Low-cost Management of Inverted Files for Online Full-Text Search

Giorgos Margaritis  Stergios V. Anastasiadis

Department of Computer Science
University of Ioannina
Ioannina 45110, GREECE
{gmargari,stergios}@cs.uoi.gr

ABSTRACT

In dynamic environments with frequent content updates, we require online full-text search that scales to large data collections and achieves low search latency. Several recent methods that support fast incremental indexing of documents typically keep on disk multiple partial index structures that they continuously update as new documents are added. However, spreading indexing information across multiple locations on disk tends to considerably decrease the search responsiveness of the system. In the present paper, we take a fresh look at the problem of online full-text search with consideration of the architectural features of modern systems. Selective Range Flush is a greedy method that we introduce to manage the index in the system by using fixed-size blocks to organize the data on disk and dynamically keep low the cost of data transfer between memory and disk. As we experimentally demonstrate with the Proteus prototype implementation that we developed, we retrieve indexing information at latency that matches the lowest achieved by existing methods. Additionally, we reduce the total building cost by 30% in comparison to methods with similar retrieval time.

Categories and Subject Descriptors
E.5 [Files]: Organization/Structure; H.3.2 [Information Storage]: File Organization; H.3.3 [Information Search and Retrieval]: Search Process; H.3.4 [Systems and Software]: Performance evaluation

General Terms
Algorithms, Design, Experimentation, Performance

Keywords
Online index maintenance, search engines, software design, prototype implementation.

1. INTRODUCTION

As the cost of storage space drops and the amount of accumulated digital content grows, automated full-text search for file systems, mail services and electronic commerce environments becomes equally important to the search support in digital libraries and the web [14, 24]. Modern commercial search engines rebuild the entire index periodically by processing tens of petabytes of data every day with the assistance of customized systems infrastructure and data processing tools [8]. They operate sufficiently well for the web because they track changes that occur relatively infrequently and would be almost infeasible to follow continuously due to their enormous volume. On the other hand, search environments that require immediate visibility of newly added documents are currently actively investigated with respect to their index organization and their update algorithms [4, 6, 11, 14, 24]. The main challenge is to achieve fast update and search operation at low cost.

Inverted file is an index that for each term stores a list of pointers to all the documents that contain the term. Each pointer to a document is usually called posting and each list of postings for a particular term is called posting list. The lexicon of the inverted file associates every term that appeared in the dataset into the posting list 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].

In order to index static datasets, one needs to parse documents offline into partial indexes and periodically flush the accumulated postings from memory to disk. Eventually, external sorting can be used to merge the multiple index files into a single file that handles queries for the entire dataset [26]. Online approaches periodically merge the partial indexes on disk to support search operations concurrently with index updates. There is a typical trade-off between index building time and search time. Ideally, each term search should involve a number of steps that only depends on the number of postings rather than the total index size. However, existing methods either take polynomial time to build an index of constant search time or require logarithmic search time for linear building time.

Unlike the latest methods that keep the merging cost low through balanced-tree schemes [6, 11, 15], in the present paper we follow the more straightforward approach of maintaining the postings on disk in fixed-size blocks. Each fixed-size block may contain the postings of a single frequent term or the posting lists of a lexically 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 search cost that only depends on the number of retrieved postings, and index building time that is substantially lower than that of methods with similar retrieval time.

In Figure 1, we depict the number of partial indexes maintained by a linear building approach during the processing of a standard text collection. We notice that there are phases during the building process 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. A typical SATA disk has seek time 8ms, average rotation latency 4ms and nominal transfer time 70 MB/s [23]. Then, in a 12ms time period of the head positioning overhead, the disk can read sequentially about 800 KB. Since the posting list for the majority of the terms occupies up to a few megabytes, the access overhead required to read multiple runs may exceed the corresponding useful transfer time. Ideally, we should retrieve each posting list in a single disk access, without substantial increase of the index building time.

The main contributions of the paper include: (i) grouping of infrequent terms into lexicographic ranges, (ii) partial flushing of both frequent and infrequent terms to disk, (iii) dynamic balance between frequent and infrequent terms flushed to disk, (iv) block-based storage management of all terms on disk. To the best of our knowledge this is the first time that a method simultaneously combines the above features. Previous methods distributed the infrequent terms randomly across different blocks [25] unless they managed them individually [3, 27], only flushed partially the frequent terms from memory to disk [4, 5], and obtained limited benefits from block-based storage management because they only considered small blocks of a few kilobytes [3, 25].

In Section 2 we summarize the previous related work and categorize 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 go over the experimentation environment that we used. In Section 5 we present the results from experiments with alternative system parameters and comparisons with other systems and in Section 6 we outline our conclusions.

2. RELATED WORK

Published literature on text retrieval separates offline index construction from online index maintenance [22, 26]. In comparison to online maintenance, offline index construction is simpler and more efficient because it does not handle document queries until its completion. In particular, 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, 8].

During index building, a system typically parses new documents into posting lists of terms that temporarily maintains in main memory for improved efficiency [7]. 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 [25]. Storage contiguity may improve access efficiency for both query processing and index maintenance, but introduces the need for complex dynamic storage management and frequent or bulky relocations of postings. Alternatively, the disk access efficiency may be improved by partitioning the terms into lexicographic ranges and keeping the postings of different terms from the same range in correspondingly different neighboring blocks on disk [12].

In-place methods build 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 [17, 18]. One can amortize the cost of relocation by preallocating list space for future appends using various criteria [25]. If the system keeps the postings lists non-contiguously on disk, then it avoids relocations but may need multiple seeks during query processing to retrieve a posting list. The merge-based methods merge postings from memory and disk into a single file on disk. The latest related methods amortize the cost by permitting the creation of multiple inverted files on disk and merging them according to specific patterns [15, 16]. Even though in-place index maintenance has linear asymptotic disk cost that is lower than the polynomial cost of merge-based methods, merge-based methods are experimentally shown to use sequential disk transfers and outperform in-place methods [17].

The problem of merging postings lists is very similar to external sorting. We call run a collection of posting lists lexicographically sorted by term. One way to specify the sequence of merging steps is to use a tree representation [13]. The leaves of the tree correspond to the initial runs, while their internal nodes refer to the runs that result from the merging of their descendants. Previous research in database systems has identified the optimization objective of merging to perform as few merge steps and move as few records as possible [9]. Known heuristics always merge the smallest existing runs or mostly use maximal fan-in. One approach previously mentioned but not verified only merges or concatenates fractions of runs [10]. In the present paper we introduce the concept of term range to apply the above idea for first time, and examine its benefit in the storage management of inverted files.

Hybrid methods separate terms into short and long. One early approach hashed short terms accumulated in memory into fixed-size disk regions called buckets. If a bucket filled up, the method categorized the term with the most postings as long and kept it at a separate disk region from that point on [25]. In several recent hybrid methods, the systems use a merge-based approach for the short terms and in-place appends for the long ones [6]. They treat each term as short or long depending on the number of postings that have shown up in total until the current moment, or currently participate in the merging process. The Wumpus prototype implementation of the above methods weakens the storage contiguity requirement
by keeping the postings of each long term into multiple different locations of a file that consists of 64KB blocks [4, 6]. We extend this approach by storing the postings of each long term across large blocks (with default size 8MB) and also lexicographically grouping the short terms into ranges that fit in large blocks. Subsequently, we merge to disk those ranges with a substantial number of accumulated postings in memory. Thus we reduce the data transfers between memory and disk to the cases where we estimate them as efficient.

In Table 1, we summarize the asymptotic cost of known methods to manage inverted files in secondary storage. The No Merge method flushes its postings to a new run on disk without any merging, every time memory gets full. Although impractical to process queries, No Merge provides a baseline for the minimum possible building time. The Immediate Merge method repeatedly merges the postings in memory with the entire inverted file on disk. The Geometric Partitioning or Logarithmic Merge method uses a balanced-postings in memory with the entire inverted file on disk. The Geometric Partitioning method with ≤ p partitions adjusts continuously 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 with ≤ p is also known in the literature as Square Root Merge method [6].

<table>
<thead>
<tr>
<th>Index Maintenance Method</th>
<th>Building Cost</th>
<th>Search Cost</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Merge</td>
<td>Θ(N )</td>
<td>Θ(N/M )</td>
<td>[6, 12, 15, 29]</td>
</tr>
<tr>
<td>Immediate Merge</td>
<td>Θ((N^2/M) )</td>
<td>O(1)</td>
<td>[4, 6, 7, 15]</td>
</tr>
<tr>
<td>Logarithmic Merge (or Geometric Partitioning)</td>
<td>Θ(N log(N/M) )</td>
<td>O(log(N/M))</td>
<td>[6, 15]</td>
</tr>
<tr>
<td>Geometric Partitioning with ≤ p partitions</td>
<td>Θ( N (N/M)^(1/p) )</td>
<td>O(1)</td>
<td>[15]</td>
</tr>
<tr>
<td>Hybrid Immediate Merge</td>
<td>Θ( (N^1.86/M) )</td>
<td>O(1)</td>
<td>[4, 6]</td>
</tr>
<tr>
<td>Hybrid Logarithmic Merge</td>
<td>Θ(N )</td>
<td>O(log(N/M))</td>
<td>[6]</td>
</tr>
</tbody>
</table>

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 Θ(N ), but none of the above methods achieves that. On the other hand, experimental research has shown that building time may also depend on storage system parameters not always included in asymptotic cost estimates [4, 17].

3. THE PROTEUS ARCHITECTURE

Even though the index building process involves the parsing of documents to extract the postings, in the present paper we focus our interest on the management of the inverted file. In our design, we set two objectives: (i) retrieve posting lists at cost that only depends on their length and not the size of the index, and (ii) build the inverted file with minimal disk transfer cost under the above constraint. We consider these objectives consistent with the requirements of a search engine designed for dynamic environments, where new documents are added frequently and users expect to search them shortly after their addition. In order to achieve our goal, we make the following decision designs: (i) Categorize terms into short or long based on the total space of their postings. (ii) Manage short terms in lexicographic ranges and long terms individually. (iii) Flush postings lists to disk selectively by amount of postings in memory. (iv) Allocate disk storage space in fixed-size blocks.

As we add new documents to a collection, we accumulate term postings in the available memory space and eventually transfer them to disk. We use a lexicon to keep track of the individual terms and associate them with postings lists in memory or on disk. Through experimentation we verified that in-place management of terms with few postings bears significant overhead for appending them to disk. Similarly, merge-based management of terms with lots of postings incurs significant cost for merging them to disk. Thus, we consider a term short or long, respectively, depending on whether its current total posting space is less or exceeds the preconfigured system parameter term threshold T_l. For the sake of conciseness, we use the name long or short not only for terms but also for their corresponding postings, posting lists or ranges.

Initially all terms are short. When the total size of postings 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 remove a long term from memory, we simply append its postings to the existing list on disk using the in-place approach. Since individual short terms are infrequent, we group them into lexicographically ordered ranges. We move their postings to disk by merging them in memory with older postings on 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 as needed. A range of short terms takes its own block on disk to store the postings lists lexicographically ordered by term. When the posting block of a short range gets full, we split the range across two postings blocks. 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 exclusive block. From that point on, we no longer keep any postings of the long term in the corresponding short range that previously contained that term.

Finally, 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 need to get all the postings of the long term in the corresponding short range that previously contained that term.

### 3.1 Selective Range Flush

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 posting memory gets full, we need some policy to determine which particular postings to transfer to disk and make space for new ones [4]. In order to minimize the total index building time, we need to minimize the total number of disk operations and maximize their efficiency. In fact, the overall building cost depends not only on the efficiency of each individual transfer from memory to disk but also on the total amount of data brought from disk to memory, when we apply the merge-based method. For long postings, we use the in-place method and prefer to have only few large appends to disk. For short postings, we apply the merge-based method, and want to minimize the number and maximize the size of the transfers to disk. Thus, we group short terms into lexicographic ranges and keep their postings in memory as long as possible to avoid the repetitive disk reads and writes involved during merges.

Short-term flushing seems similar to the page replacement problem in the sense that good candidates for flushing are those ranges which won’t get new postings in the future. One main difference from paging is that we accumulate new postings in memory without needing the older postings until we flush them to disk. Additionally, we need to balance flushes of short and long terms according to their cost. Long postings incur an one-time cost for flushing, while short ranges require repetitive reads from disk before flushing new postings. Furthermore, writes occur asynchronously and may incur delays during subsequent reads - of new documents, for example, that we process next - due to the need for cleaning dirty buffers from the page cache [2].

For this unique problem of disk transfer scheduling, we came up with a new policy called Selective Range Flush. We show the pseudocode of the method in Algorithm 1. Every time our posting buffer gets full, we sort the posting lists according to the space they occupy in memory (Line 4). We compare the byte size of the largest long list against the byte size of the largest short range in memory (Line 10). We pick for flushing next the largest long list, unless it is $F_p$ times smaller than the largest short range (Lines 11-14). In the latter case we flush the short range instead (Lines 16-23). We repeat the above process until we flush to disk Flashed Memory ($M_f$) bytes of postings. Our approach generalizes in several ways the partial flushing introduced previously [4]. We avoid inefficient flushes of long terms by only flushing $M_f$ bytes instead of the entire posting memory. In addition to long terms, we selectively flush short ranges, when their size is sufficiently large.

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 repetitive transfers.
between memory and disk of a short range. We postpone the
flushing of the largest short range until the size of the largest long list be-
comes \( F_p \) times smaller. Then the flushing overhead of the long list
takes too much for the amount of data flushed. At the same time we
prefer to keep the short postings in memory and avoid their merging
into disk. The parameter \( F_p \) may depend on the performance char-
acteristics of the system architecture, such as the head-movement
overhead, the sequential throughput of the disk, and the statistics
of the indexed document collection (e.g. 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 for
the dataset that we used.

Our algorithm behaves greedily because it only considers the
space that a long term or short range currently occupies in memory.
We use the parameter \( F_p \) to approximate the cost of flushing a short
range relative to a long term. Similarly, we use the occupied space
of postings in the indexed dataset to categorize the terms into short
or long. We experimented extensively with alternative approaches
that estimate the posting flushing throughput or choose to aggres-
sively flush terms up to a minimum size of posting lists. The simple
approach of Selective Range Flush to flush few tens of megabytes
from the largest lists in memory gave the best performance overall.

As a baseline for search efficiency, we provide a version of
the SRF algorithm with the long lists contiguousously stored on disk. Each
long list starts as a single block with the default size. As the size of
the list exceeds the current capacity of the block, we reallo-
cate a block with twice the size and relocate the postings of the
list to the new block. In our figures, we depict this implement-
ment as SRF/CNT-Proteus in order to separate it from the origi-
nal SRF/PFG-Proteus version, where each long list is fragmented
across multiple blocks of the default size.

### 3.2 Prototype Implementation

We built a prototype implementation of our own inverted-file
management in the Proteus architecture (Figure 2(a)). We retained
the parsing and search components of the open-source Zettair search
engine (version 0.9.3) [20]. The focused and modular design of
Zettair made it a good choice for our needs. In our study, we mainly
investigate the I/O aspects of search and make no engineering effort
to optimize processing-related tasks beyond reasonable choices. In
particular, we use the standard memory management of libc, al-
though a customization to the needs of Selective Range Flush could
reduce the related processing cost.

We maintain in memory a hash table that we call **hashtable**,
where we store the posting lists that we extract from the parsed
documents (Fig. 2(b)). In the posting list of each term, the doc-
ument 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 [26]. Overall, compression reduces considerably the space requirements of postings in memory and on
disk.

We categorize the terms into 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 (Figure 2(c)). We use binary search to look for particular
terms in the termtable. Organization of the termtable as an array of
 pointers to descriptors makes relatively inexpensive the shifting of
existing terms and the insertion of new ones at arbitrary positions.
Each descriptor contains the term name, the size of the postings
in memory, a pointer to the last block, the amount of free space
at the last block on disk, and a linked list of nodes that we call
**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.

In memory, we keep for the short terms a sorted array that we
call **rangetable** (Figure 2(d)). Each rangetable entry corresponds to
a range of terms whose postings are stored in a single block. The
entry contains the size of the postings, the names of the first
and last term in the range, and also a pointer to the block and the
amount of the free space at this block. In a partial index that we
call **search bucket**, we maintain the name and location of the term
that occurs every 128KB along each posting block. The search
bucket allows us to approximately retrieve only the required part
of the posting block that may contain a term. From our experi-
ence, any more detailed index to each posting block may increase
significantly the maintenance overhead of the rangetable.

Initially, the termtable is empty and the rangetable contains a sin-
gle entry for all terms. When the posting memory fills up, we 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 Selec-
tive Range Flush algorithm. For each range we maintain a linked
list of all the associated terms that have non-empty posting lists in
memory. Before we flush a range, we retrieve its postings already
stored on disk and merge them with those accumulated in memory.
If the range that emerges from merging exceeds the capacity of a
posting block, we split the range into multiple half-filled blocks.
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

---

**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. (c) 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. (d) 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.
for the last one. After the flush, we update the tables in memory to accurately reflect the postings currently available in memory.

4. EXPERIMENTATION ENVIRONMENT

For our experiments we used servers running the Debian distribution of Linux kernel version 2.6.18. Each server is equipped with one quad-core x86 2.33GHz processor, 3GB RAM, one linked gigabit ethernet port, and two 7200RPM SATA disks of 500GB each. The disk vendors specify buffer size 16MB, seek time 8.9ms, and sustained transfer rate 72MB/s. We store the document collection and the generated index on the two disks separately. We 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 standard dataset from the TREC Terabyte Track [19]. Unless otherwise specified, we set the parameter values of Proteus $B_P = 8MB, M_P = 1GB, M_f = 20MB, F_c = 3$ and $T_c = 1MB$. In the case of GOV2, the hashtable that we maintain in main memory occupies 4MB, the termtable and rangetable together 0.5MB, the block lists of the long terms 0.12MB, and the range buckets of the short terms reserve 36.5MB. In total, our auxiliary structures in memory require less than 50MB.

In order to keep our comparative measurements consistent, we do all the related experiments on a single server and observe negligible (~1%) measurement variations across different repetitions of the same experiment on one machine. Proteus generates index size of 70GB which is comparable to the 64GB created by the Wumpus system [6]. Even though the two systems manage the storage space differently, we verified that the posting list of the same term occupies space within a few percent of each other across the two systems.

5. PERFORMANCE EVALUATION

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. We consider a subset of index maintenance methods that are known to cover a wide range of tradeoffs between building and search efficiency (Table 1). We experimented with the above methods as implemented in the Wumpus system with activated partial flushing and automated threshold adjustment [4, 6]. The original Zettair implementation builds a lexicon for term searches at the end of the index building; this makes it offline and we don’t examine it any further here. In Proteus, we maintain a lexicon that allows us to retrieve all the posting lists from memory and disk during the building process.

To keep Wumpus and Proteus functionally comparable, we activate full stemming across both systems when we compare them with each other (Porter’s option [21]). Full stemming reduces terms to their root form by stripping suffixes. Thus, it retrieves relevant documents with words that do not match exactly those searched. In Proteus we use an unoptimized version of Porter’s algorithm as implemented in Zettair. This makes the reported parsing time of the Proteus index building a pessimistic estimate that may be further improved with sufficient engineering effort. When we examine the sensitivity of Proteus to the configuration parameters, we use a less aggressive option called light stemming instead, which is the default setting in the original Zettair parsing implementation.
Table 3: Selective Range Flush/FRG of Proteus stores the posting list of each short term contiguously in part of a posting block; it exclusively occupies multiple posting blocks for the posting list of each long term. 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. 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.

5.1 Reading a Posting List

We measure the time to retrieve the posting list of different unique terms in the index of the entire GOV2 dataset. In our experiments, we use the terms (25673 short and 5121 long terms on average across the policies) contained in the Efficiency Topics query set of the TREC 2005 Terabyte Track [19]. We ensure that our measurements include the delay of the disk transfers by flushing the memory cache before we retrieve the posting list of a term.

In Sections 2 and 3, we already explained the policies that we examine. We note that SRF/CNT keeps the postings of each term contiguously in either the in-place or the merge-based part of the index but not both. SRF/FRG is similar to SRF/CNT with the only difference that it fragments the postings of the long terms across multiple posting blocks. Hybrid Immediate Merge (HIM) has one merge-based run and one in-place run. It keeps the postings of each short term in the merge-based run, and the postings of each long term in both the runs. Hybrid Square Root Merge (HSM) has one in-place run and number of merge-based runs that varies between one and two depending on the index size. At the end of GOV2 processing, HSM ends up having one merge-based run. Thus, the postings of each short term are stored in one run, and the postings of each long term in up to two runs. Hybrid Logarithmic Merge (HLM) has one in-place run and number of merge-based runs that logarithmically depends on the index size. During the processing of GOV2, the number of merge-based runs varies between one and six, and at the end it becomes four. Thus in our measurements of HLM, each short term has postings in four runs, while each long term has postings in five runs. We note that the in-place run of the Wumpus prototype consists of 64KB segments. As a result, the postings of each long term in Wumpus are fragmented across multiple segments which are not contiguously stored.

In Figure 3(a) we measure the average retrieval time for terms with postings up to 1MB in the system. We notice that HIM achieves the lowest time. HSM is similar to HIM because it happens to have single-run merge-based index at the end of GOV2. However, with experiments that we also did (not shown here) for the case of two-run merge-based index, HSM requires about 50% more time for terms with total postings up to 1MB. Essentially, the search behavior of HIM varies depending on the status of the merging process at which we do the experiment. SRF/CNT is 10% higher than HIM and SRF/FRG 13.5% higher. We attribute this difference between SRF and HIM to seek overheads that arise because we spread the postings of the short terms across a disk region of 70GB index. Instead, we found that Wumpus keeps the merge-based index of HIM in a smaller region of 10GB. Finally, HLM requires about four times longer time than the other policies, due to the four-run merge-based index. The discrepancy would be even higher in the case of a six-run merge-based index (as shown in Figure 1).

In Figure 3(b), we measured the retrieval time for terms with postings of more than 1MB. We observe that SRF requires retrieval time about two to three times lower in comparison to the other policies. We attribute this difference to the mechanism of SRF and the default size of 8MB posting block in Proteus. In comparison to short terms, the retrieval time of long terms may be less crucial depending on the type of the search operator. For example, in conjunction queries only a subset of the posting list needs to be brought to memory. We also note that SRF/CNT achieves 13% lower retrieval time in comparison to SRF/FRG. We can explain this discrepancy if we consider the contiguity of the postings in SRF/CNT that reduces disk access overheads during the retrieval of a long term.

In Figure 3(c), we group the terms into intervals according to their posting retrieval time. We choose the width of the interval equal to 5ms. Thus, the y axis shows the number of terms whose retrieval time lies within the same interval. We see that HIM has its peak just one interval left of SRF. HLM varies its runs in the merge-based part between 1 and 6. In our experiments HLM ended up with four runs resulting into histogram curve distinctly on the right of the other policies. HSM has merge-based index with one run at the end of GOV2, therefore it has retrieval time similar to HIM. We also examined the case (not shown) of HSM with two-run merge-based index that is created if we stop the processing of GOV2 a few gigabytes short of the entire dataset. Then, the curve of HIM lies halfway between SRF and HLM.

In Table 3, we can see the exact number of runs accessed during a search with the HLM and SRF/FRG policies. For example, HLM maintains short postings for term anim across three runs and long postings across 89 segments of 64KB. Instead, SRF/FRG categorizes anim as long term and keeps all its postings in a single block. As a result, the total retrieval time is 99ms for SRG/FRG policy and 238ms for the HLM. Overall, HIM and SRF achieve the lowest retrieval time for posting lists of short terms, while HIM and HLM may take longer due to the multiple runs that they possibly maintain. When we retrieve the list of a short term we need to access all the runs in the merged-based part as a result of the manner that the index is constructed. This is even needed in the case that there is no occurrence of the searched term in the indexed dataset.
Algorithm over Proteus we depict the cases of fragmented long lists and free buffer space for newly read documents. For the SRF algorithm over Proteus we depict the cases of fragmented long lists (SRF/CNT) and contiguous long lists (SRF/FRG). We observe that SRF/FRG takes 30% less time to build than HIM.

5.2 Building the Inverted File

In Figure 4, we break down the index building time into the Parsing part to read and parse the dataset into postings, and the Flushing to gather and transfer the postings to disk. The different policies cover a wide range of building times between 325 min for No Merge and 730 min for HIM. In comparison to HIM, the total building time is 30% lower for SRF/FRG and 28% for SRF/CNT. We also notice that SRF only needs 14 additional minutes of building time for postings relocations that keep the long lists contiguous on disk (CNT versus FRG). We found that the corresponding increase in the index size due to additional empty space within the posting blocks of SRF/CNT is 8%.

The other two hybrid policies of Wumpus that we examine, HSM and HLM respectively, achieve 37% and 46% reduction in comparison to HIM. As we already explained in the previous section, HSM keeps the postings of each term across a number of runs that varies between 1 and 3 (one in-place run and two merge-based runs) during the index construction. Furthermore, this number varies between 1 and 7 in the case of HLM during the processing of GOV2. The retrieval time of the short terms may increase significantly due to the additional empty space in blocks of long terms, leading to low index building time.

Figure 4: We break down the index building time into document parsing and postings flushing parts across different maintenance policies implemented in Wumpus and Proteus. Parsing includes the time to clean dirty pages of the index as needed to free buffer space for newly read documents. For the SRF algorithm over Proteus we depict the cases of fragmented long lists (SRF/CNT) and contiguous long lists (SRF/FRG). We observe that SRF/FRG takes 30% less time to build than HIM.

5.2.1 Posting Block Size $B_p$

The size of the posting block $B_p$ is a critical configuration parameter that specifies the amount of postings contained in each range of short terms. Therefore, it directly affects the amount of bytes transferred while we merge the short postings of memory and disk. Figure 5(a) demonstrates this effect through the amount of bytes read and written during flushes. The less data that we read during short term flushing, the lower total building time we achieve. When we append long postings to disk there is almost no read involved. As a result, the block size does not change the amount of transferred bytes but it may affect the required number of disk transfers. Overall, Figure 5(b) shows that an increasing block size reduces the total 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 5(c) we break down the inverted file into a part that contains postings, another that is empty space in blocks of short terms, and a third that is empty space in blocks of long terms. The large block size tends to increase substantially the empty space in blocks of long terms, leading to larger inverted file. With our default choice of $B_p = 8MB$ we get an index of 70GB, where about 41GB are actual data and the rest is empty space. However, large posting blocks lead to low disk access overhead when we retrieve long terms. For example, for a disk with 12ms access overhead and 70MB/s sequential throughput, a block size of $B_p = 8MB$ is anticipated to keep access overhead less than 10%.

5.2.2 Preference Factor $F_p$

The preference factor specifies how aggressively we flush long postings relatively to short ones. Term flushes cannot be done efficiently unless a term has a sufficient number of postings in memory. Otherwise, there is a high head movement cost for appending or merging postings. Figure 6(a) makes it clear that we spend more time for short flushes if we assume equal cost for short and long flushes with $F_p = 1$. Instead, if we increase $F_p$ beyond 8, the long flushes dominate the I/O cost of index building. Reads of postings from disk are asynchronous and directly affect the total building time, while postings writes are asynchronous and only partially account for the building time.

In Figure 6(b) we also see that long terms are involved in flushes much more often than short ones. We can explain this behavior by the fact that the most popular long terms accumulate postings fast and become the first choice to flush into disk, while the flushing cost of long terms is relatively lower. Should the frequency of term occurrence change significantly across the indexed documents, we may have to adjust the choice of $F_p$ accordingly. In our experience, a fixed small value around $F_p = 3$ keeps the index building time low throughout the processing of the dataset.

5.2.3 Posting Memory $M_p$

The Posting Memory $M_p$ specifies the memory space that we reserve for temporary storage of postings. In Figure 6(c), we notice that as we increase $M_p$ from 0.5GB to 1GB, the build time drops substantially. Further increase to 1.5GB reduces slightly the build time, while further increase to 2GB keeps build time the same. In the rest of the experiments, we chose as default value $M_p = 1GB$. The postings later during parsing, when the reads have to clean dirty buffers and free space in order to fit the new documents in memory before processing them. Such system behavior has been already documented in the literature [2].

In the rest of the current section, we examine the sensitivity of the index building time across the configuration parameters of the system using the SRF/FRG policy.
We investigate the problem of online inverted-file maintenance. From previous work, there is a known trade-off between index building time and search latency that makes existing systems most 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 the memory gets full with new postings, we only flush selectively the terms with most postings in memory using the Selective Range Flush method. We implement the proposed method in the Proteus prototype and examine extensively its efficiency using a standard dataset of half-terabyte. We find that Selective Range Flush retrieves the posting lists of infrequent terms at amount of time that matches one of the fastest known methods, the Hybrid Immediate Merge (with partial flushing and automatic threshold adjustment). The corresponding retrieval time of Selective Range Flush for frequent terms is several factors lower in comparison to alternative methods. Furthermore, the index building time of Selective Range Flush is 30% lower in comparison to Hybrid Immediate Merge. We examine the sensitivity of our method to various configuration parameters of the system through extensive experimentation.

In our future work, we plan to further investigate in the context of the Proteus architecture alternative cost models for the flushing overhead, and consider using different block sizes for the short and long terms. Additional directions for exploration include the analytical study of Selective Range Flush, and the automatic adjustment of its configuration parameters according to the characteristics of the indexed files and the underlying hardware.

7. ACKNOWLEDGMENTS

In part supported by project Interstore of the Interreg IIIA Greece-Italy EU Community Initiative program under contract no. I2101005.

8. REFERENCES

Figure 6: (a) Setting $F_p = 3$ is a reasonable choice that minimizes the total building time. (b) As the preference factor $F_p$ becomes higher, the flushes of long terms increase and become too expensive in comparison to merges of short postings. (c) We observe diminishing reduction in build time as we increase the posting memory size from 0.5GB to 2GB. (d) Flushing more than a few tens of megabytes ($M_f$) at a time incurs high cost due to the overhead of small I/O operations for terms or ranges.