Wang, L., Lin, J., and 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'11)*. 105–114.

Zobel, J. and Moffat, A. 2006. Inverted files for text search engines. *ACM Comput. Sur. 38,* 6, 1–56.

Received August 2012; revised January, March 2013; accepted March 2013