Building a Distributed Full-Text Index for the Web

Sergey Melnik Sriram Raghavan Beverly Yang Hector Garcia-Molina
{melnik, rsram, byang, hector}@cs.stanford.edu
Computer Science Department, Stanford University

Abstract

We identify crucial design issues in building a distributed inverted index for a large collection of web pages. We introduce a novel pipelining technique for structuring the core index-building system that substantially reduces the index construction time. We also propose a storage scheme for creating and managing inverted files using an embedded database system. We propose and compare different strategies for addressing various issues relevant to distributed index construction. Finally, we present performance results from experiments on a testbed distributed indexing system that we have implemented.

1 Introduction

A number of different access methods have been developed by the information retrieval community to support efficient search and retrieval over text document collections. Examples of such access methods include *suffix arrays* [13], *inverted files or inverted indexes* [22, 29], and *signature files* [6]. Inverted files have traditionally been the index structure of choice on the Web. They perform well for simple and short Web search queries that typically consist of conjunctions of search terms. Commercial search engines use custom network architectures and high-performance hardware to achieve sub-second query response times using such inverted indexes.[1]

An inverted index over a collection of Web pages consists of a set of *inverted lists*, one for each word (or *index term*) occurring in that collection. The inverted list for a term is a sorted list of *locations* where the term appears in the collection. A location consists of a page identifier and the position of the term within the page. Sometimes, when it is not necessary to track each occurrence of a term within a page, a location will include just a page identifier (optionally accompanied by a count of the number of occurrences of that term in the page). Given an index term w, and a corresponding location l, we will refer to the pair (w, l) as a *posting* for w.

Conceptually, building an inverted index over a collection of Web pages involves processing each page to extract postings, sorting the postings first on index terms and then on locations, and finally writing out the sorted postings as a collection of inverted lists on disk. For application scenarios in which the collection is small and static, and where indexing is a rare activity, extensive optimization of the index-building process is not as critical as optimizing run-time query processing and retrieval. However, two factors make the development of Web-scale inverted index building techniques a challenging and important problem:

Scale and Growth rate The size and rate of growth of the Web [12, 28] require the indexing architecture and implementation to be highly scalable, much beyond the requirements of traditional

[1] Despite the fact that the semantic content of the link structure on the Web is being utilized to produce high-quality search results, text-based retrieval continues to be the primary method for identifying the relevant pages for a search query. In most commercial search engines, a combination text and link-based methods are employed.

IR systems. As a measure of comparison, the 40 million page WebBase repository [10] represents only about 4% of the *publicly indexable web* but is already larger than the 100 GB *very large TREC-7 collection* [9], the benchmark for large IR systems.

Rate of change Since the content on the Web changes extremely rapidly [5], there is a need to periodically crawl the Web and rebuild the inverted index. Many techniques for incremental update of inverted indexes reportedly perform poorly when confronted with large whole-scale changes commonly observed between successive crawls of the Web. Others require additional work at query time to handle new and modified pages. Hence, incremental updates to indexes are usually used only as a short-term measure and the index is periodically rebuilt [15] to maintain retrieval efficiency.

We have implemented a testbed system to build inverted files on a cluster of *nodes* (workstations in our case). Using this testbed, we make the following contributions in this paper:

- We propose the technique of constructing a *software pipeline* on each indexing node to enhance performance through intra-node parallelism.
- We argue that the use of an embedded database system (such as *The Berkeley Database* [19]) for storing inverted files has a number of important advantages. We propose an appropriate format for inverted files that makes optimal use of the features of such a database system.
- We present a qualitative discussion on two different strategies for interleaving page distribution and index construction.
- Any distributed system for building inverted indexes needs to address the issue of collecting global statistics (e.g., reciprocal of the total number of documents in which a word occurs, the so-called *inverse document frequency IDF*). We examine different strategies for collecting such statistics from a distributed collection.
- For each of the above issues, wherever appropriate, we present experiments and performance studies to compare the different alternatives.

We emphasize that the focus of this paper is on the actual process of building an inverted index and not on using this index to process search queries. As a result, we do not address issues such as ranking functions, relevance feedback [22, 29], and distributed query processing [11, 24]. Furthermore, our aim is not to construct a feature-rich inverted index that supports phrase searching, approximate matching, and proximity queries [22, 29]. Rather, our studies are more fundamental, and apply to the construction of any distributed inverted index that supports some or all of these features.

The rest of this paper is organized as follows. In Section 2, we describe the architecture of our testbed, identify the various components of our system, and present an overview of the index building process. Our discussion of the key design issues is organized into two sections. Section 3 deals with the design of the core indexing engine. The issues and techniques that we present in that section are common to any index-building system, irrespective of whether the system operates on a single node or is distributed. Section 4 deals with issues that arise when inverted files are built using a distributed architecture. We discuss related work in Section 5 and conclude in Section 6.

2 Testbed Architecture

Our testbed system for building inverted indexes operates on a distributed shared-nothing architecture consisting of a collection of nodes connected by a local area network. We identify three types of nodes in the system (Figure 1):

Distributors These nodes store, on their local disks, the collection of web pages to be indexed.

Indexers These nodes execute the core of the index building engine.

Query servers Each of these nodes stores a portion of the final inverted index. Depending on the organization of the index files, some or all of the query servers may be involved in answering a search query.

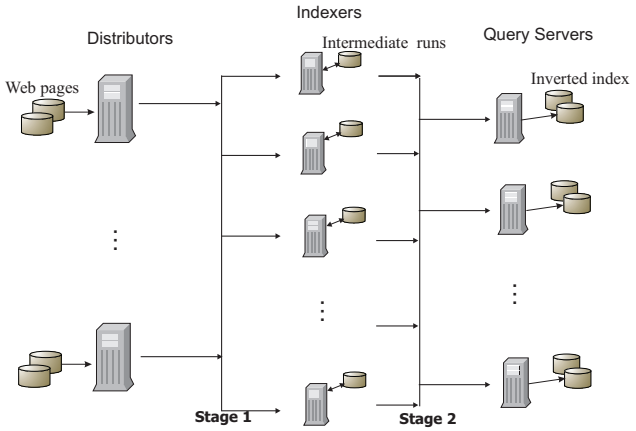

Figure 1: Testbed architecture

The input to the indexing system is a collection of web pages stored on the distributor nodes. The output is a set of (inverted file,*lexicon*) pairs, one on each query server. The inverted file on a query server covers a subset of the documents in the collection. The *lexicon* lists all the index terms in the corresponding inverted file along with their associated statistics.

The inverted index is built in two stages. In the first stage, each distributor node runs a *distributor process* that disseminates the collection of web pages to the indexers. Each indexer receives a mutually disjoint subset of pages and their associated identifiers. The indexers parse and extract postings from the pages, sort the postings in memory, and flush them to intermediate structures on disk.

In the second stage, these intermediate structures are merged together to create one or more inverted files and their associated lexicons. An (inverted file, lexicon) pair is generated by merging a subset of the sorted runs. Each (inverted file, lexicon) pair is transferred to one or more query servers depending on the degree of replication. In this paper, for simplicity, we assume that each indexer builds only one such pair.

Distributed inverted index organization In a distributed environment, there are two basic strategies for distributing the inverted index over a collection of query servers [14, 20, 23]. One strategy is to partition the document collection so that each query server is responsible for a disjoint subset of documents in the collection (called *local inverted files* in [20]). The other option is to partition based on the index terms so that each query server stores inverted lists only for a subset of the index terms in the collection (called *global inverted files* in [20]). Performance studies described in [23] indicate that the local inverted file organization uses system resources effectively and provides good query throughput in most cases. This organization is particularly effective for processing search queries that are Boolean conjunctions of search terms (see B); the most common type of queries on the Web. For the above reasons, our testbed employs the local inverted file organization.

Property	Value
Average number of words per page	438
Average number of *distinct* words per page	171
Average size of each page (as HTML)	8650
Average size of each page after removing HTML tags[a]	2815
Average size of a word in the vocabulary	8

[a]Multiple consecutive whitespace characters were also replaced by a single whitespace character in the process.

Table 1: Properties of the WebBase collection

Testbed environment Our indexing testbed uses a large repository of web pages provided by the WebBase project [10] as the test corpus for the performance experiments. The storage manager of the WebBase system receives pages from the web crawler [5] and populates the distributor nodes. The indexers and the query servers are single processor PC's with 350-500 MHz processors, 300-500 MB of main memory, and equipped with multiple IDE disks. The distributor nodes are dual-processor machines with SCSI disks housing the repository. All the machines are interconnected by a 100 Mbps Ethernet LAN network.

2.1 The WebBase collection

To study some properties of web pages that are relevant to text indexing, we analyzed 5 samples, of 100000 pages each, from different portions of the WebBase repository. Table 1 summarizes some of the salient features of the WebBase collection.

We also used these 5 samples to study the rate at which the *vocabulary* (number of distinct words encountered) grows as more pages are processed. Figure 2 plots the growth of the vocabulary, both as a function of the number of pages processed by the indexer, and as a function of the total size of HTML text processed. We were able to regression fit simple *power-law* curves (of the form $y = ax^b$) to the vocabulary growth data collected from the five samples. It is interesting to note that the exponent b of the power-law is almost the same for both cases though the multiplicative factor a is different. We also note that the exponent for our collection of Web pages seems to be larger (implying that the vocabulary grows faster) than exponents normally observed in other text collections [21].

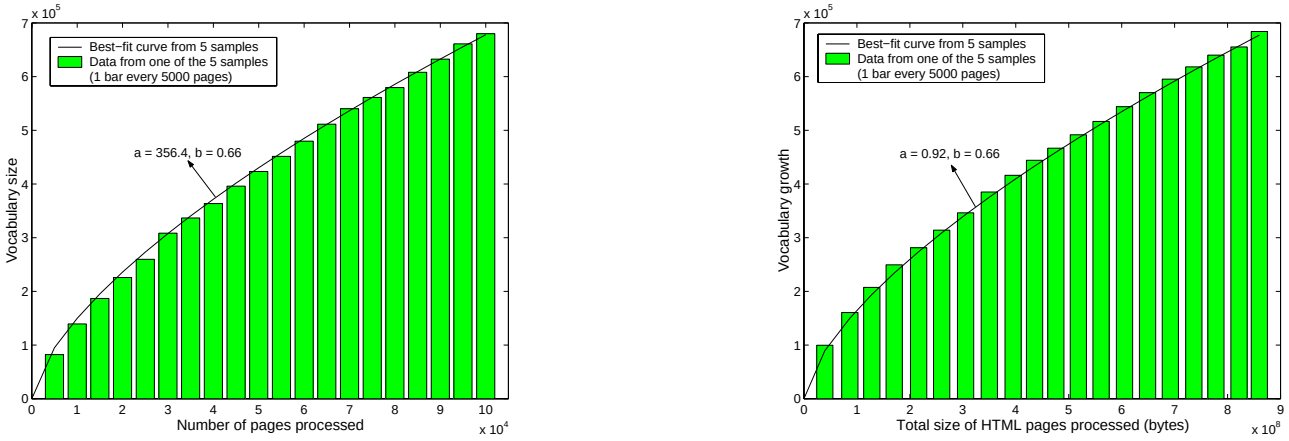

Figure 2: Vocabulary growth

3 Design of the Indexer

The core of our indexing system is the *index-builder* process that executes on each indexer. In this section, we describe two novel aspects of the design of this process:

- Introducing parallelism by structuring the implementation of the index-builder as a *software pipeline* (Section 3.1)
- Using an embedded database system to store the inverted files generated by the process (Section 3.2)

Note that these two techniques are applicable to any inverted index building system, independent of whether the implementation architecture is a single node or a distributed collection of nodes.

In our testbed, the input to the index-builder process is a sequence of web pages and their associated identifiers.[2] The output of the index-builder is a set of *sorted runs*. Each sorted run contains postings extracted from a subset of the pages received by the index-builder.

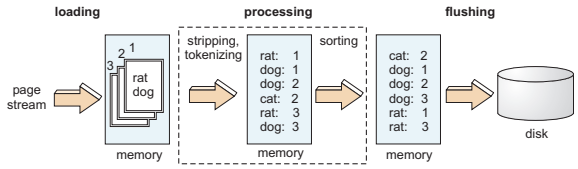

Figure 3: Logical phases of constructing a sorted run

The process of generating these sorted runs can logically be split into three phases as illustrated in Figure 3. We refer to these phases as *loading*, *processing*, and *flushing*. During the loading phase, some number of pages are read from the input stream and stored in memory. The processing phase involves two steps. First, the pages are parsed to remove HTML tagging, tokenized into individual terms, and stored as a set of term-location pairs (or *postings*) in a memory buffer. In the second step, the postings are sorted in-place, first by term, and then by location. During the flushing phase, the sorted postings in the memory buffer are saved on disk as a sorted run. These three phases are executed repeatedly until the entire input stream of pages has been consumed.

3.1 Pipelined index construction

Loading, processing and flushing tend to use disjoint sets of system resources. Processing is obviously CPU-intensive, whereas flushing primarily exerts secondary storage, and loading can be done directly from the network, tape, or a separate disk. Therefore, indexing performance can be improved by executing these three phases concurrently. Since the execution order of loading, processing and flushing is fixed, these three phases together form a *software pipeline*.

Figure 4 illustrates the benefits of pipelined parallelism during index construction. The figure shows a portion of an indexing process that uses three concurrent threads, operates on three reusable memory buffers, and generates six sorted runs on disk.

The goal of our pipelining technique is to design an execution schedule for the different indexing phases that will result in minimal overall running time (also called *makespan* in the scheduling literature). Our problem differs from a typical *job scheduling* problem [4] in that we can vary the sizes of the incoming *jobs*, i.e., in every loading phase we can choose the number of pages to load. In the

[2]The URLs are replaced by numeric identifiers for compactness

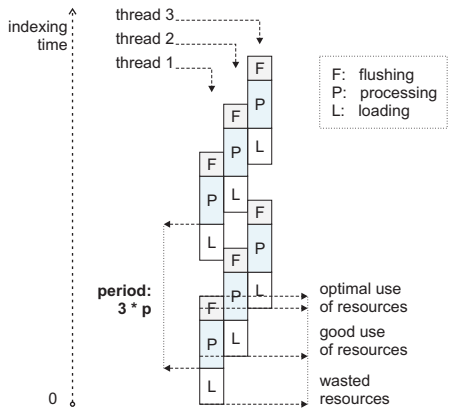

Figure 4: Multi-threaded execution of indexing phases

rest of this section, we describe how we make effective use of this flexibility. First, we derive, under certain simplifying assumptions, the characteristics of an *optimal pipeline schedule* and determine the theoretical speedup achievable through pipelining. Next, we describe experiments that illustrate how observed performance gains differ from the theoretical predictions.

3.1.1 Pipeline design

Let us consider an indexer node that has one resource of each type - a single CPU, a single disk, and a single network connection over which to receive the pages. How should we design the pipeline shown in Figure 3 to minimize index construction time?

First, notice that executing concurrent phases of the same kind, such as two disk flushes, is futile. Since we have only one resource of each type, we can at best achieve the same running time as when executing the phases sequentially. In practice, the overhead for thread switching would tend to make concurrent execution perform worse.

Consider an index-builder that uses N executions of the pipeline to process the entire collection of pages and generate N sorted runs. By an *execution of the pipeline*, we refer to the sequence of three phases - loading, processing, and flushing - that transform some set of pages into a sorted run. Let B_i, $i = 1 \ldots N$, be the buffer sizes used during these N executions. The sum $\sum_{i=1}^{N} B_i = B_{total}$ is fixed for a given amount of text input. Our aim is to come up with a way of choosing the B_i values so as to minimize the overall running time.

Now, loading and flushing take time linear in the size of the buffer. Processing time has a linear component (representing time for removing HTML and tokenizing) and a linear-logarithmic component (representing sorting time). Let $l_i = \lambda B_i$, $f_i = \varphi B_i$, and $p_i = \delta B_i + \sigma B_i \log B_i$ represent the durations of the loading, flushing, and processing phases for the i^{th} execution of the pipeline.[3] For large N, the overall indexing time is determined by the scarcest resource (the CPU, in Figure 4) and can be approximated by $T_p = \max\{\sum_{i=1}^{N} l_i, \sum_{i=1}^{N} p_i, \sum_{i=1}^{N} f_i\}$.

It is easy to show (see A) that T_p is minimized when all N pipeline executions use the same buffer size B, where $B = B_1 \ldots = B_N = \frac{B_{total}}{N}$. Let $l = \lambda B$, $f = \varphi B$, and $p = \delta B + \sigma B \log B$ be the durations of the loading, processing, and flushing phases respectively. We must choose a value of B that maximizes the speedup gained through pipelining.

We calculate speedup as follows. Pipelined execution takes time $T_p = N \max(l, p, f)$ ($6p$ in Figure 4) and uses 3 buffers, each of size B. In comparison, sequential execution using a single buffer of size $3B$

[3] $\lambda = \lambda_1 \lambda_2$, where λ_1 is the rate at which pages can be loaded into memory from the network and λ_2 is the average ratio between the size of a page and the total size of the postings generated from that page.

Constant	*Value*
λ	1.26×10^{-3}
φ	4.62×10^{-4}
δ	6.74×10^{-4}
σ	2.44×10^{-5}

Table 2: Constants determined from measurement runs

will take time $T_s = \frac{N}{3}(l' + p' + f')$, where $l' = \lambda(3B)$, $f' = \varphi(3B)$, and $p' = \delta(3B) + \sigma(3B)\log(3B)$. Thus, in a node with a single resource of each type, the maximal theoretical speedup that we can achieve through pipelining is (after simplification):

$$\begin{aligned} \theta &= \frac{T_s}{T_p} \\ &= \frac{(l+p+f)}{\max(l,p,f)} + \frac{\sigma \log 3}{\max(\lambda, \varphi, \delta + \sigma \log B)} \\ &= \theta_1 + \theta_2 \qquad \text{(say)} \end{aligned}$$

Clearly, $\theta_1 \geq 1$ whereas $\theta_2 \leq \frac{\sigma \log 3}{\max(\lambda,\varphi)} << 1$ for typical values of λ, φ, and σ (refer Table 2). Therefore, we ignore θ_2 and concentrate on choosing the value of B that maximizes θ_1. The maximum value of θ_1 is 3, which is reached when $l = p = f$, i.e., when all three phases are of equal duration. We cannot guarantee $l = f$ since that requires $\lambda = \varphi$. However, we can maximize θ_1 by choosing $p = \max(l, f)$ so that $\theta_1 = 2 + \frac{\min(l,f)}{\max(l,f)}$.

For example, in Figure 4, the ratio between the phases is $l : p : f = 3 : 4 : 2$. Thus, θ_1 for this setting is $\frac{3+4+2}{4} = 2.25$. We could improve θ_1 by changing the ratio to 3:3:2, so that $\theta_1 = 2 + \frac{2}{3} \approx 2.67$. In general, setting $\delta B + \sigma B \log B = \max\{\lambda B, \varphi B\}$, we obtain

$$B = 2^{\frac{\max\{\lambda,\varphi\}-\delta}{\sigma}} \tag{1}$$

This expression represents the size of the postings buffer that must be used to maximize the pipeline speedup, on an indexer with a single resource of each type. If we use a buffer of size less than the one specified by equation 1, loading or flushing (depending on the relative magnitudes of λ and φ) will be the bottleneck and the processing phase will be forced to periodically wait for the other phases to complete. An analogous effect will take place for buffer sizes greater than the one prescribed by equation 1. We can generalize equation 1 for an indexer with c identical CPUs, d identical disks and i input streams, to obtain $B = 2^{\frac{\max\{\lambda\lceil\frac{\mu}{i}\rceil,\varphi\lceil\frac{\mu}{d}\rceil\}-\delta\lceil\frac{\mu}{c}\rceil}{\sigma\lceil\frac{\mu}{c}\rceil}}$ where $\mu = \max\{c, d, i\}$.

3.1.2 Experimental results

To study the impact of the pipelining technique on indexing performance, we conducted a number of experiments using a single indexer node supplied with a stream of web pages from a distributor node.

We first ran the index-builder process in *measurement mode* where we monitor and record the execution times of the various phases in the indexing process - loading, parsing, sorting, and flushing. Based on a number of measurement runs using multiple samples (of 100,000 pages each) from the repository, and using the expressions for l, p, and f from above, we determined the values of λ, φ, σ, and δ (Table 2). Using the values of these constants in equation 1, we evaluate B to be 16 MB. Therefore, the optimal total size of the postings buffers, as predicted by our theoretical analysis, is $3B = 48$ MB.

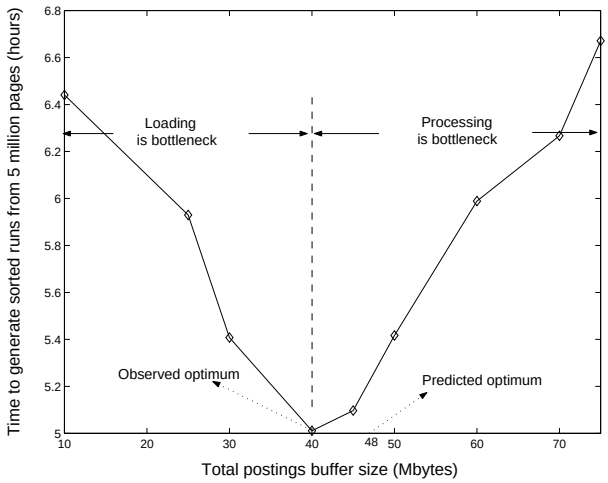

Figure 5: Optimal buffer size

Impact of buffer size on performance Figure 5 illustrates how the performance of the index-builder process varies with the size of the buffer used. The optimal total buffer size based on actual experiments turned out be 40 MB. For values less than 40, loading proved to be the bottleneck and both the processing and flushing phases had to periodically wait for the loading phase to complete. However, as the buffer size increased beyond 40, the processing phase dominated the execution time as larger and larger buffers of postings had to be sorted.

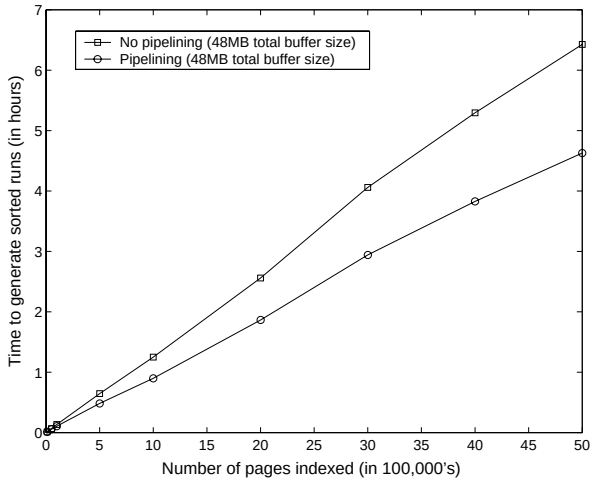

Figure 6: Performance gain through pipelining

Performance gain through pipelining Figure 6 shows how pipelining impacts the time taken to process and generate sorted runs for a variety of input sizes. Note that for small collections of pages, the performance gain through pipelining, though noticeable, is not substantial. This is because small collections require very few pipeline executions and the overall time is dominated by the time required at startup (to load up the buffers) and at shutdown (to flush the buffers). However, as collection sizes increase, the gain becomes more significant and for a collection of 5 million pages, pipelining completes almost 1.5 hours earlier than a purely sequential implementation. Our experiments showed that in

Collection size	*Speedup in generating sorted runs* (seconds)	*Overall speedup including merging* (seconds)
100,000	96	76
500,000	584	550
1,000,000	1264	1134
2,000,000	2505	2265

Table 3: Overall speedup after including merging

general, for large collections, a sequential index-builder is about 30-40% slower than a pipelined index-builder. Note that the observed speedup is lower than the speedup predicted by the theoretical analysis described in the previous section. That analysis was based on an "ideal pipeline," in which loading, processing and flushing do not interfere with each other in any way. In practice, however, network and disk operations do use processor cycles and access main memory. Hence, any two concurrently running phases, even of different types, do slow down each other.

Note that for a given buffer size, pipelined execution will generate sorted runs that are approximately 3 times smaller than those generated by a sequential indexer. Consequently, 3 times as many sorted runs will need to be merged in the second stage of indexing. However, as indicated in Table 3, our experiments show that even for very large collection sizes, the potential increase in merging time is more than offset by the time gained in the first stage through pipelining. We expect that as long as there is enough main memory at merge time to allocate buffers for the sorted runs, performance will not be substantially affected.

3.2 Managing inverted files using an embedded database system

Inverted files can either be stored and managed using a custom implementation or by leveraging existing relational or object data management systems [3, 8]. The advantage of a custom implementation is that it enables very effective optimizations tuned to the specific operations on inverted files (e.g., caching frequently used inverted lists, compressing rarely used inverted lists using expensive methods that may take longer to decompress). Leveraging existing data management systems does not allow such fine-grained control over the implementation but reduces development time and complexity. Also, additional features such as concurrency control and recovery are available without extra development effort.

However, the challenge lies in designing a scheme for storing inverted files that makes optimal use of the storage structures provided by the data management system. The storage scheme must be space efficient and must ensure that the basic lookup operation on an inverted file (i.e., retrieving some or all of the inverted list for a given index term) can be efficiently implemented using the native access methods of the data management system.

In our testbed, we use a freely available embedded database system called Berkeley DB [19] to store and manage inverted files. An embedded database is a library or toolkit that provides database support for applications through a well-defined programming API. Unlike traditional database systems that are designed to be accessed by applications, embedded databases are linked (at compile-time or run-time) into an application and act as its persistent storage manager. They provide device-sensitive file allocation, database access methods (such as B-trees and hash indexes), and optimized caching, with optional support for transactions, locking, and recovery. They also have the advantage of much smaller footprints compared to full-fledged client-server database systems.

In the following, we briefly the sketch the capabilities of Berkeley DB and propose a B-tree based inverted file storage scheme called the *mixed-list scheme*. We qualitatively compare the mixed-list

scheme with two other schemes for storing inverted lists in Berkeley DB databases. We present performance results that indicate that the mixed-list scheme provides performance and storage characteristics comparable to those reported by other inverted file implementations.

3.2.1 Rationale and implementation

Berkeley DB provides a programming library for managing `(key,value)` pairs, both of which can be arbitrary binary data of any length. It offers four access methods, including B-trees and linear hashing, and supports transactions, locking, and recovery.[4] We chose to use the B-tree access method in our implementation because it efficiently supports prefix searches (e.g., retrieve inverted lists for all terms beginning with "pre") and has higher reference locality than hash-based indexes.

The standard organization of a B-tree based inverted file involves storing the index terms in the B-tree along with pointers to inverted lists that are stored separately. Such an organization, though easy to implement using Berkeley DB, does not fully utilize the capabilities of the database system. Since Berkeley DB efficiently handles arbitrary sized keys and values, it is more efficient to store both the index terms and their inverted lists within the database. This enables us to leverage Berkeley DB's sophisticated caching schemes while retrieving large inverted lists with a minimum number of disk operations.

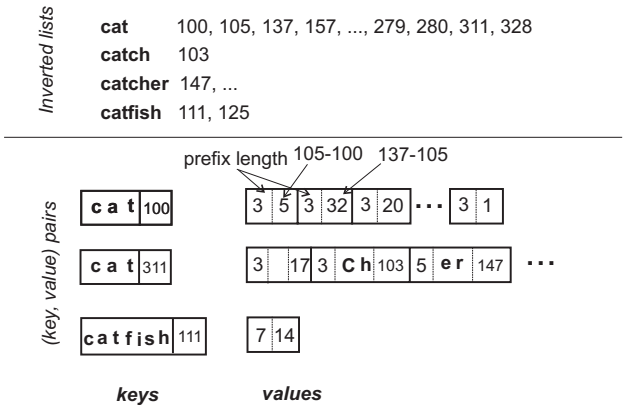

Figure 7: Mixed list storage scheme

Storage schemes We considered three storage schemes for storing inverted files as sets of `(key,value)` pairs in a B-tree:

1. *Full list*: The key is an index term, and the value is the complete inverted list for that term.

2. *Single payload*: Each posting (an index term, location pair) is a separate key.[5] The value can either be empty or may contain additional information about the posting.

3. *Mixed list*: The key is again a posting, i.e., an index term and a location. However, the value contains a number of successive postings in sorted order, even those referring to different index terms. The postings in the value field are compressed and in every value field, the number of postings is chosen so that the length of the field is approximately the same. Note that in this

[4] All these features can be turned off, if desired, for efficiency.

[5] Storing the indexing term in the key and a single location in the value is not a viable option as the locations for a given term are not guaranteed to be in sorted order.

Scheme	Index size	Zig-zag joins	Hot updates
single payload	−−	+	+
full list	+−	−	−
mixed list	+−	+−	+−

Table 4: Comparison of storage schemes

scheme, the inverted list for a given index term may be spread across multiple `(key,value)` pairs.

Figure 7 illustrates the mixed-list storage scheme. The top half of the figure depicts inverted lists for four successive index terms and the bottom half shows how they are stored as `(key,value)` pairs using the mixed-list scheme. For example, the second `(key,value)` pair in the figure, stores the set of postings `(cat,311)`, `(cat,328)`, `(catch,103)`, `(catcher,147)`, etc., with the first posting stored in the key and the remaining postings stored in the value. As indicated in the figure, the postings in the value are compressed by using prefix compression for the index terms and by representing successive location identifiers in terms of their numerical difference. For example, the posting `(cat,328)` is represented by the sequence of entries `3` <an empty field> `17`, where `3` indicates the length of the common prefix between the words for postings `(cat,311)` and `(cat,328)`, the `<empty field>` indicates that both postings refer to the same word, and `17` is the difference between the locations `328` and `311`. Similarly, the posting `(catch,103)` is represented by the sequence of entries `3 ch 103`, where `3` is the length of the common prefix of `cat` and `catch`, `ch` is the remaining suffix for `catch`, and `103` is the location.

A qualitative comparison of these storage schemes is summarized in Table 4.

Index size The crucial factors determining index size are the number of internal pages (a function of the height of the B-tree) and the number of overflow pages.[6] In the *single payload* scheme, every posting corresponds to a new key, resulting in rapid growth in the number of internal pages of the database. For large collections, the database size becomes prohibitive even though Berkeley DB employs prefix compression on keys. Also, at query time, many performance-impeding disk accesses are needed. The situation is significantly better with the *full list* scheme. A database key is created only for every distinct term, and the value field can be well compressed. However, many terms occur only a few times in the collection whereas others may occur in almost every page. To accommodate such large variations in the size of the value field, many overflow pages are created in the database. In comparison, with the *mixed list* scheme, the length of the value field is approximately constant. This limits the number of overflow pages. Moreover, the total number of keys (and hence the number of internal pages) can be further reduced by choosing a larger size for the value field. However, since the value field can contain postings of different index terms, it is not compressed as well as with full lists.

Zig-zag joins The ability to selectively retrieve portions of an inverted list can be very useful when processing conjunctive search queries on an inverted file. For example, consider the query `green AND catchflies`. The term `green` occurs on the Web in millions of documents, whereas `catchflies` produces only a couple of dozen hits. A zig-zag join [7] between the inverted lists for `green` and `catchflies` allows us to answer the query without reading out the complete inverted list for `green`. The single payload scheme provides the best support for zig-zag joins as each posting can be retrieved individually. In the full list scheme, the entire list must be retrieved to compute the join, whereas

[6]Since values can be of arbitrary length, Berkeley DB uses overflow pages to handle large value fields.

with the mixed list scheme, access to specific portions of the inverted list is available. For example, in Figure 7, to retrieve locations for `cat` starting at 311, we do not have to read the portion of the list for locations 100-280.

The skipped-list and random inverted-list structures of [17] and [18] also provides selective access to portions of an inverted list, by diving the inverted list into blocks each containing a fixed number of postings. However, those schemes assume a custom inverted file implementation and are not built on top of an existing data management system.

Hot updates Hot updates refers to the ability to modify the index at query time. In all three schemes, the concurrency control mechanisms of the database can be used to support such hot updates while maintaining consistency. However, the crucial performance factor is the length of the inverted list that must be read, modified, and written back to achieve the update. Since we limit the length of the value field, hot updates are faster with mixed lists than with full lists. The single payload scheme provides the best update performance as individual postings can be accessed and modified.

Notice that all three schemes significantly benefit from the fact that the postings are first sorted and then inserted. Inserting keys into the B-tree in a random order negatively affects the page-fill factor, and expensive tree reorganization is needed. Berkeley DB is optimized for sorted insertions so that high performance and a near-one page-fill factor can be achieved in the initial index construction phase.

Table 4 shows that the mixed-list scheme provides the best balance between small index size and support for efficient zig-zag joins. In the following section, we present performance results from experiments that we conducted using the mixed-list scheme.

3.2.2 Experimental results

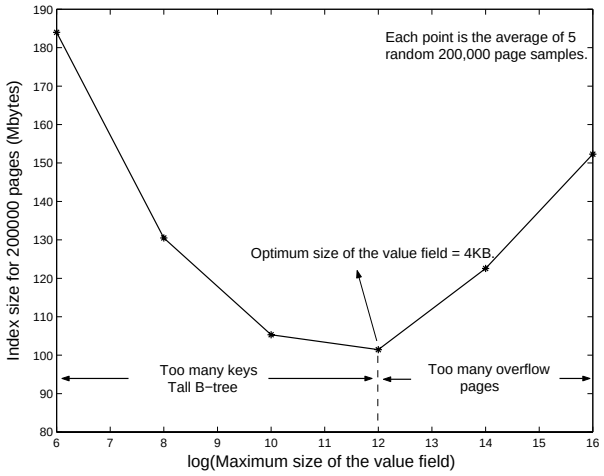

Figure 8: Variation of index size with value field size

Figure 8 illustrates the impact of the size of the value field on the size of the resulting inverted index when using the mixed-list scheme. Both very small and very large value fields have an adverse impact on index size. Very small value fields will require a large number of internal database pages (and a potentially taller B-tree index) to accommodate all the postings. Very large value fields will cause the allocation of a number of overflow pages which in turn lead to a larger index.

Table 5 shows how the index size (using the mixed-list scheme) varies with the size of the input

Number of pages (million)	Input size (GB)	Index size (GB)	Index size (%age)
0.1	0.81	0.05	6.68
0.5	4.03	0.27	6.70
2.0	16.11	1.13	7.04
5.0	40.28	2.78	6.90

Table 5: Mixed-list scheme index sizes

collection.[7] The numbers for Table 5 were generated by using mixed-lists with the optimal value field size of 4 KB derived from Figure 8. Our implementation of the mixed-list scheme also used the BER (Basic Encoding Rules) technique to represent integers in the key and the value. Table 5 shows that the mixed-list storage scheme scales very well to large collections. The size of the index is consistently about 7% the size of the input HTML text. This compares favorably with the sizes reported for the VLC2 track (which also used crawled web pages) at TREC-7 [9] where the best reported index size was approximately 7.7% the size of the input HTML. Our index sizes are also comparable to other recently reported sizes for non-Web document collections using compressed inverted files [18].

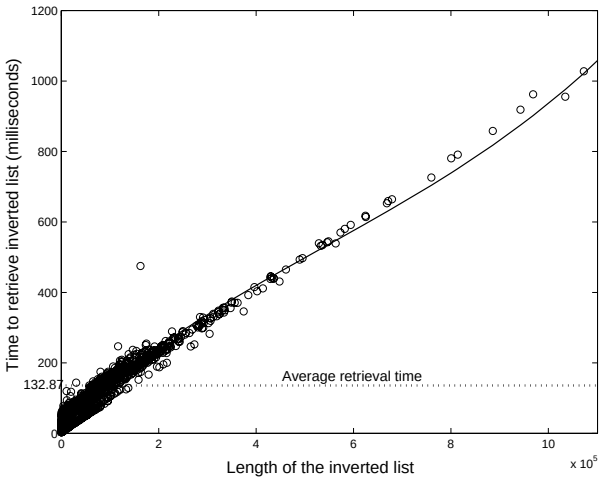

Figure 9: Time to retrieve inverted lists

Finally, Figure 9 illustrates the inverted list retrieval performance of our mixed-list scheme. The graph was produced by generating about 100,000 random query terms and measuring the time required to retrieve the entire inverted list for each query term from a 1.13 GB inverted file (obtained by inverting a collection of 2 million pages). Figure 9 indicates that most inverted lists, even those containing more than a million entries, can be retrieved in less than a second. The average retrieval time is less than 150 milliseconds as most lists are not very long (the median list length was only about 20,000 entries). Note that an efficient implementation for processing conjunctive search queries would make good use of the zig-zag join technique and in many cases, would avoid retrieving all the entries from a long inverted list. The database cache size did not have a very significant impact on this test as the query terms were randomly generated.

[7] Only one posting was generated for all the occurrences of a word in a page.

4 Distributed indexing

In the following sections, we discuss two problems that must be addressed when building an inverted index on a distributed architecture:

- Page distribution: the question of when and how to distribute pages to the indexing nodes.
- Collecting global statistics: the question of where, when, and how to compute and distribute global (or collection-wide) statistics.

4.1 Page Distribution Strategies

Very often, the text collection to be indexed is available on a set of nodes disjoint from the nodes used for indexing. For example, in our testbed, the collection of web pages is stored on the distributors and needs to be shipped to the indexers for the purposes of indexing. We consider two strategies for interleaving page distribution and index construction:

A priori distribution: In this strategy, all the pages are first retrieved from the distributors and stored on the local disks of the indexers. During indexing, the index-builder processes retrieve the pages from the local disk instead of receiving them over the network. A priori distribution is the approach adopted in [21].

Runtime distribution: In this strategy, collection distribution takes place concurrently with index-building. The index-builder processes directly operate on the pages as they are received over the network from the distributor(s).

A priori distribution requires **additional space** on the indexer nodes to store the pages. Typically, our testbed generates indexes whose size is less than 10% that of the collection. Therefore, a priori distribution requires more than 10 times as much space on the indexer as runtime distribution.

With runtime distribution, it is possible to easily achieve some rudimentary **load balancing** during index construction. Distributor processes can be designed to deliver pages to different indexer nodes at different rates, depending on the processing speed of the individual indexers.[8] As a result, an indexer receives a portion of the collection whose size is commensurate with its processing capability. For example, an indexer which is able to index pages twice as fast as another will get to process, on average, about twice the number of pages as the slower indexer.[9] With a priori distribution, such runtime load balancing cannot be implemented since pages are pre-assigned to indexer nodes. Of course, it is possible to try and simulate load balancing based on some estimates of the relative processing capabilities of the different indexers.

Runtime distribution also leads to more **effective pipelining** (Section 3.1). Note that for maximum performance gain, the individual phases of the pipeline must use disjoint system resources to the extent possible. With runtime distribution, the loading phase exercises the networking subsystem of the indexer nodes whereas flushing exercises the disk subsystem. With a priori distribution, both loading and flushing require disk access and are likely to interfere strongly with each other. This is especially true if both phases access the same disk.

[8] In our testbed, distributor processes are fully multi-threaded. Each thread of the process is responsible for transferring pages to exactly one indexer.

[9] Since Web search services employ hundreds (and sometimes even thousands) of machines for indexing and querying, it is very likely that hardware upgrades to these machines take place in batches. Therefore, the set of machines in operation at any given time, are unlikely to be identical in performance characteristics.

The main disadvantage of runtime distribution is the **complexity in dealing with node failures**. When an indexer node (or the index-builder process running on the indexer node) fails during index construction, the set of pages that it was processing in memory are lost. When multiple such failures occur, a number of small subsets of the collection do not get processed by the indexer nodes. These subsets must be retransmitted and processed before the second stage of indexing can commence. However, such selective transmission of specific portions of the collection is often not supported by Web repositories, where the most common access mode is a continuous stream of pages.

4.2 Collecting Global Statistics

Most information retrieval systems use some kind of collection-wide information to increase effectiveness of retrieval [26]. One popular example is the inverse document frequency (IDF) statistics used in ranking functions. The IDF of a term is the inverse of the number of documents in the collection that contain that term. If query servers have only IDF values over their local collections, then rankings would be skewed in favor of pages from query servers that return few results. In order to offer effective rankings to the user, query servers must know global IDF values.

In this section, we analyze the problem of gathering collection-wide information with minimum overhead. We present two techniques that are capable of gathering different types of collection-wide information, though here we focus on the problem of collecting term-level global statistics, such as IDF values.[10]

4.2.1 Design

While some authors suggest computing global statistics at query time, using this approach would require an extra round of communication among the query servers at query time to exchange local statistics. This communication adversely impacts query processing performance, especially for large collections spread over many servers. Since query response times in most systems are critical, our system precomputes and stores statistics at the query servers during index creation instead.

A dedicated server known as the statistician is used for computing statistics. Having a dedicated statistician allows most computation to be done in parallel with other indexing activities. It also minimizes the number of conversations among servers, since indexers exchange statistical data with only one statistician. Local information is sent to the statistician at various stages of index creation, and the statistician returns global statistics to the indexers in the merging phase. Indexers then store the global statistics in the local lexicons. A lexicon consists of entries of the form *(term, term-id, local-statistics, global-statistics)*, where the terms stored in a lexicon are only those terms occurring in the associated inverted file (Section 2).

In order to avoid extra disk I/O, local information is sent to the statistician only when it is already in memory. We have identified two phases in which this occurs: *flushing* - when sorted runs are written to disk, and *merging* - when sorted runs are merged to form inverted lists and the lexicon. Sending information in both of these phases leads to two different strategies, both with various tradeoffs which are discussed in the next section. We note here only that local information can be sent to the statistician in these phases without any I/O overhead; thus, a huge fraction of statistic collection cost is eliminated.

Sending information to the statistician is further optimized by summarizing the postings. In both identified phases, postings occur in at least partially sorted order, meaning multiple postings for a term pass through memory in groups. Groups are condensed into *(term, local aggregated information)*

[10] *Term-level* refers to the fact that any gathered statistic describes only single terms, and not higher level entities such as pages or documents.

	Phase	Statistician load	Memory usage	Parallelism
ME	merging	+-	+	+-
FL	flushing	-	-	++

Table 6: Comparison of schemes for collecting global statistics

pairs which are sent to the statistician. For example, if an indexer holds 10000 pages that contain the term "cat", instead of sending 10000 individual postings to the statistician, the indexer can count the postings as they pass through memory in a group and send the summary *("cat", 10000)* to the statistician. The statistician receives local counts from all indexers, and aggregates these values to produce the global document frequency for "cat". This technique greatly reduces network overhead in collecting statistics.

4.2.2 Statistic Gathering Strategies

Here we describe and compare the two strategies mentioned above for sending information to the statistician. Table 6 summarizes their characteristics. The column titled "Parallelism," refers to the degree of parallelism possible within each strategy.

ME Strategy: sending local information during merging. Summaries for each term are aggregated as inverted lists are created in memory, and sent to the statistician. The statistician receives parallel sorted streams of *(term, local-aggregate-information)* values from each indexer and merges these streams by term, aggregating the sub-aggregates for each term to produce global statistics. The statistics are then sent back to the indexers in sorted term order. This approach is entirely stream based, and does not require in-memory or on-disk data structures at the statistician or indexer to store intermediate results. However, using streams means that the progress of each indexer is synchronized with that of the statistician, which in turn causes indexers to be synchronized with each other. As a result, the slowest indexer in the group becomes the bottleneck, holding back the progress of faster indexers.

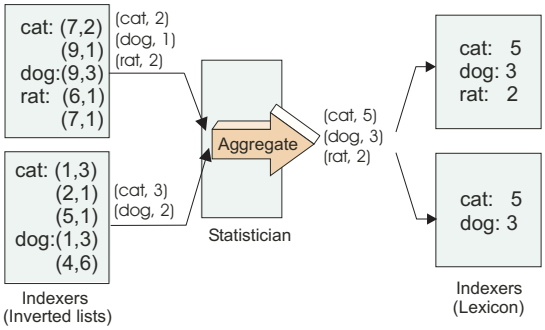

Figure 10: ME strategy

Figure 10 illustrates the ME strategy for collecting document frequency statistics for each term. Inverted lists are summarized into sorted streams of (term, local document frequency) pairs and sent to the statistician. At the statistician, incoming streams are merged and aggregated. The output is a sorted stream of global document frequencies for each term that is sent to the indexers. The bottom lexicon does not include statistics for "rat" because the term is not present in the local collection.

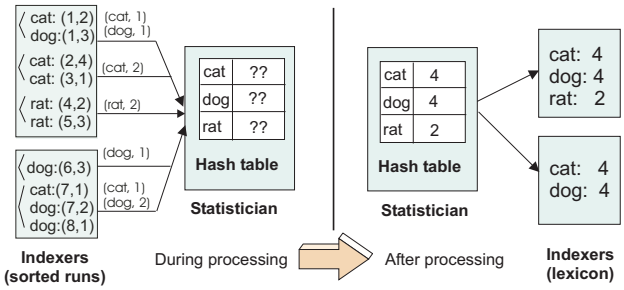

Figure 11: FL strategy

FL Strategy: sending local information during flushing. As sorted runs are flushed to disk, postings are summarized and the summaries sent to the statistician. Since sorted runs are accessed sequentially during processing, the statistician receives streams of summaries in globally *unsorted* order. To compute statistics from the unsorted streams, the statistician keeps an in-memory hash table of all terms and their related statistics, and updates the statistics as summaries for a term are received. Because the vocabulary exhibits sub-linear growth with respect to the size of the collection (see Section 2.1), the hash table is expected to fit easily within memory. At the end of the processing phase, the statistician sorts the statistics in memory and sends them back to the indexers. Figure 11 illustrates the FL strategy for collecting document frequency statistics.

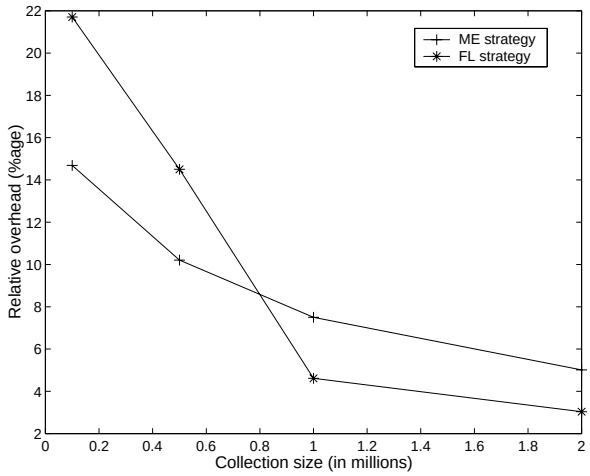

Figure 12: Overhead of statistics collection

4.2.3 Experiments

To demonstrate the performance and scalability of the collection strategies, we ran the index-builder and merging processes over a hardware configuration of four indexers on various collection sizes. Figure 12 shows the *relative overhead* (time overhead expressed as a percentage of index creation time with *no* statistics collection) for both strategies. In general, experiments show the FL strategy outperforming ME, although they seem to converge as the collection size becomes large. Furthermore, as the collection size grows, the relative overheads of both strategies decrease.

Comparison of strategies. At first glance ME might be expected to outperform FL: since the statistician receives many summary streams in FL, but only one from each indexer in ME, it performs more comparison and aggregation in FL than in ME. However, as mentioned earlier, merging progress in ME is synchronized among the servers. Hence, a good portion of computation done at the statistician cannot be done in parallel with merging activities at the indexer.

In FL, on the other hand, the indexer simply writes summaries to the network and continues with its other work. The statistician can then asynchronously process summary information from the network buffer in parallel. However, not all work can be done in parallel, since the statistician consumes summaries at a slower rate than the indexer writes them to network, and the network buffer generally cannot hold all the summaries from a sorted run. Hence there is still nontrivial waiting at the indexer during flushing as summaries are sent to the statistician.

Enhancing parallelism. In the ME strategy, synchronization occurs when an indexer creates a lexicon entry and summary for a term, sends the summary to the statistician, and then waits for the global statistic to be returned so that the lexicon entry can be completed. To reduce the effect of synchronization, the merging process can instead write lexicon entries to a *lexicon buffer*, and a *separate* process will wait for global statistics and include them in the entries. In this way, the first process need not block while waiting, and both processes can operate in parallel.

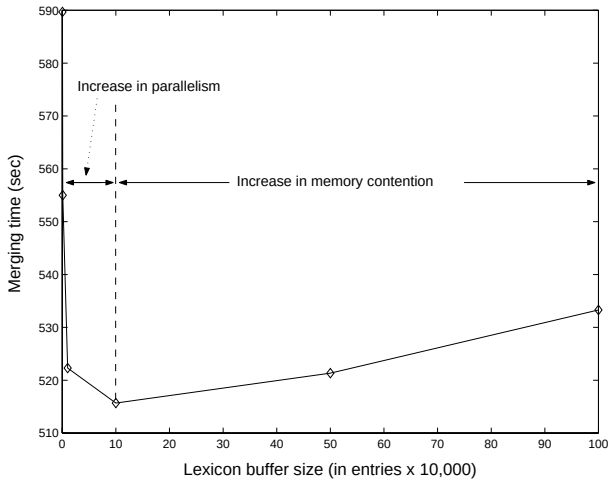

Figure 13: Varying lexicon buffer size

With parallelized merging, the size of the lexicon buffer becomes an important parameter. Figure 13 shows the effect of lexicon buffer size on merging performance over a collection of a million pages. Because lexicon entries are created faster than global statistics are returned on all indexers but the slowest, the lexicon buffer often becomes full. When this occurs, the process creating lexicon entries must block until the current state changes. Because larger lexicon buffers reduce the possibility of saturation, we expect and see that initial increases in size result in large performance gains. As lexicon buffer size becomes very large, however, performance slowly deteriorates due to memory contention. Although the entire buffer need not be present in memory at any one time, the lexicon buffer is accessed cyclically; therefore LRU replacement and the fast rate at which lexicon entries are created cause buffer pages to cycle rapidly through memory, swapping out other non-buffer pages.

Sub-linear growth of overhead. The constant decrease of the ME and FL relative overhead in Figure 12 can be explained by the fact that the number of distinct terms in a page collection is a

sub-linear function of collection size. The overhead incurred by gathering statistics grows linearly with the number of terms in the collection, while the cost of index creation grows linearly with the size of the collection. As a result, overhead of statistic collection will display sub-linear growth with respect to index creation time. This prediction is consistent with our experimental results.

However, the decreasing relative overhead for FL is subject to the constraint that the hashtable can fit in memory. Considering that a collection of a billion pages would require a hash table of roughly 5-6 GB in size[11], this constraint may become a problem for very large collections. While a memory of 6 GB is not completely unreasonable, a simple alternative using only commodity hardware would be to run several statisticians in parallel, and partition the terms alphabetically between statisticians. In this way, each statistician can collect and sort a moderately sized set of global statistics. We have not implemented this option in our system.

5 Related Work

Motivated by the Web, there has been recent interest in designing scalable techniques to speed up inverted index construction using distributed architectures. In [21], Ribeiro-Neto, et. al describe three techniques to efficiently build an inverted index using a distributed architecture. However, they focus on building global, rather than local, inverted files. Furthermore, they do not address issues such as global statistics collection and optimization of the indexing process on each individual node.

Our technique for structuring the core index engine as a pipeline has much in common with pipelined query execution strategies employed in relational database systems [7]. Chakrabarti, et. al. [4] present a variety of algorithms for resource scheduling with applications to scheduling pipeline stages.

There has been prior work on using relational or object-oriented data stores to manage and process inverted files [1, 3, 8]. Brown, et. al. [3] describe the architecture and performance of an information retrieval system that uses a persistent object store to manage inverted files. Their results show that using an *off-the-shelf* data management facility improves the performance of an information retrieval system, primarily due to intelligent caching and device-sensitive file allocation. We experienced similar performance improvements for the same reasons by employing an embedded database system. Our storage format differs greatly from theirs because we utilize a B-tree storage system and not an object store.

References [27] and [26] discuss the questions of when and how to maintain global statistics in a distributed text index, but their techniques only deal with challenges that arise from incremental updates. Our system does not employ incremental updates; we wished to explore strategies for gathering statistics during index construction.

A great deal of work has been done on several other issues, relevant to inverted-index based information retrieval, that have not been discussed in this paper. Such issues include index compression [16, 18, 29], incremental updates [2, 11, 25, 29, 30], and distributed query performance [23, 24].

6 Conclusions

In this paper, we addressed the problem of efficiently constructing inverted indexes over large collections of web pages. We proposed a new pipelining technique to speed up index construction and addressed the issue of choosing the right buffer sizes to maximize performance. We demonstrated that for large collection sizes, the pipelining technique can speed up index construction by several

[11] From Section 2.1, a billion pages will contain roughly 310 million distinct terms, and each term using 20 bytes of storage results in a hashtable of 5.77 GB.

hours. We proposed and compared different schemes for storing and managing inverted files using an embedded database system. We showed that an intelligent scheme for packing inverted lists in the storage structures of the database can provide performance and storage efficiency comparable to tailored inverted file implementations.

We presented a qualitative discussion and comparison of the "a priori" and "runtime" approaches to page distribution. We showed that there is a tradeoff between ease of implementation and superior performance. Finally, we identified the key characteristics of methods for efficiently collecting global statistics from distributed inverted indexes. We proposed two such methods and compared and analyzed the tradeoffs thereof.

In the future, we intend to extend our testbed to incorporate distributed query processing and explore algorithms and caching strategies for efficiently executing queries. We also intend to experiment with indexing and querying over larger collections and integration of our text-indexing system with indexes on the link structure of the Web.

References

[1] D. C. Blair. An extended relational document retrieval model. *Information Processing and Management*, 24(3):349–371, 1988.

[2] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of 20th International Conference on Very Large Databases*, pages 192–202, September 1994.

[3] Eric W. Brown, James P. Callan, W. Bruce Croft, and J. Eliot B. Moss. Supporting full-text information retrieval with a persistent object store. In *4th International Conference on Extending Database Technology*, pages 365–378, March 1994.

[4] S. Chakrabarti and S. Muthukrishnan. Resource scheduling for parallel database and scientific applications. In *8th ACM Symposium on Parallel Algorithms and Architectures*, pages 329–335, June 1996.

[5] Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. To appear in the 26th International Conference on Very Large Databases, September 2000.

[6] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems*, 2(4):267–288, October 1984.

[7] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation.* Prentice-Hall, 2000.

[8] D. A. Gorssman and J. R. Driscoll. Structuring text within a relation system. In *Proceedings of the 3rd International Conference on Database and Expert System Applications*, pages 72–77, September 1992.

[9] D. Hawking and N. Craswell. Overview of TREC-7 very large collection track. In *Proceeedings of the Seventh Text Retrieval Conference*, pages 91–104, November 1998.

[10] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. WebBase: A repository of web pages. In *Proceedings of the 9th International World Wide Web Conference*, pages 277–293, May 2000.

[11] B-S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, February 1995.

[12] Steve Lawrence and C. Lee Giles. Accessibility of information on the web. *Nature*, 400:107–109, 1999.

[13] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.

[14] Patrick Martin, Ian A. Macleod, and Brent Nordin. A design of a distributed full text retrieval system. In *Proceedings of the ACM Conference on Research and Development in Information Retrieval*, pages 131–137, September 1986.

[15] Mike Burrows. Personal Communication.

[16] A. Moffat and T. Bell. In situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550, 1995.

[17] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.

[18] Anh NgocVo and Alistair Moffat. Compressed inverted files with reduced decoding overheads. In *Proceedings of the 21st International Conference on Research and Development in Information Retrieval*, pages 290–297, August 1998.

[19] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the 1999 Summer Usenix Technical Conference*, June 1999.

[20] B. Ribeiro-Neto and R. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the 3rd ACM Conference on Digital Libraries*, pages 182–190, June 1998.

[21] Berthier Ribeiro-Neto, Edleno S. Moura, Marden S. Neubert, and Nivio Ziviani. Efficient distributed algorithms to build inverted files. In *Proceedings of the 22nd ACM Conference on Research and Development in Information Retrieval*, pages 105–112, August 1999.

[22] G. Salton. *Information Retrieval: Data Structures and Algorithms.* Addison-Wesley, Reading, Massachussetts, 1989.

[23] Anthony Tomasic and Hector Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 8–17, January 1993.

[24] Anthony Tomasic and Hector Garcia-Molina. Query processing and inverted indices in shared-nothing document information retrieval systems. *VLDB Journal*, 2(3):243–275, 1993.

[25] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental update of inverted list for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 289–300, May 1994.

[26] Charles L. Viles. Maintaining state in a distributed information retrieval system. In *32nd Southeast Conference of the ACM*, pages 157–161, 1994.

[27] Charles L. Viles and James C. French. Dissemination of collection wide information in a distributed information retrieval system. In *Proceedings of the 18th International ACM Conference on Research and Development in Information Retrieval*, pages 12–20, July 1995.

[28] Inktomi WebMap. http://www.inktomi.com/webmap/.

[29] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kauffman Publishing, San Francisco, 2nd edition, 1999.

[30] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In *18th International Conference on Very Large Databases*, pages 352–362, August 1992.

A Proof of optimality of equisize buffers

We are given $T_p = \max\{\sum_{i=1}^{N} l_i, \sum_{i=1}^{N} p_i, \sum_{i=1}^{N} f_i\}$. If loading or flushing is the bottleneck, T_p is either λB_{total} or φB_{total}, and has the same value for all distributions of B_i including an equisize distribution. If processing is the critical phase, $T_p = \sum_{i=1}^{N} (\delta B_i + \sigma B_i log B_i)$. Under the constraint $\sum_{i=1}^{N} B_i = B_{total}$, the absolute minimum of T_p is reached when $B_i = \frac{B_{total}}{N}$ for each i, i.e., when all buffers have equal sizes. This global extremum can be easily determined using standard techniques such as Lagrange multipliers.

B Global versus Local Inverted Files

The *local inverted file* organization partitions the document collection so that each query server is responsible for a disjoint subset of documents in the collection. A search query would be broadcast to all the query servers, each of which would return disjoint lists of document identifiers containing the search terms.

The other option, called *global inverted file*, is to partition based on index terms so that each query server stores inverted lists only for a subset of the index terms in the collection. For example, in a global inverted file organization with two query servers A and B, A could store the inverted lists for all index terms that begin with characters in the ranges *[a-q]* whereas B could store the inverted list for the remaining index terms. Therefore a search query that asks for documents containing the term *"process"* would only involve A.

We believe that the local inverted file strategy has certain important advantages especially in the context of answering Web search queries.

Resilience to failures In the global inverted file organization, when a query server that is responsible for some set of index terms fails, no search queries dealing with that set of index terms can be answered. On the other hand, a similar single-node failure in a local inverted file organization does not prevent any search query from being answered, though the result set might not contain all the relevant documents in the collection. For very large collections such as the Web, the temporary loss of some subset of documents from the index may not be very critical.

Lower network load The most common search queries on the Web are Boolean conjunctions of a small number of search terms. For such queries, each query server in a local inverted file organization can perform the necessary Boolean conjunctions locally and return, over the network, only document identifiers that belong to the result set. On the other hand, in a global inverted file organization, if the search terms in a query involve more than one query server, unnecessary document identifiers may be sent over the network. For huge collections such as the web, if one of the search terms happens to occur in a large percentage of documents, query performance is likely to be significantly affected.