# High Performance Index Build Algorithms for Intranet Search Engines

Marcus Fontoura Eugene Shekita Jason Y. Zien Sridhar Rajagopalan Andreas Neumann

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
USA
trevi@almaden.ibm.com

## Abstract

There has been a substantial amount of research on high-performance algorithms for constructing an inverted text index. However, constructing the inverted index in a intranet search engine is only the final step in a more complicated *index build process*. Among other things, this process requires an analysis of all the data being indexed to compute measures like PageRank. The time to perform this *global analysis* step is significant compared to the time to construct the inverted index, yet it has not received much attention in the research literature. In this paper, we describe how the use of slightly outdated information from global analysis and a fast index construction algorithm based on radix sorting can be combined in a novel way to significantly speed up the index build process without sacrificing search quality.

## 1 Introduction

Most web and intranet search engines use an inverted text index to execute queries [29]. Because inverted indexes are expensive to update [7, 8, 28], search engines typically reconstruct their index from scratch on a periodic basis. This is simplier and faster than trying to incrementally update the index. The more frequently an index can be reconstructed, the faster updates will be reflected in search results, which in turn improves search quality. Therefore, the time to construct an inverted index is an important issue in search engines.

There has been a substantial amount of research on high-performance algorithms for constructing an inverted text index [1, 17, 22, 29]. However, constructing the inverted index in a intranet search engine is only the final step in a more complicated *index build process*. Before data is indexed, it has to be analyzed as a whole in a *global analysis* (GA) step. Examples of GA computations include static rank [23], duplicate detection [5, 6], anchor text extraction [14, 15], and template detection [2]. The information gleaned from GA is then used as input to construct the inverted text index. For example, static rank can used to put the index in rank order [19]. The time to perform GA is significant compared to the time to construct an inverted index, since all the data being indexed has be analyzed. Moreover, the proportion of time spent in GA is likely to grow as analysis becomes more sophisticated [12].

In this paper, we describe how the use of slightly outdated GA information, that is *lagging GA*, and a fast index construction algorithm based on radix sorting can be combined in a novel way to significantly speed up the index build process without sacrificing search quality. Results from the Trevi search engine are presented. Trevi was developed in IBM Research and is currently used to support all the searches on IBM's global intranet, which runs in 7x24 mode and supports over 350,000 employees. The main contributions of this paper include:

- An index build process that uses lagging GA, allowing Trevi's inverted index to be constructed with just one pass over the data.
- A fast index construction algorithm based on a pipelined radix sort.
- Experimental results showing that lagging GA can speed up the index build process, with only a negligible degradation in precision.

*Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.*

**Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004**

- Experimental results showing that Trevi's index construction algorithm is significantly faster than alternative approaches described in the research literature.

## 2 Architecture Overview

Figure 1 shows Trevi's hardware and software architecture for IBM's intranet. As shown, there are four processing nodes in a cluster. Each node is a commodity two-way x86 SMP running Linux, with its own direct-attached RAID for storage. The cluster is connected with a local gigabit Ethernet to ensure ample bandwidth for copying data between nodes.

As shown in Figure 1, each node in the Trevi cluster is assigned a particular task. This partitioning of nodes by task makes it easy to crawl, construct inverted indexes, and execute queries in parallel. The *Crawler* is responsible for crawling data. It stores raw documents (HTML pages, PDF files, XML, etc.) along with associated metadata in a database. Raw documents are copied to the *Index Build* node, which is responsible for periodically running the index build process. Once an index has been constructed, it is copied to the *Query Servers*, which execute end-user searches. Two Query Servers are used for fault-tolerance and load balancing.

The RAID storage on each node is spread over two physical arrays. and data is never updated in-place, much like in a log-structured file system [25]. When a new index is constructed, data is always read from one disk array and written to the other array. This dual-array storage architecture greatly improves the performance of the index build process by making all disk I/O sequential. It also makes it possible to atomically install a new index on a Query Server without taking the server down.

## 3 Index Build Data Structures

The main focus of this paper is on the index build process, so we do not present further details on the Crawler and Query Servers. This section describes the main data structures used in the index build process, which we subsequently refer to as simply *index build*. The data structures used during index build are the *Store*, the *Index*, the *DeltaStore*, and the *DeltaIndex*. All of these data structures are maintained on the Index Build node and copied to the Query Servers, where they are accessed in read-only mode.

### 3.1 The Store

The Store is a repository for the tokenized version of each document. Documents read from the Crawler database are parsed, tokenized, and then added to the Store. The reason for storing tokenized content in the Store is performance. Multi-format parsing and multilingual tokenization of a document is extremely CPU intensive, on the order of 50 times slower than a random disk I/O. Consequently, we want to do it only once. After the bulk of the documents on the intranet have been crawled, there are typically just small additions to the Store, on the order of 5% to 10% per day. When a document changes it will eventually be recrawled.

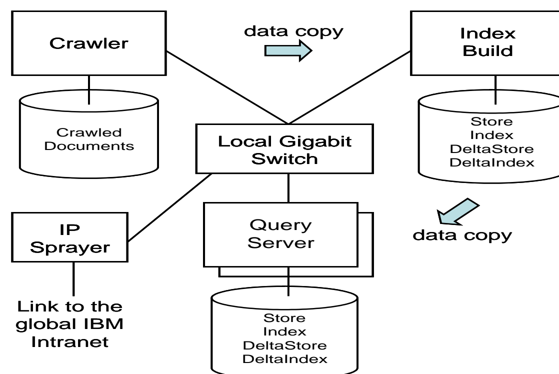


Figure 1: High-level view of Trevi's hardware and software architecture

Tokenized documents tend to be small, so they are aggregated into *bundles* in the Store to enable big-block I/O for improved performance. Currently, each bundle in Trevi corresponds to a 8 MB file. The Store is scanned sequentially during index build, but an API also exists for randomly accessing documents. This API is used during query processing to generate summaries for results. Each document in the Store has an associated *store locator* that encodes the bundle and offset within that bundle for a document.

Each document in the Store is identified by a 64-bit hash of its URL. We refer to this as the *URL hash* of a document. Auxiliary data structures are kept that make it possible to obtain the store locator of a document given its URL hash.

### 3.2 The Index

Each token occurrence (posting) in the inverted text index contains a position and an attribute, i.e, *posting=(position, attribute)*. The position encodes the document ID (docID) and the offset within the document in which the token appeared. Positional information is necessary for implementing phrase search and proximity queries. The attribute field encodes ranking and display information about the token occurrence, e.g. if the occurrence was in a title, anchor text, or normal text. This attribute field is similar to the fancy postings used in [4], and is used at query time for ranking purposes. Attributes are typically one byte in size, although we allow arbitrary sizes. The index can be viewed as a collection of posting lists, one for each token. Each posting list is a set of postings ordered by position.

### 3.3 The DeltaStore and DeltaIndex

In order to make new documents and updates to existing documents appear as soon as possible in search results, Trevi uses a DeltaStore and a DeltaIndex. The DeltaStore is used to accumulate changes to the Store, while the DeltaIndex is an index of the documents in the DeltaStore. The DeltaStore and DeltaIndex basically mirror the structure and functionality of the Store and Index, respectively.

The index construction algorithm used to build the DeltaIndex is similar to the one used to build the Index. In particular, the DeltaIndex is reconstructed from scratch each time. However, one key difference is that GA is not done when the DeltaIndex is constructed, since it would take too long. The result of this is that, although the DeltaIndex enables updates to appear in search results as soon as possible, it does not eliminate the need for a fast index build process that includes GA.

## 4 Global Analysis

In this section we describe the main computations performed by GA. These are duplicate detection, anchor text processing, and static ranking. All of these have been discussed at length in the literature [5, 6, 14, 15, 23], so we only highlight how they are implemented in Trevi. The main focus of this section is to provide background for the index build algorithm presented in Sections 5 and 6.

### 4.1 Duplicate Detection

Duplicate detection identifies and discards duplicate documents. During tokenization, each document is annotated with a fingerprint, which is computed by hashing the document's content. This information is put in the *Fingerprint Table*. Duplicate detection uses a union-find algorithm [10] on fingerprints to identify groups of documents with the same (or nearly the same) content.

After identifying groups of similar documents, duplicate detection picks a *master* document for each group. In order to avoid duplicate answers to search queries, only master documents are indexed by Trevi. There are several heuristics that can be used to pick the masters, such giving priority to documents with shorter URLs. The output of duplicate detection is the *Dup Table*, which assigns a master to each document in the Store. Given the URL hash of a document $D$, the Dup Table can be used to determine if $D$ is a master or a duplicate.

### 4.2 Anchor Text Processing

We define the anchor text of a document $D$ as the collection of text contained in anchors that point to $D$ from other documents. For example, if a given document points to http://trevi.ibm.com with text "search engine", then "search engine" is part of the anchor text for http://trevi.ibm.com. Several studies have shown that indexing anchor text significantly improves search quality [11, 15]. The intuition is that, very often, anchor text resembles end-user queries [18].

Trevi's anchor text algorithm extracts the links and their surrounding text from all the documents in the Store and puts them in the *Link Table*. The data kept for each link includes the link's source URL hash, its destination URL hash, and its associated anchor tokens. The Link Table is sorted and then aggregated on the destination URL hash to create a virtual anchor document for each destination. These anchor documents are written sequentially to the *AnchorStore*. A separate storage area is created to avoid updating the Store in-place, which would require seeking all over disk.

### 4.3 Static Ranking

Trevi assigns a static rank to each document in the Store. Currently, the static rank of a document $D$ is simply set to the the number of different hosts that point to $D$, i.e., the hostcount. The higher the hostcount, the higher the static rank. Although more sophisticated techniques for assigning static rank are available, such as PageRank [23], hostcounts are easy to compute and have produced satisfactory results on IBM's intranet. To compute static ranks, Trevi simply runs a count on the sorted Link Table.

The result of the static rank computation is the *Rank Table*, which is a mapping from the URL hash of a document to its static rank. Given the URL hash of a document $D$, the Rank Table can be used to determine $D$'s static rank. As in many search engines, docID is synonymous with static rank in Trevi. This effectively puts Trevi's inverted indexes in static rank order, making it possible to terminate top-k queries early [19].

## 5 The Index Build Algorithm

In this section we first present the index build algorithm at a high level, ignoring GA computations. We then describe a straightforward way to incorporate GA computations, which requires two passes over the data being indexed. Finally, we present an index build algorithm that uses lagging GA, allowing just one pass over the data being indexed.

### 5.1 The Basic Index Build Algorithm

The basic high-level flow of Trevi's index build algorithm is illustrated in Figure 2(a). The algorithm takes the current version of the Store, that is, $Store_{i+1}$ and merges it with the current version of the DeltaStore to generate the new version of the Store and the new Index, that is, $Store_{i+1}$ and $Index_{i+1}$. The Store and Index always move together in time this way, with

$Index_{i+1}$ over $Store_{i+1}$. By generating a new version of the Store each iteration we get to garbage collect the Store, keeping it sequential on disk and free of old, deleted, or duplicate documents. Since the performance of index build is proportional to the time to scan the Store, this greatly improves performance.

The DeltaStore and the DeltaIndex also move together in time, but at a faster clip than the Store and the Index. As shown in Figure 2(b), $DeltaStore_j$ is merged with newly crawled documents to generate $DeltaStore_{j+1}$ and $DeltaIndex_{j+1}$. Note that newly crawled documents in Figure 2 (b) are analogous to the DeltaStore in Figure 2 (a). Consequently, the underlying algorithm for both figures is effectively the same.

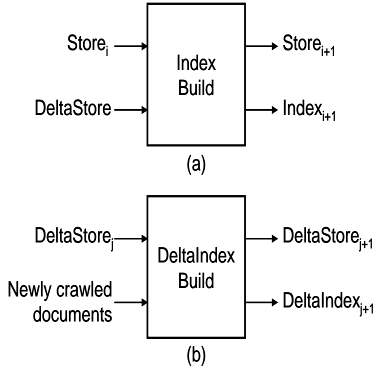


Figure 2: High-level view of (a) index build and (b) DeltaIndex build

It is important to note that the performance of the algorithms for constructing both the Index and the DeltaIndex benefit greatly from Trevi's dual-array storage architecture. For example, while the $Store_i$ and DeltaStore are sequentially read from one disk array, $Store_{i+1}$ and $Index_{i+1}$ can be written in parallel to the other array.

### 5.2 Incorporating Global Analysis

We now provide a more detailed description of the index build algorithm that incorporates GA. The naive approach to do this is:

1. *Duplicate Elimination*: Scan the Store to generate the Fingerprint Table, which maps URL hashes to fingerprints and is input to duplicate detection. When the Fingerprint Table is ready, duplicate detection is performed, generating the Dup table.

2. *Anchor Text Extraction*: Scan the Store to extract links from each document. Once all links are extracted, they are saved in the Link Table. The anchor text algorithm then generates the AnchorStore from the Link Table.

3. *Static Ranking*: Use the Link table to assign static ranks to documents in the Store and to generate the Rank Table. Note that no sort is needed in this phase since the Link Table was already sorted for anchor text extraction.

4. *Index Construction*: Scan the Store and DeltaStore to construct the Index. The tables generated during the GA phases are needed as input for index construction to remove duplicates, index anchor text, and obtain each document's static rank.

5. *Creation the New Store*: Scan the Store and DeltaStore to create the next generation of the Store. After this step, the DeltaStore and the DeltaIndex are reset.

Clearly, this naive approach is inefficient, requiring four passes over the Store. But we can easily reduce the number of passes to just two. First, observe that a single Store scan can be used to generate the input tables for duplicate elimination and anchor text extraction. Then if we are clever about the way we scan the Store, we can also combine index construction with the creation of the next generation of the Store.

These optimizations are illustrated in Algorithm 1. In the algorithm, a subscript is used with each data structure to denote which version of the Store it reflects. For example, $Rank_{i+1}$ corresponds to the Rank Table for $Store_{i+1}$.

**Algorithm 1: Straightforward Index Build**

---

1. Scan $Store_i$ and the $DeltaStore$ to generate $FingerPrint_{i+1}$ and $Link_{i+1}$
2. $Dup_{i+1}$ = duplicate detection($FingerPrint_{i+1}$)
3. $AnchorStore_{i+1}$ = anchor text processing($Link_{i+1}$)
4. $Rank_{i+1}$ = static ranking($Link_{i+1}$)
5. $GA_{i+1}$ = $Dup_{i+1}$, $AnchorStore_{i+1}$, and $Rank_{i+1}$
6. Scan $Store_i$, the $DeltaStore$, and using $GA_{i+1}$, generate $Store_{i+1}$ and $Index_{i+1}$

---

As shown, in step 1, the algorithm scans $Store_i$ and the DeltaStore to generate the inputs for GA, which runs in steps 2-5 and produces $GA_{i+1}$. In step 6, the algorithm scans the stores again and using $GA_{i+1}$ generates $Store_{i+1}$ and $Index_{i+1}$. After Algorithm 1 finishes, the DeltaIndex build process is resumed. It cycles at its own rate, generating a new DeltaStore and DeltaIndex each cycle.

### 5.3 Index Build with Lagging Global Analysis

The performance of index build is largely bound by the time to do a disk scan of the Store and to perform GA. Two disk scans of the Store are required in Algorithm 1, one to generate $GA_{i+1}$ and another to generate $Store_{i+1}$ and $Index_{i+1}$. The number of scans can be reduced to just one by using $GA_i$ to generate $Index_{i+1}$ rather than $GA_{i+1}$, that is, by using *lagging GA*. Lagging GA causes some loss of index precision,

but, as we will show, the loss is negligible because information like the static rank of a document does not change drastically from generation $i$ to $i+1$. Index build with lagging GA is illustrated in Algorithm 2.

**Algorithm 2: Index Build with Lagging GA**

---

1. Scan $Store_i$, the *DeltaStore*, and using $GA_i$ as input, generate $Store_{i+1}$, $Index_{i+1}$, $FingerPrint_{i+1}$, and $Link_{i+1}$
2. $Dup_{i+1}$ = duplicate detection($FingerPrint_{i+1}$)
3. $AnchorStore_{i+1}$ = anchor text processing($Link_{i+1}$)
4. $Rank_{i+1}$ = static ranking($Link_{i+1}$)
5. $GA_{i+1}$ = $Dup_{i+1}$, $AnchorStore_{i+1}$, and $Rank_{i+1}$

---

An important point to note in Algorithm 2 is that, in step 1, both $Store_{i+1}$ and $Index_{i+1}$ are generated using the same disk scan of $Store_i$. Another important point to note is that, because $GA_i$ is used in step 1, new documents will have no static rank computed for them when they are indexed. New documents are assigned a low, default static rank, under the assumption they will have few incoming links. The next iteration of index build will fix this if it is not true. Finally, note that $Index_{i+1}$ is ready after step 1, enabling DeltaIndex construction to run in parallel with $GA_{i+1}$. This in turn reduces the cycle time of index build, as illustrated in Figure 3.

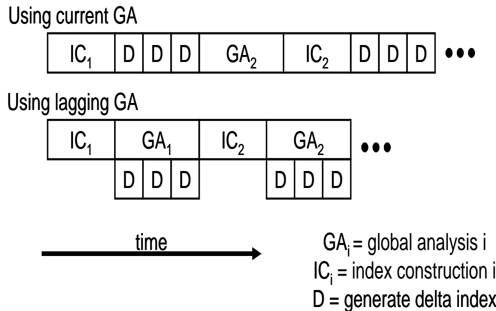


Figure 3: Index build cycle time using the current GA versus lagging GA

Algorithm 2 has to be run twice initially, with the appropriate inputs set to null, to generate $Index_0$. More specifically, bootstrapping begins with $Store_0$ and no GA.

### 5.4 Creating the Next Generation Store

This section describes how $Store_{i+1}$ is generated in step 1 of Algorithm 2 using $Store_i$, the DeltaStore, and $GA_i$ as input.

The document bundles in $Store_i$ and the DeltaStore are scanned in LIFO order to generate $Store_{i+1}$. In order to index only the most recent version of each document, a Bloom Filter [3] on URL hash is used to filter out older versions of documents. More specifically, if document $D$ in $Store_i$ has been replaced by a newer version of $D$ in the DeltaStore (call it $D'$), then only $D'$ will appear in $Store_{i+1}$. In addition, $Dup_i$ is used to eliminate duplicates. Surprisingly, eliminating duplicates reduces the size of the Store by 50% on the IBM intranet.

Figure 4 illustrates how $Store_{i+1}$ is generated. In the example shown, documents $D1$ and $D5$ have newer versions, $D1'$ and $D5'$, that appear in the DeltaStore. Therefore, the original versions of these documents are garbage collected and do not appear in $Store_{i+1}$. The DeltaStore is handled similarly. As noted earlier, to improve performance, $Store_i$ and the DeltaStore are read from one disk array, while $Store_{i+1}$ is written to the other array.

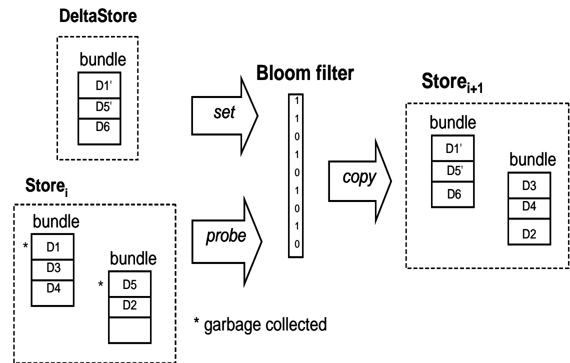


Figure 4: Generating $Store_{i+1}$

## 6 High-Performance Index Construction

This section describes how $Index_{i+1}$ is constructed in step 1 of Algorithm 2 using $Store_i$, the DeltaStore, and $GA_i$ as input. The index construction algorithm takes the tokenized documents from the Store and DeltaStore as input and produces the inverted index. Recall that the index is basically a collection of posting lists, one list for each token that appears in the corpus. Each posting is a pair *(position, attribute)*, and each position in turn consists of a *(docID, offset)*. To reduce storage and I/O costs, posting lists are compressed using a simple variable-byte scheme based on computing the deltas between positions.

Trevi's index construction is based on sorting. Document tokens are streamed into a sort, which is used to create the posting lists. The primary sort key is on token, the secondary sort key is on docID (static rank), and the tertiary sort key is on offset within a document. By sorting on this compound key, token occurrences are effectively grouped into ordered posting lists.

The well-known sort-merge technique [1, 29] is used for sorting. Sort-merge is particularly suitable for

batch index construction, where all of the data is indexed at once, which is true in most search engines. Sort-merge has two main phases in Trevi's index construction:

1. *Sort Phase*: Scan $Store_i$ and the DeltaStore, streaming document tokens into a memory buffer. Each time the memory buffer fills, it is sorted and then written to disk in compressed form. This process is repeated until there are no more documents to scan.

2. *Merge Phase*: Create a memory heap to merge the sorted runs. Perform a multi-way merge of the runs to generate the final compressed posting lists.

We made two important optimizations that greatly improved the performance of these phases. First, we used a highly tuned radix sort [10, 26] to generate sorted runs. Second, we used a pipelined software architecture to enable I/O and CPU to be overlapped. We describe each of these optimizations below.

### 6.1 Using Radix Sort

Radix sort has two important characteristics that Trevi exploits to improve performance: it sorts in linear time and is stable, meaning it preserves the original order of input data when there are ties on the sort key. However, radix sort requires fixed-length keys, whereas text tokens are variable length. Therefore, tokens need to be transformed into fixed-length *tokenIDs*.

Transforming tokens to tokenIDs can be done in a variety of ways – for example, by generating unique sequential IDs or by using a hash function. The disadvantage of generating unique IDs is that a potentially large map of tokens to IDs needs to be kept. Hashing does not suffer from this problem, but there may be collisions when two or more tokens hash to the same tokenID. However, by using enough bits in the hashing function, the probability of a collision can be brought to nearly zero, which is good enough for a search engine. So we opted for hashing in Trevi, using a 64-bit Pearson's hash function [24]. On the IBM intranet, which has over 260 million unique tokens, there were no collisions using this hash function.

Using tokenID's, the fixed-length sort keys for index construction have the form *(tokenID, docID, offset)*, where tokenID is a 64-bit hash value, docID is 32 bits, and offset within a document is 32 bits. The encoding of the sort key is illustrated in Figure 5. As shown, the upper bit in the offset is used to denote the section of a document. This is so a given document can be streamed into the sort in different sections. More will be said about this shortly. Note that, by sorting the full 128-bit key, we are able to simultaneously:

- Group tokens into posting lists.
- Order each posting list by docID, effectively putting it in static rank order.
- After docID, order each posting list by the offset within a document and bring different sections of a document together.

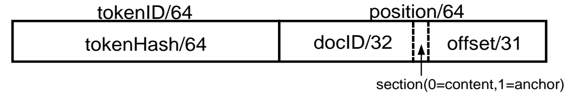


Figure 5: Sort Key

To index anchor text tokens, the AnchorStore is scanned and streamed into the sort after the Store and the DeltaStore are scanned. The section bit of the offset is used to indicate whether a token is for content or anchor text. Anchor text tokens have their section bit set to "1". Consequently, after sorting, the anchor text tokens for a document $D$ follow the content tokens for $D$. By using more section bits, this approach could be generalized, allowing documents to have multiple sections, with each section stored separately.

Within a document $D$, tokens are streamed into the sort in the order in which they appear in $D$, that is, in offset order. Taking advantage of the fact that radix sort is stable, this allows us to use a 96-bit sort key that excludes offset, rather than sorting on the full 128-bit key. Note that if posting lists did not need to be put in static rank order, a 64-bit radix sort on just tokenID would be sufficient.

Trevi's radix sort was implemented using a 16-bit radix, so for the 96-bit sort key, this requires six linear passes through the data to accumulate the radix counters. With a 16-bit radix, we needed radix counter tables that are able to hold 65K 32-bit integer values. Experiments showed that using this large radix was faster than using a smaller one because fewer passes through the data were required.

Note that we can incorporate new sort criteria in Trevi by simply changing its sort key. So by using a key-based radix sort for index construction, we obtain both high performance and flexibility. This is in contrast to an approach based on accumulating postings using a dictionary [17], which would require an additional sort on each posting list to incorporate new sort criteria.

### 6.2 Pipelining the Merge-Sort

Overlapping I/O and CPU is the key to good sorting performance. This is accomplished by using a software pipeline to create sort runs, as shown in Figure 6. Although it is not shown, we also use a two-stage pipeline for the merge phase, with one stage reading and merging runs from disk, and the other stage compressing and writing the final postings to disk. Our

work here follows from [22], except that we replaced a comparison-based sort for each run with a more efficient radix sort. In addition, we used a two-stage pipeline rather than a three-stage pipeline to create sort runs. We found that the flush stage was so fast compared to the other stages that combining it with the radix sort resulted in a more balanced pipeline.

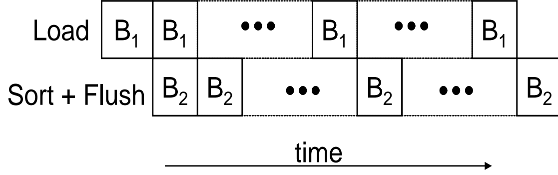


Figure 6: The pipeline for generating sort runs; $B_1$ and $B_2$ are the sort buffers

Determining an optimal sort buffer size was a key contribution of [22]. They found that the buffer size should be neither too small nor too large – a balance has to be struck. This is intuitive because, on the one hand, a larger buffer improves I/O efficiency. But on the other hand, comparison-based sorting is nonlinear, which means sorting can become a CPU bottleneck if the sort buffer is too large.

With radix sort, there is no balance that needs to be struck, since the time to sort a run is linear in the buffer size. Moreover, radix sort is so fast that the fraction of time spent in the merge phase becomes larger. The time spent in the merge phase is $O(S \log N)$ where $S$ is the total size of the data and $N$ is the number of runs. These observations imply that we want as large a sort buffer as possible – to maximize I/O efficiency and to minimize $N$. We ran experiments to verify this, as shown in Figure 7. Note that, for Trevi's two-stage pipeline to work, the sort buffer is actually split into two equal-sized sub-buffers.

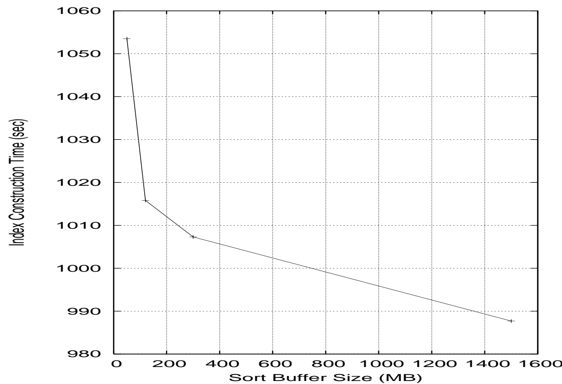


Figure 7: Index construction time with different sort buffer sizes and 600,000 documents

We consciously tried to balance the work in the pipeline stages. For example, instead of accumulating the radix counters as tokens were loaded into the sort buffer, we performed this computation in the Sort+Flush stage because it resulted in a more balanced pipeline.

We can calculate the maximal speedup of the merge-sort from pipeling. First consider the sort phase. Let $B_i$ denote the sort buffer size. The Load stage is linear in the buffer size and takes time $\lambda B_i$. The Sort+Flush stage is also linear in the buffer size and takes time $\phi B_i$. Thru experimentation, we found that $\lambda = 1.20$ and $\phi = 1.00$. In other words, the Load stage, which includes a fair bit of per-token processing, was actually more costly than the Sort+Flush stage. Based on our values for $\lambda$ and $\phi$, the theoretical speedup of the sort phase is the serial execution time divided by the pipelined execution time, that is, $(1.20 + 1.00)/1.20$ or 1.83. The merge phase tends to be I/O bound, and intuitively has a speedup of at most 2.0.

The fraction of time spent in the merge phase was found to be 45%. Hence, the maximal speedup of the merge-sort from pipelining is $0.55 \cdot 1.83 + 0.45 \cdot 2.0$ or 1.91. Therefore, we should be able to effectively use two CPUs and no more. Experimentally, we saw a speedup of about 1.30 using two CPUs. Using two CPUs also provides a nice balance with Trevi's dual-array storage architecture.

### 6.3 Aggregating Token Occurrences

Aggregating all the occurrences of a particular token within a document $D$ just once when $D$ is tokenized and stored can speed up index construction [4]. Although the volume of data to be sorted is not dramatically reduced, fewer individual tokens need to be sorted, which in turn reduces CPU usage. In all of our experiments, unless otherwise noted, we used this approach.

## 7 Experimental Results

In this section, we present experimental results from the Trevi search engine. Results are provided for Trevi's index build process, showing that lagging GA can improve the performance of index build without sacrificing search quality. Results are also provided for Trevi's index construction algorithm, showing that it is significantly faster than alternative approaches described in the research literature. We ran all our experiments on a two-way SMP with dual 2.4 Ghz Intel Xeon processors running Linux. The disk storage was configured as two physical RAID0 arrays, each with 6 drives. We were able to read from one disk array and write to the other array at a rate of 95MB/s.

### 7.1 Global Analysis

In these experiments, we constructed several generations of the index, starting from a 3.5 million document Store and adding 500,000 documents each generation.

Our data set was based on a partial crawl of IBM's intranet. The *change interval* of 500,000 was chosen because that is the daily rate of change we see in IBM's intranet. This includes changes to existing documents and newly crawled documents.

In our first set of experiments, we used the Kendall's tau distance for top $k$ lists by Fagin et. al. [16] to measure the *discrepancy* in static rank between different generations of the index, i.e., how static rank changes from $Index_i$ to $Index_j$. Similar experiments were carried out for anchor text. Given two ordered lists, Kendall's tau computes a similarity measure by checking every possible pair $\{i, j\}$ of items in the two lists and applying a penalty whenever the order of items $i$ and $j$ differ in the input lists. We use a normalized version of the measure that scales the values to be between 0 (when the lists are identical) and 1 (when the lists are in the opposite order). This measure has been widely used before in different contexts, such as index pruning [9] and rank aggregation [13].

Figure 8 shows how the discrepancy measure changes for the documents with the top 100,000 static ranks. The bottom curve shows how the ranks vary from generation $i$ to $i + 1$, while the top curve one shows how the ranks vary over time, comparing the ranks in generation 1 with the ranks in generation $i$.

Figure 8 shows that, in the steady state, the top 100,000 static ranks differ by no more than 2% between two consecutive generations, and by up to 18% from the start after 7 generations. This means that by using lagging static ranks we lose less than 2% accuracy on static ranks if a 500,000 change interval is used to trigger the construction of a new index. This graph also shows that the loss of accuracy in the static rank grows linearly with the number of changed documents that accumulate between generations. Another interesting observation is that the difference between the static ranks in two consecutive generations decreases over time, from about 6% to about 1.5%.

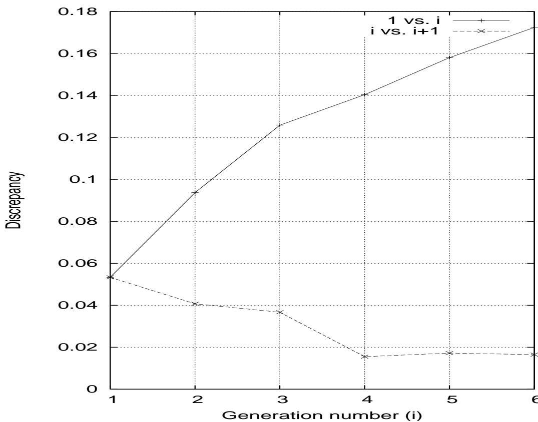


Figure 8: Discrepancy between the top 100,000 static ranks in different index generations

One of the reasons for this behavior is that in intranets the crawl date is a very good static rank, as described in [15]. This means that the documents with the highest static rank tend to be crawled first since they are closer to the intranet's "root". After the first few index generations, when most of the "important" documents are already in the Store, the ranks are quite stable.

Another characteristic of the data is the Zipfian distribution of the static ranks. Figure 9 shows the distribution of the ranks on a log-log scale. It is interesting to notice that after 10,000 documents all the static rank values are extremely low and they are 0 after 600,000 (this point is not shown in the graph). This means that after the most important documents have been crawled, the discrepancy in the top static ranks tends to be very low. This also explains why the bottom curve of Figure 8 decays and stabilizes after a few generations.

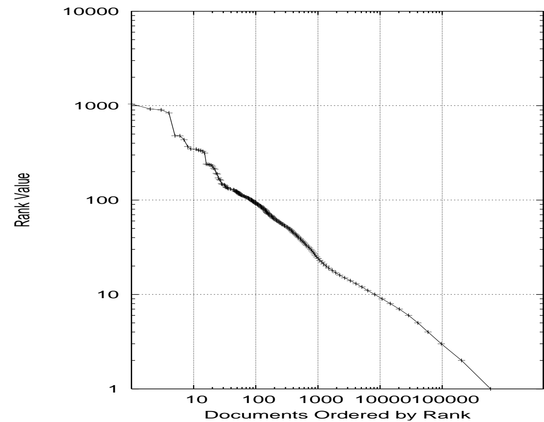


Figure 9: Distribution of the static ranks for the IBM intranet

Figure 10 shows similar discrepancy results for anchor text. We generated lists of the 100,000 most frequent anchor tokens for every index generation and applied the same discrepancy measure as before. The graphs for anchor text and ranks exhibit a similar shape and range of discrepancy values. The difference between consecutive generations is between 2% and 4% and after the 7 generations it is still less than 14%. The main point here is that both static ranks and anchor texts are very stable between consecutive generations for the change interval we considered.

The use of lagging duplicate detection might cause a potential loss of precision. This is due to the fact that, if we use lagging information, all documents added to the Store between generation $i$ and $i + 1$ are added to $Index_{i+1}$, even if they are duplicates and should be filtered out. If there are no duplicates among the documents added to the Store in a given generation there is no degradation in quality, since no duplicates are indexed. On the other hand, if there are duplicates

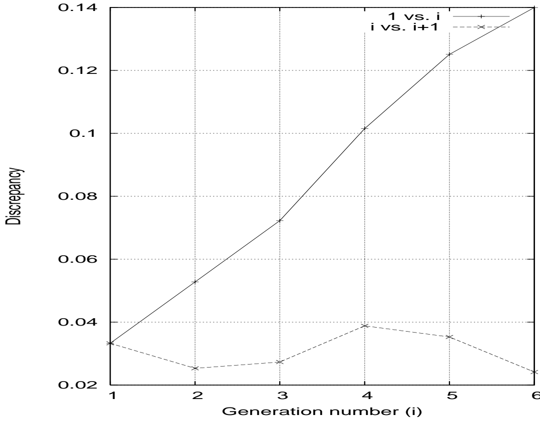


Figure 10: Discrepancy between the top 100,000 anchor text tokens in different index generations

added to the Store there can be a degradation in quality, since there is a chance that the index might return duplicate documents in the results of a query.

We analyze this in Figure 11 by comparing the ratio of duplicate documents added in a generation to the total number of documents. The graph shows that, if we use lagging GA, only 2.6% to 5.1% of the documents added to the Store are incorrectly classified by duplicate detection. Moreover, these are new documents that are assigned a low, default static rank. Consequently, this makes it highly unlikely that duplicate documents will be returned in search results, unless the query is very specific and not enough highly ranked documents appear in the results.

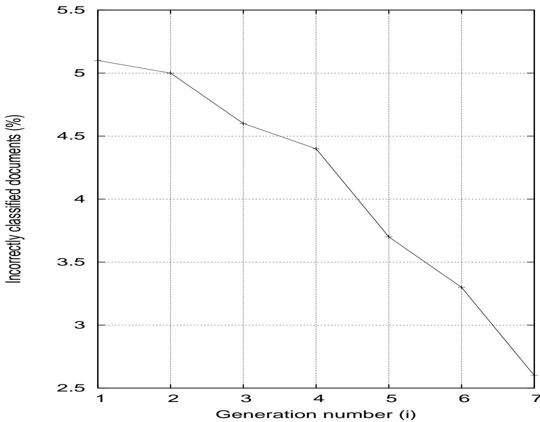


Figure 11: Analysis of the duplicate detection for several index generations

We also ran a quality test by selecting 180 queries from the Trevi query log, manually identifying relevant documents for these queries. We measured the average precision at 1 and at 10 over all the queries for the different index generations. The results were stable, varying less than 2% in both cases over the 7 generations. Precision at 1 varied from 0.639 to 0.650 while precision at 10 varied from 0.215 to 0.219. This shows that no precision is lost over the generations due to the use of lagging GA.

Figure 12 shows the difference in the time to perform index build with and without lagging GA. Note that that the time to scan the Store and run GA, which is the difference between the two curves, grows linearly with the number of documents. The improvement in index build from lagging GA depends on the complexity of GA. In Trevi, GA is very simple and yet the performance gains are significant, on the order of 25%. In the extreme case, GA could be orders of magnitude slower than index construction. For that scenario, Trevi's index build algorithm could be modified to let GA lag for two or more generations to reduce its impact. Another advantage of using lagging GA, which is not reflected in Figure 12, is that it allows delta indexes to be built in parallel with GA.

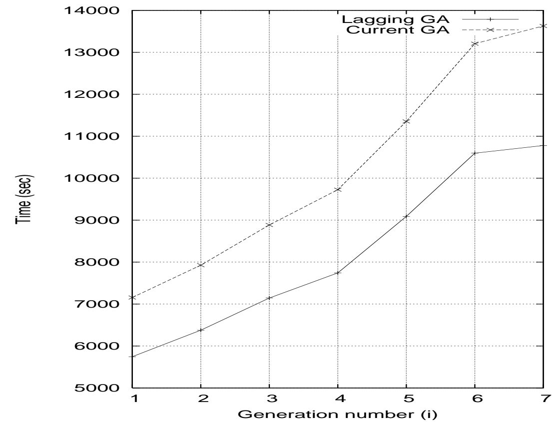


Figure 12: Performance improvement in index build from lagging GA

### 7.2 The Performance of Index Construction

We begin this section by looking at how Trevi's index construction algorithm scales. In Figure 13, we see that it scales nearly linearly with the index size. Our time complexity is actually $O(S \log N)$ where $S$ is the total size of the data and $N$ is the number of runs, but since we use very large sort buffer sizes (1.5GB by default), $N$ tends to be small.

All the points in Figure 13 correspond to a partial crawl of the IBM intranet except for the last point, which corresponds to a full 10,666,580 document crawl. The crawl included 6,515,728 unique documents after duplicate elimination plus 4,150,852 anchor documents. Only the last point in the graph includes anchor documents, which tend to be smaller than the average document.

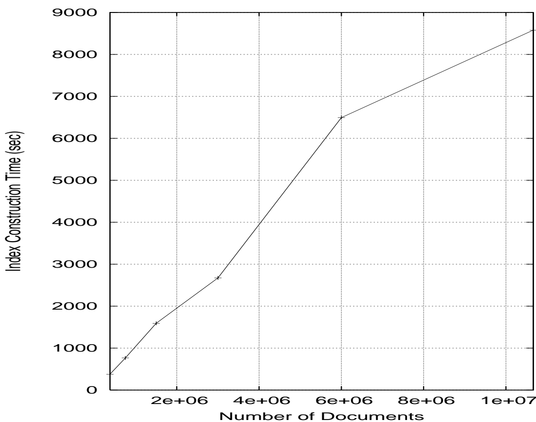


Figure 13: Nearly linear scaling of index construction

#### 7.2.1 Comparison to Other Index Construction Algorithms

It is difficult to compare Trevi's index construction algorithm to alternative approaches described in the research literature. This is because different index features may have been supported, or a different hardware platform was used, or different input data was used, and so on. Nonetheless, we still tried to carry out a comparison in as fair a manner as possible. All of the results shown below are from constructing an index on a partial crawl of IBM's intranet with 2,000,000 documents. The document collection did not include anchor text documents, and was about 23GB. The tokenized documents included stemmed and unstemmed tokens.

- *Trevi with Token Aggregation*: 4,000,000 docs/hour. This was using Trevi's default index construction algorithm, where all the occurrences of a particular token in a document $D$ are aggregated when $D$ is tokenized and stored, as described in Section 6.3. Postings contain full position information, a one-byte attribute, and they are ordered by rank. The final index size was 10.5GB.
- *Trevi without Token Aggregation*: 2,800,000 docs/hour. This was using Trevi's default algorithm but without aggregated token occurances.
- *Trevi without Attributes*: 3,100,000 docs/hour. This is using Trevi's default algorithm but without the one-byte attribute in postings to try and be more comparable with Lucy (see below).
- *Lucy*: 3,000,000 docs/hour This was using Lucy [21] on our hardware and our data set. Lucy is a C-based open source text index from RMIT University. At this time, we know of no papers published describing Lucy, but it does not appear to order postings by rank or support attributes in postings, so it had considerably less work to do than Trevi. Lucy's index was 7.3GB, which was smaller than Trevi's index because it stored much less information per posting. There were 2.7 billion postings in the index, so if a one-byte attribute was added to every posting, Lucy's index size would be have been 10.0GB, which is nearly identical in size to Trevi's index.
- *Melnik et al.*: 1,100,000 docs/hour. This was the result reported in [22]. Experiments were run on a 350-500 Mhz PC with 300-500MB of RAM and multiple IDE disks. Their index does not appear to contain per-token attributes or document offsets, and postings are not ordered by rank. Also, the result reported seems to only include the time to write sorted runs and excludes the time to merge them. When we obtained an executable version of their algorithm and ran it on our hardware and our data set, we obtained 600,000 docs/hour.
- *Long and Suel*: 700,000 docs/hour. This was the result reported in [19]. Experiments were run on a Dell Optiplex 240GX, 1.6Ghz Pentium 4, with 1GB RAM and two 80GB Seagate Barracuda hard disks. Postings are ordered by rank and contain attribute information, and most likely also position information. They actually had a larger data set and ran 16 machines in parallel.
- *Lucene*: 350,000 docs/hour. This was using Lucene [20] on our hardware and our data set. Lucene is an open source Java-based index [20]. Postings do not contain attribute information and are not ordered by rank.

## 8 Related Work

Although several papers have focused on specific GA algorithms [2, 5, 6, 14, 15, 23], to the best of our knowledge, this is the first paper that describes how to integrate these algorithms into a complete index build process. In [30], the authors describe how to design stable rank algorithms. That work is relevant to this paper since Trevi's index build process relies on the fact that document ranks should be stable across consecutive generations of its index. Another related work [19], describes how to use static rank ordering in an inverted index to improve query performance. However, the authors did not provide any details about how their index is actually construction.

Brin and Page [4] discuss an early version of the Google search engine. The authors describe a system consisting of a crawler, document repository, GA (PageRank), and index construction. However, they do not discuss how frequently GA is computed or how tightly coupled it is in their index build process. Their index construction algorithm is also not described in as much detail has here.

Witten, Moffat, and Bell [29] describe a sort-based index construction algorithm that saves temporary disk space by using an in-place merge algorithm. A table mapping index blocks to their location on disk was maintained, and an extra pass over the blocks is needed to shuffle them into sorted order. The drawback to this scheme is that it generates more I/O, and in particular more random I/O, during index construction to save disk space, which has become an inexpensive commodity. In contrast, Trevi's index construction maximizes performances without worrying about temporary disk space. It tries to do as much sequential I/O as possible, reading from one disk array while writing to the other.

Heinz and Zobel [17] describe an index construction algorithm that uses an in-memory lexicon to accumulate compressed posting lists. When memory is used up, the posting lists are dumped to disk in lexicographic order and the process is repeated. When all the documents have been processed, an in-place merge is used to produce the final results. Their index construction algorithm differs from the one described here in that they use an in-place merge, they do not attempt to order posting lists in rank order, they do not allow documents to be indexed in sections, and they do not include per-posting attribute information.

Sinha and Zobel [27] describe a new index construction algorithm based on dynamic tries. They provide results showing that their algorithm performs better than one based on radix sorting for large collections of strings. However, it is unclear whether their algorithm could be adapted to efficiently build the kind of rank-ordered posting lists described here.

## 9 Conclusions

The time to construct an inverted index is an important issue in web and intranet search engines [1, 17, 22, 29] But constructing the inverted index is only the final step in a more complicated *index build process*, which includes a global analysis (GA) of all the data being indexed to compute measures like PageRank [23]. In this paper, we showed how the use of slightly outdated GA information, that is *lagging GA*, and a fast index construction algorithm based on radix sorting can be combined in a novel way to speed up the index build process without sacrificing search quality.

We presented experimental results from the Trevi search engine, which is currently used to support all the searches on IBM's global intranet. Results show that the use of lagging GA does not compromise search quality and can reduce the time of the index build process by 25% or more. If the complexity of GA increases, perhaps due to clustering or data mining, the use of lagging GA will result in even greater time savings. Results also showed that index construction using a pipelined radix sort can outperform alternative approaches by 33% or more.

## References


[1] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, New York, NY, 1999.

[2] Ziv Bar-Yossef and Sridhar Rajagopalan. Template detection via data mining and its applications. In *Proceedings of the International World Wide Web Conference, WWW2002*, 2002.

[3] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *WWW7 / Computer Networks 30(1-7)*, pages 107–117, 1998.

[5] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *Combinatorial Pattern Matching, 11th Annual Symposium*, pages 1–10, 2000.

[6] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *WWW6 / Computer Networks 29(8-13)*, pages 1157–1166, 1997.

[7] Eric William Brown. Execution performance issues in full-text information retrieval. Technical report, University of Massacusetts, Amherst, MA, February 1996. Ph.D. Thesis.

[8] E.W. Brown, J.P. Callan, and W.B. Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, pages 192–202, Santiago, Chille, September 1994.

[9] David Carmel, Doron Cohen, Ronald Fagin, Eitan Farch, Michael Herscovici, Yoelle S. Maarek, and Aya Soffer. Static index pruning for information retrieval systems. In *Proceedings of 24th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '01)*, pages 43–50, New Orleans, Louisiana, USA, September 2001.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 2003.

[11] Nick Craswell, David Hawking, and Stephen Robertson. Effective site finding using link anchor information. In *Research and Development in Information Retrieval*, pages 250–257, 2001.

[12] Stephen Dill, Nadav Eiron, David Gibson, Daniel Gruhl, R. Guha, Anant Jhingran, Tapas Kanungo, Sridhar Rajagopalan, Andrew Tomkins, John A. Tomlin, and Jason Y. Zien. Semtag and seeker: bootstrapping the semantic web via automated semantic annotation. In *Proceedings of the Twelfth International World Wide Web Conference, WWW2003*, May 2003.

[13] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the web. In *Proceedings of the Tenth International World Wide Web Conference (WWW 10)*, pages 613–622, Hong Kong, China, May 2001.

[14] Nadav Eiron and Kevin S. McCurley. Analysis of anchor text for web search. In *SIGIR Conference*, pages 459–460, 2003.

[15] Ronald Fagin, Ravi Kumar, Kevin S. McCurley, Jasmine Novak, D. Sivakumar, John A. Tomlin, and David P. Williamson. Searching the workplace web. In *Proceedings of the Twelfth International World Wide Web Conference, WWW2003*, May 2003.

[16] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing top k lists. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA '03)*, pages 28–36, Baltimore, MD, USA, January 2003.

[17] Steffen Heinz and Justin Zobel. Efficient single-pass index construction for text databases. *JASIST*, 54(8):713–729, 2003.

[18] Reiner Kraft and Jason Zien. Mining anchor text for query refinement. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW 10)*, page (to appear), New York, NY, May 2004.

[19] Xiaohui Long and Torsten Suel. Optimized query execution in large search engines with global page ordering. In *Proceedings of 29th International Conference on Very Large Databases (VLDB 2003)*, pages 129–140, Berlin, Germany, September 2003.

[20] Lucene. http://jakarta.apache.org/lucene/.

[21] Lucy. http://www.seg.rmit.edu.au/lucy.

[22] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. In *World Wide Web*, pages 396–406, 2001.

[23] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[24] Peter K. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33(6):677–680, June 1990.

[25] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[26] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley Publishing Company, Boston, MA, 1998.

[27] Ranjan Sinha and Justin Zobel. Cache-conscious sorting of large sets of strings. In *Proceedings of the ALENEX Workshop on Algorithm Engineering and Experiments*, pages 93–105, 2003.

[28] Anthony Tomasic, Héctor García-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. pages 289–300, 1994.

[29] Ian Witten, Alistair Moffat, and Timoty Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1999.

[30] Alice X. Zheng, Andrew Y. Ng, and Michael I. Jordan. Stable algorithms for link analysis. In *Proceedings of 24th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '01)*, pages 258–266, New Orleans, Louisiana, USA, September 2001.