Proteus: Scalable Management of Inverted Files for Online Full-Text Search

Georgios Margaritis Stergios V. Anastasiadis
Department of Computer Science
University of Ioannina, GREECE

Technical Report DCS 2008-07
November 5, 2008

For dynamic environments with frequent content updates, such as file systems, we require online full-text search facilities that scale to large data collections but maintain low search latency. Although recent solutions manage to reduce substantially the related cost of index building, they may raise search latency up to several factors depending on the design of the update method. The dominant techniques keep on disk several partial structures, that they continuously update as new files are added to the indexed dataset. Nevertheless, spreading indexing information across multiple locations on disk tends to considerably increase the time involved in search operations. In the present paper, we take a fresh look at the problem of online full-text search and propose a simple approach that manages the index on disk using fixed-size blocks. As we experimentally demonstrate with a prototype implementation, our architecture touches a limited number of blocks over time and keeps the total building cost a linear function of the index size. Furthermore, our latency for retrieving indexing information matches the lowest delays that previous methods only achieved at higher maintenance costs.

1 Introduction

As the cost of storage space drops and the amount of accumulated digital content grows, automated full-text search over desktop and distributed file systems becomes equally important to the search support in digital libraries and the web. Modern commercial search engines are capable of processing tens of petabytes of data on a daily basis thanks to the customized systems infrastructure and data processing tools that they operate [9]. For the web, periodical index rebuilding for the entire dataset works sufficiently well because the web has a decentralized architecture, its content changes relatively infrequently and the amount of resources made available for its indexing are enormous.

In comparison to the research activity that the web attracted, the search automation of file systems received less attention. Although the basic operation of locating terms in documents remains the same across alternative environments, file systems are substantially different in their features and corresponding requirements:

- Directory and file contents change relatively frequently.
- Users anticipate file updates to be searchable within a short period of time.
- Centralized management makes easier the tracking of modifications.
- Search operations may share resources with the indexed file system.

Additional specific properties of file systems already covered by previous work include the access control restrictions that usually apply to stored files [5], and the lack of hyperlink associations between files which makes them less amenable to web ranking methods [20].

Lexicon is the list of all individual terms that appear in a dataset. *Inverted file* is an index that for each term in the lexicon stores a list of pointers to all the documents that contain the term. Each pointer to a document is usually called *posting* and each entry in the inverted file *posting list*. We assume that a pointer specifies the exact position of a document where a terms occurs. Additionally, we consider posting lists consisting of document identifiers sorted in increasing order. We focus on datasets that allow insertions of new documents over time and examine methods to maintain inverted files efficiently on secondary storage. Index maintenance for the more general case of document updates and deletions is an interesting problem on its own that we won't consider further here [11].

If we only need to index datasets offline, we parse documents into partial indexes that we initially keep in main memory and periodically flush to disk. As a final step, we use external sorting to merge the multiple index files into a single file that handles queries for the entire dataset [24]. The above process is appropriate for static datasets. In order to support search operations concurrently with index updates, researchers came up with online approaches that periodically merge the partial indexes on disk. Typically, a trade-off arises between index building time and search time. In the rest of this paragraph, time refers to the number of disk I/O operations needed to transfer a posting list

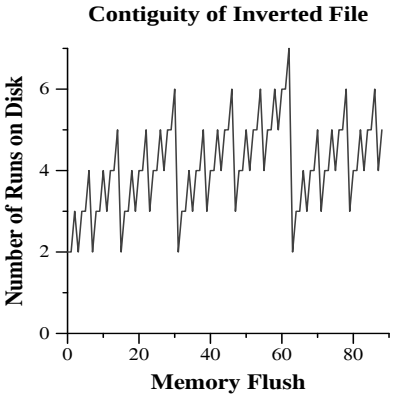

Figure 1: A recently proposed system maintains on disk a variable number of partial indexes (runs), across which it stores the inverted file. When we retrieve the postings of a term, we need to access multiple runs of the inverted file. The x axis refers to the instances that memory contents are flushed to disk. For this measurement, we used the Wumpus system with the Hybrid Logarithmic Merge policy over the 426GB GOV2 dataset [7].

from disk to memory. Ideally, each term search would involve latency that only depends on the number of postings rather than the total index size. Unfortunately, the currently known methods either take polynomial time to build an index that guarantees constant search time or require logarithmic search time for linear building time.

In Figure 1, we observe the number of partial indexes maintained by a linear building approach during the processing of a standard text collection. There are phases where we may need as many as 7 disk accesses to retrieve the posting list of a term regardless of the storage space it occupies. Let's assume typical SATA disks with seek time 8ms, average rotation latency 4ms and nominal transfer time 70 MB/s. Then, in a time period of 12ms that is equal to the head positioning overhead, the disk can read sequentially about 800 KB. Since the posting list for the majority of the terms occupies less than a few megabytes, the total retrieval time for a term may reach or exceed twice the transfer time due to access overheads. On the other hand, it takes twice the time with current technology to build an index with posting lists contiguously stored on disk.

Unlike the latest methods that keep low the merging cost through balanced-tree schemes, in the present paper we follow the more straightforward approach of maintaining the postings in fixed-size blocks. Each fixed-size block may contain the postings of a single frequent term or a lexicographically ordered subset of several infrequent terms. During the index construction, we dynamically determine the subset of terms whose postings gathered in main memory can be more efficiently flushed to disk. Thus, at each flush we only update a small number of terms on disk at cost that remains relatively constant during the index building. Depending on the frequency of a searched term, its postings on disk either (i) are stored contiguously as part of a single block, or (ii) are exclusively occupying a collection of multiple blocks. As we show experimentally, we achieve constant search cost that only depends on the number of retrieved postings and index building time that is linear function of the index size.

In Section 2, we classify existing approaches for managing inverted files online. In Section 3, we introduce our index maintenance method and describe the Proteus prototype implementation, while in Section 4, we provide some information about the experimentation environment that we used. In Section 5, we present the results from experiments with alternative system parameters and comparisons with other systems. In Section 6, we briefly summarize previous related research and in Section 7, we outline our conclusions.

2 Background

For the index maintenance, the system parses new documents into posting lists of terms temporarily maintained in main memory for improved efficiency [8]. When memory gets full, the system flushes the postings lists to disk. Early work recognizes as main requirement in the above process the contiguous storage on disk of the postings belonging to each term. By enabling the sequential retrieval of postings, storage contiguity may improve access efficiency for both query processing and index maintenance. On the other hand, storage contiguity introduces the need for complex dynamic storage management and frequent or bulky relocations during index building.

In-place methods. In-place maintenance builds each posting list incrementally as new documents are processed. The need for contiguity makes it necessary to relocate the lists when they run out of empty space at their end. Relocation incurs data movement overhead and free space management complexity. We can amortize the cost of relocation by preallocating list space for future appends using various criteria. If we accept non-contiguous posting lists, then we avoid relocations but may need multiple seeks during query processing to retrieve a posting list. If the fragments of the postings lists are sufficiently large, then it is possible to keep the seek overhead arbitrarily low [7]. On the other hand, if we append the new postings of each term separately to its posting list on disk, we may undergo substantial overhead for infrequent terms with relatively low volume of accumulated postings over time.

Merge-based methods. The alternative approach of merge-based maintenance merges postings from memory

Index Maintenance Method	Building Cost	Search Cost	References
No Merge	$\Theta(N)$	$\Theta(N/M)$	[7, 13, 22]
Immediate Merge	$\Theta(N^2/M)$	O(1)	[7, 8, 13]
Logarithmic Merge (or Geometric Partitioning)	$\Theta(N \log(N/M))$	$O(\log(N/M))$	[7, 13]
Geometric Partitioning with $\leq p$ partitions	$\Theta(N\,(N/M)^{1/p})$	O(1)	[13]
Hybrid Immediate Merge	$\Theta(N^{1.833}/M)$	O(1)	[7]
Hybrid Logarithmic Merge	$\Theta(N)$	$O(\log(N/M))$	[7]

Table 1: The table summarizes the asymptotic cost (in I/O operations) required to build and search inverted files for online full-text search. The rightmost column contains references to published literature where the corresponding methods appeared. N is the number of indexed postings and M is the amount of main memory used for postings gathering. Ideally, we would prefer to have a method that offers constant search time $O(1)$ for linear building cost $\Theta(N)$, but none of the above methods achieves that.

and disk into a single file on disk. Even though the retrieval and storage of posting lists during the merging operation exploits the sequential throughput of the disk, the entire on-disk posting list collection has to be accessed at each merging. The latest related methods amortize the cost of merging by permitting the creation of multiple posting files on disk. Only when the files meet particular conditions related to their number or size, the system merges them according to specific patterns.

The problem of merging postings lists is very similar to external sorting. One extra complication that we have with dynamic collections is that data arrive over time rather being entirely available at the beginning of the procedure. Let's call *run* a collection of posting lists lexicographically sorted by term. One way to specify the sequence of steps involved in merging multiple runs into one is to use a *tree representation*. The leaves of the tree correspond to the initial runs, while the internal nodes refer to the run that results from the merging of its descendants. The sum of the depth of all the leaves specifies the total number of times that the initial runs are processed during the merging. Although several sophisticated merging patterns are available for external sorting, it is difficult to decide which is best in a given situation [12].

Previous research in database systems has identified as the goal of merge optimization to reduce the number of runs to one while performing as few merge steps and move as few records as possible. Known heuristics that work effectively always merge the smallest existing runs or mostly use maximal fan-in. One particular approach that was previously mentioned but not investigated is to only merge or concatenate fractions of runs [10]. In our case, we can efficiently combine runs that contain non-overlapping ranges of terms without any merging. Similarly, we can improve efficiency, if we limit the merging across two runs into specific ranges of terms. In the rest of the paper we apply the above ideas for first time, and examine their merit in the context of storage management for inverted files.

Hybrid methods. Hybrid methods separate terms into short and long. One early approach hashes short terms into fixed-size disk regions called buckets [22]. Every time the memory fills up with new postings, the method merges the memory postings of each short term into those of the corresponding bucket. If the bucket also fills up, it categorizes the short term with the most postings as long. Subsequently, it moves the postings of the term from the overfilled bucket to a separate disk region, where it maintains its postings from that point on. In several new hybrid methods that were recently proposed, the system uses a merge-based approach for the short terms and in-place appends for the long ones [7]. The terms are treated as short or long depending on the number of postings participating in the merging between the postings in memory and disk. By maintaining the long terms separately using the in-place technique, hybrid methods reduce the index construction time while their search efficiency varies depending on their merge-based approach.

Discussion In Table 1, we summarize the asymptotic cost of known methods to manage inverted files in secondary storage. From the search cost that is not constant, we realize that several recent methods tend to relax the requirement for contiguity of the postings lists. For example, several efficient merge-based methods maintain on disk more than one index files. Similarly, a recent in-place method stores on disk the postings of a term in multiple groups of a minimum size rather than a single contiguous collection [7]. Nevertheless, it remains a challenge to reduce the index construction cost with the minimum impact to the search efficiency.

Based on the above observations, we simplify the inverted-file management by deliberately organizing posting lists using fixed-size blocks. We apply the in-place approach by appending the new postings of a frequent term to an increasing number of separate fixed-size blocks that we allocate on-demand. The choice of the block size keeps under control the amount of wasted space at the end of the last block and also the percentage of disk access overhead during posting list retrieval. Similarly, we sim-

Symbol	Name	Description	Default Value
B_p	Posting Block	Fixed size of each block storing postings on disk	8MB
M_p	Posting Buffer	Total buffer space for accumulating postings in memory	1GB
M_f	Memory Flushing	Bytes flushed to disk every time the posting buffer gets full	20MB
F_p	Preference Factor	Factor of flushing preference for long or short terms	3
T_t	Term Threshold	Posting list size that differentiates short terms from long	1MB

Table 2: Summary of parameters used in the proposed architecture.

plify the merge-based approach by storing multiple posting lists on each fixed-size block. We only apply merging for infrequent terms and restrict it to those blocks that contain terms with new postings in memory. We improve the efficiency of merging by managing infrequent terms in ranges rather than individually that was done previously [22].

3 *Proteus* Architecture

Even though index building involves the parsing of documents to extract the postings, in the present paper we only examine the management of the inverted file. In our design, we set two objectives: (i) build the inverted file at cost that is linear function of the number of postings, and (ii) retrieve posting lists at cost that only depends on their length and not the size of the index. To the best of our knowledge, existing systems do not achieve simultaneously both the above objectives in online full-text search. We consider the above objectives consistent with the requirements of a search engine designed for dynamic environments such as a file system, where new documents are added frequently and users expect to be able to search them shortly after their addition. In order to achieve our goal, we make the following design decisions:

1. Categorize terms into short or long based on the total space of their postings.
2. Manage short terms in lexicographic ranges and long terms individually.
3. Flush postings lists to disk selectively by amount of postings in memory.
4. Allocate disk storage space in fixed-size blocks.

We assume that as we add new documents to a collection, we can accumulate term postings in memory until we run out of the space that we reserved for that purpose. At this point we need to transfer at least some of the postings to disk to make space for the document processing to continue. We use a lexicon to keep track of all the individual terms and associate them with their postings lists on disk.

Initially, the index is empty. As we experimentally verified, if we use in-place management for terms with few postings, then we will bear significant overhead for appending their postings to disk. If alternatively we use merge-based management for terms with lots of postings, then we incur significant cost for repetitively reading posting lists to memory and merging them with new ones. Thus, we consider a term *short* or *long* respectively, depending on whether the total amount of space that its postings occupy in the system is less or exceeds the preconfigured system parameter *term threshold* T_t.

Initially we consider all terms as short. When the total number of postings in the system for a term exceeds the threshold T_t, then we categorize the term as long. We anticipate that long terms are relatively frequent and will continue to accumulate postings in memory. When we transfer the postings of a long term from memory to disk, we simply remove the postings of the term from memory and append them to the existing posting list on disk following the in-place approach. In contrast, we expect that the short terms are infrequent and we group them in lexicographically ordered ranges. We use the merge-based approach to transfer the postings of a range to disk. This means that we read from disk into memory the old postings of the range, we merge them with the new postings and transfer the entire collection of posting lists back to disk.

We store the postings on disk using fixed-size blocks of size B_p, that we call *posting blocks*. The postings of a long term exclusively occupy one or multiple blocks that we allocate dynamically over time as needed. A range of short terms takes its own block on disk where we store the postings lists lexicographically ordered by term. When the posting block of a short range gets full, we split the range into two ranges and adjust the postings across two postings blocks accordingly. Similarly, if the posting block of a long term overflows, we allocate a new posting block and move the overflow postings there. When a term changes category from short to long, we remove all its postings from the range and store them into a new dedicated block that we allocate exclusively for the new long term. The range continues to retain the postings of its terms that remain short. For simplicity, we use the name long or short not only for terms but also for their corresponding postings, posting lists or ranges.

According to our design, if a term is short its postings are contiguously stored as part of a single posting block

Algorithm 1 Flush term or range postings from memory to disk

```
1: Algorithm: SelectiveRangeFlush
2: Input: index in memory/on disk
3: Output: updated index in memory/on disk
4: Sort long terms/short ranges by memory space of postings
5: while (flushed memory space < M_f) do
6:    T_long := long term of max memory space
7:    R_short := short range of max memory space
8:    {Compare terms/ranges by memory space of postings}
9:    if (sizeof(R_short)/sizeof(T_long) < F_p) then
10:      Remove the postings of T_long from memory
11:      Allocate new posting blocks as needed
12:      Append memory postings to the posting blocks
13:   else
14:      Remove the postings of R_short from memory
15:      Merge postings into posting block of R_short
16:      if (posting block overflown) then
17:         Allocate new posting blocks as needed
18:         Split R_short equally across posting blocks
19:      end if
20:   end if
21: end while
```

on disk. When we retrieve the posting list, we only need a single disk transfer. Nevertheless, if a term is long, then its postings occupy one or multiple posting blocks. In this case, in order to retrieve the posting list, we need a number of disk transfers equal to the number of posting blocks. However, if the posting block has size in the order of a few megabytes, the corresponding disk overhead is an order of magnitude less than the transfer time involved. Overall, in our design, the retrieval time of a posting list is independent of the total size of the index. Admittedly, for a large index the posting blocks may be spread across large areas of the disk platters, which may increase the average seek time. However, when we access a disk through the file system, we don't usually control space allocation to prevent block spreading even for a small index.

Additionally, we maintain a partial index on top of each long posting list. This allows us to only retrieve the postings for specific ranges of documents. For example, this is needed in the relatively common case that we answer conjunction queries and merge the posting lists of the most frequent terms against those of the most infrequent. The answer is the intersection of the documents that contain the terms of the query.

3.1 Selective Range Flush

We call *posting buffer* the space of capacity M_p that we reserve in main memory to temporarily accumulate the postings from new documents. When the posting buffer gets full, we need some policy to determine which particular postings to transfer to disk and make space for new ones. In order to minimize the index building time, we need to keep as low as possible the total number of disk operations. Since we apply the in-place approach for the long terms, we prefer to have only few large appends of postings from memory to disk. For the short terms, whose flushing involves merging with existing postings on disk, we want to minimize the number of times each posting is read and written back to disk, and also maximize the new postings in each transfer. Thus, we group short terms into ranges and keep their postings in memory as long as possible to avoid the repetitive disk reads and writes involved during merges.

At first sight, the handling of short term flushing bears some similarity to the page replacement problem. If a range doesn't get any new postings in the future, it would make a good candidate for flushing. As with page replacement, we usually don't know in advance when a range will get new postings. However, the flushing problem is substantially different from paging because we can accumulate new postings in memory without the need to retrieve the older postings of the same range; we only need the older postings in memory when we flush the new ones to disk. Therefore, traditional approaches of tracking the latest time or the frequency with which a page is accessed are not directly applicable to the problem of deciding which particular short range to flush at any time [17]. Also, we need an appropriate balance between flushes of short and long terms. Long postings only incur an one-time cost for flushing, while short ranges potentially require repetitive reading for all the future occurrences of postings that fall within the same range. Furthermore, disk reads and writes are interdependent in the sense that writes occur asynchronously and may show up as delays during subsequent reads due to the need for cleaning dirty pages from the page cache [3].

We conclude that we face a unique problem of disk transfer scheduling. After extensive experimentation, we came up with a simple rule, called *Selective Range Flush*, that performs impressively well. Every time our posting buffer gets full, we sort the posting lists belonging to long terms or ranges of short terms according to the space they occupy in memory. We compare the size of the largest long posting list against the size of the short range that currently occupies the most space. We pick for flushing next the largest long list unless it is F_p times smaller than the largest short range. In the last case we flush the short range instead. We repeat the above process until we flush to disk an amount of postings with total capacity M_f bytes. We show the pseudocode of the method as Algorithm 1.

The constant F_p is a fixed configuration parameter that we call *preference factor*. Its choice reflects our preference to undergo the one-time cost of flushing a long posting list instead of the repetitive cost for transferring a short range back and forth to disk. In other words, we postpone the flushing of the largest short range until the size of the largest long list is F_p times smaller and the corresponding flushing overhead of the long list too high for the amount of data flushed. At the same time we express a preference to keep the postings of the short ranges in memory and

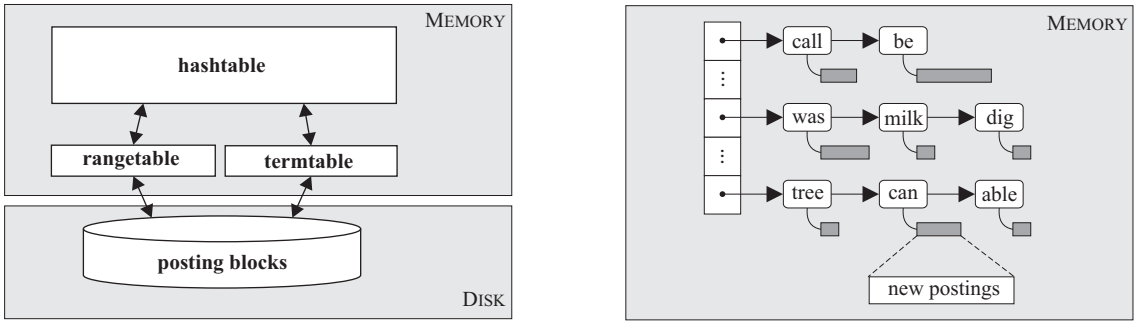

Figure 2: (a). The prototype implementation of *Proteus*. (b) We maintain the hashtable in memory to keep track of the postings that we have not flushed yet to disk.

avoid their merging into disk. The parameter F_p depends on the performance characteristics of the system architecture, such as the head-movement overhead and sequential throughput of the disk, and also the statistics of the indexed document collection, such as the frequency of the terms across the different documents. In our experiments, small values between $F_p = 2$ and $F_p = 3$ achieved the lowest building time.

Our algorithm behaves greedily in the sense that it only considers the amount of bytes that a long term or short range currently occupies in memory. We use the parameter F_p in the effort to get an estimate of the cost to flush a short range relatively to a long term. Similarly, we count the size of postings in the indexed dataset to determine the type of flushing that we apply to each term. We also tried alternative approaches that more accurately estimate the flushing throughput of posting lists or aggressively flush posting lists up to a minimum size. It turned out that the approach of *Selective Range Flush* to flush a few tens of megabytes of the largest posting lists in memory gave the best performance overall.

3.2 Prototype Implementation

We experimented extensively with a prototype implementation of the *Proteus* architecture that we developed (Figure 2(a)). We built our own inverted-file management including the Selective Range Flush algorithm, but retained the parsing and search components from an open-source distribution of the Zettair search engine (version 0.9.3) [16]. The focused and modular design of Zettair made it a good choice for our needs. We note that our main interest has been to optimize the disk-related operations and made no effort to optimize processing-related tasks beyond reasonable choices. For example, we use the standard memory management of libc, although an optimized version customized to the needs of our flushing algorithm would probably reduce measurably the processing cost.

Consistently with the original Zettair implementation, we maintain a hash table (called *hashtable*) in memory, where we store the posting list that we extract from the parsed documents (Figure 2.(b)). In the posting list of each term, the document identifiers and the corresponding locations of the terms are sorted in ascending order. Then, each list is stored as an initial identifier or position and a list of gaps compressed using variable-length byte-aligned encoding [24]. Overall, compression reduces considerably the space requirements of postings on disk and in memory.

As we already explained, we categorize terms *short* or *long* according to the total space of their postings in the system. We use a sorted array, called *termtable*, to keep track of the posting blocks associated with each long term, as shown in Figure 3(a). We use binary search to look for particular terms in the termtable. Insertion of new terms is relatively inexpensive by having the termtable consisting of pointers to descriptors rather than the descriptors themselves. Each descriptor contains the term name, the size of the postings held in memory, the amount of free space at the end of the last block on disk, and a linked list of nodes, called *blocklist*. Each node of the blocklist contains a pointer to a posting block on disk, and also the first and last document identifier held by the corresponding posting block. This is useful information for the case that we need to retrieve only a subset of the posting blocks that contain specific document identifiers.

For the short terms, we keep in memory a sorted array that we call *rangetable* (Figure 3(b)). As with the termtable, we keep insertion relatively cheap by having each entry containing a pointer to a descriptor. The descriptor contains the size of the postings held and also the names of the first and last term in the range. Additionally, it maintains a partial index, called *search bucket*, with the name and location of the term that occurs every 128KB along the posting block of the range. The search

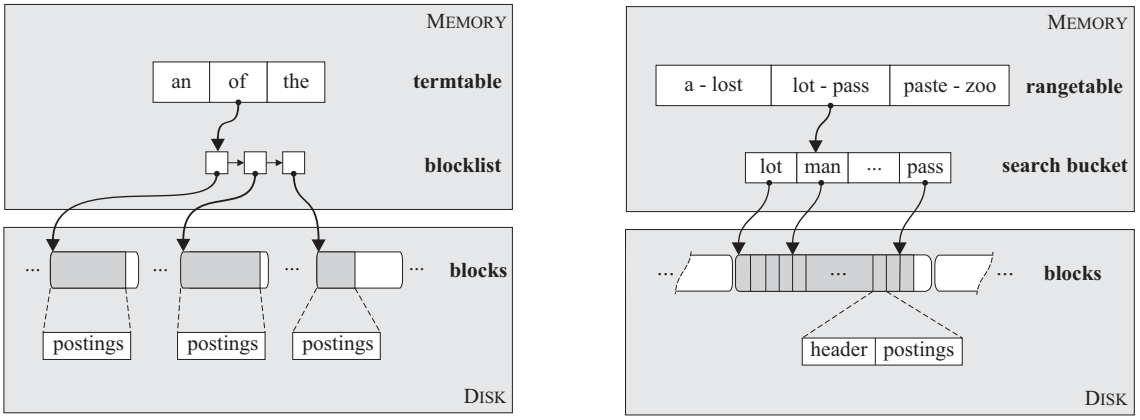

Figure 3: (a). Each entry of the termtable corresponds to a long term and points to the blocklist that keeps track of the associated posting blocks on disk. (b). Each entry of the rangetable corresponds to a range of short terms, and points to the search bucket that serves as partial index of the corresponding posting block.

bucket allows us to only retrieve a small part of the posting block that may contain a term. We still need to retrieve the indexed part of the posting block into memory in order to make sure that we have postings for a particular term. From our experience, any more detailed index to each posting block may increase significantly the maintenance overhead of the rangetable.

When the buffer space that we reserve for postings in memory fills up, we use two temporary arrays to sort by posting space the short and long ranges currently in memory. We pick the actual term or range that we flush next based on the Selective Range Flush algorithm. For each range (or long term) we maintain a linked list that connects its entry in the rangetable (or termtable) with all the associated terms that have non-empty posting lists in memory. The linked list guides us to copy into a buffer all the memory postings of the chosen range or term. Before we flush a range, we retrieve its postings already stored on disk and merge them with those accumulated in memory. If the new range that emerges exceeds the capacity of a posting block, we split the range into multiple half-filled blocks, as needed. We flush a long term by simply appending its postings to its last block on disk. If we exceed the capacity of the last posting block, we allocate more blocks on disk and completely fill them up except for the last one. After the flush, we update the tables in memory to accurately reflect the postings currently available in memory.

4 Experimental Environment

For our experiments, we used rack-mounted nodes running the Debian distribution of Linux kernel version 2.6.18. Each node is equipped with one quad-core x86 2.33GHz processor, 2GB RAM, an active gigabit ethernet port, and two 7200RPM SATA disks, one 250GB and the other 500GB. The vendors of the two disks specify the same buffer size 16MB, average seek time in the range 8.9-9ms, and sustained transfer rate 61-72MB/s. We store the document collection on the large disk and the generated index on the small one. Even though accessing the disks through their raw interface would give us better control of their disk allocation, it would also necessitate building our own filesystem. Instead, we decided to access the disks through the default filesystem of Linux (ext3). All the reported numbers correspond to system operation with negligible swapping activity.

In our experiments we use the full 426GB GOV2 dataset from the TREC Terabyte track [15]. For Proteus we use the default parameter values mentioned in Table 2, unless otherwise specified. In experiments where we do detailed statistics gathering, the total time required for the system operation may increase depending on the logging rate, but the general behavior of the system remains the same. We observed relatively limited variations ($< 5\%$), when we repeated the same experiment on the same or different node. Proteus generates index size of 70GB which is comparable to the 64GB created by Wumpus. The small discrepancy is reasonable since the two systems manage differently the storage space. Nevertheless, the posting list of the same term occupies space within a few percent of each other across the two systems.

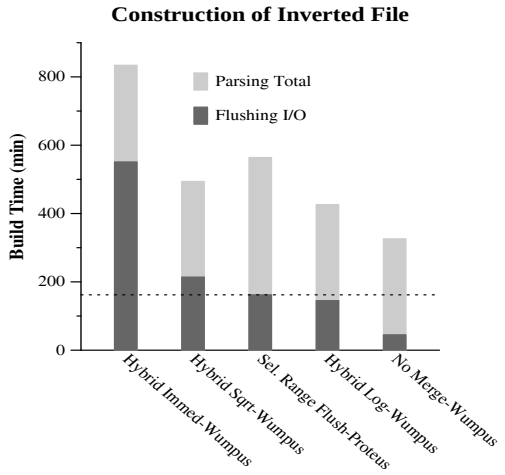

Figure 4: We break down the index building time into flushing and parsing across different maintenance policies implemented in Wumpus and Proteus. We observe that the flushing cost of Proteus falls between those of Hybrid Square Root Merge and Hybrid Logarithmic Merge, while being substantially lower than Hybrid Immediate Merge. Since parsing is not our main concern in the present study, the difference in parsing time between Wumpus and Proteus is not surprising given that, for this function, we used existing code (from Zettair) without any custom optimizations to the Proteus' needs.

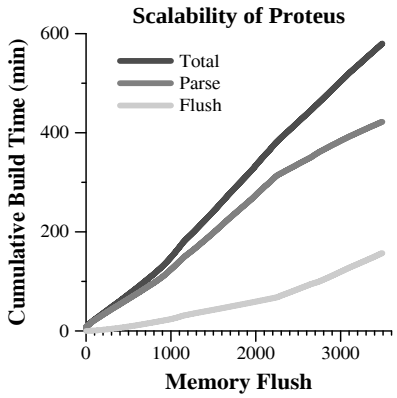

Figure 5: We measure the time of Proteus to process input documents (Parse), and flush postings to disk every time memory gets full (Flush). The x axis corresponds to instances at which Proteus flushes postings to disk. Overall, we observe that build time increases approximately linearly as a function of the index size. At the last third of the flushes, input documents contain substantially more actual terms leading to higher postings density. This has an effect to the rate at which the system reads documents from disk and flushes postings, which explains the light bending of the Parse and Flush lines at the right part of the figure.

5 Performance Evaluation

In the present Section, we compare the index building and term retrieval behavior of Proteus against alternative configurations of Wumpus. Subsequently, we examine the effects of the system configuration parameters to the performance of the Proteus prototype.

5.1 Building the Inverted File

For the experimental comparisons, we chose a subset of index maintenance methods that are known to cover a wide range of tradeoffs between building and search efficiency (Table 1). The Immediate Merge method repeatedly merges the postings in memory with the entire inverted file on disk. The Square Root method uses a balanced-tree pattern to merge the postings of memory and the runs on disk. Additionally, it continuously adjusts the fanout of the merging tree to keep the number of runs on disk at most two ($p = 2$). The Logarithmic Method uses a binary-tree pattern to merge the postings of memory with the runs on disk. In the Hybrid versions of all the above, the system partitions the index into in-place and merge-based parts. During merging, it moves to the in-place part of the index the postings of terms that accumulated more than T (typically 10^6) postings collectively in memory and the merge-based part. The No Merge method flushes its postings to a new run on disk without any merging, every time memory gets full. Although it is not practical to process an online query by retrieving postings from a large number of runs, No Merge can provide a baseline for the minimum possible building time.

We experimented with the above methods using their implementation in the Wumpus system [7]. The original Zettair implementation builds a lexicon (B-tree) to support term searches at the end of the dataset processing, which essentially makes it an offline system that we don't examine any further. In Proteus, we continuously maintain a collection of tables that serve the purpose of a lexicon and allow us to retrieve the posting lists of terms throughput the building process. We strive to improve search efficiency at low building cost, thus we mostly focus our effort on the flushing I/O rather than the processing tasks. In both Wumpus and Proteus, we choose to activate full stemming during index construction. Full stemming reduces terms to their root form by stripping one or more suffixes. Practically, stemming helps retrieve relevant documents even if the searched words do not match exactly those found in the documents. The alternative of deferring stemming to query processing makes searches slower and increases the index size due to lower compression efficiency [23].

As shown in Figure 4, the different policies cover a wide range of building times between 326 min for No Merge and 834 min for Hybrid Immediate Merge (HIM). We break down building into the time required for reading the input dataset and parsing it into postings (Parsing Total) and the time needed for gathering the postings and transferring to disk (Flushing I/O). More specifically, we observe the Flushing part of Selective Range Flush (SRF) over Proteus to lie between those of Hybrid Square Root

	Hybrid Logarithmic Merge - Wumpus					Selective Range Flush - Proteus				
Stemmed Term	**Short Bytes**	**# Runs**	**Long Bytes**	**# Segs (64KB)**	**Time ms**	**Short Bytes**	**Blks (8MB)**	**Long Bytes**	**# Blks (8MB)**	**Time ms**
anim	541211	3	4285528	89	238	0	0	4723752	1	99
colmid	1	4	0	0	89	7	1	0	0	21
gtefcu	0	4	0	0	113	0	0	0	0	30
wallet	17226	4	0	0	91	53314	1	0	0	14
floor	955182	4	1115022	21	252	0	0	2940503	1	74
spruce	93818	4	0	0	99	185226	1	0	0	31
yahoomap	29	4	0	0	84	195	1	0	0	26
degener	52819	4	0	0	74	126185	1	0	0	19
meaning	242044	4	0	0	99	778171	1	0	0	26
wage	583295	3	2618283	53	204	0	0	3515692	1	87

Table 3: Hybrid Logarithmic Merge in Wumpus spreads a posting list across several runs of the merge-based subindex and several segments of the in-place subindex. Selective Range Flush of Proteus stores the posting list of each short term contiguously in part of a posting block, while for the posting list of each long term it exclusively occupies multiple posting blocks. In comparison to Wumpus, Proteus consistently achieves a decrease of several factors in the retrieval time of the posting lists. This is a sample from the indexes created for the GOV2 collection by Proteus and Wumpus, respectively.

Merge (HSM) and Hybrid Logarithmic Merge (HLM), but also remain a factor of three lower than that of HIM. This is important because HIM is the only method that maintains one run for the merge-based part of the index and one for the in-place part similarly to SRF. On the contrary, the number of runs in the merge-based part of HSM can be up to 2 and up to 6 for HLM in the case of the GOV2. The number of runs on disk can affect significantly the query processing time, as we show in the next section.

Overall, we see that SRF manages to keep the postings of each term at only one location on disk, yet the building cost is comparable to methods that spread postings across multiple runs whose number may be constant (eg. total of 3 with HSM) or depend logarithmically on the index size (e.g. total up to 7 for HLM over 426GB). No Merge requires only 45 min to generate 73 runs on disk, but we would need an additional 80 min to merge them into a single run. In Figure 5, we measure the cumulative building time as a function of the index size. Ideally, building time should increase linearly as we add more postings to the index, independently of the total index size. Proteus approximately achieves this behavior by limiting updates to a few postings blocks every time memory gets full. This property is fundamental for the scalable operation of the system, given that data collections tend to increase over time.

5.2 Reading a Posting List

We experimentally compared the different maintenance methods by measuring the time to retrieve a posting list across different terms. In our experiments, we used the terms contained in the Efficiency Topics query set of the TREC 2005 Terabyte Track [15]. Since our main interest is the I/O efficiency of the inverted file, we flush the memory cache after every time we retrieve the posting list of a term.

In Figure 6(a), we show the number of terms that lie within the same 10ms interval of retrieval time across the different policies. We see that SRL is similar to HIM, while HSM and HLM lie on the right. HSM limits up to two the number of runs of the merge-based part. This means that as it processes the dataset, the number of runs varies between one and two. In our experiments, HSM has two runs in the merge-based part, which makes it somewhat longer in the retrieval of lists when compared to HIM and SRF. In the case that HSM has only one run (not shown), its behavior is comparable to that of HIM. Similarly, HLM varies its runs in the merge-based part between 1 and 6 (2-7 if we also consider the in-place part, Figure 1). As a result, HLM takes up to four times more to answer the short terms in our experiments with four runs. We make this more clear in Figure 6(b) where we measure the average retrieval time for terms with up to 1MB of postings in the system. Additionally, we can see the exact number of runs involved during a search for the HLM and SRF policies in Table 3.

In Figure 6(c), we repeated the experiment for terms with more than 1MB of postings. We observe that SRF takes half time in comparison to the other policies, which we attribute to the 8MB blocks that we used. Nevertheless, since in common conjunction queries we only need the intersection of the documents that contain the searched terms, only a subset of the posting list needs to be brought to memory for the long terms. Therefore, we consider the behavior of the long terms somewhat less crucial depending on the type of the search operator used.

Overall, we found that both HIM and SRF achieve the lowest time to retrieve posting lists of short terms, while HSM and HLM take more for the same purpose due to the multiple runs that they maintain. We should point out that when we retrieve the list of a short term we need to access all the runs in the merged-based part from the way that the index is constructed. This is usually needed even

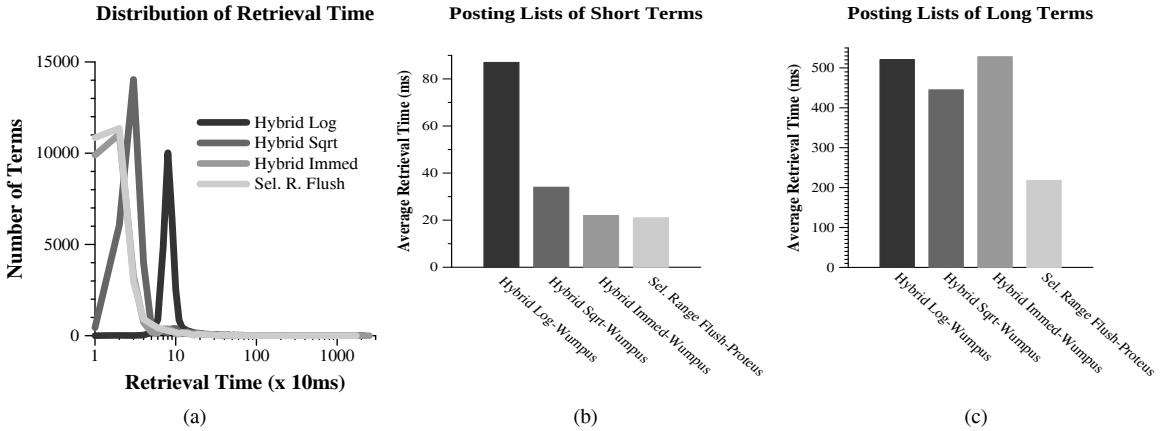

Figure 6: (a) We count the number of terms (y axis) for which the measured time of reading the posting list lies within the same 10ms interval (x axis). We observe that the curves of Hybrid Immediate Merge and Selective Range Flush almost overlap, while those of Hybrid Square Root Merge and Hybrid Logarithmic Merge lie on the right. (b) We measure the average retrieval time for the terms with total postings up to 1MB. Obviously, Hybrid Immediate Merge is comparable to Selective Range Flush. Hybrid Square Root Merge with two runs for the merge-based part takes about 50% more time, while Hybrid Logarithmic Merge with four runs take about four times more. (c) We measure the average retrieval time for terms with postings more than 1MB. Selective Range Flush requires about half time to retrieve a long list due to the 8MB blocks that we used. However, in common queries (e.g. conjunction) only a subset of the long posting list need to be actually retrieved.

in the case that there is no occurrence of the term in the indexed dataset.

5.3 Sensitivity Analysis

In this Section, we examine how the system behavior of Proteus depends on several configuration parameters.

5.3.1 Posting Block B_p

The size of the posting block B_p is probably the most critical configuration parameter in our system. It specifies the amount of postings contained in each range of short terms, therefore it has a significant impact to the amount of bytes transferred when we merge the postings of short terms in memory with those already on disk. Figure 7(a) demonstrates this effect very clearly through the length of the bars corresponding to bytes read and written during flushes of short terms. The less data we read during a flush of short term, the lower total building time we achieve. In fact, when we append postings of long terms to disk, the amount of bytes that we transfer is relatively insensitive to the block size since there is almost no read involved. As we see in Figure 7(b), an increasing block size reduces the flush time of long terms and raises that of short terms. Our default block size $B_p = 8MB$ balances the two trends leading to low index building time.

In Figure 7(c) we break down the inverted file into a part that contains postings, another that is empty due to the blocks used by short terms, and a third that is empty due to blocks used by long terms. Large postings blocks tend to increase substantially the empty space in blocks dedicated to long terms, which ends up as larger inverted file. With our default choice of $B_p = 8MB$ we ended up with 70GB index size, out of which about 41GB are actual data and the rest is empty space. We should note that a relatively large posting block leads to low disk access overhead when we retrieve terms with many postings. For example, for a disk with 12ms access overhead and 70MB/s sequential throughput, a block size of $B_p = 8MB$ keeps access overhead less than 10%. Nevertheless, depending on the type of queries that the system processes (e.g. conjunction), only a subset of the postings may be needed to be brought into memory rather than an entire posting list.

5.3.2 Preference Factor F_p

The preference factor specifies how aggressively we flush long postings relatively to short. We assume that in order to flush a term, a sufficient number of postings need to be gathered in memory. Otherwise, it is not worth it, either because the head movement cost will be high or the bytes that we need to read in memory and write back to disk will lead to a high merging cost. In Figure 8(a) it becomes clear, that if we assume that the short and long flushes have equal cost ($F_p = 1$), then we end up spending more time for short flushes. Alternatively, if we increase F_p beyond the value of 4, the long flushes dominate the I/O cost of index building. It is worth noting that reads during merges of short postings are synchronous and affect directly the total building time. On the other hand, the writes that we do during all flushes are asynchronous and are only partially accountable for the building time.

In Figure 8(b) we see that long terms are involved in

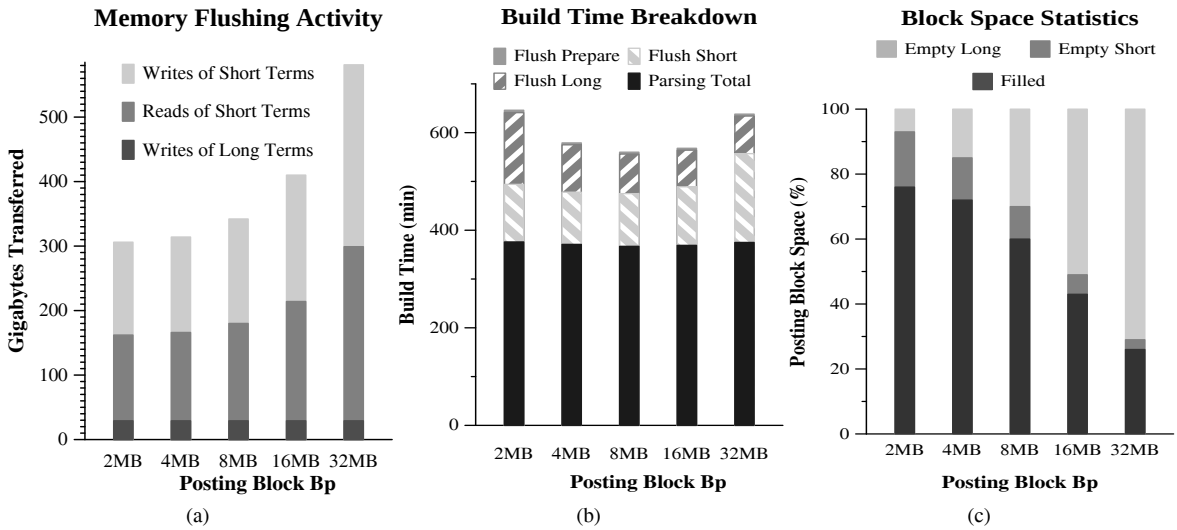

Figure 7: (a) As the size of the posting block increases, the system transfers more data into memory every time it merges postings of short terms into disk. The amount of data that the system writes back to disk grows correspondingly. (b) As the posting block becomes larger, there is a shift of the flush time from long terms to short. Setting $B_p = 8MB$ strikes a good balance between the flush time of the different term types. (c) Empty space in posting blocks increases with their size, mostly due to dedicating separate blocks to each long term.

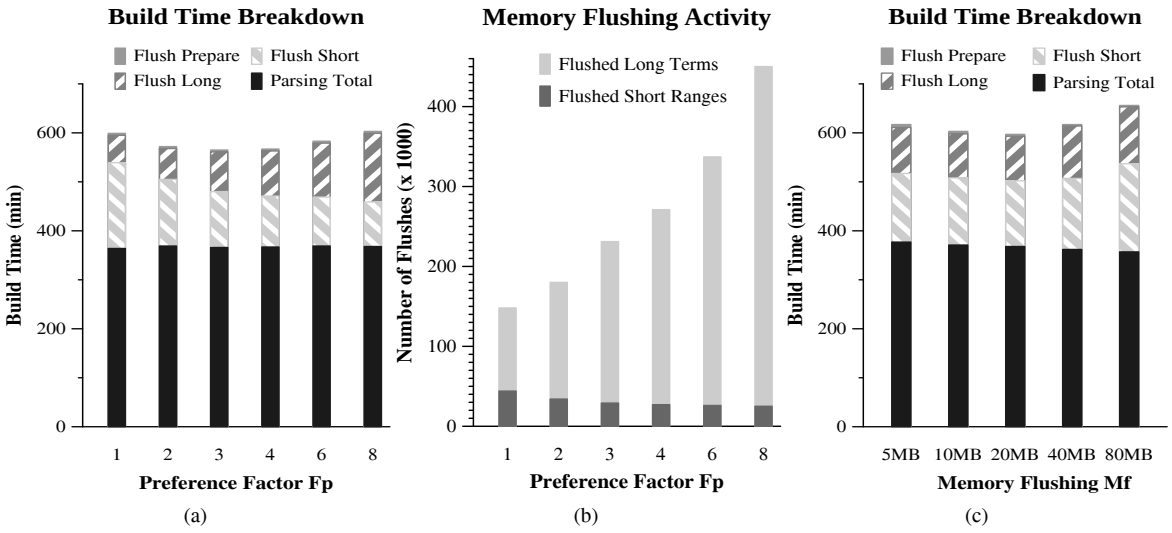

Figure 8: (a) Setting $F_p = 3$ seems to be a reasonable choice that minimizes the total building time. Larger values tend to increase the cost of flushing long postings. (b)The preference factor controls the balance between flushes of short and long terms. As F_p increases, we have a corresponding raise in the number of flushes of long terms. Eventually, disk appends of long postings become too expensive for the system in relation to merges of short postings. (c) The amount of bytes M_f that we flush every time the posting buffer gets full controls the efficiency of the flushes. Essentially, flushing more than a few tens of megabytes increases the related cost of both short and long terms, because the I/O overhead is too high for small amount of postings.

flushes much more often than short. Part of the reason is that the most popular long terms accumulate postings fast and become the first choice to flush into disk. Additionally, the flush cost of long terms is lower as we already explained and this is directly reflected into our choice of F_p. Should the frequency of term occurrence change significantly across the indexed documents, we should possibly adjust the choice of F_p accordingly. In our experience, a fixed small value around $F_p = 3$ keeps the index building time low.

5.3.3 Other parameters

The memory flushing M_f parameter refers to the amount of bytes that we flush to disk every time the posting buffer gets full (Figure 8(c)). In our experiments, we found that setting $M_f = 20MB$, as a small percentage (2%) of the posting buffer (1GB), leads to low building time of the inverted file. If we use values of M_f lower than 20MB, we don't create sufficient free space to accumulate new postings that will flush efficiently the next time memory gets full. The Zipf distribution of term occurrence in documents is known to gather most postings in a few frequent terms [7]. Instead, if we increase M_f to values much larger than 20MB, we end up flushing small amounts of postings that incur high overhead due to either head movement for appends, or reads and writes of old postings for merges.

We experimented with several other parameters that we don't show due to space limitations. In particular, the parameter term threshold T_t refers to the space occupied by the posting list of a term in the system. Its choice affects the categorization of terms into short or long and the subsequent flush method that we use for their postings. We found that the default value $T_t = 1MB$ reaches a good balance in the flush time of long and short terms, although the system behavior is relatively insensitive to values of T_t in the neighborhood of few megabytes.

6 Related Work

In recent research on online search for file systems, Ganger et al. have examined the problem of ranking results using temporal and causal associations between file accesses [19, 20]. Also, Büttcher and Clarke proposed an index structure that works reasonably efficiently without leaking information about file contents to users that lack file access permissions [5]. Additionally, in Linux, they examined event notification that is useful to track file system changes for the needs of full-text search [6]. Today, vendors such as Google, Microsoft and Yahoo! provide search tools for filesystems, but with undisclosed architectural characteristics and performance limitations, especially in demanding environments with multiple online users.

Published literature on text retrieval separates offline index construction from online index maintenance [18, 24]. In comparison to online maintenance, offline index construction is simpler and more efficient because it does not handle document queries until its completion. We already covered in Section 2 several known ways to maintain an online index [7, 13]. Recent research also considers the need to handle both insertions and deletion of documents to the indexed document set [11]. The authors propose a balanced-tree framework that keeps merge operations efficient by only having runs of comparable size being merged. In the present paper, we radically simplify online index maintenance by storing posting lists on fixed-size blocks.

Other previous work has examined the storage of posting lists onto collections of 4KB and 8KB blocks [4, 22, 25]. Although we found their basic results somewhat promising, the authors leave unclear the crucial issue of how block sizes affect index construction and query handling times given the advancement in the technical characteristics of the hard disks over the last two decades. In the present paper, we demonstrate the relevance of fixed-size block storage management to online text retrieval over modern system architectures and datasets.

The problem of external sorting has been studied extensively in the context of algorithms and database systems [10, 12]. Researchers came up with several alternative patterns and heuristics to merge multiple runs on disk, including the balanced merging recently used for maintenance of posting lists. Finally, web search research mainly focused on offline index construction, giving also emphasis on the related issues of how to crawl the web to gather documents and exploit web hyperlinking information for ranking purposes [1, 9]. Similarly, other related work examines the possibility of distributing the indexing process across multiple nodes [2, 14, 21].

7 Conclusions and Future Work

We investigate the problem of inverted-file maintenance in highly dynamic environments. In previous work, there is a known trade-off between index building time and search latency that makes existing systems successful in only one of the two directions. In the present paper we propose a simple yet innovative organization of inverted files that uses fixed-size blocks for their storage on disk. When memory gets full with new postings, we only flush selectively the terms with most postings in memory using the Selective Range Flush method. We implemented the proposed architecture in the Proteus prototype and examined extensively its efficiency using a dataset of half terabyte.

In our experiments, we find that Proteus manages to both build the inverted file and retrieve posting lists at times that closely match the fastest existing practical method for each of the two operations. In our future work, we plan to further investigate alternative cost models for the flushing overhead, and consider using different block sizes for the short and long terms. We also plan to examine workloads with concurrent insertions and deletions of files into the indexed dataset. Another interesting direction that we would like to explore is the automatic adjustment of the configuration parameters according to the characteristics of the indexed files and the underlying hardware.

8 Acknowledgements

In part supported by project INTERSTORE with contract number I2101005 of the INTERREG IIIA Greece-Italy 2000-2006 program.

References

[1] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the web. *ACM Transactions on Internet Technology*, 1(1):2–43, Aug. 2001.

[2] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, mar/apr 2003.

[3] A. Batsakis and R. Burns. Awol: An adaptive write optimizations layer. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 67–80, San Jose, CA, Feb. 2008.

[4] E. W. Brown, J. P. Callan, and W. B. Croft. Fast incremental indexing for full-text information retrieval. In *VLDB Conference*, pages 192–202, Sept. 1994.

[5] S. Büttcher and C. L. A. Clarke. A security model for full-text file system search in multi-user environments. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 169–182, San Francisco, CA, Dec. 2005.

[6] S. Büttcher and C. L. A. Clarke. adding full-text filesystem search to linux. *USENIX ;Login*, 31(3):28–33, June 2006.

[7] S. Büttcher, C. L. A. Clarke, and B. Lushman. Hybrid index maintenance for growing text collections. In *ACM SIGIR*, pages 356–363, Seattle, Washington, USA, Aug. 2006.

[8] D. Cutting and J. Pedersen. Optimizations for dynamic inverted index maintenance. In *ACM SIGIR*, pages 405–411, Brussels, Belgium, Sept. 1990.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008.

[10] G. Graefe. Implementing sorting in database systems. *ACM Computing Surveys*, 38(3):1–37, Sept. 2006.

[11] R. Guo, X. Cheng, H. Xu, and B. Wang. Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In *Conference on Information and Knowledge Management*, pages 751–759, Lisboa, Portugal, Nov. 2007.

[12] D. E. Knuth. *The Art of Computer Programming: Searching and Sorting*, volume 3. Addison Wesley Longman, 2 edition, 1998.

[13] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. In *Conference on Information and Knowledge Management*, pages 776–783, Bremen, Germany, Oct. 2005.

[14] M. Lifantsev and T. Chiueh. I/o-conscious data preparation for large-scale web search engines. In *VLDB Conference*, pages 382–393, Hong Kong, China, Aug. 2002.

[15] National Institute of Standards and Technology. Trec terabyte track. http://trec.nist.gov/data/terabyte.html.

[16] RMIT University. The zettair search engine. http://www.seg.rmit.edu.au/zettair/.

[17] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *USENIX Conference on File and Storage Techonlogies (FAST)*, pages 115–130, San Francisco, CA, Feb. 2003.

[18] B. Ribeiro-Neto, E. S. Moura, M. S. Neubert, and N. Ziviani. Efficient distributed algorithms to build inverted files. In *ACM SIGIR*, pages 105–112, Berkeley, CA, Aug. 1999.

[19] S. Shah, C. A. N. Soules, G. R. Ganger, and B. D. Noble. Using provenance to aid in personal file search. In *USENIX Annual Technical Conference*, pages 171–184, Santa Clara, CA, June 2007.

[20] C. A. N. Soules and G. R. Ganger. Connections: Using context to enhance file search. In *ACM Symposium on Operating System Principles*, pages 119–132, Brighton, UK, Oct. 2005.

[21] C. Tang and S. Dwarkadas. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 211–224, San Francisco, CA, Mar. 2004.

[22] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *ACM SIGMOD Conference*, pages 289–300, Minneapolis, Minnesota, May 1994.

[23] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, 1994.

[24] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), July 2006.

[25] J. Zobel, A. Moffat, and R. Sacks-Davis. Storage management for files of dynamic records. In *Australian Database Conference*, pages 26–38, Brisbane, Australia, 1993.