Incremental Text Indexing for Fast Disk-Based Search

GIORGOS MARGARITIS and STERGIOS V. ANASTASIADIS, University of Ioannina

Real-time search requires to incrementally ingest content updates and almost immediately make them searchable while serving search queries at low latency. This is currently feasible for datasets of moderate size by fully maintaining the index in the main memory of multiple machines. Instead, disk-based methods for incremental index maintenance substantially increase search latency with the index fragmented across multiple disk locations. For the support of fast search over disk-based storage, we take a fresh look at incremental text indexing in the context of current architectural features. We introduce a greedy method called Selective Range Flush (SRF) to contiguously organize the index over disk blocks and dynamically update it at low cost. We show that SRF requires substantial experimental effort to tune specific parameters for performance efficiency. Subsequently, we propose the Unified Range Flush (URF) method, which is conceptually simpler than SRF, achieves similar or better performance with fewer parameters and less tuning, and is amenable to I/O complexity analysis. We implement interesting variations of the two methods in the Proteus prototype search engine that we developed and do extensive experiments with three different Web datasets of size up to 1TB. Across different systems, we show that our methods offer search latency that matches or reduces up to half the lowest achieved by existing disk-based methods. In comparison to an existing method of comparable search latency on the same system, our methods reduce by a factor of 2.0–2.4 the I/O part of build time and by 21–24% the total build time.

Categories and Subject Descriptors: H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File organization*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software

General Terms: Algorithms, Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Inverted files, online maintenance, search engines, prototype implementation, performance evaluation

ACM Reference Format:
Giorgos Margaritis and Stergios V. Anastasiadis. 2014. Incremental text indexing for fast disk-based search. ACM Trans. Web 8, 3, Article 16 (June 2014), 31 pages.
DOI: http://dx.doi.org/10.1145/2560800

1. INTRODUCTION

Digital data is accumulated at exponential rate due to the low cost of storage space and the easy access by individuals to applications and Web services that support fast content creation and data exchange. Traditionally, Web search engines periodically rebuild in batch mode their entire index by ingesting tens of petabytes of data with the assistance of customized systems infrastructure and data processing tools [Brewer 2005; Dean and Ghemawat 2008; Dean and Barroso 2013]. This approach is sufficient for Web sites whose content changes relatively infrequently, or their enormous data volume makes infeasible their continuous tracking.

Part of the manuscript appeared in preliminary form in *Proceedings of the 2009 ACM Conference on Information and Knowledge Management (CIKM)*, pp. 455–464.
This work was supported in part by a Graduate Scholarship from Bodossaki Foundation, Athens, Greece.
Authors' addresses: G. Margaritis and S. V. Anastasiadis (corresponding author), Department of Computer Science and Engineering, University of Ioannina, Ioannina 45110, Greece: email: stergios@cs.uoi.gr.

DOI: http://dx.doi.org/10.1145/2560800

Today, users are routinely interested to search the new text material that is frequently added across different online services, such as news Web sites, social media, mail servers, and file systems [Lempel et al. 2007; Shah et al. 2007; Büttcher and Clarke 2008; Bjorklund et al. 2010; Geer 2010; Busch et al. 2012]. Indeed, the sources of frequently changing content are highly popular Web destinations that demand almost immediate search visibility of their latest additions [Lempel et al. 2007; Lester et al. 2008; Büttcher and Clarke 2008; Peng and Dabek 2010; Busch et al. 2012]. Real-time search refers to the fast indexing of fresh content and the concurrent support of interactive search; it is increasingly deployed in production environments (e.g., Twitter, Facebook) and actively investigated with respect to the applied indexing organization and algorithms.

Text-based retrieval remains the primary method to identify the pages related to a Web query, while the *inverted file* is the typical index structure used for Web search [Arasu et al. 2001; Brewer 2005; Chen et al. 2011]. An inverted file stores for each term a list of pointers to all the documents that contain the term. A pointer to a document is called a *posting*, and a list of postings for a particular term is called an *inverted list* [Zobel and Moffat 2006]. The *lexicon* (or *vocabulary*) of the inverted file associates every term that appeared in the dataset to its inverted list. In a *word-level* index a posting specifies the exact position where a term occurs in the document, unlike a *document-level* index that only indicates the appearance of a term in a document. Word positions are valuable in Web search because they are used to identify the adjacency or proximity of terms, such as in phrase queries (see also Section 7) [Arasu et al. 2001; Williams et al. 2004; Brewer 2005; Zobel and Moffat 2006].

A Web-scale index applies a distributed text indexing architecture over multiple machines [Arasu et al. 2001; Barroso et al. 2003; Brewer 2005; Baeza-Yates et al. 2007a; Leibert et al. 2011]. Scalability is commonly achieved through an index organization called *document partitioning*. The system partitions the document collection into disjoint subcollections across multiple machines and builds a separate inverted index (*index shard*) on every machine. A client submits a search query to a single machine (*master* or *broker*). The master broadcasts the query to the machines of the search engine and receives back disjoint lists of documents that satisfy the search criteria. Subsequently, it collates the results and returns them in ranked order to the client. Thus, a standalone search engine running on a single machine provides the basic building block for the distributed architectures that provide scalable search over massive document collections.

When a fresh collection of documents is crawled from the Web, a *batch update scheme* rebuilds the index from scratch. Input documents are parsed into postings with the accumulated postings periodically flushed from memory into a new partial index on disk. Techniques similar to external sorting merge the multiple index files into a single file at each machine [Zobel and Moffat 2006]. Due to fragmentation of each inverted list across multiple partial indices on a machine, search is supported by an older index during the update. Instead, an *incremental update scheme* continuously inserts the freshly crawled documents into the existing inverted lists and periodically merges the generated partial indices to dynamically maintain low search latency [Hirai et al. 2000; Lempel et al. 2007].

Disk-based storage is known as a performance bottleneck in search. Thus, index pruning techniques have been developed to always keep in memory the inverted lists of the most relevant keywords or documents, but lead to higher complexity in index updating and context-sensitive query handling [Broder et al. 2003; Zobel and Moffat 2006; Anh and Moffat 2006; Ntoulas and Cho 2007; Strohman and Croft 2007]. Although the latency and throughput requirements of real-time search are also currently met by distributing the full index on the main memory of multiple machines [Busch et al. 2012], the purchase cost of DRAM is two orders of magnitude higher than that of disk storage capacity [Saxena et al. 2012]. Therefore, it is crucial to develop disk-based data structures, algorithmic methods, and implementation techniques for incremental text indexing to interactively handle queries without the entire index in memory.

In this study we examine the fundamental question of whether disk-based text indexing can efficiently support incremental maintenance at low search latency. We focus on incremental methods that allow fast insertions of new documents and interactive search over the indexed collection. We introduce two new methods, the *Selective Range Flush* and *Unified Range Flush*. Incoming queries are handled based on postings residing in memory and the disk. Our key insight is to simplify index maintenance by partitioning the inverted file into disk blocks. A block may contain postings of a single frequent term or the inverted lists that belong to a range of several infrequent terms in lexicographic order. We choose the right block size to enable sequential disk accesses for search and update. When memory gets full during index construction, we only flush to disk the postings of those terms whose blocks can be efficiently updated. Due to the breadth of the examined problem, we leave outside the study scope several orthogonal issues that certainly have to be addressed in a production-grade system, such as concurrency control [Lempel et al. 2007], automatic failover [Leibert et al. 2011], or the handling of document modifications and deletions [Lim et al. 2003; Guo et al. 2007].

For comparison purposes, we experiment with a software prototype that we developed, but we also apply asymptotic analysis. In experiments with various datasets, we achieve search latency that depends on the number of retrieved postings rather than fragmentation overhead, as well as index building time that is substantially lower than that of other methods with similar search latency. To the best of our knowledge, our indexing approach is the first to group infrequent terms into lexicographic ranges, partially flush both frequent and infrequent terms to disk, and combine the aforesaid with block-based storage management on disk. Prior maintenance methods for inverted files randomly distributed the infrequent terms across different blocks [Tomasic et al. 1994], or handled each term individually [Zobel et al. 1993; Brown et al. 1994]. Alternatively, they partially flushed to disk only the frequent terms [Büttcher et al. 2006a; Büttcher and Clarke 2008], or used disk blocks of a few kilobytes with limited benefits [Brown et al. 1994; Tomasic et al. 1994].

The main contributions of the present manuscript include:

(i) reconsideration of incremental text indexing for fast disk-based search;
(ii) introduction of two innovative methods and a storage organization scheme for disk-based text indexing;
(iii) prototype implementation of our methods into a functional system;
(iv) experimental measurement of search and update time across representative indexing methods across different systems;
(v) comparative evaluation of different configuration parameters, storage and memory allocation methods, and datasets; and
(vi) asymptotic I/O analysis of build cost for Unified Range Flush.

In Section 2 we provide background information on incremental text indexing before we experimentally motivate our work in Section 3. In Section 4 we specify the studied problem, present the system structure, and introduce our indexing methods. We describe our prototype implementation in Section 5. In Section 6 we go over our experimentation environment and present build and search performance measurements from experiments with alternative methods, system parameters, datasets, and systems. We comprehensively compare our study with previous related work in Section 7 and outline our conclusions in Section 8. Finally, in Appendix A we provide an I/O complexity analysis of Unified Range Flush.

Table I. Asymptotic Cost (in I/O operations) Required to Incrementally Build Inverted Files and Retrieve Terms for Query Handling

Index Maintenance Method	Build Cost	Search Cost
Nomerge [Tomasic et al. 1994; Lester et al. 2008] [Büttcher et al. 2006b; Heinz and Zobel 2003]	$\Theta(N)$	N/M
Immediate Merge [Lester et al. 2008; Cutting and Pedersen 1990] [Büttcher and Clarke 2008]	$\Theta(N^2/M)$	1
Logarithmic Merge [Büttcher et al. 2006b], *Geometric Partitioning* [Lester et al. 2005, 2008]	$\Theta(N \cdot \log(N/M))$	$\log(N/M)$
Geometric Partitioning with $\leq p$ *partitions* [Lester et al. 2008]	$\Theta(N \cdot (N/M)^{1/p})$	p
Hybrid Immediate Merge [Büttcher et al. 2006b; Büttcher and Clarke 2008] *Unified Range Flush [Appendix A]*	$\Theta(N^{1+1/a}/M)$	1 or 2 (according to the list threshold)
Hybrid Logarithmic Merge [Büttcher et al. 2006b]	$\Theta(N)$	$\log(N/M)$

N is the number of indexed postings and M is the amount of main memory used for postings gathering. The parameter a (e.g., $a = 1.2$) refers to the Zipf distribution (Appendix A).

2. BACKGROUND

In this section, we summarize the current general approaches of incremental text indexing and factor out the relative differences of existing methods with respect to the new methods that we introduce.

Merge-based methods maintain on disk a limited number of files that contain fragments of inverted lists in lexicographic order. During a merge, the methods read sequentially the lists from disk, merge each list with the new postings from memory, and write the updated lists back to a new file on disk. The methods amortize the I/O cost if they create on disk multiple inverted files and merge them in specific patterns [Lester et al. 2005, 2008]. *In-place methods* avoid reading the whole disk index and incrementally build each inverted list by appending new memory postings at the end of the list on disk. The need for contiguity makes it necessary to relocate the lists when the disk runs out of empty space at their end [Tomasic et al. 1994; Lester et al. 2004, 2006; Büttcher and Clarke 2008]. Although the linear I/O cost of in-place methods is lower than the polynomial of merge-based, the sequential disk transfers of merge-based indexing practically outperform the random I/O of in-place [Lester et al. 2006].

Hybrid methods separate terms into short and long according to the term popularity in the indexed dataset. Recent methods use a merge-based approach for the short terms and in-place appends for the long ones. The system treats a term as short or long depending on the number of postings that either have shown up in total until now (*contiguous*) [Büttcher et al. 2006a] or participate in the current merging process (*noncontiguous*) [Büttcher et al. 2006b]. In the noncontiguous case, if a term contributes more than T (e.g., $T = 1\text{MB}$) postings to the merging process, the method moves the postings from the merge-based index to the in-place index; this reduces the build time but may slightly increase the retrieval time of long terms due to their storage on both the in-place and merge-based indices. Depending on their functional availability, we consider alternative variations of the Wumpus [2011] methods in our experiments (Section 6).

As shown in Table I for representative methods, the asymptotic complexity of index building is estimated by the number of I/O operations expressed as a function of the number of indexed postings. We include the search cost as the number of partial indices (also called *runs*) across which an inverted list is stored. The *Nomerge* method flushes

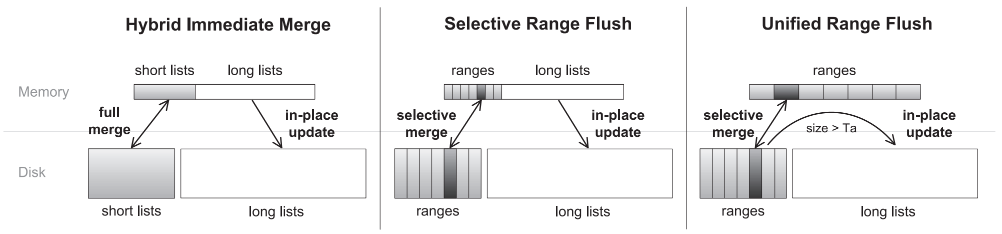


Fig. 1. Hybrid Immediate Merge only applies partial flushing to long (frequent) terms, while Selective Range Flush (SRF) and Unified Range Flush (URF) partially flush both short (infrequent) and long terms. Unlike SRF, URF organizes all postings in memory as ranges, allows a term to span both the in-place and merge-based indices, and transfers postings of a term from the merge-based to the in-place index every time they reach a size threshold T_a (see also Section 4.5).

its postings to a new run on disk every time memory gets full. Although impractical to process queries, Nomerge provides a baseline for the minimum indexing time. The *Immediate Merge* method repeatedly merges the postings in memory with the entire inverted file on disk every time memory gets full. The *Geometric Partitioning* and *Logarithmic Merge* methods keep multiple runs on disk and use a hierarchical pattern to merge the postings of memory and the runs on disk. The Geometric Partitioning method with $\leq p$ partitions continuously adjusts the fanout of the merging tree to keep the number of runs on disk at most p. In the particular case of $p = 2$, Geometric Partitioning is also known as *Square Root Merge* [Büttcher et al. 2006b]. Hybrid versions of the preceding methods partition the index into in-place and merge-based indices.

Our methods, Selective Range Flush and Unified Range Flush, differ from existing ones because we organize the infrequent terms into ranges that fit into individual disk blocks and store each frequent term into dedicated disk blocks (Figure 1). Additionally, we only partially flush frequent and infrequent terms from memory to preserve the disk I/O efficiency. The two methods differ with respect to the criteria that they apply to categorize the terms as short or long and also to determine which terms should be flushed from memory to disk. In Table I we include the asymptotic costs of Unified Range Flush as estimated in Appendix A.

According to experimental research, build time may additionally depend on system structures and parameters not always captured by asymptotic cost estimates [Lester et al. 2006; Büttcher and Clarke 2008]. Thus, although the Hybrid Immediate Merge and Unified Range Flush have the same asymptotic complexity as shown in Table I, we experimentally find their measured merge performance to substantially differ by a factor of 2. More generally, the potential discrepancy between theoretical and empirical results is a known issue in literature. For instance, online problems are the type of optimization problems that receive input and produce output in online manner, but each output affects the cost of the overall solution. Several paging algorithms are examples of online algorithms that theoretically incur the same relative cost (competitive ratio) to an optimal algorithm, but they clearly differ with respect to experimentally measured performance [Borodin and El-Yaniv 1998].

In Table II, we factor out the main functional differences among representative methods that we consider. The index update varies from simple creation of new runs, to purely merge-based and hybrid schemes. In hybrid schemes, term postings are respectively stored at the merge-based or in-place index according to their count in the entire index (*Total*) or the index part currently being merged (*Merge*). The merging pattern varies from sequential with a single run on disk, to hierarchical that tightly controls the number of runs and range-based that splits the index into nonoverlapping intervals of sorted terms. When the memory fills up, most existing methods flush the

Table II. Main Functional Differences among Existing and Our New Methods of Incremental Text Indexing

Index Maintenance Method	Update Scheme	Threshold Count	Merging Pattern	Partial Flushing	Flushing Criterion	Storage Unit
Nomerge	new run	none	none	none	full mem.	runs
Immediate Merge	merge	none	sequential	none	full mem.	single run
Geometric Partition.	merge	none	hierarchical	none	full mem.	partitions
Hybrid Log. Merge	hybrid	merge/total	hierarchical	none	full mem.	segments
Hybrid Imm. Merge	hybrid	merge/total	sequential	in-place	list size	segments
Select. Range Flush	hybrid	total	range-based	both	list ratio	blocks
Unified Range Flush	hybrid	merge	range-based	both	range size	blocks

entire memory to disk except for the Hybrid Immediate Merge that partially flushes long terms; in contrast, our methods apply partial flushing to both frequent and infrequent terms (Figure 1). The criterion of partial memory flushing alternatively considers the posting count of individual terms and term ranges or their ratio. Most methods allocate the space of disk storage as either one or multiple runs (alternatively called partitions or segments [Büttcher et al. 2006b]) of overlapping sorted terms, while we use blocks of nonoverlapping ranges.

3. THE SEARCH COST OF STORAGE FRAGMENTATION

In this section we experimentally highlight that search latency is primarily spent on disk I/O to retrieve inverted lists. Across different queries, latency can be relatively high even when stop-words are used or caching is applied, which makes the efficiency of storage access highly relevant in fast disk-based search [Zobel and Moffat 2006; Baeza-Yates et al. 2007b].

Early research on disk-based indexing recognized as a main requirement the contiguous storage of each inverted list [Cutting and Pedersen 1990; Tomasic et al. 1994]. Although storage contiguity improves access efficiency in search and update, it also leads to complex dynamic storage management and frequent or bulky relocations of postings. Recent methods tend to relax the contiguity of inverted lists so that they lower the cost of index building. One particular study partitioned the postings of each term across multiple index files and stored the inverted list of each long term as a chain of multiple noncontiguous segments on disk [Büttcher et al. 2006b]. Not surprisingly, it has been experimentally shown across different systems that multiple disk accesses (e.g., 7 in GOV2) may be needed to retrieve a fragmented inverted list regardless of the list length [Lester et al. 2008; Margaritis and Anastasiadis 2009]. List contiguity is particularly important for short terms because they dominate text datasets and are severely affected by list fragmentation. From the Zipf distribution of term frequency, the inverted file of a 426GB text collection has more than 99% of inverted lists smaller than 10KB [Cutting and Pedersen 1990; Büttcher et al. 2006b]. If a list of such size is fragmented into k runs, the delay of head movement in a hard disk typically increases the list retrieval time by a factor of k.

We examine the importance of query I/O efficiency by using the Zettair search engine with an indexing method that stores the inverted lists contiguously on disk [Zettair 2009]. Using the index of the GOV2 (426GB) text collection, we evaluate 1,000 standard queries [TREC 2006] in a server as specified in Section 6.1 with a disabled buffer cache. Thus, we measure the time to return the 20 most relevant documents per query along with the percentage of time spent on I/O. We sort the queries by increasing response time and calculate the average query time for the 50%, 75%, 90%, 95%, and 100% fastest. According to the last row of Table III, 64% of the average query time is spent

Table III. Search Time and its I/O Fraction in GOV2/Zettair

Queries (%)	Avg (ms)	I/O (%)
50	105	67
75	255	58
90	680	58
95	1,053	61
100	1,726	64

Table IV. Average, Median and 99th Percentile of Search Latency when Different Numbers of Stop-Words are Applied with and without Page Caching Using the GOV2 Dataset over the Zettair Search Engine

stop words	w/out caching			w/ caching		
	avg	med	99th	avg	med	99th
0	1,726	291	19,616	1,315	274	13,044
10	640	247	5,283	508	217	4,182
20	489	242	3,221	413	204	2,873
100	411	232	2,398	341	188	1,959

on I/O for reading inverted lists from disk. The percentage becomes 67% for the 50% fastest queries, which mostly consist of nonfrequent terms with small inverted lists.

Caching keeps in memory the postings of frequently queried terms, while stop-words are frequent terms usually ignored during query handling [Zobel and Moffat 2006; Baeza-Yates et al. 2007b]. From Table IV it follows that enabling the page cache decreases by 24% the average latency, 6% the median, and 34% the 99th percentile. Caching is generally known to reduce the latency of interactive services, but cannot directly address the problem of variable responsiveness in distributed systems unless the entire working set resides in main memory [Dean and Barroso 2013]. If we additionally omit the 10, 20, or 100 most common stop-words during query handling, the enabled buffer cache still decreases latency by about 18% on average. For instance, using 10 stop-words combined with caching lowers the average latency by 71% from 1.7s to 508ms, of which 45% is still spent on I/O.

Query latency is often evaluated using average measurements that do not convey the high variations across different queries [Büttcher et al. 2006b; Lester et al. 2005]. In Table III, the average query latency is about 1.7s, even though the 50% fastest queries only take an average of 105ms. If we presumably double the duration of the 50% fastest queries, the average latency across all the queries is only increased by 3%. Similarly, the discrepancy between the average and median latency measurements in Table IV further demonstrates the effect from the few long queries to the measured statistics. A long query involves a large number of documents or terms and greatly affects the user experience. Although sequential query execution is preferable at heavy load, parallel evaluation reduces the response time at low or moderate load. Additionally, multithreaded query handling processes the list chunks in their sequential order [Jeon et al. 2013]. Therefore, the efficient handling of long queries is facilitated through the preservation of storage contiguity.

Given the substantial time fraction of query handling spent on I/O, we advocate to preserve the list contiguity of frequent and infrequent terms through the design and storage-level implementation of the indexing method. Additionally, we aim to achieve low query latency, both on average and across different percentiles.

4. SYSTEM ARCHITECTURE

We first describe the studied problem along with our goals, and then explain the data structures and the SRF method to solve it. Motivated by our long experimental effort to tune SRF, we then proceed to the description of the URF method with simpler structure but similar (or even better, sometimes) build and search performance.

4.1. Problem Definition

In this study we mainly focus on the incremental maintenance of inverted files for efficient index building and search. We do not examine the related problems of parsing input documents to extract new postings, or the ranking of retrieved postings for query relevance. We primarily aim to minimize the I/O time required to retrieve the term postings of a query and the total I/O time involved in index building. More formally we set the following two goals:

$$\text{query handling: minimize } \sum_i \text{I/O time to read the postings of term}_i \tag{1}$$

$$\text{index building: minimize } \sum_j \text{I/O time to flush posting}_j, \tag{2}$$

where i refers to the terms of a query and j refers to the postings of the indexed document collection. The I/O time of query handling depends on the data volume read from disk along with the respective access overhead. Similarly, the total I/O time of index building is determined by the volume of data transferred between memory and disk along with the corresponding overhead. Accordingly, we aim to minimize the amount of read data during query handling, the amount of read and written data during index building, and minimize access overheads in all cases.

Ideally, for efficient query handling we would store the postings of each term contiguously on disk in order to retrieve a requested term with a single I/O. Also, for efficient index building, we would prefer to flush new postings from memory to disk with a single write I/O and without any involvement of reads. One challenge that we face in index building is that we do not know in advance the term occurrences of the incoming documents. As a result, we cannot optimally plan which postings to flush for maximum I/O efficiency every time memory gets full.

In fact, the preceding goals are conflicting because the I/O efficiency of query handling depends on the organization of term postings by index building. In the extreme case that we write new postings to disk without care for storage contiguity, query handling becomes impractical due to the excessive access overhead involved to read the fragmented postings from disk. As a reasonable compromise, we only permit limited degree of storage fragmentation in the postings of a term, and also ensure sufficient contiguity to read a term roughly sequentially during query handling. At the same time we limit the volume of data read during index building but with low penalty in the I/O sequentiality of disk reads and writes. Next we explain in detail how we achieve this.

4.2. System Structure

As we add new documents to a collection, we accumulate their term postings in memory and eventually transfer them to disk. We lexicographically order the terms and group them into ranges that fit into disk blocks (called *rangeblocks*) of fixed size B_r. Rangeblocks simplify the maintenance of inverted files because they allow us to selectively update parts of the index. We flush the postings of a range R from memory by merging them into the respective rangeblock on disk. If the merged postings overflow the rangeblock, we equally divide the postings –and their range– across the original rangeblock and any number of additional rangeblocks that we allocate as needed.

For several reasons, we do not store all the lists in rangeblocks. First, the list of a frequent term may exceed the size of a single rangeblock. Second, the fewer the postings in a rangeblock, the lower the update cost, because the merge operation transfers fewer bytes from disk to memory and back. Third, we should defer the overflow of a rangeblock because the ranges that emerge after a split will accumulate fewer postings than the original range, leading to higher merging cost. Finally, we experimentally confirm that merge-based management involves repetitive reads and writes that are mostly efficient for collections of infrequent terms, while in-place management uses list appends that are preferable for terms with large number of postings. Consequently, we store the list of a frequent term on exclusively occupied disk blocks that we call *termblocks*. We dynamically allocate new termblocks as existing termblocks run out of empty space. For efficiency, we choose the size B_t of the termblock to be different from the rangeblock B_r (Section 5). Where clarity permits, we collectively call *posting blocks* the rangeblocks and termblocks.

The lexicon is expected to map each term to the memory and disk locations where we keep the respective postings. The B-tree provides an attractive mapping structure because it concisely supports ranges and flexibly handles large numbers of indexed items. However, we experimentally noticed that the B-tree introduces multiple disk seeks during lookups, thus substantially increasing the latency of index search and update. As an alternative lexicon structure we considered a simple sorted table (called *indextable*) that fully resides in memory. For each range or frequent term, the indextable uses an entry to store the locations of the postings across the memory and disk. For terms within a range, the indextable plays the role of a sparse structure that only approximately specifies their position through the range location. For every terabyte of indexed dataset, the indextable along with the auxiliary structures occupy memory space in the order of few tens of megabytes. Therefore, the memory configuration of a typical server makes the indextable an affordable approach to build an efficient lexicon. We explain in detail the indextable structure in Section 5.

4.3. The Selective Range Flush Method

We call *posting memory* the space of capacity M_p that we reserve in main memory to temporarily accumulate the postings from new documents. When it gets full, we need to flush postings from memory to disk. We consider a term *short* or *long* if it, respectively, occupies total space up to or higher than the parameter *term threshold* T_t. For conciseness, we also use the name short or long to identify the postings and inverted lists of a corresponding term.

Initially all terms are short, grouped into ranges, and transferred to disk via merging. Whenever during a range merge the posting space of a term exceeds the threshold T_t, we permanently categorize the term as long and move all its postings into a new termblock. Any subsequent flushing of new postings for a long term is simply an append to a termblock on disk (Section 5). We still need to determine the particular ranges and long terms that we will flush to disk when memory gets full. Long postings incur a one-time flushing cost, while short ranges require repetitive disk reads and writes for flushing over time. From an I/O efficiency point of view, we prefer to only perform a few large in-place appends and totally avoid merges or small appends. Although writes appear to occur asynchronously and return almost instantly, they often incur increased latency during subsequent disk reads due to the cleaning delays of dirty buffers [Batsakis et al. 2008].

For efficient memory flushing, next we introduce the *Selective Range Flush* method (Algorithm 1). We maintain the long terms and the term ranges sorted by the space their postings occupy in memory (lines 1 and 2). We compare the memory space (bytes) of the largest long list against that of the largest range (line 7). Subsequently, we flush the largest long list (lines 8–13), unless its postings are F_p times fewer than those of the respective range, in which case we flush the range (lines 15–28). We repeat the preceding process until *flushed memory* (M_f) bytes of postings are flushed to disk. Our approach generalizes a previous method of partial memory flushing [Büttcher and Clarke 2008] in two ways: (i) We avoid inefficiently flushing the entire posting memory because we only move to disk M_f bytes per memory fill-up. (ii) In addition to long terms, we also selectively flush ranges when their size becomes sufficiently large with respect to that of long terms.

The constant F_p is a fixed configuration parameter that we call *preference factor*. Its choice reflects our preference for the one-time flushing cost of a long list rather than the

ALGORITHM 1: Pseudocode of the SELECTIVE RANGE FLUSH method

```
1:  Sort long terms by memory space of postings
2:  Sort ranges by memory space of postings
3:  while (flushed memory space < $M_f$) do
4:     $T$ := long term of max memory space
5:     $R$ := range of max memory space
6:     // Compare T and R by memory space of postings
7:     if ($R$.mem_postings < $F_p \times T$.mem_postings ) then
8:        // Append postings of T to on-disk index
9:        if (append overflows the last termblock of list) then
10:          Allocate new termblocks (relocate the list if needed)
11:       end if
12:       Append memory postings to termblocks
13:       Delete the postings of $T$ from memory
14:    else
15:       // Merge postings of R with on-disk index
16:       Read the lists from the rangeblock of $R$
17:       Merge the lists with new memory postings
18:       if (list size of term $w > T_t$) then
19:          Categorize term $w$ as long
20:          Move the inverted list of $w$ to new exclusive termblock
21:       end if
22:       if (rangeblock overflows) then
23:          Allocate new rangeblocks
24:          Split merged lists equally across rangeblocks
25:       else
26:          Store merged lists on rangeblock
27:       end if
28:       Delete the postings of $R$ from memory
29:    end if
30: end while
```

repetitive transfers between memory and disk of a range. We flush a range only when the size of the largest long list becomes F_p times smaller. Then the flushing overhead of the long list takes too much for the amount of data flushed. We also prefer to keep the short postings in memory and avoid their merging into disk. The parameter F_p may depend on the performance characteristics of the system architecture, such as the head movement overhead, the sequential throughput of the disk, and the statistics of the indexed document collection, such as the frequency of terms across the documents.

4.4. Sensitivity of Selective Range Flush

The SRF method behaves greedily because it only considers the memory space occupied by a range or long term, and simply estimates the flushing cost of a range as F_p times that of a long term. We extensively experimented with alternative or complementary flushing rules that either: (i) directly estimate the disk I/O throughput of ranges and long terms to prioritize their flushing, or (ii) aggressively flush the long terms with memory space exceeding a minimum limit to exploit the append I/O efficiency, or (iii) flush the ranges with fewest postings currently on disk to reduce the merging cost, or (iv) periodically flush the ranges or long terms with low rate of posting accumulation. In the context of the SRF algorithm, the approach to selectively flush a few tens of megabytes from the largest terms or ranges in memory performed more robustly overall.

Table V. Sensitivity to Interactions between Rangeblock Size B_r and Preference Factor F_p

Index Building Time (min) of Selective Range Flush

	F_p						
B_r (MB)	5	10	20	40	80	160	max inc
8	42.83	42.57	42.83	43.55	44.97	47.52	+11.6%
16	**42.58**	**41.63**	**41.22**	41.48	42.08	43.28	+5.0%
32	43.42	41.68	41.38	**40.43**	**40.73**	**41.18**	+7.4%
64	46.90	42.85	41.77	41.02	41.15	41.28	+14.3%
128	51.77	46.82	43.73	41.87	41.75	41.52	+24.7%
256	62.18	53.28	46.75	43.57	43.13	42.40	+**46.7%**
max inc	+46.0%	+28.0%	+13.4%	+7.7%	+10.4%	+15.4%	

We underline the lowest measurement on each row and use **bold** for the lowest on each column. The highest measured time is 62.18min (B_r = 256MB, F_p = 5), that is, 53.8% higher than the lowest 40.32s (B_r = 32MB, F_p = 40).

Table VI. Parameters of Selective Range Flush (SRF) and Unified Range Flush (URF)

Symbol	Name	Description	Value
B_r	Rangeblock	Byte size of rangeblock on disk	32MB
B_t	Termblock	Byte size of termblock on disk	2MB
M_p	Posting Memory	Total memory for accumulating postings	1GB
M_f	Flushed Memory	Bytes flushed to disk each time memory gets full	20MB
F_p	Preference Factor	Preference to flush short or long terms by SRF	20
T_t	Term Threshold	Term categorization into short or long by SRF	1MB
T_a	Append Threshold	Amount of postings flushed to termblock by URF	256KB

In the last column we include their default values used in our prototype.

SRF combines low indexing time with list contiguity on disk [Margaritis and Anastasiadis 2009], but also has several shortcomings. First, if the statistics of a processed dataset change over time, it is possible that a term initially reserves some memory space, but then stops accumulating new postings to trigger flushing. Second, in order for SRF to behave efficiently across different datasets, it requires tuning of several parameters and their interactions for specific datasets or systems [Büttcher and Clarke 2008]. For example, the optimal preference factor F_p and term threshold T_t may vary across different term frequencies or interact in complex ways with other system parameters, such as the rangeblock size B_r. Third, as the dataset processing progresses, the number of ranges increases due to rangeblock overflows; consequently, the number of memory postings per range decreases, leading to low flushing efficiency.

In Table V we examine the variation of the SRF index building time across all possible combinations of 7 values for B_r and 6 for F_p (42 pairs in total). Due to the large number of experiments involved, we limit the indexing to the first 50GB of GOV2. The elements in the last row and column of the table report the largest increase of build time with respect to the minimum measurement of the respective column and row. We notice that as B_r increases, for example, due to restrictions from the file system, then we have to tune the preference factor to retain low build time. Otherwise, the build time may increase as high as 47% with respect to the lowest measurement for a specific B_r value. The respective increase of the highest measured value to the lowest in the entire table is 53.8%. After a large number of experiments across different combinations of parameters, we identified as default values for build and search efficiency those specified in Table VI (see also Sections 4.4 and 6.4).

ALGORITHM 2: Pseudocode of the UNIFIED RANGE FLUSH method

```
 1: Sort ranges by total memory space of postings
 2: while (flushed memory space < $M_f$) do
 3:    $R$ := range of max memory space
 4:    // Merge postings of $R$ with on-disk index
 5:    Read the inverted lists from the rangeblock of $R$
 6:    Merge the lists with new memory postings
 7:    if (list size of term $w > T_a$) then
 8:       // Move postings of $w$ to exclusive termblock
 9:       if ($w$ does not have termblock or append will overflow last termblock) then
10:          Allocate new termblocks (relocate list, if needed)
11:       end if
12:       Append memory postings to termblocks
13:    end if
14:    if (rangeblock overflows) then
15:       Allocate new rangeblocks
16:       Split merged lists equally across rangeblocks
17:    else
18:       Store merged lists on rangeblock
19:    end if
20:    Remove the postings of R from memory
21: end while
```

4.5. The Unified Range Flush Method

In order to facilitate the practical application of SRF, we evolved it to a new method that we call *Unified Range Flush* (*URF*). In this method we assign each memory posting to the lexicographic range of the respective term without the categorization as short or long. Thus, we omit the term threshold T_t and preference factor F_p of SRF along with their interactions against other parameters. When the posting memory gets full, we always pick the range with the most postings currently in memory and merge it to disk, including the terms that SRF would normally handle as long. In order to reduce the data volume of merge, we introduce the *append threshold* (T_a) parameter. If the postings of a term in a merge occupy memory space $> T_a$, we move them (*append postings*) from the rangeblock to an exclusive termblock. Subsequently, the range continues to accumulate the new postings of the term in the rangeblock, until their amount reaches the number T_a again.

The pseudocode of URF is shown in Algorithm 2. In comparison to SRF, it is strikingly simpler because we skip the distinct handling of short and long terms. Algorithm 2 simply identifies the range with the most postings in memory at line 3 and merges it with the corresponding rangeblock on disk at lines 5–6 and 14–20. If there are terms with amount of postings exceeding the threshold T_a, URF flushes them to their corresponding termblocks at lines 7–13. From our experience across different datasets, the parameter T_a controls the disk efficiency of the append operation and primarily depends on performance characteristics of the I/O subsystem, such as the geometry of the disk. Instead, the term threshold T_t of SRF permanently categorizes a term as long and prevents it from merge-based flushing, even at low amount of posting memory occupied by the term. The general approach of dynamic threshold adjustment followed by previous research would only further complicate the method operation (e.g., in Section 6.2 we examine the automated threshold adjustment θ_{PF} = auto [Büttcher and Clarke 2008]).

The description of T_a bears some similarity to the definition of long-term threshold T introduced previously [Büttcher et al. 2006b]. However, the URF algorithm has

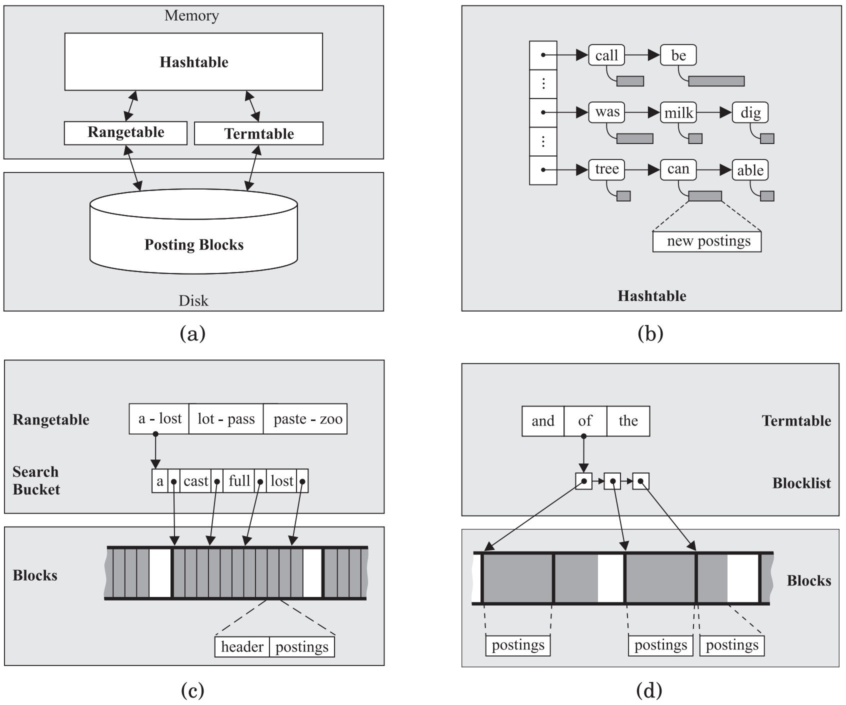


Fig. 2. (a) The prototype implementation of *Proteus*; (b) we maintain the hashtable in memory to keep track of the postings that we have not yet flushed to disk; (c) each entry of the rangetable corresponds to a term range and points to the search bucket, which serves as partial index of the corresponding rangeblock; (d) each entry of the termtable corresponds to a term and points to the blocklist that keeps track of the associated termblocks on disk.

fundamental differences from the hybrid approach of Büttcher et al. First, every invocation of hybrid merge flushes all the postings currently gathered in memory. Instead, we only flush the largest ranges with total amount of postings in memory at least M_f. Second, the choice of T_a only affects the efficiency of the index building process, because we separately control the search efficiency through the termblock size B_t. In contrast, T determines the storage fragmentation of each long list; choosing small T improves the update performance but reduces the efficiency of query processing. Indeed, we experimentally found that it is possible to achieve lowest building time and search latency for T_a around 128KB–256KB, but such small values for T would significantly lower query processing performance due to the excessive fragmentation in long lists.

We summarize the parameters of our architecture in Table VI. In the following sections, we show that URF in comparison to SRF: (i) has fewer parameters and lower sensitivity to their values, (ii) has similar index maintenance performance (or better over a large dataset) and search performance, and (iii) has more tractable behavior that allows us to do complexity analysis of index building in Appendix A.

5. PROTOTYPE IMPLEMENTATION

The *Proteus* system is a prototype implementation that we developed to investigate our inverted file management (Figure 2(a)). We retained the parsing and search components of the open-source Zettair search engine (v. 0.9.3) [Zettair 2009]. Unlike the original implementation of Zettair that builds a lexicon for search at the end of index building, we dynamically maintain the lexicon in Proteus throughout the building process. We store the postings extracted from the parsed documents in a memory-based hashtable that we call *hashtable* (Figure 2(b)). The inverted list of each term consists of the document identifiers along with the corresponding term locations in ascending order. We store each list as an initial document identifier and location followed by a list of gaps compressed with variable-length byte-aligned encoding [Zobel and Moffat 2006]. Compression considerably reduces the space requirements of postings across memory and disk.

We keep track of the term ranges in a memory-based sorted array that we call *rangetable* (Figure 2(c)). Each entry corresponds to the range of a single rangeblock and contains the space size of the disk postings along with the names of the first and last term in the range. In a sparse index that we call *search bucket*, we maintain the name and location of the term that occurs every 128KB along each rangeblock. The search bucket allows us to only retrieve the exact 128KB that may contain a term instead of the entire rangeblock. In our experience, any extra detail in rangeblock indexing tends to significantly increase the maintenance overhead and lookup time without considerable benefits in performance of query evaluation (Section 6.3). We use a sorted array (*termtable*) to keep track of the termblocks that, respectively, store the long terms of SRF or the append postings of URF (Figure 2(d)). We organize the termtable as an array of *descriptors*. Each descriptor contains the term name, a pointer to the memory postings, their size, the amount of free space at the last termblock on disk, and a linked list of nodes called *blocklist*. Each node contains a pointer to a termblock on disk.

The rangetable along with the termtable together implement the index table in our system (Section 4). Initially, the termtable is empty and the rangetable contains a single range that covers all possible terms. The inverted lists in memory that belong to the same range are connected through a linked list. If the inverted lists after a merge exceed the capacity of the respective rangeblock, we split the range into multiple half-filled rangeblocks. Similarly, if we exceed the capacity of the last termblock, we allocate new termblocks and fill them up. After a flush, we update the tables to accurately reflect the postings that they currently hold. When the capacity of a termblock is exceeded, we allocate a new termblock following one of three alternative approaches.

(1) *Fragmented (FRG).* Allocate a new termblock of size B_t to store the overflown postings.
(2) *Doubling (DBL).* Allocate a new termblock of twice the current size to store the new postings of the list.
(3) *Contiguous (CNT).* Allocate a termblock of twice the current size and relocate the entire list to the new termblock to keep the list contiguous on disk. This is our default setting.

For a term, the DBL allocation leads to number of termblocks that is logarithmic with respect to the number of postings, while FRG makes it linear. In our evaluation, we consider the implementation of the previous approaches over Proteus for both SRF and URF (Section 6.5).

5.1. Memory Management and I/O

For every inverted list in memory, the hashtable stores into a *posting descriptor* information about the inverted list along with pointers to the term string and the list itself (Figure 2(b)). For the postings of the inverted list, we allocate a simple byte array whose size is doubled every time it fills up. When an inverted list is flushed to disk, we free the respective posting descriptor, term string, and byte array. The efficiency of these memory (de-)allocations is crucial for the system performance because they are invoked extremely often.

Initially, we relied on the standard `libc` library for the memory management of the inverted lists. On allocation, the library traverses a list of free memory blocks (*free list*) to find a sufficiently large block. On deallocation, the freed block is put back into the free list and merged with adjacent free blocks to reduce external fragmentation. We refer to this scheme as *default*. If a program runs for long time and uses a large amount of memory, the free list becomes long and the memory fragmented, increasing the management cost. In order to handle this issue, we use a single call to allocate both the descriptor and term, or instead to deallocate them after an inverted list is flushed to disk. The byte array is not included in the preceding optimization, because we cannot selectively free or reallocate portions of an allocated chunk every time we double the array size. We refer to this scheme as *singlecall*.

We further reduce the management cost by using a single call to get a memory chunk (typically 4KB) and store the posting descriptors and term strings of a range. In a chunk, we allocate objects (strings and descriptors) in a stack-like manner. Processor cache locality is also improved when we store together the objects of each range. If the current chunk has insufficient remaining space, we allocate an object from a new chunk that we link to the current one. When we flush a range to disk, we traverse the chunk list to free all the term strings and descriptors of the range. We refer to this scheme as *chunkstack*.

In our prototype system, we store the disk-based index over the default file system. Hence, we cannot guarantee the physical contiguity of disk files that are incrementally created and extended over time. Disk blocks are allocated on demand as new data is written to a storage volume, leading to file fragmentation across the physical storage space. To prevent the file fragmentation caused by the system, we examined using the *preallocation* of index files. Index building includes document parsing, which reads documents from disk to memory (I/O intensive) and extracts their postings (CPU intensive). *Prefetching* asynchronously retrieves the next part of a dataset (during the processing of the dataset) to prevent the blocking of subsequent disk reads during the processing of the dataset [Patterson et al. 1995]. We evaluate all the aforesaid approaches of memory management and I/O optimization in Section 6.5.

6. PERFORMANCE EVALUATION

We compare the index build and search performance across a representative collection of methods (from Table I) over Wumpus and Proteus. In our experiments we include the performance sensitivity of the URF method across several configuration parameters, storage, and memory allocation techniques, and other I/O optimizations. We also explore the relative build performance of SRF and URF over different datasets.

6.1. Experimentation Environment

We execute our experiments on servers running the Debian distribution of Linux kernel (v.2.6.18). Each server is equipped with one quad-core$\times$86 2.33GHz processor, 3GB RAM, and two SATA disks. We store the generated index and the document collection on two different disks over the Linux ext3 file system. Different repetitions of an experiment on the same server lead to negligible measurement variations ($<$1%).

We mostly use the full 426GB GOV2 standard dataset from the TREC terabyte track [TREC 2006]. Additionally, we examine the scalability properties of our methods with the 200GB dataset from Wikipedia [2008], and the first 1TB of the ClueWeb09 dataset from CMU [ClueWeb 2009]. We mainly use 7.2KRPM disks of 500GB capacity, 16MB cache, 9–9.25ms seek time, and 72–105MB/s sustained transfer rate. In some experiments (ClueWeb, Section 6.6), we store the data on a 7.2KRPM SATA disk of 2TB capacity, 64MB cache, and 138MB/s sustained transfer rate.

We use the latest public code of Wumpus [2011] and set the threshold T equal to 1MB, as suggested for a reasonable balance between update and query performance. We measure the build performance of HIM in Wumpus with activated partial flushing and automated threshold adjustment [Büttcher and Clarke 2008]. In both systems we set $M_p = 1$GB. In Proteus, unless otherwise specified, we set the parameter values $B_t = 2$MB, $B_r = 32$MB, $M_f = 20$MB, $F_p = 20$, $T_t = $1MB, and $T_a = 256$KB (Table VI, Sections 4.4 and 6.4). The auxiliary structures of URF and SRF for GOV2 in Proteus occupy less than 42MB in main memory. In particular, with URF (SRF) we found the hashtable to occupy 4MB, the termtable and rangetable together 0.5MB, the blocklists 0.25MB (0.12MB), and the range buckets 31.2MB (36.5MB).

To keep Wumpus and Proteus functionally comparable, we activate full stemming across both systems (Porter's option [Porter 1980]). Full stemming reduces terms to their root form through suffix stripping. As a result, document parsing generates a smaller index and takes longer time; also query processing often takes more time due to the longer lists of some terms, and only approximately matches the searched terms over the indexed documents. In Proteus we use an unoptimized version of Porter's algorithm as implemented in Zettair. This makes the parsing performance of Proteus pessimistic and amenable to further optimizations. When we examine the performance sensitivity of Proteus to configuration parameters, we use a less aggressive option called *light stemming*, that is the default in Zettair.

6.2. Building the Inverted File

First we examine the build time of several methods implemented over Wumpus and Proteus. In the case of GOV2, Hybrid Immediate Merge (HIM) keeps each short term in one merge-based run, and each long term in one in-place and one merge-based run. Hybrid Square Root Merge (HSM) keeps each short term in two merge-based runs, and each long term in one in-place and two merge-based runs. Hybrid Logarithmic Merge (HLM) has each short term over four merge-based runs, and each long term over one in-place run and four merge-based runs. Nomerge fragments the postings across 42 runs. SRF maintains the postings of each term in a unique rangeblock or termblock, while URF keeps each infrequent term in one rangeblock and each frequent term in up to one rangeblock and one termblock.

In Figure 3 we consider the build time of the methods Nomerge$_W$, HLM$_W$, HSM$_W$, and HIM$_W$ in Wumpus, and the methods HIM$_P$, SRF$_P$, and URF$_P$ as we implemented them in Proteus. HIM$_W$ is the contiguous version of HIM (HIM$_C$ [Büttcher and Clarke 2008], Section 2, Appendix A) with all the applicable optimizations and the lowest build time among the Wumpus variations of HIM as we experimentally verified. According to the Wumpus implementation of contiguous and noncontiguous methods, the postings of a long term are dynamically relocated to ensure storage on multiple segments of size up to 64MB each [Wumpus 2011]. In order to ensure a fair comparison of different methods on the same platform, we also implemented HIM$_C$ in Proteus (HIM$_P$) with the CNT storage allocation by default. The index size varied from 69GB for URF$_P$ down to 60GB for SRF$_P$ and HIM$_P$, due to about 10GB difference in the empty space within the respective disk files.

The Wumpus methods take between 254min (baseline Nomerge$_W$) and 523min (HIM$_W$). HSM$_W$ and HLM$_W$ reduce the time of HIM$_W$ by 20% and 31%, respectively, but fragment the merge-based index across two and four runs. This behavior is known to substantially increase the I/O time of query processing, and consequently we do not consider HSM$_W$ and HLM$_W$ any further [Büttcher and Clarke 2008; Margaritis and Anastasiadis 2009]. The 531min of HIM$_P$ is comparable to the 523min required by HIM$_W$; in part, this observation validates our HIM implementation over Proteus. Instead, SRF$_P$ and URF$_P$ take 404min and 421min, respectively, which is 24% and

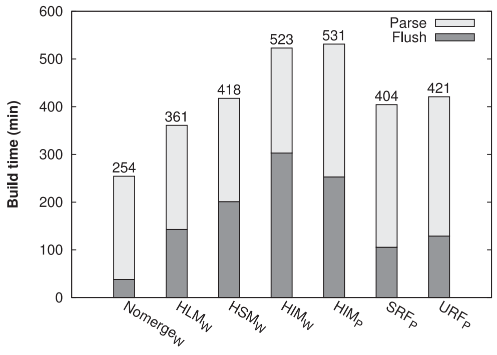


Fig. 3. We consider the index building time for different indexing methods across Wumpus and Proteus, both with full stemming. Over Wumpus, we examine Nomerge (Nomerge$_W$), Hybrid Logarithmic Merge (HLM$_W$), Hybrid Square Root Merge (HSM$_W$), and Hybrid Immediate Merge (HIM$_W$). Over Proteus, we include Hybrid Immediate Merge (HIM$_P$), Selective Range Flush (SRF$_P$), and Unified Range Flush (URF$_P$). URF$_P$ takes 421min to process the 426GB of GOV2 achieving 18.1MB/s indexing throughput (see also Figure 9 for other datasets).

21% below HIM$_P$. URF$_P$ takes 4.2% more than SRF$_P$ to incrementally index GOV2, although URF$_P$ is faster than SRF$_P$ in a different dataset (Section 6.6).

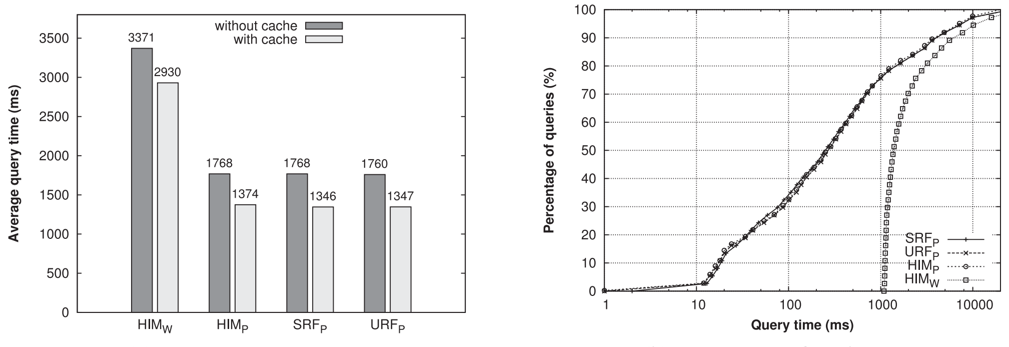
In Figure 3, we also break down the build time into *parse*, to read the datasets and parse them into postings, and *flush*, to gather the postings and transfer them to disk. HIM$_P$ reduces the flush time of HIM$_W$ from 303min to 253min, but HIM$_P$ has longer parse time partly due to the unoptimized stemming. Instead, SRF$_P$ and URF$_P$ only take 105min and 129min for flushing, respectively, thanks to their I/O efficiency. Therefore, our methods reduce the flush time of HIM$_P$ by a factor of 2.0–2.4, and that of HIM$_W$ by a factor of 2.4–2.9.

Somewhat puzzled by the longer parse time of Proteus, we recorded traces of disk transfer activity during index building. Every time we retrieved new documents for processing, we noticed substantial write activity with tens of several megabytes transferred to the index disk. Normally, parsing should only create read activity to retrieve documents and no write activity at all. However, when we flush index postings to disk, the system temporarily copies postings to the system buffer cache. In order to accommodate new documents in memory later during parsing, read requests clean dirty buffers and free memory space. Overall, SRF$_P$ and URF$_P$ reduce by about a factor of 2–3 the flush time of HIM$_P$ and HIM$_W$, and achieve a reduction of the respective total build time by 20–24%.

6.3. Query Handling

Next we examine the query time across different indexing methods and systems. In our experiments, we use the GOV2 dataset and the first 1,000 queries of the Efficiency Topics query set in the TREC 2005 terabyte track [TREC 2006]. We consider both the alternative cases of having the buffer cache disabled and enabled during query handling. As representative method of Wumpus we study the HIM$_W$, while in Proteus we consider HIM$_P$, SRF$_P$, and URF$_P$.

In the latest publicly available version of Wumpus (but also the older versions), we noticed that the implemented contiguous variation (HIM$_C$) of HIM was constantly crashing during search at a broken assertion. For that reason, in our search experiments we used the noncontiguous variation instead (HIM$_{NC}$ [Büttcher et al. 2006b]). Although the previous two Wumpus variations of HIM differ in their efficiency of index building, they have similar design with respect to query handling. They both store



(a) average query time (b) query time distribution

Fig. 4. We consider Hybrid Immediate Merge over Wumpus (HIM_W) or Proteus (HIM_P), along with Selective Range Flush (SRF_P) and Unified Range Flush (URF_P) over Proteus: (a) We measure the average query time with alternatively disabled and enabled system buffer cache across different queries in the two systems with full stemming; (b) we look at the distribution of query time over the two systems with enabled buffer cache.

each short term in one run; however, HIM_{NC} allows a long term to be stored in two runs, while HIM_C always stores it in one run. Given the long transfer time involved in the retrieval I/O of long terms, we do not expect the aforesaid difference by one disk positioning overhead to practically affect the query performance.

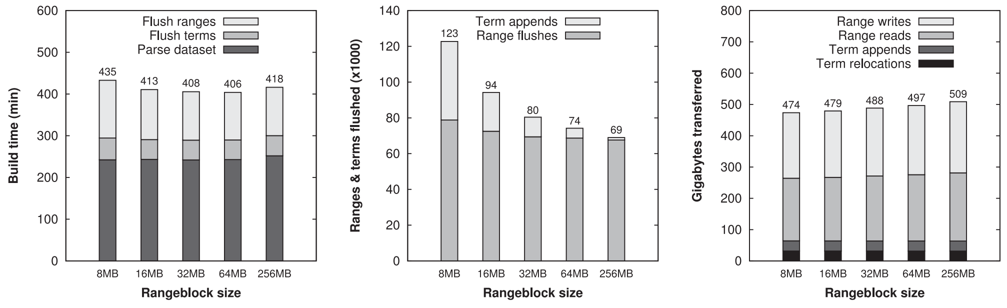
From Figure 4(a), caching activation reduces the query time of HIM_W by 13%, and by about 22–24% that of HIM_P, SRF_P, and URF_P. Across both the caching scenarios, HIM_W over Wumpus takes about twice the average query time of the Proteus methods. Given that our HIM_P implementation is based on the published description of HIM_W, we attribute this discrepancy to issues orthogonal to the indexing method, such as the query handling and storage management of the search engine. In Proteus, the average query times of HIM_P, SRF_P, and URF_P remain within 2% of each other. Therefore, both SRF_P and URF_P achieve the query performance of HIM_P, even though they are considerably more efficient in index building (Section 6.2).

We use measurement distributions to further compare the query time of the four methods with enabled system caching (Figure 4(b)). Although the median query time of Proteus lies in the range 246–272ms, that of HIM_W is 1.378s, namely a factor of 5 higher. In fact, HIM_W requires about 1s to handle even the shortest queries. Also, the 99th percentile of HIM_W is 68% higher than that of the Proteus methods. Instead, the 99th percentiles of the Proteus methods lie within 2% each other, while the median measurements lie within 10%. We conclude that HIM_P, URF_P, and SRF_P are similar in query performance, but faster by a factor of 2 on average with respect to HIM_W.

6.4. Sensitivity of Unified Range Flush

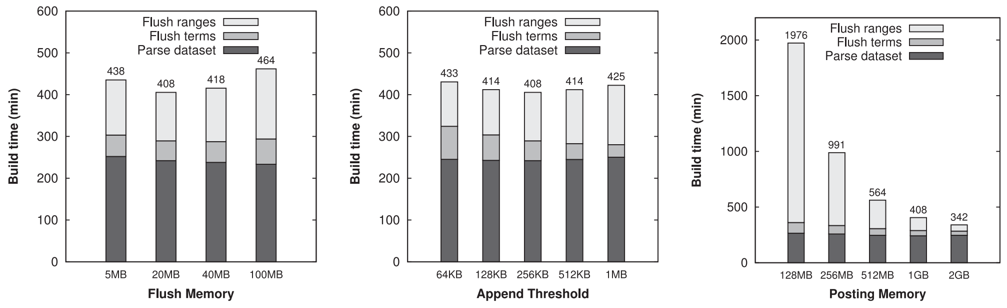
Subsequently, we consider the sensitivity of the URF build performance to the rangeblock size B_r, flush memory M_f, append threshold T_a, and posting memory M_p.

Rangeblock B_r. The rangeblock size B_r determines the posting capacity of a range; it directly affects the data amount transferred during range flushes and the I/O time spent across range and term flushes. We observed the lowest build time for B_r at 32–64MB (Figure 5(a)). Setting B_r less than 32MB generates more ranges and raises the total number of term and range flushes (Figure 5(b)). On the contrary, setting B_r higher than 64MB increases the amount of transferred data during range merges (Figure 5(c)) leading to longer I/O. Our default value $B_r = 32$MB balances the preceding two trends into build time equal to 408min. A much larger B_r practically emulates the HIM method, for instance, with $B_r = 1$GB (not shown) we measured 510min build time.



(a) index building (b) number of flushes (c) I/O activity

Fig. 5. (a) Setting the rangeblock size B_r below 32MB or above 64MB raises the build time of Unified Range Flush. Increasing the B_r tends to; (b) decrease the number of flushes; and (c) increase the data amount transferred during merges. We use Proteus with light stemming.



(a) sensitivity to M_f (b) sensitivity to T_a (c) sensitivity to M_p

Fig. 6. (a) Flushing more than a few tens of megabytes (M_f) leads to longer build time for Unified Range Flush (URF). This results from the more intense I/O activity across term and range flushes; (b) setting the append threshold to $T_a = 256$KB minimizes the total I/O time of range and term flushes; (c) the build time of range merge in URF decreases approximately in proportion to the increasing size of posting memory (M_p). The Proteus system with light stemming is used.

For sensitivity comparison with SRF, we also measured the URF build time for the first 50GB of GOV2. With B_r in the interval 8MB–256MB, we found the maximum increase in build time equal to 9.1%, that is, almost 6× times lower than the respective 53.8% of SRF (Table V).

Flush Memory M_f. The parameter M_f refers to the amount of bytes that we flush to disk every time posting memory gets full (Figure 6(a)). Build time is reduced to 408min if we set $M_f = 20$MB, namely 2% of the posting memory $M_p = 1$GB. Despite the Zipfian distribution of postings [Büttcher et al. 2006b], setting M_f below 20MB leaves limited free space to gather new postings at particular ranges (or terms) for efficient I/O. At M_f much higher than 20MB, we end up flushing terms and ranges with small amounts of new postings, leading to frequent head movement in appends and heavy disk traffic in merges. If we set $M_f = M_p$ (=1GB), we deactivate partial flushing and build time becomes 632min (not shown).

Append Threshold T_a. This parameter specifies the minimum amount of accumulated postings required during a merge to flush a term to the in-place index. It directly affects the efficiency of term appends, and indirectly their relative frequency to range flushes. In Figure 6(b) we observe that $T_a = 256$KB minimizes the URF build time. If we increase T_a to 1MB (=T_t) we end up with build time higher by 6%. Unlike

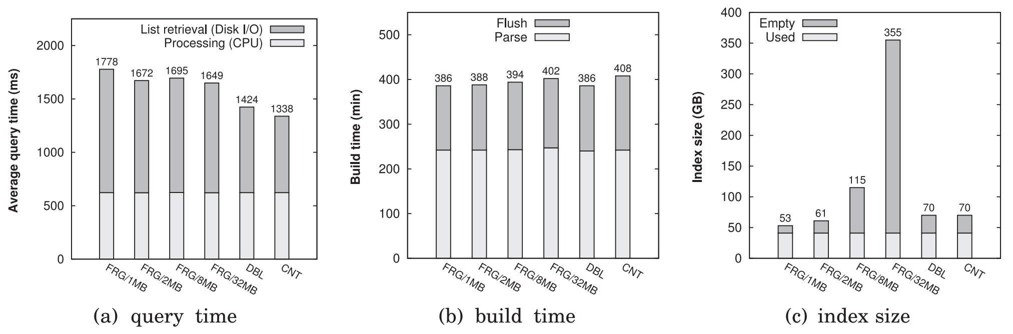


Fig. 7. We examine the behavior of Unified Range Flush over Proteus with the following storage allocation methods: (i) contiguous (CNT), (ii) doubling (DBL), and (iii) fragmented (FRG) with block sizes 1MB, 2MB, 8MB, and 32MB. (a) CNT achieves the lowest query time on average closely followed by DBL. We keep the system buffer cache enabled across the different queries; (b) build time across the different allocation methods varies within 5.7% of 386min (FRG/1MB and DBL); (c) unlike CNT and DBL, FRG tends to increase the index size, especially for larger termblock.

T_t of SRF that permanently categorizes a term as long, T_a specifies the minimum append size and tends to create larger merged ranges by including postings that SRF would permanently treat as long instead. Overall, the URF performance shows little sensitivity across reasonable values of T_a.

Posting Memory M_p. The parameter M_p specifies the memory space that we reserve for temporary storage of postings. Smaller values of M_p increase the cost of range flushes, because they enforce frequent range flushes and limit the gathering of postings from frequent terms in memory. As we increase M_p from 128MB to 1GB in Figure 6(c), the time of range merge drops almost proportionally, resulting in substantial decrease of the total build time. Further increase of M_p to 2GB only slightly reduces the build time, because at $M_p = 1$GB most time (59.3%) is already spent on parsing. As default value in our experiments we set $M_p = 1$GB.

6.5. Storage and Memory Management

Next we examine the effect of storage allocation to the build and query time of URF. Based on the description of Section 5, we consider FRG with alternative termblock sizes 1MB, 2MB, 8MB, and 32MB (respectively denoted as FRG/1MB, FRG/2MB, FRG/8MB, FRG/32MB), and also the DBL and CNT allocation methods. In Figure 7(a) we show the average CPU and I/O time of query processing in a system with activated buffer cache. The average query time varies from 1649min with FRG/32MB to 1778min with FRG/1MB, while it drops to 1424min by DBL and 1338min by CNT. Essentially, CNT reduces the query time of FRB/1MB by 25% and of DBL by 6%. The preceding variations are mainly caused by differences in I/O time given that the CPU time remains almost constant at 622min (<47% of total). Unlike query time, from Figure 7(b) we notice the build time only slightly varies from 386min for both FRG/1MB and DBL to 408min for CNT (5.7% higher). In these measurements, the flush time is about 40% of the total build time. Due to differences in the empty space of termblocks, the index size varies from 53GB for FRG/1MB to 355GB for FRG/32MB, and 70GB for DBL and CNT (Figure 7(c)). We conclude that our CNT default setting is a reasonable choice because it achieves improved query time at low added build time or index size.

In Section 5.1 we mentioned three alternative approaches to manage the memory of postings: (i) default (D), (ii) singlecall (S), and (iii) chunkstack (C). The methods differ in terms of function invocation frequency, memory fragmentation, and bookkeeping space overhead. Memory allocation affects the time spent on dataset parsing when

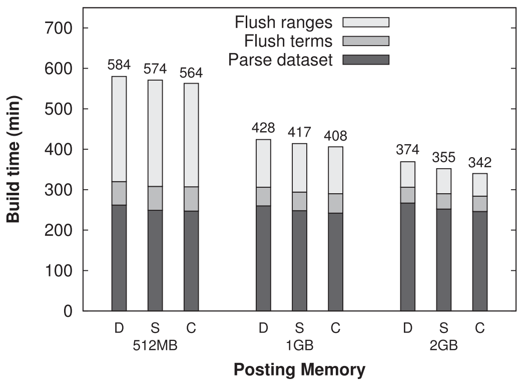


Fig. 8. We consider three methods of memory allocation during index building by Unified Range Flush: (i) default (D), (ii) singlecall (S), and (iii) chunkstack (C). The sensitivity of build time to memory management is higher (up to 8.6% decrease with C) for larger values of M_p. We use Proteus with light stemming.

Table VII. Effect of Alternative Optimizations to the Query and Build Time of Unified Range Flush

Average Build and Query Time - Unified Range Flush						
Memory and I/O Optimizations	Total Build (min)	Parse Time (min)	Flush Time Ranges (min)	Flush Time Terms (min)	Query W/out (ms)	Query W/Cache (ms)
None	543	374	124	40	2082	1537
Preallocation	534	373	112	45	1728	1316
Preallocation+Prefetching	429	260	118	47	1724	1318
Preallocation+Prefetching+Chunkstack	408	242	116	48	1726	1315

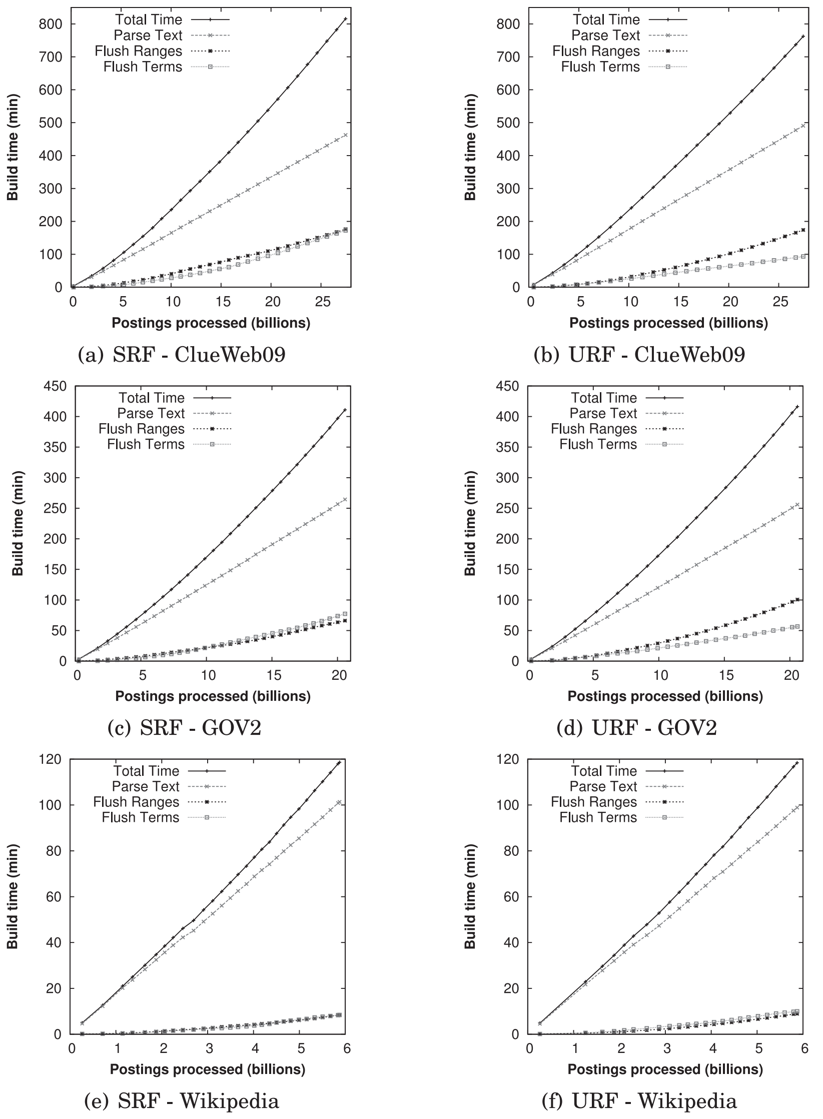
Preallocation reduces the average query time, while prefetching and chunkstack reduce the build time.

we add new postings to memory and the duration of term and range flushes when we remove postings. In Figure 8 we consider the three allocation methods with URF across different values of posting memory. Memory management increasingly affects build time as posting memory grows from 512MB to 2GB. More specifically, the transition from the default policy to chunkstack reduces build time by 3.4% for $M_p = 512$MB, 4.7% for $M_p = 1$GB, and 8.6% for $M_p = 2$GB. Therefore, larger amounts of memory space require increased efficiency in memory management to accelerate index building.

In Table VII we compare the effects of several memory and I/O optimizations to the build and search time of SRF. File preallocation of the index lowers by 14–17% the average query time as a result of reduced storage fragmentation at the file system level. For aggressive prefetching, we increase the Linux readahead window to 1MB, making it equal to the size of the parsing buffer. Thus, during the processing of 1MB text, we read the next 1MB from disk in the background. As a result, the build time drops by 20% from 534min to 429min. When we activate the chunkstack method in memory management, build time further drops by 5% from 429min to 408min. We have all these optimizations activated throughout the experimentation with Proteus.

6.6. Scalability across Different Datasets

Finally, we measure the total build time of the CNT variants of SRF and URF for three different datasets: ClueWeb09 (first 1TB), GOV2 (426GB), and Wikipedia (200GB). In our evaluation, we use the default parameter values shown in Table VI. In Figures 9(a) and 9(b) we break down into parse and flush time the SRF and URF build time for the ClueWeb09 dataset. Even though SRF better balances the flush time of ranges and terms against each other, URF actually reduces the total build time of SRF by 7% from



(a) SRF - ClueWeb09
(b) URF - ClueWeb09
(c) SRF - GOV2
(d) URF - GOV2
(e) SRF - Wikipedia
(f) URF - Wikipedia

Fig. 9. We show the scaling of build time with Selective Range Flush (SRF) and Unified Range Flush (URF). We use the ClueWeb09 (first 1TB), GOV2 (426GB), and Wikipedia (200GB) datasets over Proteus with light stemming. URF takes 53.5min (7%) less time for ClueWeb09, about the same for Wikipedia, and 16.4min (4%) more for GOV2 in comparison to SRF.

815.8min to 762.3min. This improvement is accompanied by a respective reduction of the total flush time by 81.5min (23%) from 353.2min to 271.7min.

In Figures 9(c) and 9(d) we examine the scaling of build time for the GOV2 dataset. SRF reduces the build time of URF by 16.4min (4%) from 420.8min to 404.4min. The total number of indexed postings is 20.58bn in GOV2 (426GB) and 27.45bn in ClueWeb09 (1TB). However, GOV2 has about half the text size of ClueWeb09, and the index building of GOV2 takes almost half the time, spent for ClueWeb09. In fact, the parsing of GOV2 seems to take more than 70% of the total build time, partly due to cleaning of pages written during flushing (Section 6.2). In the Wikipedia dataset, parsing takes about 84–85% of the total build time, but both URF and SRF require the same time (about 118.5min) to build the index (Figures 9(e) and 9(f)).

Across Figure 9, the total build time of URF and SRF (e.g., ClueWeb09 and GOV2) demonstrates a nonlinearity mainly caused by the range flush time rather than the parsing and term flushing. We explored this issue by using the least-squares method to approximate the build time of GOV2 as a function of the number of postings. In our regression, we alternatively consider the linear function $f(x) = a_1 + b_1 \cdot x$ and the polynomial function $f(x) = a_2 \cdot x^{b_2}$. Using the coefficient of determination R^2 to quantify the goodness of fit, we find that both the total build time and the time of range flushing are accurately tracked by the polynomial function [Jain 1991]. By contrast, the respective times of parsing and term flushing achieve good quality of fit with linear approximation.

7. RELATED WORK

We examine related literature in organization of inverted lists, query evaluation, hybrid incremental indexing, real-time search, and scalable indexing for Web analytics.

List Organization. Modern search engines typically keep their inverted lists compressed on disk in order to reduce the space occupied by the inverted index and the time required for query evaluation. Index compression adds extra computation cost, but the gain of reduced data traffic to and from disk is relatively higher [Zobel and Moffat 2006; Lester et al. 2006]. Each new document added to the collection is assigned a monotonically increasing identifier. Thus, an inverted list consists of document identifiers sorted in increasing order (*document ordered*) and can be represented as a sequence of differences between successive document identifiers (*d-gaps*). The differences are usually smaller than the initial identifiers and can be efficiently encoded with an integer coding scheme [Zobel and Moffat 2006]. Document-ordered inverted lists are widely used for incremental index maintenance because they are updated simply by appending new postings at their end [Lester et al. 2008]. Depending on the query type and the system performance, query evaluation may require to retrieve in memory the entire document-ordered inverted list of each query term [Lester et al. 2006]. Alternatively, an inverted list is sorted according to decreasing frequency (*frequency ordered*) of term occurrence in a document or decreasing contribution (*impact ordered*) to the query-document similarity score [Zobel and Moffat 2006]. Such organizations allow inverted lists to be retrieved in blocks rather than in their entirety, making their contiguous storage relevant for the individual blocks. In comparison to a document-ordered list organization, however, the alternative organizations require additional cost (e.g., for I/O or decoding) to handle index updates and complex queries (e.g., term proximity or Boolean queries) [Zobel and Moffat 2006; Zhu et al. 2008].

Query Evaluation. Query evaluation strategies calculate the similarity of the indexed documents to a query by evaluating the contribution of each query term to all document scores (*term-at-a-time*), all query terms to a single document score (*document-at-a-time*), or the postings with highest impact to document scores (*score-at-a-time*) [Anh and Moffat 2006]. Traditionally, document-at-a-time evaluation is commonly used in Web search because it more efficiently handles context-sensitive queries for which the relation (e.g., proximity) among terms is crucial [Broder et al. 2003]. Given that a high percentage of users (e.g., 80%) only examine a few tens of relevant documents, search engines may prune their index to quickly compute the first batches of results for popular documents and keywords. Thus, a two-tier search architecture directs all incoming queries to a first-tier pruned index, and only directs to a second-tier full index those queries not sufficiently handled by the first tier [Ntoulas and Cho 2007]. In order to overcome the bottleneck of disk-based storage, pruning of an impact-sorted index allows inverted lists to be stored in memory for significantly improved performance of score-at-a-time query evaluation [Strohman and Croft 2007].

Hybrid Methods. One early hybrid approach hashed the short terms accumulated in memory into fixed-size disk regions called buckets. When a bucket filled up, the term with the most postings was categorized as long and stored at a separate disk region from that point on [Tomasic et al. 1994]. A recent hybrid method separates the terms into frequent and nonfrequent ones (according to their appearance in query logs) and maintains them in separate subindices of multiple partitions each [Gurajada and Kumar 2009]. Frequent terms use a merge strategy designed for better query performance, while infrequent terms rely on a merge strategy that attains better update performance. Block-based maintenance of term ranges was previously used to build in batch mode a data structure (*half-inverted index*) for the fast processing of certain types of advanced queries [Celikik and Bast 2009].

In previous work, we described the Selective Range Flush (SRF) method and experimentally demonstrated some of its performance benefits in build time and term retrieval I/O [Margaritis and Anastasiadis 2009]. In the present manuscript, we further motivate our study by measuring the I/O part of query time along with the effect of caching and stop-words, and demonstrate the tuning effort needed by SRF to achieve performance efficiency. Then, we propose the Unified Range Flush (URF) method for efficient index maintenance that is conceptually simpler, achieves robust performance with less tuning, and is amenable to complexity analysis. We also investigate the performance effects of several storage and memory allocation methods and examine the scaling properties of our methods across three different standard Web datasets. Finally, we provide a worst-case complexity analysis of the I/O cost required by URF index building.

Another hybrid method uses *partial flushing* to delay merges of short terms by only flushing the long terms with occupied memory that exceeds an automatically adjusted threshold [Büttcher and Clarke 2008]. When transfer efficiency drops, then all long and short postings are flushed from memory to disk. Instead, we free a small amount of memory space every time memory gets full by selectively flushing terms based on their size in memory. Recent research also considers document deletions from the indexed document set [Guo et al. 2007]. With our work, we radically simplify the general problem of online index maintenance by keeping the inverted lists on blocks rather than contiguous files. Other previous work has examined the storage of inverted lists onto collections of blocks with size up to 64KB [Zobel et al. 1993; Tomasic et al. 1994; Brown et al. 1994]. Those studies were done with architectural assumptions of two decades ago. Consequently, they undervalued the benefits of block-based maintenance due to reported overheads related to query processing and unused storage space.

Real-Time Search. Given the high cost of incremental updates and their interference with concurrent search queries, a main index can be combined with a smaller index that is frequently rebuilt (e.g., hourly) and a Just-in-Time Index (JiTI) [Lempel et al. 2007]. JiTI provides (nearly) instant retrieval for content that arrives between rebuilds. Instead of dynamically updating the index on disk, it creates a small inverted file for each incoming document and chains together the inverted lists of the same term among the different documents. Earlier work on Web search also pointed out the need to update an inverted file with document insertions and deletions in real time [Chiueh and Huang 1999]. Instead of a word-level index, the Codir system uses a single bit to keep track of multiple term occurrences in a document block (*partial inverted index*) and processes search queries by combining a transient memory-based index of recent updates with a permanent disk-based index.

Twitter commercially provided the first real-time search engine, although other companies (e.g., Google, Facebook) are also launching real-time search features [Geer 2010].

Real-time search at Twitter is recently supported by the Earlybird system that consists of inverted indices maintained in the main memory of multiple machines [Busch et al. 2012]. Earlybird reuses query evaluation code from the Lucene search engine [McCandless et al. 2010], but also implements the term vocabulary as an optimized hashtable and the inverted list as a collection of document-ordered segments with increasing size. The authors state that the topic of real-time search appears largely unexplored in the academic literature [Busch et al. 2012].

Web Analytics. In comparison to the batch scheme used until recently, the incremental update scheme of Percolator from Google reduces the average latency of document processing by a factor of 100, although it is still considered insufficient for real-time search [Peng and Dabek 2010; Busch et al. 2012]. Stateful incremental processing has also been proposed as a general approach to improve the efficiency of Web analytics over large-scale datasets running periodically over MapReduce [Dean and Ghemawat 2008; Logothetis et al. 2010]. A different study shows that the throughput achieved by a method optimized for construction of inverted files across a cluster of multicore machines is substantially higher than the best performance achieved by algorithms based on MapReduce [Wei and Jaja 2012]. Earlier work on batch index building proposed a software pipeline organization to parallelize the phases of loading the documents, processing them into postings, and flushing the sorted postings to disk as a sorted file [Melnik et al. 2001].

Techniques developed for incremental index maintenance of inverted files are applicable to the more general problem of maintaining online key-value pairs across memory and disk. The Bigtable system accumulates multidimensional data values in multiple tables across memory and disk [Chang et al. 2006]; when a size threshold is reached, it transfers tables from memory to disk and periodically merges multiple tables on disk. The Anvil system manages key-value pairs on disk applying a merge-based method similar to the one described by Lester et al. [2005] and Mammarella et al. [2009]. A similar method is also used by the Lucene open-source text indexing software [McCandless et al. 2010], the Zoie variation of Lucene employed by the Linkedin social network [Zoie 2008], and the Spyglass metadata search engine [Leung et al. 2009]. Finally, Lucene provides a feature called near-real-time search, that immediately flushes to disk the postings of newly added (or deleted) documents so that they are included in subsequent search queries without additional buffering delay.

8. CONCLUSIONS AND FUTURE WORK

We investigate the problem of incremental maintenance of a disk-based inverted file. Our objective is to improve both the search latency and index building time at low resource requirements. We propose a simple yet innovative disk organization of inverted files based on blocks and introduce two new incremental indexing methods, the Selective Range Flush (SRF) and Unified Range Flush (URF). We implemented our two methods in the Proteus prototype that we built. We extensively examine their efficiency and performance robustness using three different datasets of size up to 1TB. SRF requires considerable tuning effort across different parameter combinations to perform well. In comparison to SRF, URF has similar or even better performance, while it is also simpler, easier to tune, and amenable to I/O complexity analysis. Both in Proteus and the existing Wumpus system, we experimentally examine the search performance of the known Hybrid Immediate Merge (HIM) method with partial flushing and automatic threshold adjustment. Our two methods achieve the same search latency as HIM in Proteus while reducing into half the search latency of HIM in Wumpus. Additionally, our methods reduce by a factor of 2–3 the I/O time of HIM during index building and lower the total build time by 20% or more. Based on the performance benefits of our methods, in our future work we plan to investigate issues related to adaptive caching, concurrency control, automatic failover, parsing efficiency, and handling of document modifications and deletions.

APPENDIX

A. COMPLEXITY ANALYSIS OF UNIFIED RANGE FLUSH

For complexity comparison with existing methods of index building, we estimate the worst-case asymptotic I/O cost of our approach. We focus on the URF method because the simple flushing of the largest ranges makes the analysis more tractable. For simplicity, we assume that a termblock is not relocated when overflown. During index building, URF allows a term list to be split across the in-place and the merge-based indices. This approach is also followed by the *noncontiguous* methods of hybrid index maintenance [Büttcher et al. 2006b]. Accordingly, if the size of a short list during a merge exceeds the threshold value T, Büttcher et al. move the postings of the term that participate in the merge to the in-place index. They define as $\hat{L}(N, T)$ the number of postings accumulated in the in-place index and $\hat{P}(N, T)$ the number of postings in the merge-based index for a collection of N postings. Next, they provide the following asymptotic estimates:

$$\begin{aligned} \hat{L}(N,T) &= N - c \cdot T^{(1-1/a)} \cdot N^{1/a}, \\ \hat{P}(N,T) &= c \cdot T^{(1-1/a)} \cdot N^{1/a}, \\ c &= \frac{1}{(a-1)(\gamma + \frac{1}{a-1})^{1/a}}. \end{aligned} \tag{3}$$

The parameter $\gamma \approx 0.577216$ is the Euler-Mascheroni constant, while a is the parameter of Zipf distribution that models the frequency of term occurrences.

In Eq. (3), the counts of short and long postings result from the terms distribution rather than the method used to maintain each part of the index on disk. Therefore, if we replace T with the append threshold T_a, the previous estimates also apply to the number of postings stored in the rangeblocks and termblocks of URF. In order to indicate the intuition of URF in our analysis, we use the symbols $P_{append}(N)$ and $P_{merge}(N)$ instead of the respective $\hat{L}(N, T)$ and $\hat{P}(N, T)$.

For a collection of N postings, the total I/O cost C_{total} to build the index with URF is the sum of costs for appends, C_{append}, and merges, C_{merge}:

$$\begin{aligned} C_{total}(N) &= C_{append}(N) + C_{merge}(N) \\ &= k_{append}(N) \cdot c_{append}(N) + k_{merge}(N) \cdot c_{merge}(N), \end{aligned} \tag{4}$$

where $k_{append}()$ and $k_{merge}()$ are the respective numbers of appends and merges, whereas $c_{append}()$ and $c_{merge}()$ are the respective costs per append and merge.

If a list participates in a range merge and has size greater than T_a, we append the postings of the list to a termblock on disk. After N postings have been processed, we assume that each append takes a fixed amount of time that only depends on the disk geometry and the threshold T_a

$$c_{append}(N) \approx c_{write}(T_a) = c_{append}, \tag{5}$$

where $c_{write}()$ approximates the delay of a disk write. For a collection of N postings, each append flushes at least T_a postings to a termblock and the total number of appends does not exceed $\lfloor P_{append}(N)/T_a \rfloor$:

$$k_{append}(N) \leq \left\lfloor \frac{P_{append}(N)}{T_a} \right\rfloor = \left\lfloor N \cdot \frac{1}{T_a} - N^{1/a} \cdot \frac{c}{T_a^{1/a}} \right\rfloor \in O(N). \tag{6}$$

Instead, a range merge involves the following steps: (i) read the rangeblock to memory, (ii) merge the disk postings with new postings in memory, and (iii) write the merged postings back to the rangeblock on disk. If the rangeblock overflows, we split it into two half-filled rangeblocks. Since a rangeblock begins 50% filled and splits when 100% full, we assume that a rangeblock is 75% full on average. Thus, in a posting collection of size N, the cost of a range merge can be estimated as $c_{read}(0.75 \cdot B_r) + c_{merge}(0.75 \cdot B_r + p) + c_{write}(0.75 \cdot B_r + p)$, where p is the number of new postings accumulated in memory for the range. The $c_{merge}()$ refers to processor activity mainly for string comparisons and memory copies; we do not consider it further because we focus on disk operations. From the merged postings of amount $0.75 \cdot B_r + p$, some will be moved to termblocks because they exceed the threshold T_a. Since the number p of new postings is usually small relative to the amount of merged postings, we can also omit p and approximate $c_{merge}(N)$ with a constant:

$$c_{merge}(N) \approx c_{read}(0.75 \cdot B_r) + c_{write}(0.75 \cdot B_r) = c_{merge}. \tag{7}$$

To process a collection of N postings, we do $\lceil N/M_f \rceil$ flushes. During the i-th flush, we perform m_i range merge operations to flush a total of M_f postings. We first estimate an upper bound for m_i, before we derive an upper bound for the total number of merge operations $k_{merge}(N)$. Suppose the posting memory is exhausted for i-th time and we need to flush M_f postings. The URF method flushes the minimum number of ranges m_i needed to transfer M_f postings to disk.

In the worst-case analysis, we aim to maximize m_i. For M_p postings and R_i ranges currently in memory, m_i is maximized if the postings in memory are equally distributed across all ranges. Before a range is flushed to disk, the respective number of new postings accumulated in memory for the range has to reach $p_i = M_p/R_i$. Then, the number of ranges m_i flushed during the i-th flush operation is equal to $m_i = \frac{M_f}{p_i} = \frac{M_f \cdot R_i}{M_p}$.

Just before the i-th flush, a total of $(i-1) \cdot M_f$ postings were written to disk. From them, $P_{merge}((i-1) \cdot M_f)$ postings are stored over rangeblocks. Since each rangeblock stores an average of $0.75 \cdot B_r$ postings, the number of rangeblocks on disk is $P_{merge}((i-1) \cdot M_f)/(0.75 \cdot B_r)$. The number of ranges in the rangetable just before the i-th flush will be equal to the number of rangeblocks on disk, because each range is associated with exactly one rangeblock on disk: $R_i = \frac{P_{merge}((i-1) \cdot M_f)}{0.75 \cdot B_r}$.

Based on the preceding equations of m_i and R_i, for a collection of N postings we can derive an upper bound for the total number of range merges:

$$\begin{aligned}
k_{merge}(N) &= \sum_{i=1}^{(\#\text{ flushes})} (\#\text{ merges during } i\text{-th flush}) = \sum_{i=1}^{\lceil N/M_f \rceil} m_i \\
&= \sum_{i=1}^{\lceil N/M_f \rceil} \frac{(i-1)^{1/a} \cdot T_a^{(1-1/a)} \cdot {M_f}^{1+1/a} \cdot c}{0.75 \cdot M_p \cdot B_r} \\
&\leq \frac{T_a^{(1-1/a)} \cdot {M_f}^{1+1/a} \cdot c}{0.75 \cdot M_p \cdot B_r} \cdot \sum_{i=1}^{\lceil N/M_f \rceil} i^{1/a} && (8) \\
&\leq \frac{T_a^{(1-1/a)} \cdot {M_f}^{1+1/a} \cdot c}{0.75 \cdot M_p \cdot B_r} \cdot \left\lceil \frac{N}{M_f} \right\rceil^{1+1/a} \\
&\approx \frac{T_a^{(1-1/a)} \cdot c}{0.75 \cdot M_p \cdot B_r} \cdot N^{1+1/a} \in O(N^{1+1/a}). && (9)
\end{aligned}$$

According to Eqs. (4)–(7) and (9), the total I/O cost of index building has the following upper bound:

$$C_{total}(N) \in O(N^{1+1/a}). \tag{10}$$

From Table I, the upper-bound index building cost of Eq. (10) makes URF asymptotically comparable to HIM [Büttcher et al. 2006b]. Additionally, the approach of URF to store the postings of each term across up to two subindices makes the I/O cost of term retrieval constant.

A.1. Special Case

To cross-validate our result, we use a special case of URF to emulate the behavior of HIM [Büttcher et al. 2006b]. We set $M_{flush} = M_{total}$ to force a full flush when we run out of memory. We also append to termblocks any list with more than T_a postings and choose a large B_r value for URF to approximate the sequential merging of HIM. Each range merge transfers $0.75 \cdot B_r$ postings to disk. For collection size N, the total amount of postings written to disk across $k_{merge}(N)$ merges follows from Eq. (8):

$$\begin{aligned} P_{merge\_written}(N) &= k_{merge}(N) \cdot (0.75 \cdot B_r) \\ &= \frac{T_a^{1-1/a} \cdot {M_p}^{1+1/a} \cdot c \cdot 0.75 \cdot B_r}{0.75 \cdot M_p \cdot B_r} \cdot \sum_{i=1}^{\lceil N/M_p \rceil} i^{1/a} \\ &= \sum_{i=1}^{\lceil N/M_p \rceil} c \cdot T_a^{1-1/a} (i \cdot M_p)^{1/a} \\ &\leq c \cdot T_a^{1-1/a} \cdot \frac{N^{1+1/a}}{M_p}. \end{aligned} \tag{11}$$

After we add the linear I/O cost from appends (Eqs. (5) and (6)), and replace T_a with T at the right part of inequality (11), we estimate the worst-case cost of HIM to be that of Eq. (6) by Büttcher et al. [2006b]. Thus we asymptotically confirm that the behavior of HIM is approximated as special case of the URF method.

ACKNOWLEDGMENTS

The authors are thankful to the anonymous reviewers for their constructive comments that helped improve the manuscript. They would also like to thank Argyris Kalogeratos for his detailed feedback on an earlier draft. Access to the infrastructure of the Information Retrieval Facility (IRF) of Vienna, Austria, for experimentation purposes is gratefully acknowledged. Permission to use the ClueWeb09 Text Research Collections has been granted by Organization Agreement with Carnegie Mellon University.

REFERENCES

Vo Ngoc Anh and Alistair Moffat. 2006. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th Annual ACM SIGIR International Conference on Research and Development in Information Retrieval*. 372–379.

Arvind Arasu, Junghoo Cho, Hector Garcia-Molina, Andreas Paepcke, and Sriram Raghavan. 2001. Searching the web. *ACM Trans. Internet Technol.* 1, 1, 2–43.

Ricardo Baeza-Yates, Carlos Castillo, Flavio Junqueira, Vassilis Plachouras, and Fabrizio Silvestri. 2007a. Challenges on distributed web retrieval. In *Proceedings of the IEEE International Conference on Data Engineering.* 6–20.

Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. 2007b. The impact of caching on search engines. In *Proceedings of the 30th Annual ACM SIGIR International Conference on Research and Development in Information Retrieval*. 183–190.

Luiz Andre Barroso, Jeffrey Dean, and Urs Holzle. 2003. Web search for a planet: The Google cluster architecture. *IEEE Micro* 23, 2, 22–28.

Alexandros Batsakis, Randal Burns, Arkady Kanevsky, James Lentini, and Thomas Talpey. 2008. AWOL: An adaptive write optimizations layer. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 67–80.

Truls A. Bjorklund, Michaela Gotz, and Johannes Gerhke. 2010. Search in social networks with access control. In *Proceedings of the International Workshop on Keyword Search on Structured Data*. ACM Press, New York, 4:1–4:6.

Allan Borodin and Ran El-Yaniv. 1998. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, UK.

Eric A. Brewer. 2005. Combining systems and databases: A search engine retrospective. In *Readings in Database Systems*, 4th Ed., Joseph M. Hellerstein and Michael Stonebraker, Eds., MIT Press, Cambridge, MA, 711–724.

Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the ACM Conference on Information and Knowledge Management*. 426–434.

Eric W. Brown, James P. Callan, and W. Bruce Croft. 1994. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference on Very Large Data Bases.* 192–202.

Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy Lin. 2012. Early-bird: Real-time search at twitter. In *Proceedings of the IEEE International Conference on Data Engineering*. 1360–1369.

Stefan Buttcher and Charles L. A. Clarke. 2008. Hybrid index maintenance for contiguous inverted lists. *Inf. Retr.* 11, 3, 197–207.

Stefan Buttcher, Charles L. A. Clarke, and Brad Lushman. 2006a. A hybrid approach to index maintenance in dynamic text retrieval systems. In *Proceedings of the European Conference on IR Research.* 229–240.

Stefan Buttcher, Charles L. A. Clarke, and Brad Lushman. 2006b. Hybrid index maintenance for growing text collections. In *Proceedings of the 29th Annual ACM SIGIR International Conference on Research and Development in Information Retrieval*. 356–363.

Marjan Celikik and Hannah Bast. 2009. Fast single-pass construction of a half-inverted index. In *Proceedings of the International Symposium on String Processing and Information Retrieval*. Lecture Notes in Computer Science, vol. 5721, Springer, 194–205.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 205–218.

Chun Chen, Feng Li, Beng Chin Ooi, and Sai Wu. 2011. TI: An efficient indexing mechanism for real-time search on tweets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 649–660.

Tzicker Chiueh and Lan Huang. 1999. Efficient real-time index updates in text retrieval systems. Tech. rep. 66, ECSL, Stony Brook University, Stony Brook, NY.

ClueWeb. 2009. The ClueWeb09 dataset. http://boston.lti.cs.cmu.edu/Data/clueweb09/.

Doug Cutting and Jan Pedersen. 1990. Optimizations for dynamic inverted index maintenance. In *Proceedings of the 13th Annual ACM SIGIR International Conference on Research and Development in Information Retrieval*. 405–411.

Jeffrey Dean and Luiz Andre Barroso. 2013. The tail at scale. *Comm. ACM* 56, 2, 74–80.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Comm. ACM* 51, 1, 107–113.

David Geer. 2010. Is it really time for real-time search? *Comput.* 43, 3, 16–19.

Ruijie Guo, Xueqi Cheng, Hongbo Xu, and Bin Wang. 2007. Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In *Proceedings of the ACM Conference on Information and Knowledge Management.* 751–759.

Sairam Gurajada and Sreenivasa Kumar. 2009. On-line index maintenance using horizontal partitioning. In *Proceedings of the ACM Conference on Information and Knowledge Management.* 435–444.

Steffen Heinz and Justin Zobel. 2003. Efficient single-pass index construction for text databases. *J. Amer. Soc. Inf. Sci. Technol.* 54, 8, 713–729.

Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. 2000. WebBase: A repository of web pages. *Comput. Netw.* 33, 1–6, 277–293.

Raj Jain. 1991. *The Art of Computer Systems Performance Analysis*. Wiley.

Myeongjae Jeon, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. 2013. Adaptive parallelism for web search. In *Proceedings of the European Conference on Computer Systems.* 155–168.

Florian Leibert, Jake Mannix, Jimmy Lin, and Babak Hamadani. 2011. Automatic management of partitioned, replicated search services. In *Proceedings of the ACM Symposium on Cloud Computing.* 27:1–27:8.

Ronnu Lempel, Yosi Mass, Shila Ofek-Koifman, Yael Petruschka, Dafna Sheinwald, and Ron Sivan. 2007. Just in time indexing for up to the second search. In *Proceedings of the ACM Conference on Information and Knowledge Management.* 97–106.

Nicholas Lester, Alistair Moffat, and Justin Zobel. 2005. Fast on-line index construction by geometric partitioning. In *Proceedings of the ACM Conference on Information and Knowledge Management.* 776–783.

Nicholas Lester, Alistair Moffat, and Justin Zobel. 2008. Efficient online index construction for text databases. *ACM Trans. Database Syst.* 33, 3, 1–33.

Nicholas Lester, Justin Zobel, and Hugh Williams. 2006. Efficient online index maintenance for contiguous inverted lists. *Inf. Process. Manag.* 42, 4, 916–933.

Nicholas Lester, Justin Zobel, and Hugh Williams. 2004. In-place versus re-build versus re-merge: Index maintenance strategies for text retrieval systems. In *Proceedings of the Australasian Computer Science Conference.* 15–23.

Andrew W. Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L. Miller. 2009. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 153–166.

Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ramesh Agarwal. 2003. Dynamic maintenance of web indexes using landmarks. In *Proceedings of the Conference on World Wide Web*. 102–111.

Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Keb Yocum. 2010. Stateful bulk processing for incremental analytics. In *Proceedings of the ACM Symposium on Cloud Computing*. 51–62.

Mike Mammarella, Shant Hovsepian, and Eddie Kohler. 2009. Modular data storage with Anvil. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 147–160.

Giorgos Margaritis and Stergios V. Anastasiadis. 2009. Low-cost management of inverted files for online full-text search. In *Proceedings of the ACM Conference on Information and Knowledge Management.* 455–464.

Michael Mccandless, Erik Hatcher, and Otis Gospodnetic. 2010. *Lucene in Action*. Manning Publications, Stamford, CT.

Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. 2001. Building a distributed full-text index for the web. *ACM Trans. Inf. Syst.* 19, 3, 217–241.

Alexandros Ntoulas and Junghoo Cho. 2007. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th Annual ACM SIGIR International Conference on Research and Development in Information Retrieval*. 191–198.

R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. 1995. Informed prefetching and caching. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 79–95.

Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation.* 251–264.

Martin F. Porter. 1980. An algorithm for suffix stripping. *Program* 14, 3, 130–137.

Mohit Saxena, Michael M. Swift, and Yiying Zhang. 2012. FlashTier: A lightweight, consistent and durable storage cache. In *Proceedings of the ACM European Conference on Computer Systems*. 267–280.

Sam Shah, Craig A. N. Soules, Gregory R. Ganger, and Brian D. Noble. 2007. Using provenance to aid in personal file search. In *Proceedings of the USENIX Annual Technical Conference*. 171–184.

Trevor Strohman and W. Bruce Croft. 2007. Efficient document retrieval in main memory. In *Proceedings of the 30th Annual ACM SIGIR International Conference on Research and Development in Information Retrieval (SIGIR'07).* 175–182.

Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. 1994. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 289–300.

Trec. 2006. TREC terabyte track. National Institute of Standards and Technology. http://trec.nist.gov/data/terabyte.html.

Zheng Wei and Joseph Jaja. 2012. An optimized high-throughput strategy for constructing inverted files. *IEEE Trans. Parallel Distrib. Syst.* 23, 11, 2033–2044.

Wikipedia. 2008. The wikipedia dataset. http://static.wikipedia.org/downloads/2008-06/en/.

Hugh E. Williams, Justin Zobel, and Dirk Bahle. 2004. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.* 22, 4, 573–594.

Wumpus. 2011. Wumpus search engine. http://www.wumpus-search.org.

Zettair. 2009. The Zettair search engine. RMIT university. http://www.seg.rmit.edu.au/zettair/.

Mingjie Zhu, Shuming Shi, Nenghai Yu, and Ji-Rong Wen. 2008. Can phrase indexing help to process non-phrase queries. In *Proceedings of the ACM Conference on Information and Knowledge Management*. 679–688.

Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *Comput. Surv.* 38, 2, 6:1–6:56.

Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. 1993. Storage management for files of dynamic records. In *Proceedings of the Australian Database Conference*. 26–38.

Zoie. 2008. Zoie real-time search and indexing system built on Apache Lucene. http://code.google.com/p/zoie/wiki/ZoieMergePolicy.

Received September 2012; revised November 2013; accepted December 2013