Earlybird: Real-Time Search at Twitter

Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy Lin
Twitter
@michibusch @krishnagade @larsonite @plok @sam @lintool

***Abstract*— The web today is increasingly characterized by social and real-time signals, which we believe represent two frontiers in information retrieval. In this paper, we present Earlybird, the core retrieval engine that powers Twitter's real-time search service. Although Earlybird builds and maintains inverted indexes like nearly all modern retrieval engines, its index structures differ from those built to support traditional web search. We describe these differences and present the rationale behind our design. A key requirement of real-time search is the ability to ingest content rapidly and make it searchable immediately, while concurrently supporting low-latency, high-throughput query evaluation. These demands are met with a single-writer, multiple-reader concurrency model and the targeted use of memory barriers. Earlybird represents a point in the design space of real-time search engines that has worked well for Twitter's needs. By sharing our experiences, we hope to spur additional interest and innovation in this exciting space.**

I. INTRODUCTION

Information retrieval (IR), particularly in the web context (i.e., web search), is a mature technology. Through commercial search engines, anyone can query billions of web pages in milliseconds for free. From the information retrieval research literature, much is known about many core aspects of web search, e.g., machine-learned ranking algorithms [1], [2], [3], techniques for exploiting log data [4], [5], [6], [7], web crawling [8], and the systems engineering aspects of building large-scale search engines [9], [10]. Although there continue to be refinements and advances, it is unclear whether we can expect major disruptive advances in *core* web search in the near future.

There is, of course, more to information retrieval than searching (relatively) static web content. The web today is increasingly characterized by social and real-time signals, which we believe form the next frontiers of information retrieval. There has been a fair amount of work on social search (e.g., [11], [12]) and the use of social signals to personalize web search has recently seen commercial deployments, but the topic of real-time search appears to be largely unexplored in the academic literature. This is the focus of our paper.

Twitter is a communications platform on which users can send short, 140-character messages, called "tweets", to their "followers" (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 Fall 2011, Twitter has over 100 million active users worldwide, who collectively post over 250 million tweets per day. One salient aspect of Twitter is that users demand to know what's happening *right now*, especially in response to breaking news stories such as the recent death of Steve Jobs, upcoming birth of Beyoncé's baby, or protests in Tahrir Square and on Wall Street. For that, they frequently turn to real-time search: our system serves over two billion queries a day, with an average query latency of 50 ms. Usually, tweets are searchable within 10 seconds after creation.

This paper describes Earlybird, the retrieval engine that lies at the core of Twitter's real-time search service. We view this work as having two main contributions. First, we describe an approach to organizing inverted indexes that supports the demands of real-time search. Second, we present a single-writer, multiple-reader lock-free algorithm that takes advantage of Java's concurrency model to perform real-time indexing with highly-concurrent query evaluation.

Roadmap. The remainder of the paper is organized as follows: We begin by discussing the requirements of real-time search (Section II) and then provide background on traditional web search architectures and algorithms (Section III). An overview of Earlybird is provided in Section IV, followed by detailed discussions of its index organization (Section V) and concurrency model (Section VI). We describe the deployment status of Earlybird and provide some performance figures in Section VII. Future directions are presented in Section VIII before concluding.

II. REQUIREMENTS OF REAL-TIME SEARCH

We begin by describing the salient requirements of real-time search, which to a large extent dictates the design of Earlybird. With the exception of the first, these requirements are somewhat different from those of web search.

Low-latency, high-throughput query evaluation. When it comes to search, users are impatient and demand results quickly, and of course, any successful search engine must cope with large query volumes. Web search engines are no stranger to the requirements of low query latency and high query throughput, and these desired characteristics apply to real-time search as well.

High ingestion rate and immediate data availability. In real-time search, documents[1] may arrive at a very high rate, often with sudden spikes corresponding to the "flash crowd" effect. Regardless of arrival rate, 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 achieve both low latency and

[1]Following standard IR parlance, we refer to the basic unit of indexing as a "document", even though in actuality it may be a web page, a tweet, a PDF, a fragment of code, etc.

high throughput. This requirement departs from common assumptions in typical search environments: that indexing can be 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 [13]. Alternatively, substantially more machine resources can be brought to bear to reduce indexing latency using an architecture such as the one implemented by Percolator [14], but this alternative still does not appear to approach the speed required for real-time search.

Concurrent reads and writes. As a result of the previous two requirements, real-time search engines must contend with large volumes of concurrent reads and writes. That is, index structures must be continuously updated as documents are ingested, while the same index structures are accessed to serve queries. This stands in contrast to web search, which is mostly read-only (albeit, highly concurrent) from static index structures. As a simple solution, indexes can be deployed in atomic swaps, from an older to a newer version of the index, meaning that it is fairly easy to design architectures where indexes are never read from and written to concurrently.

Dominance of the temporal signal. The very nature of real-time search means that the timestamp of the document is a very important (if not *the* most important) signal for ordering results. By default, a real-time search engine could display hits in reverse chronological order (initial implementation of Twitter real-time search). Even when other relevance signals are incorporated to improve this ordering (as Twitter currently does), the temporal signal remains dominant. This stands in contrast to web search, where the timestamp of a web page has a relatively minor role in determining the relevance ranking (news search being the obvious exception). The practical implication of this is that the query evaluation algorithm should traverse inverted index postings in reverse chronological order.[2] This is an access pattern that is foreign to standard web search engines.

III. Background and Related Work

A. Distributed Search Architectures

A web search service, especially at scale, is first and foremost a complex, geographically-distributed system [9], [10]. User queries are first routed to an appropriate datacenter based on a variety of considerations: query load, link latencies, and even the cost of electricity [15], [16], [17]. Within each datacenter, search is served by large clusters consisting of hundreds to thousands of machines. Most frequently, these machines are organized in a replicated, broker-coordinated, document-partitioned distributed search architecture (or some variant thereof).

[2]Although this is not an absolute requirement and other alternatives are possible, it is perhaps the most natural implementation.

At scale, the document collection is usually partitioned logically (i.e., sharded) into disjoint segments and individual machines are responsible for serving indexes that correspond to these document partitions [18], [19]. A broker is responsible for forwarding requests to the partition servers and integrating partial results from each before passing the final output back to the requester. Partitioning reduces query and indexing latency since each partition operates independently. Robustness of the search service is achieved via replication, i.e., multiple instances of the same service running independently to share the incoming request load. Replication has little impact on latency, but increases throughput in a linearly scalable fashion.

Finally, search engines also take advantage of caching [20], [21]. Most modern systems serve indexes from main memory, but caching can be applied to results of frequent queries as well as documents and other data that are frequently accessed, e.g., the webgraph [22]. Overall, these designs are not IR-specific, but represent general principles for building large-scale distributed systems [23], [24].

Despite the importance of these architectural issues, they are not the focus of this paper. Rather, we focus on query evaluation on a single machine, over a small subset (i.e., partition) of the entire collection of documents.

B. Query Evaluation

At the level of individual machines, query evaluation in modern web retrieval is generally divided into two phases [25], [26], [27]. In the first phase, a fast, "cheap" algorithm is applied to generate a candidate list of potentially-relevant documents. For example, documents might be scored with a linear combination of a relevance measure such as BM25 [28] and a query-independent measure such as PageRank [29], HITS [30], page quality score [31], etc. Candidate documents from the first phase are then reranked by a slower, "expensive" but higher quality (usually, machine-learned) algorithm, typically considering richer features (e.g., term proximity, anchor text). The wide range of implementations include gradient boosted regression trees [32], [2], additive ensembles [33], and cascades of rankers [26], [34], just to name a few.

The inverted index lies at the heart of nearly all modern retrieval engines. Conceptually, it holds a mapping from vocabulary terms to postings lists. Each postings list contains a number of postings, typically one for every document that contains the term. Each posting, in turn, holds a document id and a payload. Most often, the payload consists of the term frequency, and in positional indexes, is augmented with term position information. Alternatively, the payload can hold pre-computed quantized scores known as impacts [35], [36]. The two standard designs are to sort postings by document ids ascending or by the payload descending (i.e., higher term frequency or impact scores first). The two approaches manifest different tradeoffs (e.g., in compression schemes) and support different types of query evaluation algorithms. Generally, there is an affinity between document-sorted indexes and document-at-a-time query evaluation [37], [38], [39]. Similarly, impact- or frequency-sorted indexes usually take advantage of term-at-a-time evaluation strategies [37], [35], [40]. The intricacies of these different algorithms, as well as variants and hybrids, are beyond the scope of this paper, but Zobel and Moffat [18] provide a good (although a bit dated) overview. However, as Section V will discuss in more detail, much of the work cited above is difficult to adapt to real-time search.

C. Other Considerations

Prior to ingestion into a search service, content has to be acquired. For web search, this involves crawling the web—which itself requires a complex, geographically-distributed system that must delicately balance latency, throughput, and freshness. For example, a crawler must infer the rate at which a particular piece of content changes and decide how often to re-crawl it, taking into account other factors such as page quality, server latency, etc. Crawling must also be accompanied by extraction of the link graph, to provide both a source of additional crawl targets and a source of relevance signals for ranking documents. A review of the myriad of strategies and algorithms is beyond the scope of this paper, but we refer the reader to a recent survey [8]. In the case of Twitter, however, we do not need to deal with most of these complexities, at least for the problem of searching tweets.

Of course, it has long been recognized that the web is not homogeneous, and different types of content benefit from different treatments. This has generally evolved into two types of partitioning strategies. Dividing the web into "verticals", for example, news, images, the academic literature, etc., allows a search service to develop custom strategies and algorithms. For example, news sites are crawled far more frequently than the general web, and the temporal signal plays a larger role in ranking than in general web search. The creation of verticals, however, creates a new problem known as vertical integration, e.g., when and where to insert news stories into general web search results [41]. Twitter faces a similar issue, in that tweets may not be the only search results relevant to a user. For example, sometimes users are looking for a specific account (to follow), in which case a user profile is the best result. Twitter recently introduced a "Top News" section to showcase relevant recent news articles about a certain topic. Deciding whether (and potentially where) to insert these different types of content is a vertical integration problem.

Another standard strategy is to divide indexes into "tiers" in terms of quality [42], [43], as captured by human editorial judgments, automatic classifiers, or some combination of the two. A common two-tier index would have "high quality" documents in an upper tier and everything else in a separate lower tier. Strategies for querying the tiers (either sequentially or in parallel) manifest different tradeoffs in efficiency and effectiveness. Twitter currently does not adopt this strategy.

Note that this type of "semantic" content partitioning (by vertical, by tier) is not to be confused with the *logical* partitioning described in Section III-A. For example, the index of the news vertical might be partitioned across several physical machines. For more details, see alternative architectures described in Risvik et al. [42].

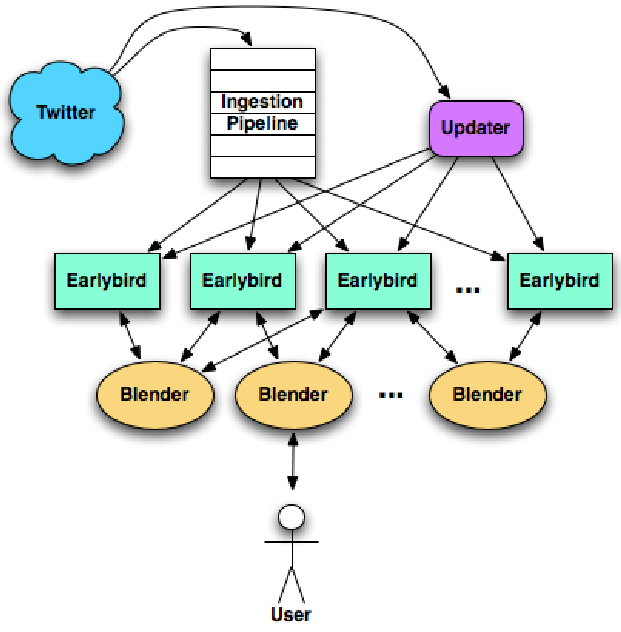

Fig. 1. Architecture of Twitter's real-time search service, showing the role of Earlybird servers, which index new tweets from the ingestion pipeline and serve front-end "Blenders".

IV. Earlybird Overview

Earlybird is Twitter's real-time retrieval engine, built on top of the open-source Lucene search engine[3] and adapted to meet the requirements discussed in Section II. Since it is specifically designed to handle tweets (and only tweets), we implemented a few optimizations that may not be applicable in the general case. Earlybird exists within the architecture shown in Figure 1 that provides Twitter real-time search.

Tweets enter the ingestion pipeline, where they are tokenized and annotated with additional metadata (for example, language). To handle large volumes, the tweets are hash partitioned across Earlybird servers, which index the tweets as soon as they have been processed. The search service currently performs relevance filtering and personalization using three types of signals:

- Static signals, directly added at indexing time.
- Resonance signals, dynamically updated over time (e.g., number of retweets a tweet receives).
- Information about the searcher, provided at search time (see below).

A component called the "Updater" pushes dynamic resonance signals to the Earlybird servers.

At query time, a "Blender" (front-end) server parses the user's query and passes it along with the user's local social graph to multiple Earlybird servers. These servers use a ranking function that combines relevance signals and the user's local social graph to compute a personalized relevance score for each tweet. The highest-ranking, most-recent tweets are

[3]http://lucene.apache.org/

returned to the Blender, which merges and re-ranks the results before returning them to the user. In production, Earlybird servers receive load from the front ends while simultaneously indexing new tweets from the ingestion pipeline. End-to-end, we typically observe a 10 second indexing latency (from tweet creation time to when the tweet is searchable) and around 50 ms query latency.

The overall distributed search architecture is beyond the scope of this paper, but for an informal description we refer the reader to a 2010 Twitter Engineering blog post.[4] Also, we will not be discussing the relevance algorithm, and so for the purposes of this paper, Earlybird simply returns a list of tweets that satisfy a boolean query, in reverse chronological order.

Earlybird is written completely in Java, primarily for three reasons: to take advantage of the existing Lucene Java code-base, to fit into Twitter's JVM-centric development environment, and to take advantage of the easy-to-understand memory model for concurrency offered by Java and the JVM. Although this decision poses inherent challenges in terms of performance, with careful engineering and memory management we believe it is possible to build systems that are comparable in performance to those written in C/C++.

As with nearly all modern retrieval engines, Earlybird maintains an inverted index, which maps terms to lists of postings. Postings are maintained in forward chronological order (most recent last) but are traversed *backwards* (most recent first); this is accomplished by maintaining a pointer to the current end of each postings list. We see this index organization as an interesting and noteworthy aspect of Earlybird, which we detail in Section V.

Earlybird supports a full boolean query language consisting of conjunctions (ANDs), disjunctions (ORs), negations (NOTs), and phrase queries. Results are returned in reverse chronological order, i.e., most recent first. Boolean query evaluation is relatively straightforward, and in fact we use Lucene query operators "out of the box", e.g., conjunctive queries correspond to intersections of postings lists, disjunctive queries correspond to unions, and phrase queries correspond to intersections with positional constraints. Lucene provides an abstraction for postings lists and traversing postings—we provide an implementation for our custom indexes, and are able to reuse existing Lucene query evaluation code otherwise. The actual query evaluation algorithm isn't particularly interesting, but the way in which we handle concurrency (concurrent index reads and writes) in a multi-threaded framework is worth discussing. Section VI is devoted to that topic.

V. Index Organization

As a quick recap, Earlybird indexes must support low-latency, high-throughput retrieval (query evaluation) while concurrently indexing tweets. Operationally, this means many concurrent reads and writes to index data structures. This section focuses on the organization of Earlybird indexes to support these demands; discussion of concurrency management is deferred to the next section.

Our strategy for managing complexity is to isolate and limit the scope of index updates. This is accomplished as follows: each instance of Earlybird manages multiple index segments (currently 12), and each segment holds a relatively small number of tweets (currently, $2^{23} \sim 8.4$ million tweets). 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 ceases to accept new tweets, we can convert it from a write-friendly structure into an optimized read-only structure.

A. Dictionary Organization

Twitter real-time search currently does not support any query operators on terms, such as wildcards, that require the term dictionary to be sorted. Therefore, the term dictionary is implemented as a simple hash table. Java's default HashMap is a poor choice, because its chained-hashing approach requires multiple objects and object pointers per entry—this is particularly bad for garbage collection due to the large number of long-living objects in the dictionary. Instead, our custom implementation uses open addressing, which requires only a small number of primitive Java arrays, independent of the number of entries.

Each dictionary term is assigned a unique, monotonically-increasing term id (i.e., a newly-encountered term is assigned the next available term id). Term data are held in parallel arrays, and contain the following two pieces of information (simplified for illustrative purposes):

- Number of postings in the postings list.
- Pointer to the tail of the postings list (see next section for more details).

Storing term data in parallel arrays is both memory efficient (dense packing, few objects) and provides fast access; lookup is accomplished by using the term id as the index into the relevant array.

B. Segment Layout: Active Index

This section provides a description of the write-friendly "active" index segment, which can support low-latency, high-throughput tweet indexing. As outlined in Section III-B, the two primary strategies for organizing inverted indexes from the information retrieval literature are to either sort by document ids (ascending) or by impact scores (descending). Unfortunately, neither approach appears to be entirely appropriate for real-time search. Since it is desirable to traverse postings in reverse temporal order for query evaluation (see Section II), we can rule out impacted-sorted indexes.

What about document-sorted indexes? If we assign document ids to new tweets in ascending order, there are two obvious possibilities:

First, we could append new postings to the ends of postings lists. However, this would require us to read postings *backwards* to achieve a reverse chronological traversal order.

[4]http://engineering.twitter.com/2011/05/engineering-behind-twitters-new-search.html

Unfortunately, this is not directly compatible with modern index compression techniques. Typically, document ids are converted into document gaps, or differences between consecutive document ids. These gaps are then compressed with integer coding techniques such as γ or Rice codes, or more recently, PForDelta [44], [45]. It would be tricky for gap-based compression (also commonly known as delta compression) to support backwards traversal. Prefix-free codes (γ and Rice codes) are meant to be decoded only in the forward direction. More recent techniques such as PForDelta are block-based, in that they code relatively large blocks of integers (e.g., 512 document ids) at a time. Reconciling this with the desire to have low-latency indexing would require additional complexity, although none of these issues are technically insurmountable.

Alternatively, we could prepend new postings to the beginnings of postings lists. This would allow us to read postings in the forward direction and preserve a reverse chronological traversal order. However, this presents memory management challenges, i.e., how would space for new postings be allocated? We are unaware of any work in the academic literature that has explored this strategy. Note that the naïve implementation using linked lists would be hopelessly inefficient. First, linked list traversal is slow since it is not a cache friendly operation, due to the lack of reference locality and predictable memory access patterns. Second, linked lists have rather large memory footprints due to object overhead and the need to store "next" pointers.

Based on the above analysis, it does not appear that real-time search capabilities can be efficiently realized with obvious extensions or adaptations of existing techniques.

For Earlybird, we implemented a much simpler approach. 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.[5] Therefore, a list of postings is simply an integer array, and indexing new documents involves inserting elements into a pre-allocated array (we discuss "extending" the arrays below). Postings traversal in reverse chronological order corresponds to iterating through the array backwards. This organization also allows every array position to be a possible entry point for postings traversal to evaluate queries (while postings continue to be inserted). In addition, it allows for binary search (to find a particular document id), and doesn't require any additional skip-list data structure to enable faster traversal through the postings lists. Finally, this organization is cache friendly, since array traversal involves linear scans through memory and this predictable access pattern provides prefetch cues to the hardware.

The next issue to address is the allocation of space for postings lists. Obviously, this process needs to be dynamic, since postings list growth is only bounded by the size of the collection itself. There are a few challenges to overcome: Postings lists vary significantly in size, since term and document frequencies are Zipfian. That is, a typical index contains a few very long postings lists and lots of short postings lists. As a result, it is tricky to choose the correct amount of memory to allocate for each term's postings (i.e., size of the integer array). 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, but also space because non-contiguous postings require pointers to chain together (in the limit, allocating one posting at a time is akin to a linked list). During postings traversal, blocks that are too small may also result in suboptimal memory access patterns (e.g., due to cache misses, lack of memory prefetching, etc.).

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

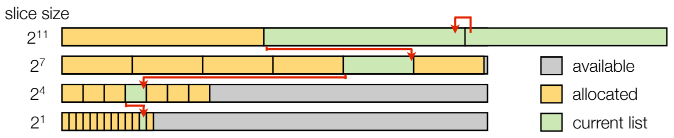

Fig. 2. Organization of the active index segment where tweets are ingested. Increasingly larger slices are allocated in the pools to hold postings. Except for slices in pool 1 (the bottom pool), the first 32 bits are used for storing the pointer that links the slices together. Pool 4 (the top pool) can hold multiple slices for a term. The green rectangles illustrate the the "current" postings list that is being written into.

Our approach to address these issues is to create four separate "pools" for holding postings. Conceptually, each pool can be treated as an unbounded integer array. In practice, pools are large integer arrays allocated in 2^{15} element blocks; that is, if a pool fills up, another block is allocated, growing the pool. In each pool, we allocate "slices", which 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 (see Figure 2). For convenience, we will 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 "previous" pointer, discussed below). After running out of space, slices are allocated in pool 3 to store the next $2^7 - 1$ term occurrences and finally $2^{11} - 1$ term occurrences in pool 4. Additional space is allocated in pool 4 in 2^{11} integer blocks 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. However, the tradeoff is that 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 the slice are used to store this "previous" pointer.

In summary, in the active index segment of Earlybird, postings are not stored in a compressed format to facilitate rapid tweet indexing. Extending postings lists is accomplished via a block allocation policy that escalates to successively-larger block sizes (capped at 2^{11} postings). This structure works well because each index segment is theoretically capped at 2^{24} document ids (and in current practice, we restrict to 2^{23} tweets). The velocity of arriving tweets means that each index segment fills up relatively quickly, and therefore doesn't spend much time in an uncompressed form.

C. Segment Layout: Optimized Index

Once an active index segment stops accepting new tweets, it is converted into an optimized read-only index. Index optimization occurs in the background; a new copy of the index is created without touching the original version. Upon completion, the original index is dropped and replaced with the optimized version. Since pools holding the postings slices are allocated in large, fixed-sized blocks, dropping index segments creates relatively little garbage.

Because the optimized read-only index cannot accept new tweets, we know exactly how much space to allocate for each postings list. Thus, postings can essentially be laid out "end-to-end" in memory as a large array of integers, and the dictionary simply holds pointers into this array.

Postings lists are divided into two types, "long" and "short", with 1000 as the arbitrary threshold. For short postings lists, we store each posting exactly as before (24-bit document id plus 8-bit position), except reversed in terms of sort order (i.e., postings are sorted reverse chronologically, therefore iterating through the array *forwards* yields the desired traversal order).

For long postings lists, we adopt a block-based compression scheme similar in spirit to PForDelta [44], [45] and the Simple9 [46] family of techniques. Postings are stored in multiple fixed-size blocks of 64 integers (i.e., 256 bytes). The first 4 bytes holds the first posting uncompressed; the second 4 bytes holds block headers (see below), leaving 248 bytes (1984 bits) to encode a variable number of postings. The original postings are traversed in reverse chronological order and converted into a sequence of (document gap, position) pairs. The goal is to encode n pairs, where each document gap and position are coded in a fixed number of bits, such that the following relation is satisfied:

$$n \cdot (\lceil \log_2(\text{gap}_{\text{max}}) \rceil + \lceil \log_2(\text{pos}_{\text{max}}) \rceil) \leq 1984$$

where gap_{max} and pos_{max} are the maximum gap and position values observed in the n document gaps and positions, respectively. The value of n and the number of bits necessary to encode the gaps and positions are stored in the block header.

This postings compression scheme is not only space efficient, but also very fast to decompress, especially compared to variable-length integer coding, which suffers from the well-known problem of processor-level branch mispredicts. As an additional optimization, since the range of possible values for $\lceil \log_2(\text{gap}_{\text{max}}) \rceil$ and $\lceil \log_2(\text{pos}_{\text{max}}) \rceil$ is relatively small, it is easy to precompute the masks and bit shifts necessary for decoding the postings under each possible case and store them all in a lookup table. Thus, decoding becomes simply a problem of applying a pre-specified "template" of bit operations. This technique is also used in the Simple9 family and PForDelta. Finally, since the first posting in each block is stored in an uncompressed format, it can be used for efficient skipping, a well-known technique in information retrieval [38].

VI. Concurrency Management

An important requirement of real-time search is the ability to concurrently handle index writes (i.e., ingesting new tweets) and index reads (i.e., query evaluation) in a multi-threaded environment. Note that this only applies to the active index segment ingesting tweets. The other index segments are read only, and do not suffer from concurrency-induced issues: multiple query evaluation threads can traverse postings concurrently. Since the later case is straightforward, we focus only on the active index segment in this section.

The complex problem of concurrent index reads and index writes can be simplified by limiting writes to a single thread. That is, a single index writer thread is responsible for ingesting tweets and updating the inverted index. On the other hand, queries are concurrently evaluated on separate index reader threads. In this context, it is important that index reader threads are presented with an up-to-date and consistent view of the index structures. This is accomplished through synchronization mechanisms, but in general, there exists a tradeoff between amount of synchronization and performance. Too much synchronization hurts performance, but too little synchronization can lead to inconsistent or incorrect results. Striking the right balance is perhaps the most difficult aspect of concurrent programming. Fortunately, Java and the JVM provide an easy-to-understand memory model for concurrency, which we exploit.

We begin this section by briefly describing the indexing process, and then discuss how concurrent query evaluation is enabled by use of a memory barrier, which ensures that memory writes made by the index writer thread are visible to index reader threads.

Indexing of a new tweet by the single index writer thread proceeds as follows: for each term in the tweet, Earlybird first looks up the corresponding dictionary entry (see Section V-A). In the dictionary, terms are mapped to term ids, which serve as indices into the parallel arrays holding the term data. Based on the pointer to the tail of the current postings list, a new posting is added. If there isn't sufficient space to insert this new posting, additional slices are allocated, as described in Section V-B. If a term has never been encountered, it is added to the dictionary and assigned the next available term id. A slice is allocated in the first pool for the new posting. In both cases (existing or new term), the term occurrence count and the tail pointer of the postings list are then updated. After all terms in a tweet have been processed in this manner, we increment the maxDoc variable, which holds the current largest document id that has been encountered—indicating that the tweet has been successfully ingested.

Concurrent queries are handled by separate threads (one per thread). Each query evaluation thread begins by reading the maxDoc variable, then looks up the postings list tail pointer corresponding to each term. These are used to initialize the appropriate postings list abstractions in Lucene, and query evaluation begins thereafter. As previously mentioned, this is simply standard Lucene code.

There are two aspects to Earlybird's consistency model. First, individual postings lists must *always* be internally consistent. For example, the tail pointers of the postings lists should always be valid. Second, we need to maintain consistency across postings lists: an index reader should see a consistent view of the index structures up to a certain, well-defined point. In this regard, Earlybird guarantees search correctness defined over all tweets with document ids less than or equal to maxDoc, at the point when the index reader begins query evaluation.

Maintaining consistency in a multi-threaded environment is challenging. Thread execution can occur in arbitrarily interleaved orders and there is no guarantee when memory writes from one thread are visible to another thread. One solution would be to make tweet indexing an atomic operation, and therefore the index reader threads are guaranteed to see consistent, up-to-date index structures. However, this level of synchronization trades off too much performance and is not practical for the volumes of tweets that we need to handle.

To guarantee search correctness in the most lightweight possible manner, we place a memory barrier at the maxDoc increment (the final step in tweet ingestion). This is accomplished by the volatile keyword in Java, which guarantees that all writes to a volatile field are visible to subsequent reads. In other words, reading a volatile field is guaranteed by the JVM to yield the most up-to-date value (see Figure 3).

To better explain, it is necessary to provide a brief overview of Java's memory model [47]:

- **Program order rule.** Each action in a thread *happens-before* every action in that thread that comes later in the program order.
- **Volatile variable rule.** A write to a volatile field *happens-before* every subsequent read of that same field.
- **Transitivity.** If A *happens-before* B, and B *happens-before* C, then A *happens-before* C.

In the index writer thread, all mutations to postings lists *happen-before* the maxDoc increment. Query evaluation begins by reading maxDoc, to make sure that the memory barrier is crossed. This *happens-before* subsequent traversal of postings. By transitivity, it is therefore guaranteed (without any other additional synchronization mechanism) that the query evaluation thread sees all mutations of the index structures

index writer	**index reader**
ingest tweet$_i$	
add postings	
...	
maxDoc++	
	read maxDoc
ingest tweet$_{i+1}$	traverse postings
add postings	...
...	

Fig. 3. Synchronization between the indexing thread (index writer) and a query evaluation thread (index reader), with "time" running from top to bottom. Using Java's volatile keyword, a memory barrier is placed at maxDoc (denoted by the solid line). In the index writer thread, index mutation *happens-before* the maxDoc increment. In the index reader thread, reading maxDoc *happens-before* subsequent postings traversal. Therefore, combining the volatile variable rule and transitivity, all index mutations (from indexing a new tweet) prior to the memory barrier are guaranteed to be visible to the index reader, without any additional synchronization.

that occur prior to the increment of the maxDoc variable (e.g., insertion of new postings). The transitivity property ensures that postings lists are internally consistent.

Memory barriers are lightweight and therefore yield good concurrent performance, but there is a downside: the non-atomic nature of tweet indexing means that Earlybird needs to explicitly handle consistency across multiple postings lists. The Java memory model guarantees that a thread's memory writes are visible by other threads after the memory barrier. However, the writes may actually be available to other threads sooner—that is, writes "leak through". This creates a number of special situations that threaten search correctness. Here, we illustrate one example: During query evaluation, it may be the case that postings lists contain document ids that are larger than maxDoc. This occurs if the index reader begins query evaluation in the middle of indexing a tweet that contains the query terms. In this case, postings may have already been added, but maxDoc has not been incremented yet since there are remaining terms to be processed. If the newly-added postings become visible (i.e., "leak through") to the index reader thread, it may see postings with document ids larger than maxDoc. This is a relatively simple case to handle: the algorithm ignores postings whose document ids are larger than maxDoc. However, there are corner cases which are more subtle, the exact explication of which is beyond the scope of this paper.

In summary, Earlybird first simplifies the concurrency management problem by adopting a single-writer, multiple-reader design. The select use of a memory barrier around the maxDoc variable supports an easy-to-understand consistency model while imposing minimal synchronization overhead.

VII. Deployment and Performance

Earlybird was introduced in October 2010, and the search architecture shown in Figure 1 was launched in May 2011. Both replaced infrastructure that was created by Summize, a company Twitter acquired in July 2008. As a first-generation system, it offers an interesting reference point for comparison. The Summize search infrastructure used Ruby on Rails for the front end and MySQL for the back end. Inverted indexes were constructed in MySQL, leveraging transactions and its B-tree data structures to support concurrent indexing and retrieval. We were able to scale our MySQL-based solution surprisingly far by partitioning the index across multiple in-memory databases and replicating the Rails front-end. In 2008, Twitter search handled an average of 20 tweets per second (TPS) and 200 queries per second (QPS). Just before it was retired in October 2010, the MySQL-based solution was handling 1000 TPS and 12000 QPS on average.

The basic configuration of an Earlybird server is a commodity machine with two quad-core processors and 72 GB memory. We typically allocate 64 GB heap space for the JVM. A fully-loaded active index segment with 16 million documents occupies about 6.7 GB memory. Index optimization typically saves around 55% memory (i.e., the optimized read-only index is a bit less than half the size of the original). On a single index segment with 16 million tweets, we achieve 17000 QPS with a 95th percentile latency of $<$100 ms and 99th percentile latency of $<$200 ms using 8 searcher threads. Pushing the query load even higher results in increases of the 95th and 99th percentile latencies to unacceptably high levels. On a fully-loaded Earlybird server (144 million tweets), we can achieve about 5000 QPS with 95th percentile latency of 120 ms and 99th percentile latency of 170 ms.

In a stress test, we evaluated Earlybird indexing performance under heavy query load. Starting with an index of around 110 million tweets (as to nearly saturate memory) and adjusting the query load to achieve near 100% CPU utilization, we simulated different tweet arrival rates (in terms of tweets per second). Under these conditions, we achieve 7000 TPS indexing rate at 95th percentile latency of 150 ms and 99th percentile latency of 180 ms. We observe that indexing latency is relatively invariant with respect to tweet arrival rate; at 1000 TPS we observe roughly the same latencies as at 7000 TPS. Attempting to push the tweet arrival rate even higher results in unacceptable latencies, primarily due to thread contention.

VIII. Future Directions

The purpose of real-time search is to provide users with a tool to understand events as they unfold and to discover items of interest. In this respect, a simple reverse chronological listing of tweets that satisfies a boolean query provides the user limited insight—especially for queries where matching tweets are arriving at a high rate. In the Twitter context, the most popular queries are the most difficult, because they tend to revolve around popular topics to which many are contributing tweets that vary widely in terms of quality. Therefore, the biggest challenge of real-time search is the development of relevance algorithms that separate truly valuable content from "noise". While the current search service does indeed incorporate relevance ranking, it is a feature that we have only begun to explore.

The notion of relevance has occupied the attention of information science and information retrieval researchers for decades. The literature is vast and deep, and for the interested reader, Mizzaro [48] provides a good starting point. Traditionally, the information retrieval community has focused on *topical* relevance, with more recent explorations in orthogonal notions such as novelty and diversity. Yet, against this rich backdrop, it is not entirely clear if we (both Twitter and the research community) have a good understanding of what relevance means in the real-time context.

Just to provide one example of an unresolved problem: a 5-grade relevance judgment ("poor", "fair", "good", "excellent", "perfect") is widely accepted in the community for assessing topical relevance. But how do different relevance grades interact with the age of the tweet? In more concrete terms, is a "perfect" tweet from five minutes ago better than a "good" tweet from five seconds ago? What about compared to a "fair" tweet from one second ago? Does it depend on the type of query and its velocity? Does it depend on the particular user? Obviously, relevance grades must be subjected to some type of temporal discounting when computing utility or "gain" for a user, but the shape and parameterizations of these "response curves" remain open questions.

To further increase complexity, we believe that relevance should be *personalized* to leverage social and interest signals that are present in the follower graph and other graphs implicitly defined by user behaviors. Personalization creates many challenges, ranging from the extraction of salient features to handling data sparsity issues when training a personalized algorithm. Despite these challenges, it seems clear that personalization is necessary to deliver high-quality results.

Information retrieval is an empirical, metrics-driven discipline, built on the ability to quantify, however imperfectly, the quality of system results. And without a clear understanding of real-time search relevance, it is difficult to formulate metrics that "make sense" to guide progress in the community. This is a challenge we are addressing, both internally and in collaboration with the community. For example, Twitter aided the development of a real-time search task at the 2011 Text Retrieval Conference (TREC), which is an annual evaluation forum for information retrieval researchers to work on shared tasks, sponsored by the National Institute of Standards and Technology (NIST). We hope that by engaging the community, we can jointly make progress on these challenging issues.

Beyond real-time search relevance, we have identified other areas of interesting future work. The first goes back to the fundamental question of what a "document" is in the real-time search context. We have been operating under the assumption that the tweet is the relevant unit of retrieval, but there are several observations that challenge this. For example, many tweets contain hyperlinks: these may point to web pages (blog posts, news stories, etc.), to images, or any other web content. This means, potentially, that real-time search interacts with web and image search in non-obvious ways—we have only begun to explore these issues and welcome support from the community in tackling these problems.

Another interesting area of work is the synthesis of information across multiple tweets, e.g., a cluster of tweets about the same topic or event, expressing the same opinion, sharing a tightly-knit group of participants, etc. To accomplish this, we need to develop basic text analysis tools, which falls under the domain of natural language processing (NLP). Indeed, there has been work on part-of-speech tagging [49], named-entity detection [50], text normalization [51], and other capabilities, specifically for Twitter. However, the idiosyncratic nature of tweets, compared to say, the newswire texts that the NLP community traditionally focuses on, remains a stumbling block to developing robust tools. Furthermore, the NLP community for the most part cares little about the efficiency (i.e., speed) of their algorithms—which limits the applicability of research results in a production context. To give a simple example, language identification is generally considered to be a solved problem by the community—using, for example, character language models. However, it is more challenging than one might think in the Twitter context. Not only is there far less text (at most 140 characters) to base a classification decision on, we also observe frequent code switching (mixing of multiple languages) as well as many other idiosyncratic properties and user behaviors. On top of all these challenges, the language identification module must operate under tight time constraints.

IX. Conclusion

In this paper, we present salient aspects of Earlybird, which powers Twitter's real-time search service. Due to the demands of real-time search, the Earlybird design uses two types of indexes: an optimized, read-only index format and an active "write-friendly", block-allocated index that supports both rapid tweet indexing and query evaluation. In the latter type of index, we adopt a single-writer, multiple-reader model that enforces consistency using a simple memory barrier.

The Earlybird project has been highly successful in laying the foundation for real-time search at Twitter. However, we have only begun to develop the suite of capabilities necessary to deliver individually-personalized, highly-relevant results, and more broadly, the technology to keep millions of users around the world plugged into "what's happening". Real-time search forms an exciting frontier in information retrieval, and it is our hope that this paper spurs more exploration.

Acknowledgments

The following engineers at Twitter contributed to the real-time search architecture described in this paper: Paul Burstein, Stephen Fedele, Mike Hayes, Justin Hurley, Eric Jensen, Abhi Khune, Frost Li, Zhenghua Li, Jake Mannix, Gilad Mishne, Raghavendra Prabu, Jonathan Reichhold, Aneesh Sharma, Tian Wang, and Yi Zhuang. We would also like to acknowledge the following managers for their dedication to the search product at Twitter: Michael Abbott, Abdur Chowdhury, Doug Cook. Thanks also goes out to Jeff Dalton, Donald Metzler, and Ian Soboroff for comments on earlier drafts of this paper.

References

[1] T.-Y. Liu, "Learning to rank for information retrieval," *Foundations and Trends in Information Retrieval*, vol. 3, no. 3, pp. 225–331, 2009.

[2] Y. Ganjisaffar, R. Caruana, and C. V. Lopes, "Bagging gradient-boosted trees for high precision, low variance ranking models," in *Proceedings of the 34rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2011)*, Beijing, China, 2011, pp. 85–94.

[3] H. Li, *Learning to Rank for Information Retrieval and Natural Language Processing*. Morgan & Claypool Publishers, 2011.

[4] H. Cui, J.-R. Wen, and W.-Y. Ma, "Probabilistic query expansion using query logs," in *Proceedings of the Eleventh International World Wide Web Conference (WWW 2002)*, Honolulu, Hawaii, 2002, pp. 325–332.

[5] E. Agichtein, E. Brill, S. Dumais, and R. Ragno, "Learning user interaction models for predicting Web search result preferences," in *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2006)*, Seattle, Washington, 2006, pp. 3–10.

[6] T. Joachims, L. Granka, B. Pan, H. Hembrooke, F. Radlinski, and G. Gay, "Evaluating the accuracy of implicit feedback from clicks and query reformulations in Web search," *ACM Transactions on Information Systems*, vol. 25, no. 2, pp. 1–27, 2007.

[7] S. M. Beitzel, E. C. Jensen, D. D. Lewis, A. Chowdhury, and O. Frieder, "Automatic classification of Web queries using very large unlabeled query logs," *ACM Transactions on Information Systems*, vol. 25, no. 2, pp. 1–29, 2007.

[8] C. Olston and M. Najork, "Web crawling," *Foundations and Trends in Information Retrieval*, vol. 4, no. 3, pp. 175–246, 2010.

[9] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: The Google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, 2003.

[10] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri, "Challenges on distributed web retrieval," in *Proceedings of the IEEE 23rd International Conference on Data Engineering (ICDE 2007)*, Istanbul, Turkey, 2007, pp. 6–20.

[11] M. V. Vieira, B. M. Fonseca, R. Damazio, P. B. Golgher, D. de Castro Reis, and B. Ribeiro-Neto, "Efficient search ranking in social networks," in *Proceedings of the Sixteenth International Conference on Information and Knowledge Management (CIKM 2007)*, Lisbon, Portugal, 2007, pp. 563–572.

[12] D. Horowitz and S. D. Kamvar, "The anatomy of a large-scale *social* search engine," in *Proceedings of the 19th International World Wide Web Conference (WWW 2010)*, Raleigh, North Carolina, 2010, pp. 431–440.

[13] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation (OSDI 2004)*, San Francisco, California, 2004, pp. 137–150.

[14] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proceedings of the 9th USENIX Symposium on Operating System Design and Implementation (OSDI 2010)*, Vancouver, British Columbia, Canada, 2010, pp. 251–264.

[15] B. B. Cambazoglu, V. Plachouras, and R. Baeza-Yates, "Quantifying performance and quality gains in distributed web search engines," in *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2009)*, 2009, pp. 411–418.

[16] B. B. Cambazoglu, E. Varol, E. Kayaaslan, C. Aykanat, and R. Baeza-Yates, "Query forwarding in geographically distributed search engines," in *Proceedings of the 33rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2010)*, Geneva, Switzerland, 2010, pp. 90–97.

[17] E. Kayaaslan, B. B. Cambazoglu, R. Blanco, F. P. Junqueira, and C. Aykanat, "Energy-price-driven query processing in multi-center web search engines," in *Proceedings of the 34rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2011)*, Beijing, China, 2011, pp. 983–992.

[18] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Computing Surveys*, vol. 38, no. 6, pp. 1–56, 2006.

[19] F. Leibert, J. Mannix, J. Lin, and B. Hamadani, "Automatic management of partitioned, replicated search services," in *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC '11)*, Cascais, Portugal, 2011.

[20] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri, "The impact of caching on search engines," in *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2007)*, Amsterdam, The Netherlands, 2007, pp. 183–190.

[21] G. Skobeltsyn, F. P. Junqueira, V. Plachouras, and R. Baeza-Yates, "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 2008)*, Singapore, 2008, pp. 131–138.

[22] M. Najork, "The Scalable Hyperlink Store," in *Proceedings of the 20th ACM Conference on Hypertext and Hypermedia (HT 2009)*, Torino, Italy, 2009, pp. 89–98.

[23] J. Hamilton, "On designing and deploying Internet-scale services," in *Proceedings of the 21st Large Installation System Administration Conference (LISA '07)*, Dallas, Texas, 2007, pp. 233–244.

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

[25] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien, "Efficient query evaluation using a two-level retrieval process," in *Proceedings of the Twelfth International Conference on Information and Knowledge Management (CIKM 2003)*, New Orleans, Louisiana, 2003, pp. 426–434.

[26] I. Matveeva, C. Burges, T. Burkard, A. Laucius, and L. Wong, "High accuracy retrieval with multiple nested ranker," in *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2006)*, Seattle, Washington, 2006, pp. 437–444.

[27] S. Tatikonda, B. B. Cambazoglu, and F. P. Junqueira, "Posting list intersection on multicore architectures," in *Proceedings of the 34rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2011)*, Beijing, China, 2011, pp. 963–972.

[28] S. E. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford, "Okapi at TREC-3," in *Proceedings of the 3rd Text REtrieval Conference (TREC-3)*, Gaithersburg, Maryland, 1994, pp. 109–126.

[29] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the Web," Stanford University, Stanford Digital Library Working Paper SIDL-WP-1999-0120, 1999.

[30] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *Journal of the ACM*, vol. 46, no. 5, pp. 604–632, 1999.

[31] J. Lin, D. Metzler, T. Elsayed, and L. Wang, "Of Ivory and Smurfs: Loxodontan MapReduce experiments for web search," in *Proceedings of the Eighteenth Text REtrieval Conference (TREC 2009)*, Gaithersburg, Maryland, 2009.

[32] C. J. C. Burges, "From RankNet to LambdaRank to LambdaMART: An overview," Microsoft Research, Tech. Rep. MSR-TR-2010-82, 2010.

[33] B. B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt, "Early exit optimizations for additive machine learned ranking systems," in *Proceedings of the Third ACM International Conference on Web Search and Data Mining (WSDM 2010)*, New York, New York, 2010, pp. 411–420.

[34] L. Wang, J. Lin, and D. Metzler, "A cascade ranking model for efficient ranked retrieval," in *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2011)*, Beijing, China, 2011, pp. 105–114.

[35] V. N. Anh, O. de Kretser, and A. Moffat, "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 2001)*, New Orleans, Louisiana, 2001, pp. 35–42.

[36] V. N. Anh and A. Moffat, "Simplified similarity scoring using term ranks," in *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2005)*, Salvador, Brazil, 2005, pp. 226–233.

[37] H. Turtle and J. Flood, "Query evaluation: Strategies and optimizations," *Information Processing and Management*, vol. 31, no. 6, pp. 831–850, 1995.

[38] A. Moffat and J. Zobel, "Self-indexing inverted files for fast text retrieval," *ACM Transactions on Information Systems*, vol. 14, no. 4, pp. 349–379, 1996.

[39] T. Strohman, H. Turtle, and W. B. Croft, "Optimization strategies for complex queries," in *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2005)*, Salvador, Brazil, 2005, pp. 219–225.

[40] T. Strohman and W. B. Croft, "Efficient document retrieval in main memory," in *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2007)*, Amsterdam, The Netherlands, 2007, pp. 175–182.

[41] F. Diaz, "Integration of news content into web results," in *Proceedings of the Second ACM International Conference on Web Search and Data Mining (WSDM 2009)*, Barcelona, Spain, 2009, pp. 182–191.

[42] K. M. Risvik, Y. Aasheim, and M. Lidal, "Multi-tier architecture for web search engines," in *Proceedings of the 1st Latin American Web Congress*, Santiago, Chile, 2003, pp. 132–143.

[43] R. A. Baeza-Yates, V. Murdock, and C. Hauff, "Efficiency trade-offs in two-tier web search systems," in *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2009)*, 2009, pp. 163–170.

[44] H. Yan, S. Ding, and T. Suel, "Inverted index compression and query processing with optimized document ordering," in *Proceedings of the 18th International World Wide Web Conference (WWW 2009)*, Madrid, Spain, 2009, pp. 401–410.

[45] J. Zhang, X. Long, and T. Suel, "Performance of compressed inverted list caching in search engines," in *Proceedings of the 17th International World Wide Web Conference (WWW 2008)*, Beijing, China, 2008, pp. 387–396.

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

[47] B. Goetz, *Java Concurrency in Practice*. Addison-Wesley, 2006.

[48] S. Mizzaro, "How many relevances in information retrieval?" *Interacting With Computers*, vol. 10, no. 3, pp. 305–322, 1998.

[49] K. Gimpel, N. Schneider, B. O'Connor, D. Das, D. Mills, J. Eisenstein, M. Heilman, D. Yogatama, J. Flanigan, and N. A. Smith, "Part-of-speech tagging for Twitter: Annotation, features, and experiments," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT 2011)*, Portland, Oregon, 2011, pp. 42–47.

[50] A. Ritter, S. Clark, Mausam, and O. Etzioni, "Named entity recognition in Tweets: An experimental study," in *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing (EMNLP 2010)*, Edinburgh, Scotland, 2011, pp. 1524–1534.

[51] S. Gouws, D. Metzler, C. Cai, and E. Hovy, "Contextual bearing on linguistic variation in social media," in *Proceedings of the ACL-HLT 2011 Workshop on Language in Social Media*, Portland, Oregon, 2011.