# Inverted Files for Text Search Engines

JUSTIN ZOBEL

*RMIT University, Australia*

AND

ALISTAIR MOFFAT

*The University of Melbourne, Australia*

The technology underlying text search engines has advanced dramatically in the past decade. The development of a family of new index representations has led to a wide range of innovations in index storage, index construction, and query evaluation. While some of these developments have been consolidated in textbooks, many specific techniques are not widely known or the textbook descriptions are out of date. In this tutorial, we introduce the key techniques in the area, describing both a core implementation and how the core can be enhanced through a range of extensions. We conclude with a comprehensive bibliography of text indexing literature.

Categories and Subject Descriptors: H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File orzanization*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Query formulation, retrieval models, search process*; D.4.3 [**Operating Systems**]: File Systems Management—*Access methods, file organization*; E.4 [**Coding and Information Theory**]—*Data compaction and compression*; E.5 [**Files**]—*Organization / structure*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*; I.7.3 [**index Generation**]

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Inverted file indexing, document database, text retrieval, information retrieval, Web search engine

## 1. INTRODUCTION

Text search is a key technology. Search engines that index the Web provide a breadth and ease of access to information that was inconceivable only a decade ago. Text search has also grown in importance at the other end of the size spectrum. For example, the help services built into operating systems rely on efficient text search, and desktop search systems help users locate files on their personal computers.

---

The Australian Research Council and the Victorian Partnership for Advanced Computing have provided valuable financial support as have our two home institutions.

Authors' addresses: J. Zobel, School of Computer Science & Information Technology, RMIT University, Australia; email: jz@cs.rmit.edu.au; A. Moffat, Department of Computer Science & Software Engineering, The University of Melbourne, Australia.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
©2006 ACM 0360-0300/2006/07-ART6 $5.00 DOI: 10.1145/1132956/1132959 http://doi.acm.org/10.1145/1132956.1132959



المنارة للاستشارات
www.manaraa.com

Search engines are structurally similar to database systems. Documents are stored in a repository, and an index is maintained. Queries are evaluated by processing the index to identify matches which are then returned to the user. However, there are also many differences. Database systems must contend with arbitrarily complex queries, whereas the vast majority of queries to search engines are lists of terms and phrases. In a database system, a match is a record that meets a specified logical condition; in a search engine, a match is a document that is appropriate to the query according to statistical heuristics and may not even contain all of the query terms. Database systems return all matching records; search engines return a fixed number of matches, which are *ranked* by their statistical similarity. Database systems assign a unique access key to each record and allow searching on that key; for querying on a Web collection, there may be many millions of documents with nonzero similarity to a query. Thus, while search engines do not have the costs associated with operations such as relational join, there are significant obstacles to fast response, that is, a query term may occur in a large number of the documents, and each document typically contains a large number of terms.

The challenges presented by text search have led to the development of a wide range of algorithms and data structures. These include representations for text indexes, index construction techniques, and algorithms for evaluation of text queries. Indexes based on these techniques are crucial to the rapid response provided by the major Web search engines. Through the use of compression and careful organization, the space needed for indexes and the time and disk traffic required during query evaluation are reduced to a small fraction of previous requirements.

In this tutorial, we explain how to implement high-performance text indexing. Rather than explore all the alternative approaches that have been described (some speculative, some ineffective, and some proven in practice), we describe a simple, effective solution that has been shown to work well in a range of contexts. This *indexing core* includes algorithms for construction of a document-level index and for basic ranked query evaluation.

Given the indexing core, we then sketch some of the principal refinements that have been devised. These include index reorganization, phrase querying, distribution, index maintenance, and, in particular, index compression. The most compelling application of the indexing techniques described in this tutorial is their use in Web search engines and so we briefly review search in the context of the Web. In the context of the different approaches, we identify significant papers and indicate their contribution.

Other research topics in text search include innovations such as metasearch, compression techniques for stored documents, and improvements to fundamental technologies such as sorting, storing, and searching of sets of strings. Domains of applicability beyond text include genomic string searching, pattern matching, and proprietary document management systems. These developments are of wide practical importance but are beyond the scope of this tutorial.

This article has a slightly unusual structure. The citations are collected into Section 13 which is a critical survey of work in the area. The earlier sections are free of citations, and reflect our desire to present the material as a tutorial that might be useful for people new to the area. To allow the corresponding citations to be accessed, the structure within Section 13 of the previous sections. That is, Section 13 can be used as a "further reading" overview for each topic.

## 2. TEXT SEARCH AND INFORMATION RETRIEVAL

Search engines are tools for finding the documents in a collection that are good matches to user queries. Typical kinds of document collection include Web pages, newspaper articles, academic publications, company reports, research grant applications, manual



المنارة للاستشارات
www.manaraa.com

pages, encyclopedias, parliamentary proceedings, bibliographies, historical records, electronic mail, and court transcripts.

These collections range dramatically in size. The plain text of a complete set of papers written by a researcher over ten years might occupy 10 megabytes, and the same researcher's (plain text, non-spam) 10-year email archive might occupy 100 megabytes. A thousand times bigger, the text of all the books held in a small university library might occupy around 100 gigabytes. In 2005, the complete text of the Web was probably some several tens of terabytes.

Collections also vary in the way they change over time. A newswire archive or digital library might grow only slowly, perhaps by a few thousand documents a day; deletions are rare. Web collections, in contrast, can be highly dynamic. Fortunately, many of the same search and storage techniques are useful for these collections.

Text is not the only kind of content that is stored in document collections. Research papers and newspaper articles include images, email includes attachments, and Web collections include audio and video formats. The sizes discussed previously are for text only; the indexing of media other than text is beyond the scope of this tutorial.

*Query Modes.* In traditional databases, the primary method of searching is by key or record identifier. Such searching is rare in text databases. Text in some kinds of collections does have structured attributes such as `<author>` tags and metadata such as the subject labels used for categorizing books in libraries, but these are only occasionally useful for content-based search and are not as useful as are keys in a relational database.

The dominant mode of text search is by its *content* in order to satisfy an *information need*. People search in a wide variety of ways. Perhaps the commonest mode of searching is to issue an initial query, scan a list of suggested answers, and follow pointers to specific documents. If this approach does not lead to discovery of useful documents, the user refines or modifies the query and may use advanced querying features such as restricting the search domain or forcing inclusion or omission of specific query terms. In this model of searching, an information need is represented by a *query*, and the user may issue several queries in pursuit of one information need. Users expect to be able to match documents according to any of the terms they contain.

Both casual users and professionals make extensive use of search engines but their typical strategies differ. Casual users generally examine only the first page or so returned by their favorite search engine, while professionals may use a range of search strategies and tools and are often prepared to scrutinize hundreds of potential answers. However, the same kinds of retrieval technique works for both types of searcher.

Another contrast with traditional databases is the notion of *matching*. A record matches an SQL query if the record satisfies a logical condition. A document matches an information need if the user perceives it to be *relevant*. But relevance is inexact, and a document may be relevant to an information need even though it contains none of the query terms or irrelevant even though it contains them all. Users are aware that only some of the matches returned by the system will be relevant and that different systems can return different matches for the same query. This inexactitude introduces the notion of *effectiveness*: informally, a system is effective if a good proportion of the first $r$ matches returned are relevant. Also, different search mechanisms have different computational requirements and so measurement of system performance must thus consider both effectiveness and efficiency.

There are many ways in which effectiveness can be quantified. Two commonly used measures are *precision* and *recall*, respectively the fraction of the retrieved documents that are relevant and the fraction of relevant documents that are retrieved. There are many other metrics in use with differing virtues and differing areas of application. In



المنارة للاستشارات
www.manaraa.com

```
1   The old night keeper keeps the keep in the town
2   In the big old house in the big old gown.
3   The house in the town had the big old keep
4   Where the old night keeper never did sleep.
5   The night keeper keeps the keep in the night
6   And keeps in the dark and sleeps in the light.
```

**Fig. 1**. The Keeper database. It consists of six one-line documents.

this tutorial, our focus is on describing indexing techniques that are efficient, and we do not review the topic of effectiveness. Worth noting, however, is that the research that led to these efficient indexing techniques included demonstrations that they do not compromise effectiveness.

With typical search engines, the great majority of information needs are presented as *bag-of-word* queries. Many bag-of-word queries are in fact *phrases* such as proper names. Some queries have phrases marked up explicitly, in quotes. Another common approach is to use Boolean operators such as AND, perhaps to restrict answers to a specific language, or to require that all query terms must be present in an answer.

*Example Collections.* A sample collection, used as an example through this tutorial, is shown in Figure 1. In this `Keeper` database, only document 2 is about a big old house. But with a simple matching algorithm, the bag-of-words query `big old house` matches documents 2 and 3, and perhaps also documents 1 and 4 which contain `old`, but not the other terms. The phrase query `"big old house"` would match only document 2.

Another issue is the parsing method used to extract terms from text. For example, when HTML documents are being indexed, should the markup tags be indexed? or terms within tags? And should hyphenated terms be considered as one word or two? An even more elementary issue is whether to *stem* and *casefold*, that is, remove variant endings from words, and convert to lowercase. Choice of parsing technique has little impact, however, on indexing. On the other hand, the issue of *stopping* does affect the cost of indexing and removal of common words or function words such as `the` and `furthermore` can have a significant effect. Confusingly, in some information retrieval literature, the task of parsing is known as index term extraction or simply indexing. In this article, indexing is the task of constructing an index.

Without stemming, but with casefolding, the vocabulary of `Keeper` is:

```
and big dark did gown had house in keep keeper keeps light
never night old sleep sleeps the town where
```

Stemming might reduce the vocabulary to:

```
and big dark did gown had house in keep light never night old
sleep the town where
```

with the exact result dependent on the stemming method used. Stopping might then reduce the `Keeper` vocabulary to:

```
big dark gown house keep light night old sleep town
```

In addition to the example `Keeper` collection, two hypothetical collections are used to illustrate efficiency issues. The characteristics of these collections are shown in Table I, and, while they are not actual data sets, they are based on experience with real text. Specifically, they are similar to two types of collections provided by the TREC project run by the United States National Institute for Standards and Technology (NIST) (see `trec.nist.gov`). TREC has been a catalyst for research in information retrieval since 1992, and without it, robust measurement of the techniques described in this tutorial



المنارة للاستشارات
www.manaraa.com

**Table I.** Characteristics of Two Hypothetical Text Databases, Used as Examples in This Tutorial

| | NewsWire | Web |
|---|---|---|
| Size (gigabytes) | 1 | 100 |
| Documents | 400,000 | 12,000,000 |
| Word occurrences (without markup) | 180,000,000 | 11,000,000,000 |
| Distinct words (after stemming) . . . , | 400,000 | 16,000,000 |
| per document, totaled | 70,000,000 | 3,500,000,000 |

would have been difficult or impossible. The last line in the table is the number of distinct word-document pairs, that is, the number of word occurrences when duplicates within a document are not counted.

Most of the costs required for additional structures scale more or less linearly for collections larger than a gigabyte. For example, in Web data, new distinct words continue to occur at a typical rates of about one per 500–1000 word occurrences. Typical query terms occur in 0.1%–1% of the indexed documents.

Taking all of these factors into account, implementors of search engines must design their systems to balance a range of technical requirements:

—effective resolution of queries;

—use of features of conventional text, such as query term proximity, that improve effectiveness;

—use of features of hyperlinked text, such as anchor strings and URL terms, that improve effectiveness;

—fast resolution of queries;

—minimal use of other resources (disk, memory, bandwidth);

—scaling to large volumes of data;

—change in the set of documents; and

—provision of advanced features such as Boolean restriction and phrase querying.

This tutorial describes techniques for supporting all of these requirements.

*Similarity Measures.* All current search engines use *ranking* to identify potential answers. In a ranked query, a statistical similarity heuristic or *similarity measure* is used to assess the closeness of each document to the textual query. The underlying principle is that the higher the similarity score awarded to a document, the greater the estimated likelihood that a human would judge it to be relevant. The $r$ "most similar according to the heuristic" documents are returned to the user as suggested answers. To describe the implementation of text retrieval, we first consider evaluation of bag-of-words queries in which similarity is determined by simple statistics. In Section 4, we extend these techniques to phrase queries.

Most similarity measures use some composition of a small number of fundamental statistical values:

—$f_{d,t}$, the frequency of term $t$ in document $d$;

—$f_{q,t}$, the frequency of term $t$ in the query;

—$f_t$, the number of documents containing one or more occurrences of term $t$;

—$F_t$, the number of occurrences of term $t$ in the collection;

—$N$, the number of documents in the collection; and

—$n$, the number of indexed terms in the collection.

These basic values are combined in a way that results in three monotonicity observations being enforced.



المنارة للاستشارات
www.manaraa.com

(1) Less weight is given to terms that appear in many documents;

(2) More weight is given to terms that appear many times in a document; and

(3) Less weight is given to documents that contain many terms.

The intention is to bias the score towards relevant documents by favoring terms that appear to be discriminatory and reducing the impact of terms that appear to be randomly distributed.

A typical older formulation that is effective in practice calculates the cosine of the angle in $n$-dimensional space between a query vector $\langle w_{q,t} \rangle$ and a document vector $\langle w_{d,t} \rangle$. There are many variations of the cosine formulation. An example is:

$$
\begin{aligned}
w_{q,t} &= \ln\left(1 + \frac{N}{f_t}\right) & \qquad w_{d,t} &= 1 + \ln f_{d,t} \\
W_d &= \sqrt{\sum_t w_{d,t}^2} & \qquad W_q &= \sqrt{\sum_t w_{q,t}^2} \\
S_{q,d} &= \frac{\sum_t w_{d,t} \cdot w_{q,t}}{W_d \cdot W_q}.
\end{aligned}
\tag{1}
$$

The term $W_q$ can be neglected as it is a constant for a given query and does not affect the ordering of documents. Variations on this theme are to avoid use of logarithms, or replace $N$ by $\max_t\{f_t\}$ in the expression for $w_{q,t}$, or multiply the query-term weights $w_{q,t}$ by $1 + \ln f_{q,t}$ when queries are long, or use a different way of combining $w_{q,t}$ and $w_{d,t}$, or take $W_d$ to be the length of the document in words or in bytes, and so on.

What all of these variants share is that the quantity $w_{q,t}$ typically captures the property often described as the inverse document frequency of the term, or IDF, while $w_{d,t}$ captures the term frequency, or TF, hence the common description of similarity measures as TF×IDF formulations. Observing that the negative log of a probability is the information content, the score assigned to a document can very loosely be interpreted from an entropy-based perspective as being a sum of information conveyed by the query terms in that document, which maps to the same ordering as the product of their respective probabilities.

Similarity formulations that are directly grounded in statistical principles have also proven successful in TREC. The best known of these is the Okapi computation,

$$
\begin{aligned}
w_{q,t} &= \ln\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right) \cdot \frac{(k_3 + 1) \cdot f_{q,t}}{k_3 + f_{q,t}} \\
w_{d,t} &= \frac{(k_1 + 1) f_{d,t}}{K_d + f_{d,t}} \\
K_d &= k_1\left((1 - b) + b\frac{W_d}{W_A}\right) \\
S_{q,d} &= \sum_{t \in q} w_{q,t} \cdot w_{d,t},
\end{aligned}
\tag{2}
$$

in which the values $k_1$ and $b$ are parameters, set to 1.2 and 0.75 respectively; $k_3$ is a parameter that is set to $\infty$, so that the expression $(k_3 + 1) \cdot f_{q,t} / \left(k_3 + f_{q,t}\right)$ is assumed to be equivalent to $f_{q,t}$; and $W_d$ and $W_A$ are the document length and average document length, in any suitable units.



www.manaraa.com

*To rank a document collection with regard to a query $q$ and identify the top $r$ matching documents*:

(1) Calculate $w_{q,t}$ for each query term $t$ in $q$.
(2) For each document $d$ in the collection,
  (a) Set $S_d \leftarrow 0$.
  (b) For each query term $t$,
    Calculate or read $w_{d,t}$, and
    Set $S_d \leftarrow S_d + w_{q,t} \times w_{d,t}$.
  (c) Calculate or read $W_d$.
  (d) Set $S_d \leftarrow S_d / W_d$.
(3) Identify the $r$ greatest $S_d$ values and return the corresponding documents.

**Fig. 2**. Exhaustive computation of cosine similarity between a query $q$ and every document in a text collection. This approach is suitable only when the collection is small or is highly dynamic relative to the query rate.

More recent probabilistic approaches are based on language models. There are many variants; a straightforward language-model formulation is:

$$w_{d,t} = \log\left(\frac{|d|}{|d|+\mu}\cdot\frac{f_{d,t}}{|d|}+\frac{\mu}{|d|+\mu}\cdot\frac{F_t}{|C|}\right) \qquad (3)$$
$$S_{q,d} = \sum_{t\in q} f_{q,t}\cdot w_{d,t},$$

where $|d|$ (respectively, $|C|$) is the number of term occurrences in document $d$ (respectively, collection $C$) and $\mu$ is a parameter, typically set to 2500. The left-hand side of the sum is the observed likelihood of the term in the document, while the right-hand side modifies this (using Dirichlet smoothing) by combining it with the observed likelihood of the term in the collection. Taking the smoothed value as an estimate of the probability of the term in the document, this formulation is rank equivalent to ordering documents by the extent to which the query's entropy in the document's model differs from the query's entropy in the collection as a whole.

In this language-model formulation, the value $w_{d,t}$ is nonzero even if $t$ is not in $d$, presenting difficulties for the term-ordered query evaluation strategies we explain later. However, this problem can be addressed by transforming it into a rank-equivalent measure:

$$S_{q,d} \stackrel{rank}{=} |q|\cdot\log\left(\frac{\mu}{|d|+\mu}\right)+\sum_{t\in q\wedge d}\left(f_{q,t}\cdot\log\left(\frac{f_{d,t}}{\mu}\cdot\frac{|C|}{F_t}+1\right)\right), \qquad (4)$$

where the term-oriented component $\log(1+(f_{d,t}/\mu)\cdot(|C|/F_t))$ is zero when $t$ is not in $d$.

In all of these formulations, documents can score highly even if some of the query terms are missing. This is a common attribute of similarity heuristics. We do not explore similarity formulations in detail and take as our brief the need to implement a system in which any such computation can be efficiently computed.

Given a formulation, ranking a query against a collection of documents is in principle straightforward: each document is fetched in turn, and the similarity between it and the query calculated. The documents with the highest similarities can then be returned to the user. An algorithm for exhaustive ranking using the cosine measure is shown in Figure 2.

The drawback of this approach is that every document is explicitly considered, but for typical situations in which $r \ll N$, only a tiny fraction of documents are returned as answers. For most documents, the vast majority of similarity values are



المنارة للاستشارات
www.manaraa.com

insignificant. The exhaustive approach does, however, illustrate the main features of computing a ranking, and, with a simple reorganization, a more efficient computation can be achieved based on the key observation that, to have a nonzero score, a document must contain at least one query term.

## 3. INDEXING AND QUERY EVALUATION

Fast query evaluation makes use of an *index*: a data structure that maps terms to the documents that contain them. For example, the index of a book maps a set of selected terms to page numbers. With an index, query processing can be restricted to documents that contain at least one of the query terms.

Many different types of index have been described. The most efficient index structure for text query evaluation is the *inverted file*: a collection of lists, one per term, recording the identifiers of the documents containing that term. Other structures are briefly considered in Section 11, but they are not useful for general-purpose querying.

*Baseline Inverted File.* An inverted file index consists of two major components. The *search structure* or *vocabulary* stores for each distinct word $t$,

—a count $f_t$ of the documents containing $t$, and
—a pointer to the start of the corresponding *inverted list*.

Studies of retrieval effectiveness show that all terms should be indexed, even numbers. In particular, experience with Web collections shows that any visible component of a page might reasonably be used as a query term, including elements such as the tokens in the URL. Even stopwords—which are of questionable value for bag-of-words queries—have an important role in phrase queries.

The second component of the index is a set of *inverted lists* where each list stores for the corresponding word $t$,

—the identifiers $d$ of documents containing $t$, represented as ordinal document numbers; and
—the associated set of frequencies $f_{d,t}$ of terms $t$ in document $d$.

The lists are represented as sequences of $\langle d, f_{d,t} \rangle$ pairs. As described, this is a *document-level* index in that word positions within documents are not recorded. Together with an array of $W_d$ values (stored separately), these components provide all the information required for both Boolean and ranked query evaluation. A complete inverted file for the `Keeper` database is shown in Figure 3.

In a complete text database system, there are several other structures, including the documents themselves and a table that maps ordinal document numbers to disk locations (or other forms of document locator such as a filename or other key). We do not explore these structures in this tutorial.

In a simple representation, for the `NewsWire` data, the total index size would be approximately 435MB (megabytes) or around 40% of the size of the original data. This is comprised of 12MB for 400,000 words, pointers, and counts; 1.6MB for 400,000 $W_d$ values; 280MB for 70,000,000 document identifiers (four bytes each); and 140MB for 70,000,000 document frequencies (two bytes each). For the `Web` data, the total size is about 21GB (gigabytes) or just over 20% of the original text. The difference between these two collections is a consequence of Web pages tending to contain large volumes of unindexed markup.

An assumption made in these calculations is that each inverted list is stored contiguously. The alternative is that lists are composed of a sequence of blocks that are linked or indexed in some way. The assumption of contiguity has a range of implications. First,



المنارة للاستشارات
www.manaraa.com

| term $t$ | $f_t$ | Inverted list for $t$ |
|---|---|---|
| and | 1 | $\langle 6,2\rangle$ |
| big | 2 | $\langle 2,2\rangle$ $\langle 3,1\rangle$ |
| dark | 1 | $\langle 6,1\rangle$ |
| did | 1 | $\langle 4,1\rangle$ |
| gown | 1 | $\langle 2,1\rangle$ |
| had | 1 | $\langle 3,1\rangle$ |
| house | 2 | $\langle 2,1\rangle$ $\langle 3,1\rangle$ |
| in | 5 | $\langle 1,1\rangle$ $\langle 2,2\rangle$ $\langle 3,1\rangle$ $\langle 5,1\rangle$ $\langle 6,2\rangle$ |
| keep | 3 | $\langle 1,1\rangle$ $\langle 3,1\rangle$ $\langle 5,1\rangle$ |
| keeper | 3 | $\langle 1,1\rangle$ $\langle 4,1\rangle$ $\langle 5,1\rangle$ |
| keeps | 3 | $\langle 1,1\rangle$ $\langle 5,1\rangle$ $\langle 6,1\rangle$ |
| light | 1 | $\langle 6,1\rangle$ |
| never | 1 | $\langle 4,1\rangle$ |
| night | 3 | $\langle 1,1\rangle$ $\langle 4,1\rangle$ $\langle 5,2\rangle$ |
| old | 4 | $\langle 1,1\rangle$ $\langle 2,2\rangle$ $\langle 3,1\rangle$ $\langle 4,1\rangle$ |
| sleep | 1 | $\langle 4,1\rangle$ |
| sleeps | 1 | $\langle 6,1\rangle$ |
| the | 6 | $\langle 1,3\rangle$ $\langle 2,2\rangle$ $\langle 3,3\rangle$ $\langle 4,1\rangle$ $\langle 5,3\rangle$ $\langle 6,2\rangle$ |
| town | 2 | $\langle 1,1\rangle$ $\langle 3,1\rangle$ |
| where | 1 | $\langle 4,1\rangle$ |

| $d$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $W_d$ | 11.4 | 13.5 | 11.4 | 8.0 | 11.3 | 12.6 |

**Fig. 3**. Complete document-level inverted file for the Keeper database. The entry for each term $t$ is composed of the frequency $f_t$ and a list of pairs, each consisting of a document identifier $d$ and a document frequency $f_{d,t}$. Also shown are the $W_d$ values as computed for the cosine measure shown in Equation 1.

it means that a list can be read or written in a single operation. Accessing a sequence of blocks scattered across a disk would impose significant costs on query evaluation as the list for a typical query term on the Web data would occupy 100kB (kilobytes) to 1MB, and the inverted list for a common term could be many times this size. Adding to the difficulties for the great majority of terms, the inverted list is much less than a kilobyte, placing a severe constraint on feasible size for a fixed-size block. Second, no additional space is required for next-block pointers. Third, index update procedures must manage variable-length fragments that vary enormously in size, from tiny to vast; our experience, however, is that the benefits of contiguity greatly outweigh these costs.

An issue that is considered in detail in Section 8 is how to represent each stored value such as document numbers and in-document frequencies. The choice of any fixed number of bits or bytes to represent a value is clearly arbitrary and has potential implications for scaling (fixed-length values can overflow) and efficiency (inflation in the volume of data to be managed). Using the methods described later in this article, large gains in performance are available through the use of compressed representations of indexes.

To facilitate compression, $d$-gaps are stored rather than straight document identifiers. For example, the sorted sequence of document numbers

$$7, 18, 19, 22, 23, 25, 63, \dots$$

can be represented by gaps

$$7, 11, 1, 3, 1, 2, 38, \dots .$$



المنارة للاستشارات
www.manaraa.com

*To use an inverted index to rank a document collection with regard to a query $q$ and identify the top $r$ matching documents*:

(1) Allocate an accumulator $A_d$ for each document $d$ and set $A_d \leftarrow 0$.
(2) For each query term $t$ in $q$,
  (a) Calculate $w_{q,t}$, and fetch the inverted list for $t$.
  (b) For each pair $\langle d, f_{d,t} \rangle$ in the inverted list
    Calculate $w_{d,t}$, and
    Set $A_d \leftarrow A_d + w_{q,t} \times w_{d,t}$.
(3) Read the array of $W_d$ values.
(4) For each $A_d > 0$, set $S_d \leftarrow A_d / W_d$.
(5) Identify the $r$ greatest $S_d$ values and return the corresponding documents.

**Fig. 4**. Indexed computation of cosine similarity between a query $q$ and a text collection.

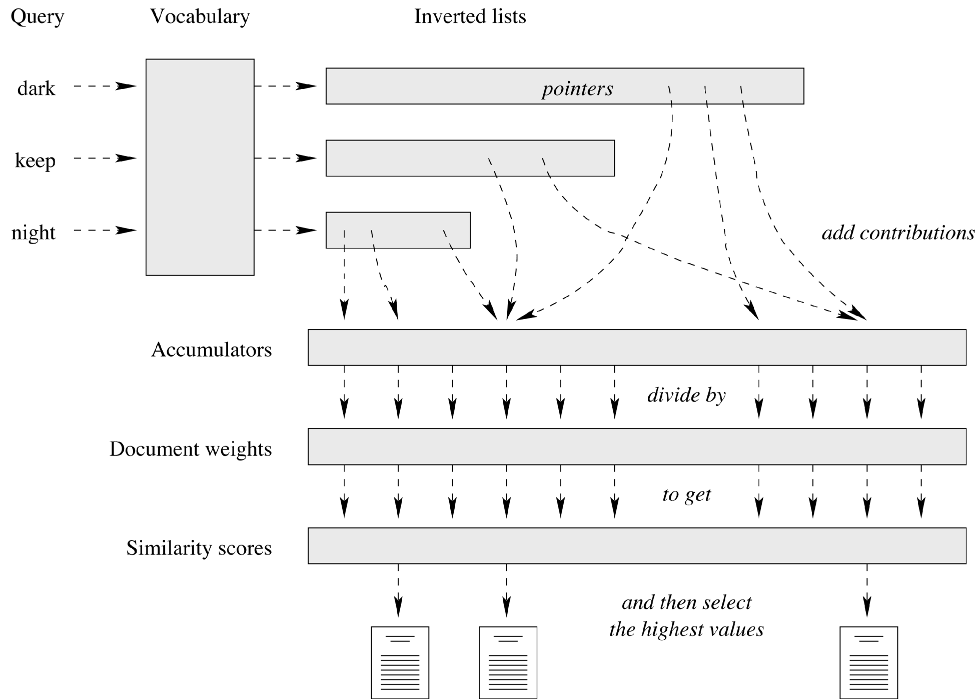


**Fig. 5**. Using an inverted file and a set of accumulators to calculate document similarity scores.

While this transformation does not reduce the maximum magnitude of the stored numbers, it does reduce the average, providing leverage for the compression techniques discussed later. Section 8 gives details of mechanisms that can exploit the advantage that is created by gaps.

*Baseline Query Evaluation.* Ranking using an inverted file is described in Figure 4 and illustrated in Figure 5. In this algorithm, the query terms are processed one at a time. Initially each document has a similarity of zero to the query; this is represented by creating an array $A$ of $N$ partial similarity scores referred to as *accumulators*, one for each document $d$. Then, for each term $t$, the accumulator $A_d$ for each document $d$ mentioned in $t$'s inverted list is increased by the contribution of $t$ to the similarity of $d$



المنارة للاستشارات
www.manaraa.com

to the query. Once all query terms have been processed, similarity scores $S_d$ are calculated by dividing each accumulator value by the corresponding value of $W_d$. Finally, the $r$ largest similarities are identified, and the corresponding documents returned to the user.

The cost of ranking via an index is far less than with the exhaustive algorithm outlined in Figure 2. Given a query of three terms, processing a query against the `Web` data involves finding the three terms in the vocabulary; fetching and then processing three inverted lists of perhaps 100kB to 1MB each; and making two linear passes over an array of 12,000,000 accumulators. The complete sequence requires well under a second on current desktop machines.

Nonetheless, the costs are still significant. *Disk space* is required for the index at 20%–60% of the size of the data for an index of the type shown in Figure 3; *memory* is required for an accumulator for each document and for some or all of the vocabulary; *CPU time* is required for processing inverted lists and accumulators; and *disk traffic* is used to fetch inverted lists. Fortunately, compared to the implementation shown in Figure 4, all of these costs can be dramatically reduced.

*Indexing Word Positions.* We have described inverted lists as sequences of index entries, each a $\langle d, f_{d,t} \rangle$ pair. An index of this form is *document-level* since it indicates whether a term occurs in a document but does not contain information about precisely where the term appears. Given that the frequency $f_{d,t}$ represents the number of occurrences of $t$ in $d$, it is straightforward to modify each entry to include the $f_{d,t}$ ordinal word positions $p$ at which $t$ occurs in $d$ and create a *word-level* inverted list containing pointers of the form $\langle d, f_{d,t}, p_1, \ldots, p_{f_{d,t}} \rangle$. Note that in this representation positions are word counts, not byte counts, so that they can be used to determine adjacency.

Word positions can be used in a variety of ways during query evaluation. Section 4 discusses one of these, phrase queries in which the user can request documents with a sequence rather than bag-of-words. Word positions can also be used in bag-of-word queries, for example, to prefer documents where the terms are close together or are close to the beginning of the document. Similarity measures that make use of such proximity mechanisms have not been particularly successful in experimental settings but, for simple queries, adjacency and proximity do appear to be of value in Web retrieval.

If the source document has a hierarchical structure, that structure can be reflected by a similar hierarchy in the inverted index. For example, a document with a structure of chapters, sections, and paragraphs might have word locations stored as $(c, s, p, w)$ tuples coded as a sequence of nested runs of $c$-gaps, $s$-gaps, $p$-gaps, and $w$-gaps. Such an index allows within-same-paragraph queries as well as phrase queries, for example, and with an appropriate representation, is only slightly more expensive to store than a nonhierarchical index.

*Core Ideas.* To end this section, we state several key implementation decisions.

—Documents have ordinal identifiers, numbered from one.

—Inverted lists are stored contiguously.

—The vocabulary consists of every term occurring in the documents and is stored in a simple extensible structure such as a B-tree.

—An inverted list consists of a sequence of pairs of document numbers and in-document frequencies, potentially augmented by word positions.

—The vocabulary may be preprocessed, by stemming and stopping.

—Ranking involves a set of accumulators and term-by-term processing of inverted lists.



للاستشارات المنارة
www.manaraa.com

This set of choices constitutes a core implementation in that it provides an approach that is simple to implement and has been used in several public-domain search systems and, we believe, many proprietary systems. In the following sections, we explore extensions to the core implementation.

## 4. PHRASE QUERYING

A small but significant fraction of the queries presented to Web search engines include an explicit phrase, such as `"philip glass" opera` or `"the great flydini"`. Users also often enter phrases without explicit quotes, issuing queries such as `Albert Einstein` or `San Francisco hotel`. Intuitively it is appealing to give high scores to pages in which terms appear in the same order and pattern as they appear in the query, and low scores to pages in which the terms are separated.

When phrases are used in Boolean queries, it is clear what is intended—the phrase itself must exist in matching documents. For example, the Boolean query `old "night keeper"` would be evaluated as if it contains two query terms, one of which is a phrase, and both terms would be required for a document to match.

In a ranked query, a phrase can be treated as an ordinary term, that is, a lexical entity that occurs in given documents with given frequencies, and contributes to the similarity score for that document when it does appear. Similarity can therefore be computed in the usual way, but it is first necessary to use the inverted lists for the terms in the phrase to construct an inverted list for the phrase itself, using a Boolean intersection algorithm. A question for information retrieval research (and outside the scope of this tutorial) is whether this is the best way to use phrases in similarity estimation. A good question for this tutorial is how to find—in a strictly Boolean sense—the documents in which a given sequence of words occur together as a phrase since, regardless of how they are eventually incorporated into a matching or ranking scheme, identification of phrases is the first step.

An obvious possibility is to use a parser at index construction time that recognizes phrases that might be queried and to index them as if they were ordinary document terms. The set of identified phrases would be added to the vocabulary and have their own inverted lists; users would then be able to query them without any alteration to query evaluation procedures. However, such indexing is potentially expensive. There is no obvious mechanism for accurately identifying which phrases might be used in queries, and the number of candidate phrases is enormous since even the number of distinct two-word phrases grows far more rapidly than the number of distinct terms. The hypothetical `Web` collection shown in Table I could easily contain a billion distinct two-word pairs.

Three main strategies for Boolean phrase query evaluation have been developed.

—Process phrase queries as Boolean bags-of-words so that the terms can occur anywhere in matching document, then postprocess the retrieved documents to eliminate false matches.

—Add word positions to some or all of the index entries so that the locations of terms in documents can be checked during query evaluation.

—Use some form of partial phrase index or word-pair index so that phrase appearances can be directly identified.

These three strategies can complement each other. However, a pure bag-of-words approach is unlikely to be satisfactory since the cost of fetching just a few nonmatching documents could exceed all other costs combined, and, for many phrases, only a tiny fraction of the documents that contain the query words also contain them as a phrase.



المنارة للاستشارات
www.manaraa.com

*Phrase Query Evaluation.* When only document-level querying is required, inclusion of positional information in the index not only takes space, but also slows query processing because of the need to skip over the positional information in each pointer. And, as discussed in more detail in the following, if bag-of-words ranked queries are to be supported efficiently, then other index organizations, such as frequency- and impact-sorted arrangements, need to be considered.

Taken together, these considerations suggest that maintenance of two separate indexes may be attractive, a word-level index for Boolean searching and phrase identification, and a document-level impact- or frequency-sorted index for processing ranked queries. Given that document-level indexes are small, the space overhead of having multiple indexes is low. Separating the indexes also brings flexibility and allows consideration of index structures designed explicitly for phrases.

Many phrases include common words. The cost of phrase query processing using a word-level inverted index is then dominated by the cost of fetching and decoding lists for those words which typically occur at the start of or in the middle of a phrase—consider `"the house in the town"`, for example. One way of avoiding this problem would be to neglect certain stop words and index them at the document-level only. For example, to evaluate the query `"the house in the town"` processing could proceed by intersecting the lists for `house`, and `town`, looking for positions $p$ of `house` such that `town` is at $p + 3$. False matches could be eliminated by post-processing, that is, by fetching candidate documents and examining them directly. The possibility of false matches could also simply be ignored. However, in some phrase queries, the common words play an important semantic role and must be included.

*Phrase Indexes.* It is also possible to build a complete index of two-word phrases using a hierarchical storage structure to avoid an overly large vocabulary. Experiments show that such an index occupies around 50% of the size of the source data, that is, perhaps 50GB for the `Web` collection. The inverted lists for phrases are on average much shorter than those of the individual words, but there are many more of them, and the vocabulary is also much bigger.

Using a two-word phrase index, evaluation of the phrase query `"the house in the town"` (which matches line three of the `Keeper` collection) involves processing the inverted lists for, say, the phrases `"the house"`, `"house in"`, and `"the town"`. The pair `"in the"` is also a phrase but is covered by the others and—making an arbitrary choice between this phrase and `"house in"`—its inverted list does not need to be processed.

For phrases composed of rare words, having a phrase index yields little advantage, as processing savings are offset by the need to access a much larger vocabulary. A successful strategy is to have an index for word pairs that begin with a common word and combine it with a word-level inverted index. For example, the preceding query could be evaluated by intersecting the inverted lists for `"the house"`, `in`, and `"the town"`. An index of all word pairs beginning with any one of the three commonest words is about 1% of the size of the `Web` data but allows phrase querying time to be approximately halved.

## 5. INDEX CONSTRUCTION

The single key problem that makes index construction challenging is that the volume of data involved cannot be held in main memory in a dynamic data structure of the kind typically used for cross-reference generation. The underlying task that needs to be performed is essentially that of matrix transposition. But the documents–terms matrix is very sparse, and is far too large to be manipulated directly as an array. Instead, index construction techniques make use of index compression methods and either distributive or comparison-based sorting techniques.



المنارة للاستشارات
www.manaraa.com

*To build an inverted index using the in-memory technique*:

(1) Make an initial pass over the collection.
For each term $t$ count its document frequency $f_t$, and determine an upper bound $u_t$ on the length of the inverted list for $t$.

(2) Allocate an in-memory array of $\sum_t u_t$ bytes, and, for each term $t$, create a pointer $c_t$ to the start of a corresponding block of $u_t$ bytes.

(3) Process the collection a second time.
For each document $d$, and for each term $t$ in $d$, append a code representing $d$ and $f_{d,t}$ at $c_t$, and update $c_t$.

(4) Make a sequential pass over the in-memory index that has been constructed.
For each $t$, copy the $f_t$ representations of the $\langle d, f_{d,t} \rangle$ pointers from the allocated $u_t$ bytes to the inverted file, compressing if desired.

**Fig. 6**. In-memory inversion.

*In-Memory Inversion.* A simple in-memory inversion algorithm is shown in Figure 6. The key idea is that a first pass through the documents collects term frequency information, sufficient for the inverted index to be laid out in memory in template form. A second pass then places pointers into their correct positions in the template, making use of the random-access capabilities of main memory. The advantage of this approach is that almost no memory is wasted compared to the final inverted file size since there is negligible fragmentation. In addition, if compression is used, the index can be represented compactly throughout the process. This technique is viable whenever the main memory available is about 10%–20% greater than the combined size of the index and vocabulary that are to be produced. It is straightforward to extend the in-memory algorithm to include word positions, but the correspondingly larger final index will more quickly challenge the memory capacity of whatever hardware is being used since individual list entries may become many kilobytes long.

It is also possible to extend the in-memory technique to data collections where index size exceeds memory size by laying out the index skeleton on disk, creating a sequence of partial indexes in memory, and then transferring each in a skip-sequential manner to a template that has been laid out as a disk file. With this extended method, and making use of compression, indexes can be built for multi-gigabyte collections using around 10–20MB of memory beyond the space required for a dynamic vocabulary.

*Sort-Based Inversion.* A shortcoming of two-pass techniques of the kind sketched in Figure 6 is that document parsing and fetching is a significant component of index construction costs, perhaps half to two-thirds of the total time for Web data. The documents could be stored parsed during index construction, but doing so implies substantial disk overheads, and the need to write the parsed text may outweigh the cost of the second parsing.

Other index construction methods are based on explicit *sorting*. In a simple form of this approach, an array or file of $\langle t, d, f_{d,t} \rangle$ triples is created in document number order, sorted into term order, and then used to generate the final inverted file.

With careful sequencing and use of a multiway merge, the sort can be carried out in-place on disk using compressed blocks. The disk space overhead is again about 10% of the final compressed index size, and memory requirements and speed are also similar to partitioned inversion. As for partitioned inversion, the complete vocabulary must be kept in memory, limiting the volume of data that can be indexed on a single machine.

*Merge-Based Inversion.* As the volumes of disk and data grow, the cost of keeping the complete vocabulary in memory is increasingly significant. Eventually, the index must be created as an amalgam of smaller parts, each of which is constructed using one of the



للاستشارات المنارة
www.manaraa.com

*To build an inverted index using a merge-based technique*:

(1) Until all documents have been processed,
  (a) Initialize an in-memory index, using a dynamic structure for the vocabularies and a static coding scheme for inverted lists; store lists either in dynamically resized arrays or in linked blocks.
  (b) Read documents and insert $\langle d, f_{d,t} \rangle$ pointers into the in-memory index, continuing until all allocated memory is consumed.
  (c) Flush this temporary index to disk, including its vocabulary.

(2) Merge the set of partial indexes to form a single index, compressing the inverted lists if required.

**Fig. 7.** Merge-based inversion.

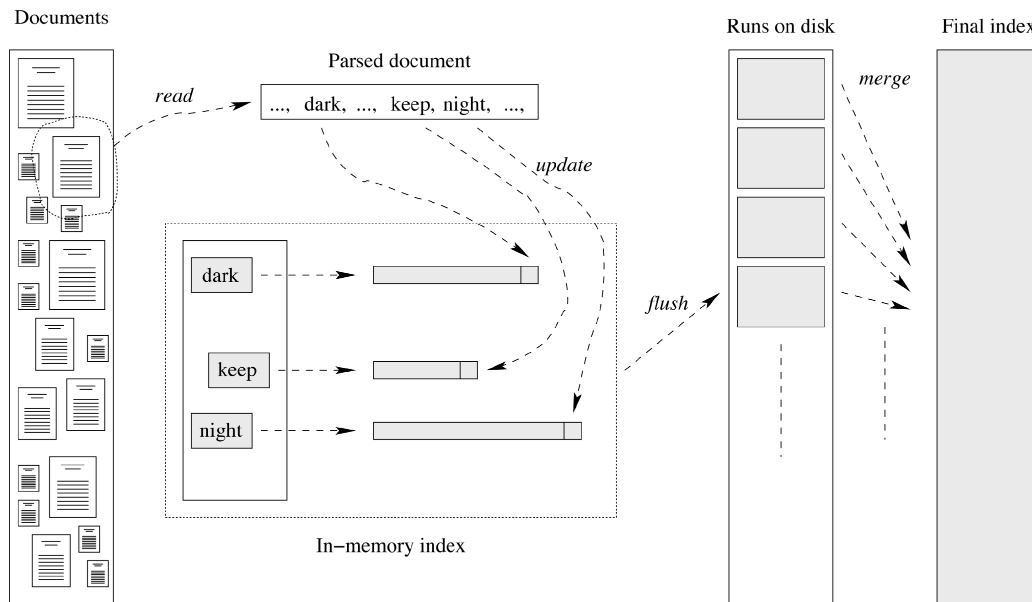


**Fig. 8.** Merge-build process.

previous techniques or using purely in-memory structures. Figures 7 and 8 illustrate this process.

In merge-based inversion, documents are read and indexed in memory until a fixed capacity is reached. Each inverted list needs to be represented in a structure that can grow as further information about the term is encountered, and dynamically resizable arrays are the best choice. When memory is full, the index (including its vocabulary) is flushed to disk as a single *run* with the inverted lists in the run stored in lexicographic order to facilitate subsequent merging. As runs are never queried, the vocabulary of a run does not need to be stored as an explicit structure; each term can, for example, be written at the head of its inverted list. Once the run is written, it is entirely deleted from memory so that construction of the next run begins with an initially empty vocabulary.

When all documents have been processed, the runs are merged to give the final index. The merging process builds the final vocabulary on the fly and, if a large read buffer is allocated to each run, is highly efficient in terms of disk accesses. If disk space is scarce, the final index can be written back into the space occupied by the runs as they are processed as the final index is typically a little smaller than the runs—vocabulary



المنارة للاستشارات
www.manaraa.com

information is not duplicated, and the final inverted lists can be represented more efficiently.

Merge-based index construction is practical for collections of all sizes. In particular, it scales well and operates effectively in as little as 100MB of memory. In addition, disk space overheads can be restricted to a small fraction of the final index; only one parsing pass is required over the data; and the method extends naturally to phrase indexing. Finally, the compression techniques described in Section 8 can further reduce the cost of index construction by reducing the number of runs required.

## 6. INDEX MAINTENANCE

Inserting one document into a text databases involves, in principle, adding a few bytes to the end of every inverted list corresponding to a term in the document. For a document of reasonable size, such an insertion involves fetching and slightly extending several hundred inverted lists and is likely to require 10–20 seconds on current hardware. In contrast, with merge-based inversion, the same hardware can index around 1,000 documents per second. That is, there is a 10,000-fold disparity in cost between these two approaches.

For fast insertion, it is necessary to avoid accessing the disk-resident inverted lists of each term. The only practical solution is to amortize the update costs over a sequence of insertions. The properties of text databases, fortunately, allow several strategies for cost amortization. In particular, for ranking, it is not always necessary for new documents to be immediately available for searches. If they are to be searchable, new documents can be made available through a temporary in-memory index—in effect, the last subindex in the merging strategy.

Three broad categories of update strategies are available: rebuild from scratch, merge an existing index with an index of new documents, and incremental update.

*Rebuild.* In some applications, the index may not to be updated online at all. Instead, it can be periodically rebuilt from scratch. Consider, for example, a university Web site. New documents are only discovered through crawling and immediate update is not essential. For a gigabyte of data, rebuilding takes just a few minutes, a small cost compared to that of fetching the documents to index.

*Intermittent Merge.* The inverted lists for even large numbers of documents can easily be maintained in the memory of a standard desktop computer. If the lists are in memory, it is cheap to insert new documents as they arrive; indeed, there is no difference between maintaining such lists and the algorithm described in Figure 7.

If existing documents are indexed in a standard inverted file and new documents are indexed in memory, the two indexes can share a common vocabulary, and all documents can be made available to queries via a straightforward adaptation of the methods described earlier. Then when memory is full, or some other criterion is met, the in-memory index is merged with the on-disk index. The old index can be used until the merge is complete, at the cost of maintaining two complete copies of the inverted lists. During the merge either a new in-memory index must be created or insertions must temporarily be blocked, and thus, during the merge, new documents are only available via exhaustive search.

It was argued earlier that inverted lists should be stored contiguously as accessing a large number of blocks would be a dominant cost of query evaluation. However, if the number of blocks is constrained (for instance, an average of three per term), the time to evaluate queries can similarly be controlled. In addition, if the index is arranged as a sequence of subindexes, each one no greater than a given fraction of the size of the next (that is, the sizes form a geometric sequence), then only a small part of the index is involved in most merge operations. This combination of techniques allow an index to



المنارة للاستشارات
www.manaraa.com

be built incrementally, and be simultaneously available for querying, in just twice the time required by an offline merge-based build.

*Incremental Update.* A final alternative is to update the main index term by term, as and when opportunity arises, with some terms' in-memory lists covering more documents than others'. This process is similar to the mechanisms used for maintaining variable-length records in a conventional database system. In such an asynchronous merge, a list is fetched from disk, the new information is integrated into the list, and the list is then written back. Using standard free-space management, the list can either be written back in place or, if there is insufficient space, written to a new location.

The per-list updates should be deferred for as long as possible to minimize the number of times each list is accessed. The simplest approach is to process as for the merging strategy and, when a memory limit is reached, then proceed through the whole index, amending each list in turn. Other possibilities are to update a list only when it is fetched in response to a query or to employ a background process that slowly cycles through the in-memory index, continuously updating entries. In practice, these methods are not as efficient as intermittent merge, which processes data on disk sequentially.

*Choosing an Update Strategy.* For reasonable collection sizes, merging is the most efficient strategy for update but has the drawback of requiring significant disk overheads. It allows relatively simple recovery as reconstruction requires only a copy of the index and the new documents. In contrast, incremental update proceeds in place with some space lost due to fragmentation. But recovery in an incremental index may be complex due to the need to track which inverted lists have been modified.

For smaller, relatively static collections, the cost of rebuild may exceed that of other methods, but still be of little consequence compared to the other costs of maintaining the database. And if the collection is highly dynamic, such as a Web site in which documents are edited as well as inserted, then inserting or deleting a single word in a document may affect all the word positions, for example, and rebuild may be the only plausible option.

## 7. DISTRIBUTED INFORMATION RETRIEVAL

When large volumes of data are involved or when high query volumes must be supported, one machine may be inadequate to support the load even when the various enhancements surveyed earlier are incorporated. For example, in mid-2004, the Google search engine processed more than 200 million queries a day against more than 20TB of crawled data, using more than 20,000 computers.

To handle the load, a combination of distribution and replication is required. *Distribution* refers to the fact that the document collection and its index are split across multiple machines and that answers to the query as a whole must be synthesized from the various collection components. *Replication* (or *mirroring*) then involves making enough identical copies of the system so that the required query load can be handled.

*Document-Distributed Architectures.* The simplest distribution regime is to partition the collection and allocate one subcollection to each of the processors. A local index is built for each subcollection; when queries arrive, they are passed to every subcollection and evaluated against every local index. The sets of subcollection answers are then combined in some way to provide an overall set of answers. The advantages of such a *document partitioned* system are several; collection growth is accommodated by designating one of the hosts as the dynamic collection so that only it needs to rebuild its index; and the computationally expensive parts of the process are distributed equally across all of the hosts in the computer cluster. The dashed region



المنارة للاستشارات
www.manaraa.com

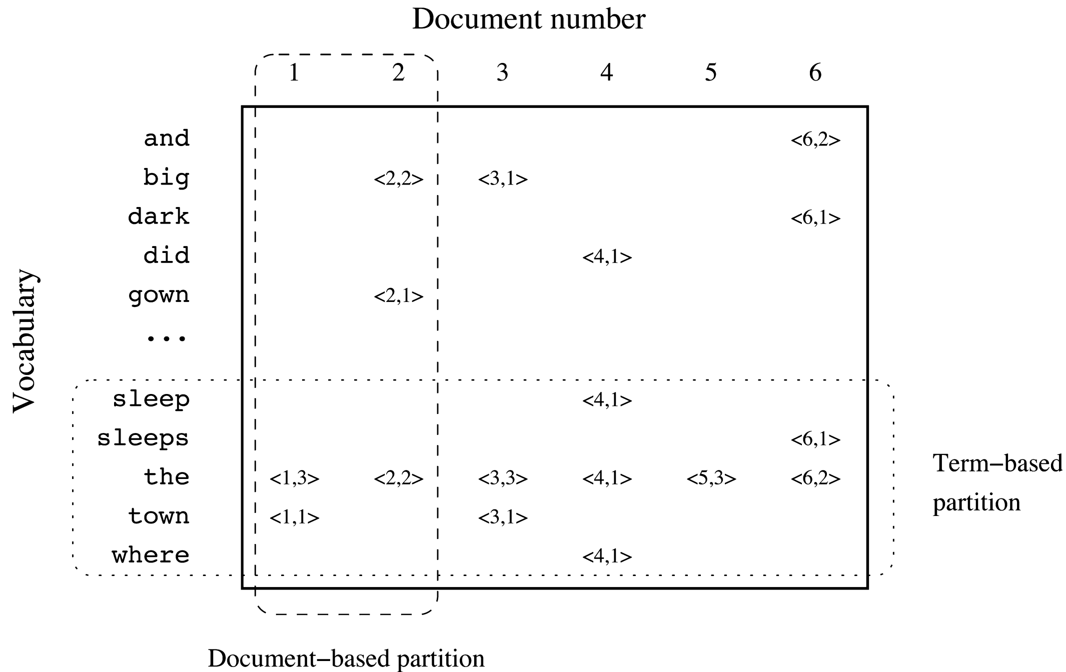


**Fig. 9**. Two ways in which the index of Figure 3 might be partitioned and distributed across a cluster of machines.

in Figure 9 shows one component of a document-distributed retrieval system with this one processor indexing all terms that appear in the first two documents of the collection.

*Term-Distributed Architectures.* An alternative strategy is *term partitioning*. In a term-partitioned index, the index is split into components by partitioning the vocabulary, with one possible partition shown by the dotted horizontal split in Figure 9. Each processor has full information about a subset of the terms, meaning that to handle a query, only the relevant subset of the processors need to respond. Term partitioning has the advantage of requiring fewer disk seek and transfer operations during query evaluation than document-partitioning because each term's inverted list is still stored contiguously on a single machine rather than in fragments across multiple machines. On the other hand, those disk transfer operations each involve more data. More significantly, in a term-partitioned arrangement, the majority of the processing load falls to the coordinating machine, and experiments have shown that it can easily become a bottleneck and starve the other processors of work.

*Choosing a Distribution Strategy.* Document distribution typically results in a better balance of workload than does term partitioning and achieves superior query throughput. It also allows more naturally for index construction and for document insertion. On the other hand, index construction in a term-partitioned index involves first of all distributing the documents and building a document-partitioned index, and then exchanging index fragments between all pairs of processors after the vocabulary split has been negotiated. Document distribution also has the pragmatic advantage of still allowing a search service to be provided even when one of the hosts is offline for some reason since any answers not resident on that machine remain available to the system.



المنارة للاستشارات
www.manaraa.com

On the other hand, in a term-distributed system, having one machine offline is likely to be immediately noticeable.

The Google implementation uses a document-partitioned index with massive replication and redundancy at all levels: the machine, the processor cluster, and the site.

It is also worth noting that document partitioning remains effective even if the collaborating systems are independent and unable to exchange their index data. A distributed system in which a final result answer list is synthesized from the possibly-overlapping answer sets provided by a range of different services is called a *metasearcher*.

## 8. EFFICIENT INDEX REPRESENTATIONS

A central idea underpinning several aspects of efficient indexing and query processing is the use of compression. With appropriate compression techniques, it is possible to simultaneously reduce both space consumption and disk traffic. Compression also reduces overall query evaluation time. This section introduces a range of integer coding techniques, and describes their use in inverted file compression.

*Compact Storage of Integers.* In a simple implementation of an inverted file index, 32-bit and 16-bit integers might be used respectively for document identifiers and term frequencies since these sizes are universally supported by compilers and are large enough to hold the likely values. However, fixed-width integers are not particularly space-efficient, and large savings can be obtained using quite simple compression techniques.

An efficient representation for an inverted list requires a method for coding integers in a variable number of bits. Using a fixed number of bits for each number, whether 32 or 20, is inconvenient: it limits the maximum value that can be stored, and is wasteful if most numbers are small. Variable-length codes can be infinite, avoiding the problem of having a fixed upper bound; and can be constructed to favor small values at the expense of large.

*Parameterless Codes.* The simplest variable-bit infinite code is *unary* which represents the value $x > 0$ as $x - 1$ "1" bits followed by a terminating "0" bit.

Unary is an example of a fixed code corresponding to a fixed distribution and is unparameterized. Other unparameterized codes for integers are Elias' *gamma* and *delta* codes. In the gamma code, integer $x > 0$ is factored into $2^e + d$ where $e = \lfloor \log_2 x \rfloor$, and $0 \le d < 2^e$. The codeword is formed as the concatenation of $e + 1$ represented in unary and $d$ represented in binary in $e$ bits. In the delta code, the value of $e + 1$ is represented using the gamma code and is followed by $d$ in binary, as for gamma.

In both codes, each codeword has two parts, a prefix and a suffix. The prefix indicates the binary magnitude of the value and tells the decoder how many bits there are in the suffix part. The suffix indicates the value of the number within the corresponding binary range. Table II gives some example codewords for each of these three unparameterized codes. The colons used in the codewords to separate the prefix and suffix parts are purely illustrative.

Which code is preferable depends on the probability distribution $\Pr(x)$ governing the values $x$ that are being coded. Using Shannon's relationship between the probability of a symbol and its ideal codeword length, $\text{len}(x) = -\log_2 \Pr(x)$, it can be seen that unary corresponds to the probability distribution $\Pr(x) = 2^{-x}$. That is, when half of all values are the number 1, a quarter are 2, an eighth are 3, and so on, then unary is the most efficient coding mechanism. When approximately half of all values are the number 1, a quarter are (equally one of) 2 or 3, an eighth are (equally one of) 4, 5, 6, or 7, and so on, and in general $\Pr(x) \approx 1/(2x^2)$, then gamma is the most efficient coding mechanism.

However, inspecting a sequence of bits one by one is relatively costly on machines for which the basic unit of access is multiples of eight bits. If each code occupies a sequence



المنارة للاستشارات
www.manaraa.com

**Table II.** Example Codewords Using the Unary, Gamma, and Delta Codes

| value | unary | gamma | delta |
|---|---|---|---|
| 1 | 0 | 0: | 0:: |
| 2 | 10 | 10:0 | 10:0:0 |
| 3 | 110 | 10:1 | 10:0:1 |
| 4 | 1110 | 110:00 | 10:1:00 |
| 10 | 1111111110 | 1110:010 | 110:00:010 |
| 100 | | 1111110:100100 | 110:11:100100 |
| 1,000 | | 1111111110:111101000 | 1110:010:111101000 |

Colons are used as a guide to show the prefix and suffix components in each codeword. All three codes can represent arbitrarily large numbers; the unary codewords for 100 and 1,000 are omitted only because of their length.

*To encode integer $x \geq 1$:*

(1) Set $x \leftarrow x - 1$.
(2) While $x \geq 128$,
  (a) $write_byte(128 + x \bmod 128)$.
  (b) Set $x \leftarrow (x \text{ div } 128) - 1$.
(3) $write_byte(x)$.

*To decode an integer $x$:*

(1) Set $b \leftarrow read_byte()$, $x \leftarrow 0$, and $p \leftarrow 1$.
(2) While $b \geq 128$,
  (a) Set $x \leftarrow x + (b - 127) \times p$ and $p \leftarrow p \times 128$.
  (b) Set $b \leftarrow read_byte()$.
(3) Set $x \leftarrow x + (b + 1) \times p$.

**Fig. 10.** Encoding and decoding variable-length byte-aligned codes. Input values to the encoder must satisfy $x \geq 1$.

of whole bytes, the bit operations can be eliminated. With this in mind, another simple unparameterized code is to use a sequence of bytes to code a value $x \geq 1$, shown in Figure 10. The idea is very simple: if $x \leq 128$, then a single byte is used to represent $x - 1$ in binary, with a leading "0" bit; otherwise, the low-order seven bits of $x - 1$ are packed into a byte with a leading "1" bit, and the quantity ($x$ div 128) is recursively coded the same way.

Thus the byte 2, with the bit-pattern 0000 0010, represents the integer 3; the byte 9, with the bit-pattern 0000 1001, represents the (decimal) integer 10; and the double-byte 147:7, with the bit-pattern 1110 0111 : 0000 0111, represents 1044 since $1044 = 128 \times (7 + 1) + (147 - 127)$. This approach is a form of gamma code in which the suffix length is a multiple of seven instead of a multiple of one. Several studies have explored bytewise coding and found it to be more efficient than bitwise alternatives. An enhanced version in which the split between continuing and terminating bytes is signaled by a different value than 128 has also been described and offers the possibility of improved compression effectiveness.

As well as offering economical decoding, the bytewise coding mechanism facilitates fast stepping through a compressed stream looking for the $k$th subsequent code. Since each codeword ends with a byte in which the top bit is a "0", it is easy to step through a compressed sequence and skip over exactly $k$ coded integers without having to decode each of them in full.



المنارة للاستشارات
www.manaraa.com

*To encode integer $x \geq 1$ using parameter $b$:*

(1) Factor $x \geq 1$ into $q \cdot b + r + 1$ where $0 \leq r < b$.
(2) Code $q + 1$ in unary.
(3) Set $e \leftarrow \lceil \log_2 b \rceil$ and $g \leftarrow 2^e - b$.
(4) If $0 \leq r < g$ then code $r$ in binary using $e - 1$ bits; otherwise, if $g \leq r < b$, then code $r + g$ in binary using $e$ bits.

**Fig. 11**. Encoding using a Golomb code with parameter $b$. Input values must satisfy $x \geq 1$.

**Table III.** Example Codewords Using Three Different Golomb Codes

| value | $b = 3$ | $b = 5$ | $b = 16$ |
|---|---|---|---|
| 1 | 0:0 | 0:00 | 0:0000 |
| 2 | 0:01 | 0:01 | 0:0001 |
| 3 | 0:11 | 0:10 | 0:0010 |
| 4 | 10:0 | 0:110 | 0:0010 |
| 10 | 1110:0 | 10:110 | 0:1001 |

Colons are used as a guide to show the prefix and suffix components in each codeword. All three codes can represent arbitrarily large numbers.

*Golomb and Rice Codes.* The Elias codes just described have the property that the integer 1 is always encoded in one bit. However, $d$-gaps of 1 are not particularly common in inverted lists. More pertinently, within the context of a single inverted list with known length, the likely size of the numbers can be accurately estimated. It is therefore desirable to consider schemes that can be parameterized in terms of the average size of the values to be coded.

Golomb codes, described in Figure 11, have this property. Integer $x \geq 1$ is coded in two parts—a unary bucket selector, and a binary offset within that bucket. The difference between this and the Elias codes is that, in Golomb codes, all buckets are the same size. The use of a variable-length binary code for the remainder $r$ means that no bit combinations are wasted even when the parameter $b$ is not an exact power of two. Table III gives some examples of the codewords generated for different values of $b$. When $b$ is a power of two, $b = 2^k$, the suffix part always contains exactly $k$ bits. These are known as Rice codes and allow simpler encoding and decoding procedures. An example Rice code appears in the last column of Table III.

*Matching Code to Distribution.* The implied probability distribution associated with the Elias gamma code was described earlier; Golomb codes are similarly minimum-redundancy when

$$\Pr(x) \approx (1 - p)^{x-1} p \,,$$

provided that

$$b = \left\lceil \frac{\log(2 - p)}{-\log(1 - p)} \right\rceil \approx 0.69 \times \frac{1}{p} \,,$$

where $p$ is the parameter of the geometric distribution (the probability of success in a sequence of independent trials). As we now explain, an inverted list can be represented as a sequence of integers that are, under a reasonable assumption, a sequence of independent (or Bernoulli) trials.

*Binary Codes.* Other coding mechanisms can also perform well and even binary codes can realize compact representations if the data is locally homogeneous. At the same time, they can provide decoding speeds as good as bytewise codes. In the simplest



المنارة للاستشارات
www.manaraa.com

example of this approach, a sequence of values is represented by fitting as many binary codes as possible into the next 32-bit output word. For example, if all of the next seven sequence values are smaller than 17, then a set of 4-bit codewords can be packed into a single output word. Similarly, if all of the next nine values are smaller than 9, an output word containing 3-bit codewords can be constructed. To allow decoding, each word of packed codes is prefixed by a short binary selector that indicates how that word should be interpreted.

As with the bytewise coding method, this approach supports fast identification of the $k$th subsequent codeword since all that is required is that the selector of each word be examined to know how many codewords it contains.

*Compressing Inverted Lists.* The representation of inverted lists introduced earlier described each inverted list as a sequence of $\langle d, f_{d,t} \rangle$ values with the restriction that each document identifier $d$ be an ordinal number. From a database perspective, imposing an ordinal identifier appears to be a poor design decision because it has implications for maintenance. However, text databases are not manipulated and updated in the ways that relational databases are, and, in practice, the use of a simple mapping table obviates the difficulties.

The frequencies $f_{d,t}$ are usually small (with a typical median of 1 or 2) and can be efficiently represented using unary or gamma. But representing raw document identifiers using these codes gives no saving since the median value in each inverted list is $N/2$.

On the other hand, if document identifiers are sorted and first-order differences (*$d$-gaps*) are stored, significant savings are possible. If a term appears in a random subset of $f_t$ of the $N$ documents in the collection, the $d$-gaps conform to a geometric distribution with probability parameter $p = f_t/N$ and can thus be coded effectively using a Golomb code or one of the other codes. For example, a Golomb-gamma-coded index of all words and numbers occurring in the documents for the `NewsWire` collection would occupy about 7% of the text size. In this approach, the representation for each inverted list consists of alternating $d$-gap values and $f_{d,t}$ values, each occupying a variable number of bits. The $d$-gap values are represented as Golomb codes, using a parameter $b$ determined from the number of $\langle d, f_{d,t} \rangle$ pairs in that inverted list, and the $f_{d,t}$ values are represented using the gamma code.

Use of Golomb codes does, however, present problems for the update strategy of intermittent merge. If Golomb codes are used for representing document identifiers, the merge process involves decoding the existing list and recoding with new parameters, but processing the existing list will then be the dominant cost of update, and should be avoided. One solution is that the old Golomb parameter could continue to be used to compress the added material at some small risk of gradual compression degradation. Alternatively, if static codes are used and lists are document-ordered, new $\langle d, f_{d,t} \rangle$ values can be directly appended to the end of the inverted lists. For other list orderings, such as those described in Section 9, there may be no alternative to complete reconstruction.

*Compression Effectiveness.* The effectiveness of compression regimes is particularly evident for inverted lists representing common words. To take an extreme case, the word `the` is likely to occur in almost every document (in an English-language collection), and the vast majority of $d$-gaps in its inverted list will be 1 and represented in just a single bit. Allowing for the corresponding $f_{d,t}$ value to be stored in perhaps (at most) 10–11 bits, around 12 bits is required per $\langle d, f_{d,t} \rangle$ pointer for common words, or one-quarter of the 48 bits that would be required if the pointer was stored uncompressed. The additional space savings that can be achieved with stopping are, for this reason, small.



المنارة للاستشارات
www.manaraa.com

However, stopping does yield significant advantages during index maintenance because stopping means that updates to the longest lists (those of common words) are avoided.

Less frequent words require longer codes for their $d$-gaps but, in general, shorter codes for their $f_{d,t}$ values. As a general rule-of-thumb, each $\langle d, f_{d,t} \rangle$ pointer in a complete document-level inverted index requires about 8 bits when compressed using a combination of Golomb and Elias codes. Use of the less-precise bytewise code for one of the two components adds around 4 bits per pointer and, for both components, adds around 8 bits per pointer. The word-aligned code has similar performance to the bytewise code for large $d$-gaps but obtains better compression when the $d$-gaps are small.

For example, a bytewise-gamma-coded index for the `NewsWire` data (with the $d$-gaps represented in a bytewise code and the $f_{d,t}$ values coded using gamma) would occupy about $70 \times 10^6 \times 12$ bits or approximately 10% of the source collection. In this index, the document identifiers are byte aligned, but the $f_{d,t}$ values are not, so the list is organized as a vector of document numbers and a separate vector of frequencies with the two lists decoded in parallel when both components are required. While the compression savings are not as great as those attained by Golomb codes, the bytewise and word-aligned codes bring other benefits.

*Compression of Word Positions.* Uncompressed, at, for instance, two bytes each (so no document can exceed 65,536 words), word positions immediately account for the bulk of index size: 360MB for `NewsWire`, and 22GB for `Web`. However, these costs can be reduced by taking differences, just as for document identifiers, and Golomb codes can be used to represent the differences in either a localized within-this-document sense, or, more usefully, in an average-across-all-documents manner. In the latter case, the vocabulary stores two $b$ values, one used to code the $d$-gaps in the way that was described earlier, and a second to code the $w$-gaps counting the word intervals between appearances of that term. Static codes—gamma, delta, bytewise, or word-aligned—also give acceptable compression efficiency and may be attractive because of their simplicity.

The choice of coding scheme also affects total fetch-and-decode times with the bytewise and word-aligned codes enjoying a clear advantage in this regard. Bitwise codes cannot be easily stepped through, and queries that require only bag-of-words processing can be considerably slower with an interleaved word-level index than with a document-level index.

An aspect of inverted indexes that is dramatically altered by the introduction of word positions is the cost of processing common words. In a document-level index, common words such as `the` are relatively cheap to store. In a word-level index, the average per-document requirement for common words is much larger because of the comparatively large number of $f_{d,t}$ word-gap codes that must be stored. In the case of the `Web` data, just a handful of inverted lists account for more than 10% of the total index size. Processing (or even storage) of such lists should be avoided whenever possible.

*Nonrandom Term Appearances.* If documents are chronological, terms tend to cluster. For example, in the 242,918 documents of the Associated Press component of TREC, there are two definite clusters for `hurricane`, one due to hurricane Gilbert in September 1988, and a second resulting from hurricane Hugo in September 1989. Similar clustering arises with web crawls. Some sites are topic-specific so that words that are rare overall may be common within a group of pages. A particular cause of this effect is indexing of documents that are written in a variety of languages.

The nonuniform distribution can be exploited by an *interpolative code* which transmits the mid-point of the list of document numbers in binary, then recursively handles the two sublists in the resulting narrowed ranges. For typical collections, this code



المنارة للاستشارات
www.manaraa.com

results in an average of 0.5 to 1.5 bits saved per $\langle d, f_{d,t} \rangle$ pointer compared to a Golomb code. The drawback of the interpolative code is that it is more complex to implement and slower in decoding.

The word-aligned binary coding method is also sensitive to localized clustering, a run of consistently small $d$-gaps is packed more tightly into output words than is a sequence containing occasional larger ones.

*Pros and Cons of Compression.* Compression has immediately obvious benefits. It reduces the disk space needed to store the index; and during query evaluation, it reduces transfer costs (the lists are shorter) and seek times (the index is smaller). Compression also reduces the costs of index construction and maintenance.

A less obvious benefit of index compression is that it improves caching. If a typical inverted list is compressed by a factor of six, then the number of inverted lists that can be retained in memory is increased by that same factor. In a retrieval system, queries arrive with a skew distribution with some queries and query terms much more common than others. Queries for `Chicago weather forecast` far exceed those for `Kalgoolie weather forecast`, for example. There can also be marked temporal effects. People use Web search engines to check questions on broadcast quiz shows, for example, so the same unusual query might arrive thousands of times in a short space of time. Increasing the effectiveness of caching can dramatically cut the cost of resolving a stream of queries.

The principal disadvantage of compression is that inverted lists must be decoded before they are used. A related problem is that they may need to be recoded when they are updated, for example, if parameters change. For some of the codes considered, the addition of new information can require complex recoding.

On current desktop machines, the decoding cost is more than offset by the reduction in disk seek costs. More importantly, as the ratio of the speed between processors and disk continues to diverge, the performance benefit available through compression is increasing. For byte- and word-aligned codes, which allow each document identifier to be decoded in just a few instruction cycles, the processing cost is more than offset by the reduction in memory-to-cache transfer costs.

Thus, if the index is larger than the available main memory or cannot be buffered for some other reason, there is no disadvantage to compression. And even if the index is in memory, processing can be faster than for uncompressed data. Use of appropriate index compression techniques is an important facet of the design of an efficient retrieval system.

## 9. LIMITING MEMORY REQUIREMENTS

If the standard query evaluation algorithm is used for queries that involve common words, most accumulators are nonzero, and an array of $N$ entries $A_d$ is the most space- and time-efficient structure. But the majority of those accumulator values are trivially small, as the only matching terms are one or more common words. Analysis of search engine logs has demonstrated that common terms are not the norm in queries, and analysis of relevance has demonstrated that common terms are of low importance. It thus makes sense to ask if there are better structures for the accumulators, requiring fewer than $N$ elements.

*Accumulator Limiting.* If only documents with low $f_t$ (i.e., rare) query terms are allowed to have an accumulator, the number of accumulators can be greatly reduced. This strategy is most readily implemented by imposing a limit $L$ on the number of accumulators as shown in Figure 12.



المنارة للاستشارات
www.manaraa.com

*To use an inverted index and an accumulator limit $L$ to rank a document collection with regard to a query $q$ and identify the top $r$ matching documents*:

(1) Create an empty set $A$ of accumulators.
(2) For each query term $t$ in $q$, ordered by decreasing $w_{q,t}$,
   (a) Fetch the inverted list for $t$.
   (b) For each pair $\langle d, f_{d,t} \rangle$ in the inverted list
      i. If $A_d$ does not exist and $|A| < L$, create accumulator $A_d$.
      ii. If $A_d$ exists, calculate $w_{d,t}$ and set $A_d \leftarrow A_d + w_{q,t} \times w_{d,t}$.
(3) Read the array of $W_d$ values.
(4) For each accumulator $A_d \in A$, set $S_d \leftarrow A_d / W_d$.
(5) Identify the $r$ greatest $S_d$ values and return the corresponding documents.

**Fig. 12**. The limiting method for restricting the number of accumulators during ranked query evaluation. The accumulator limit $L$ must be set in advance. The thresholding method involves a similar computation, but with a different test at step 2(b)i.

In ranking, using the models described earlier, a document can have a high similarity score even if several query terms are missing. In the limiting approach, as each $\langle d, f_{d,t} \rangle$ value is processed, a check is made to determine whether there is space in the set of accumulators to consider additional documents. Thus the algorithm enforces the presence of the query terms that are weighted highly because of their frequency of occurrence in the query or because of their rareness in the collection. While the behavior of the ranking is altered, it is arguably for the better since a document that is missing the most selective of the query terms is less likely to be relevant. Some of the Web search engines only return matches in which all of the query terms are present, a strategy that appears to be effective in practice for short queries but, in experimental settings, has been found to be less compelling.

*Accumulator Thresholding.* Another approach of a similar type is to use partial similarities—the contribution made by a term to a document's similarity, or $w_{q,t} \times w_{d,t}$—to determine whether an accumulator should be created. In this approach, accumulators are created if it seems likely that the document will ultimately have a sufficiently high similarity, with the test $|A| < L$ in Figure 12 replaced by a test $w_{q,t} \times w_{d,t} > S$, for some value $S$. The threshold $S$ is initially small but is increased during query processing so that it becomes harder for documents to create an accumulator as the computation proceeds.

For a set of accumulators that is limited to approximately 1%–5% of the number of documents, in the context of TREC-style long queries and evaluation, there is no negative impact on retrieval effectiveness.

However, with Web-style queries, both the limiting and the thresholding methods can be poor. In the limiting method, the limit tends to be reached part way through the processing of a query term, and thus the method favors documents with low ordinal identifiers; but if the limit is only tested between lists, then the memory requirements are highly unpredictable. Similar problems affect thresholding. A solution is to modify the thresholding approach so that, not only are accumulators created when the new contribution is significantly large, but existing accumulators are discarded when they are too small. With this approach, the number of accumulators can be fixed at less than 1% of the number of documents for large `Web`-like collections.

*Data Structures.* Comparing the two approaches, the limit method gives precise control over memory usage, whereas the threshold method is less rigid and allows even common terms to create accumulators for particular documents if they are sufficiently



المنارة للاستشارات
www.manaraa.com

frequent in those documents. Both methods reduce the number of nonzero accumulators, saving memory space without affecting retrieval effectiveness. And, as examples of more general *query pruning methods*, they can also reduce disk traffic and CPU time, using methods that are discussed shortly.

To maintain the set of accumulators in either of these two approaches, a data structure is required. An obvious solution is to store the accumulators as a list ordered by document number and successively merge it with each term's inverted list.

*Representing Document Lengths.* Storage of the set of document lengths $W_d$ is another demand on memory during ranking. Figures 2, 4, and 12 suggest that the $W_d$ values be stored in a file on disk, but execution is faster if they can be retained in memory.

There are two aspects of their use that make memory residence possible. First, they are an attribute of the document collection rather than of any particular query. This means that the array of $W$ values can be initialized at system startup time rather than for every query and that, in a multithreaded evaluation system, all active queries can access the same shared array.

The second aspect that allows memory-residence is that, like many other numeric quantities associated with ranked query evaluation, they are imprecise numbers and result from a heuristic rather than an exact process. Representing them to 64 or 32 bits of precision is, therefore, unnecessary. Experiments have shown that use of 8- or 16-bit approximate weights has negligible effect on retrieval effectiveness.

*Storage for the Vocabulary.* Except for the document mapping table and assorted buffers, the only remaining demand on main memory is the term lookup dictionary or vocabulary. For collections such as `NewsWire` and `Web`, retaining the vocabulary in main memory is expensive since each entry includes a word, ancillary fields such as the weight of that term, and the address of the corresponding inverted list.

Fortunately, access to the vocabulary is only a small component of query processing; if a B-tree-like structure is used with the leaf nodes on disk and internal nodes in memory, then a term's information can be accessed using just a single disk access, and only a relatively small amount of main memory is permanently consumed. For example, consider the `Web` collection described in Table I. The 16 million distinct terms in that collection correspond to a vocabulary file of more than 320MB but if each leaf node contains (say) 8kB of data, around 400 vocabulary entries, then the in-memory part of the structure contains only 40,000 words, occupying less than 1MB. As for the document weights, this memory is shared between active queries and is initialized at startup rather than once per query.

## 10. REDUCING RETRIEVAL COSTS

The core implementation, together with simple strategies for reducing memory consumption, provides a functional query evaluation system. For example, in the context of a desktop machine and a gigabyte-scale text collection, typical queries of a few terms can be resolved using these techniques in well under a second. In addition, with accumulator limiting, compact document weights, and a B-tree for the vocabulary, the memory footprint can be limited to a dozen or so megabytes. However, substantial further savings can be made, and, with careful attention to detail, a single standard PC can easily provide text search for a large organization such as a university or corporation.

In the core implementation (Figure 4), the principal cost is retrieval and decoding of inverted lists. That is, every $\langle d, f_{d,t} \rangle$ pair contributes to a valid accumulator. With limited-accumulator query evaluation (Figure 12), however, most $\langle d, f_{d,t} \rangle$ pairs do not correspond to a valid accumulator, and processing time spent decoding these pairs is wasted. We also argued earlier that to facilitate compression and the gains that



www.manaraa.com

compression brings, the document numbers in each list should be sorted. However, use of compression means that each number is stored in a variable number of bits or bytes so random access (e.g., to support binary search) into inverted lists is not possible.

*Skipping.* A proposal from the mid-1990s is that decoding costs be reduced by the insertion of additional structure into inverted lists. In this *skipping* approach, descriptors are periodically embedded in each compressed inverted list, dividing the list into chunks. The descriptor can be used to determine whether any $\langle d, f_{d,t} \rangle$ value in the chunk corresponds to a valid accumulator; if not, that chunk does not need to be decoded, and processing can immediately move to the next chunk. However, all chunks still need to be fetched. Note also that, if the chunks are as large as disk blocks, the great majority will need to be decoded, and, if they are smaller, more chunks can be skipped but most disk blocks will still contain a valid chunk. In experiments at the time, we found that skipping yielded worthwhile performance gains. Since then, processors have become much faster, while disk access times have not greatly improved, and the benefits provided by skipping have eroded.

To reduce disk transfer costs, it is necessary to avoid fetching the inverted lists in their entirety. One obvious method is to only fetch inverted lists for rare terms, but experiments consistently show that all terms should be processed if effectiveness is to be maintained. The more attractive option is to rearrange inverted lists so that, in a typical query, only part of each relevant list need be fetched. The schemes described earlier for accumulator limitation provide criteria for deciding whether a $\langle d, f_{d,t} \rangle$ value will be used; we now explain how these criteria can be used to restructure inverted lists without seriously degrading compression effectiveness.

*Frequency-Ordered Inverted Lists.* The principle of accumulator limitation is that large values of $w_{q,t} \times w_{d,t}$ should be processed first. The default heuristic for identifying these values is to process the low-$f_t$ terms first as their lists are likely to contain more large $w_{q,t} \times w_{d,t}$ values than would the lists of high-$f_t$ terms. However, in a typical inverted list, most $f_{d,t}$ values are small, while only a few are large. If only the large $f_{d,t}$ values are interesting, that is, can contribute to a useful $w_{q,t} \times w_{d,t}$ value, then they should be stored at the beginning of their inverted list rather than somewhere in the middle of the document-based ordering. For a given threshold $S$ and term $t$, all $w_{q,t} \times w_{d,t} > S$ values (see Equation (1)) will then be stored before any smaller ones.

To make use of this idea, the index can be reorganized. A standard inverted list is sorted by document number, for example,

$$\langle 12, 2\rangle \ \langle 17, 2\rangle \ \langle 29, 1\rangle \ \langle 32, 1\rangle \ \langle 40, 6\rangle \ \langle 78, 1\rangle \ \langle 101, 3\rangle \ \langle 106, 1\rangle.$$

When the list is reordered by $f_{d,t}$, the example list is transformed into

$$\langle 40, 6\rangle \ \langle 101, 3\rangle \ \langle 12, 2\rangle \ \langle 17, 2\rangle \ \langle 29, 1\rangle \ \langle 32, 1\rangle \ \langle 78, 1\rangle \ \langle 106, 1\rangle.$$

The repeated frequency information can then be factored out into a prefix component with a counter inserted to indicate how many documents there are with this $f_{d,t}$ value:

$$\langle 6 : 1 : 40\rangle \quad \langle 3 : 1 : 101\rangle \quad \langle 2 : 2 : 12, 17\rangle \quad \langle 1 : 4 : 29, 32, 78, 106\rangle.$$

Finally, if differences are taken in order to allow compression, we get

$$\langle 6 : 1 : 40\rangle \quad \langle 3 : 1 : 101\rangle \quad \langle 2 : 2 : 12, 5\rangle \quad \langle 1 : 4 : 29, 3, 46, 28\rangle.$$

Repeated frequencies $f_{d,t}$ are not stored, giving a considerable saving. But within each equal-frequency segment of the list, the $d$-gaps are now on average larger when the document identifiers are sorted, and so the document number part of each pointer



www.manaraa.com

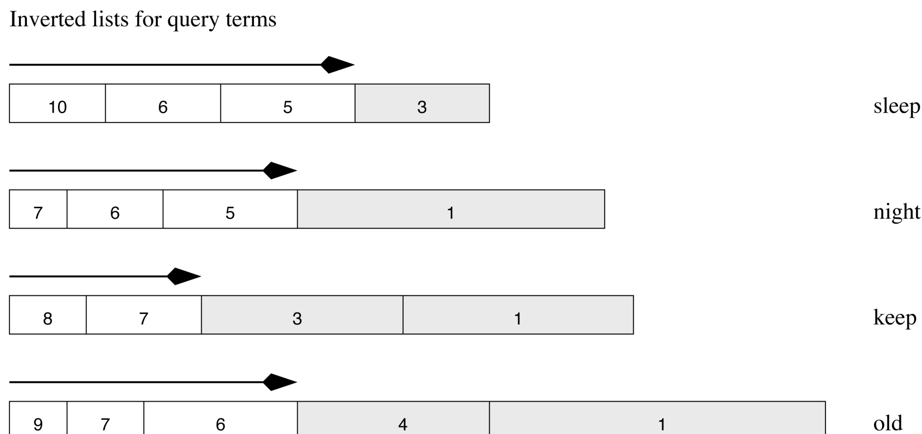


**Fig. 13**. Interleaved processing of inverted lists using a frequency- or impact-sorted index. Blocks from the front of each list are applied to the set of accumulators in decreasing score order until some stopping condition has been met. At the instant shown, 11 blocks have been processed, and, if processing continues, the next block to be processed would be for the term `old`.

increases in cost. In combination, these two effects typically result in frequency-sorted indexes that are slightly smaller than document-sorted indexes.

Using a frequency-sorted index, a simple query evaluation algorithm is to fetch each list in turn, processing $\langle d, f_{d,t}\rangle$ values only while $w_{q,t} \times w_{d,t} \geq S$, where $S$ is the threshold. If disk reads are performed one disk block at a time rather than on a whole-of-inverted-list basis, this strategy significantly reduces the volume of index data to be fetched without degrading effectiveness.

A practical alternative is for the first disk block of each list to be used to hold the $\langle d, f_{d,t}\rangle$ values with high $f_{d,t}$. These important blocks could then all be processed before the remainder of any lists, ensuring that all terms are given the opportunity to create accumulators. The remainder of the list is then stored in document order.

A more principled method of achieving the same aim is to interleave the processing of the inverted lists. In particular, once the first block of each list has been fetched and is available in memory, the list with the highest value of $w_{q,t} \times w_{d,t}$ can be selected, and its first run of pointers processed. Attention then switches to the list with the next-highest $w_{q,t} \times w_{d,t}$ run which might be in a different list or might be in the same list.

During query processing, each list could be visited zero, one, or multiple times, depending only on the perceived contribution of that term to the query. The interleaved processing strategy also raises the possibility that query evaluation can be terminated by a time-bound rather than a threshold since the most significant index information is processed first. Figure 13 shows the interleaved evaluation strategy with list sections presumed to be applied to the set of accumulators in decreasing order of numeric score. The shaded sections of each list are never applied.

*Impact-Ordered Inverted Lists*. An issue that remains even in the frequency-sorted approach is that $w_{q,t} \times w_{d,t}/W_d$ is the true contribution of term $t$ to the similarity score $S_{q,d}$, rather than $w_{q,t} \times w_{d,t}$. If it makes sense to frequency-sort the inverted lists into decreasing $w_{d,t}$ order, then it makes even better sense to order them in decreasing *impact* order using $w_{d,t}/W_d$ as a sort key. Then all that remains is to multiply each stored value by $w_{q,t}$ and add it to the corresponding accumulator. That is, an impact value incorporates the division by $W_d$ into the index.



المنارة للاستشارات
www.manaraa.com

*To use an impact-sorted inverted index and an accumulator limit $L$ to rank a document collection with regard to a query $q$ and identify the top $r$ matching documents*:

(1) Create an empty set $A$ of accumulators.
(2) Fetch the first block of each term $t$'s inverted list. Let $s_t$ be the stored impact score for that block.
(3) While processing time is not exhausted and while inverted list blocks remain
  (a) Identify the inverted list block of highest $w_{q,t} \times s_t$, breaking ties arbitrarily. Let $C$ be the integer contribution derived from $w_{q,t} \times s_t$.
  (b) For each document $d$ referenced in that block,
    i. If $A_d$ does not exist and $|A| < L$, create an accumulator $A_d$.
    ii. If $A_d$ exists, set $A_d \leftarrow A_d + C$.
  (c) Ensure that the next equi-impact block for term $t$ is available, and update $s_t$.
(4) Identify the $r$ greatest $A_d$ values and return the corresponding documents.

**Fig. 14**. Impact-ordered query evaluation.

If the inverted lists are then sorted by decreasing impact, the same interleaved processing strategy can be employed. Now the blocks in the inverted list must be created artificially rather than occurring naturally, since $w_{d,t}/W_d$ is not integer-valued, and exact sorting would destroy the $d$-gap compression. Hence, to retain compression, the impact values are quantized so that each stored value is one of a small number of distinct values, and integer surrogates are stored in the inverted lists in place of exact $w_{d,t}/W_d$ values. Experiments with this technique have shown that approximately 10–30 distinct values suffice to give unaltered retrieval effectiveness compared to full evaluation. And, unlike frequency-based methods, impact-sorted lists are just as efficient for evaluation of the Okapi similarity heuristic described in Equation (2).

With this change, each list is a document-sorted sequence of blocks of approximately-equal-impact pointers, and no $f_{d,t}$ values are stored. Compared to document- and frequency-sorted indexes, the compressed size grows slightly because the average $d$-gaps are bigger. Nevertheless, the index is still a small fraction of the size of the text being indexed.

Query evaluation with an impact-ordered index becomes a matter of processing as many blocks as can be handled in the time that is available. To process a block, an integer contribution score that incorporates the impact score associated with that block and the query term weight $w_{q,t}$ (which might be unity) is added to the accumulator of every document recorded in the block. All query-time operations are on integers, also making the final extraction phase significantly easier to manage. Impact-ordered query evaluation is shown in Figure 14.

With an impact-sorted inverted file and integer surrogate weights, query processing is extremely fast. Just as important is that time consumed by each query can be bounded, independent of the number of terms in the query. When query load is light or when a premium service has been purchased, a greater fraction of the blocks in each query can be used to influence the result ranking.

Impacts provide another illustration of the flexible constraints on design of algorithms for information retrieval. Information can be selectively thrown away or approximated without detriment to the output of the algorithm, although there may be a beneficial change to efficiency. While the set of documents returned by impact-based processing will not be the same as the set returned by the underlying computation in Equation (1), the quality of the set (proportion of documents that are relevant) may be unchanged. In practice, the effect is likely to be that one method gives better results



المنارة للاستشارات
www.manaraa.com

than the other for some queries and worse results than the other for other queries. Such change-but-no-difference effects are commonly observed in development of algorithms for text query evaluation.

*Other Considerations.* When the inverted lists are impact- or frequency-sorted, they are read in blocks rather than in their entirety, and contiguous storage is no longer a necessity. Blocks with high-impact information could be clustered on disk, further accelerating query processing. The lists for common terms—the ones with many document pointers—will never be fully read, and a great deal of disk traffic is avoided. Experiments with these techniques, using collections such as `NewsWire`, show that disk traffic and processing time are reduced by a factor of up to three compared to previous techniques involving limited accumulators. The likely gain is even greater for the larger `Web` collection. As a final advantage, the impact-sorted index dispenses with the need for an array or file of $W_d$ values.

The disadvantage of both frequency- and impact-sorted indexes is that Boolean queries and index updates are more complex. To handle Boolean queries efficiently, other index structures have been proposed. One approach is to use a *blocked* index. Rather than use $d$-gaps that are relative to the previous document number, it is feasible to code $d$-gaps relative to the start of a block. This allows use of a form of binary search at some loss of compression efficiency. Conjunctive Boolean queries can then be processed extremely quickly.

However, updates remain complex. With document-ordered list organizations, new $\langle d, f_{d,t} \rangle$ values can be appended to the list. In the enhanced organizations, the whole of each list must be fetched and decoded. Whether this is a significant disadvantage depends on the application.

*Summary.* We have described compression-based indexing and query evaluation techniques for bag-of-word queries on a text database. Compared to the uncompressed, document-sorted implementation described in Figure 4,

—*disk space* for a document-level index requires about 7%–9% of the size of the data and has been reduced by a factor of around five;

—*memory space* for accumulators requires a dozen bytes for 5% of the documents in a small collection or less than 1% of the documents in a large collection and has been reduced by a factor of around twenty;

—*CPU time* for processing inverted lists and updating accumulators is reduced by a factor of three or more;

—*disk traffic* to fetch inverted lists is reduced by a factor of five in volume and of two or more in time (The relative savings increase with collection size);

—*throughput* of queries is reduced by making better use of memory, allowing more effective caching of vocabulary entries, lists, and answers;

—*construction time* for indexes is cut by a factor of more than two, through reduction in the disk-traffic bottleneck and reduction in the number of runs required to build an index.

## 11. OTHER APPROACHES TO INDEXING

Querying can also be carried out without an index. For typical data and queries, building an index requires about 10–50 times the cost of a string search. If queries are rare, or the data is extremely volatile, it is reasonable to use a document-oriented querying strategy of the type shown in Figure 2. An example is searching of a personal email directory or search within a small collection of files.



المنارة للاستشارات
www.manaraa.com

A. *Original string, indexed by ordinal position:*

```
0         1         2         3         4
01234567890123456789012345678901234567890123456
The old night keeper keeps the keep in the town
```

B. *Selected character-pairs in the original string, indexed by ordinal position:*

```
00    04    08    14    21    27    31    36    39    43
Th..  ol..  ni..  ke..  ke..  th..  ke..  in..  th..  to..
```

C. *The same character-pairs, after lexicographic sorting:*

```
36    31    14    21    08    04    27    00    39    43
in..  ke..  ke..  ke..  ni..  ol..  th..  Th..  th..  to..
```

**Fig. 15**. Example of a suffix array.

For querying of larger collections, inverted files are not the only technology that has been proposed. The two principal alternatives are suffix arrays and signature files. In this section, we briefly outline these approaches and explain why they are not competitive with inverted files for the applications considered in this article.

*Suffix Arrays.* A suffix array stores pointers into a source string, ordered according to the strings they point at. The index can be of every character position or, for typical retrieval applications, every word position. The structure provides access via a binary search-like process which is fast if it can be assumed that both the array and the underlying source string are available in memory.

Consider the first line of the `Keeper` collection annotated with byte offsets in the string, commencing at zero, as shown in part A of Figure 15. In a word-level suffix array, a list is made of the byte addresses at which words start, shown in the first row of part B of Figure 15. The second row of part B shows the initial two characters of the semi-infinite string that is the target of each pointer. To form the suffix array, the byte pointers are reordered so that the underlying strings are sorted, using as many characters of the semi-infinite strings as are necessary to break ties, as shown in part C. Note that the second of these two lines is provided only to allow the destinations of the pointers to be visualized by the reader. All that is actually stored is the original string, and an array `A` containing one pointer for each word in the string, with `A[0]=36`, `A[1]=31`, and so on.

One way of interpreting the resultant array of byte pointers is that they represent a combined vocabulary and inverted index of every suffix string in the original text with access provided directly to the bytes rather than via ordinal document identifiers. To use the suffix array, the sorted byte pointers are binary searched, again using the underlying strings to drive the comparisons. The output of each search is a pair of indices into the suffix array, describing the first and last locations whose strings are consistent with the supplied pattern. For example, to locate occurrences of `keep` in the example array, the binary search would return 1 and 3 since all of `A[1]`, `A[2]`, and `A[3]` contain byte addresses (31, 14, and 21, respectively) pointing at semi-infinite strings in the original text that commence with the prefix `keep`. (If exactly the word `keep` was required rather than as a prefix, a trailing delimiter could be added to the search key, and then only `A[1]=31` would be returned.)

It is straightforward to extend suffix arrays to phrase searching, and a range of simple techniques, such as conflating all nonalphabetic characters to a single separator character for the purposes of string comparison, allow the same binary-searching process to be applied regardless of the length or frequency of the phrase or words in it.



المنارة للاستشارات

www.manaraa.com

Case-folding can also be achieved by making the string comparison function case-insensitive as has been presumed in the example.

For large-scale applications, suffix arrays have significant drawbacks. The pointer array is accessed via binary search so compression is not an option. For a word-aligned suffix array, a 4-byte pointer is needed for each 6 bytes or so of text, and the underlying text must also be retained. In total, the indexing system requires around 170% of the space required by the input, all of it memory-resident. A suffix array-based retrieval system for 1GB of text demands a computer with 2GB or more of memory, a requirement that scales linearly as the source message grows. Suffix array indexes are large compared to inverted files because the repeated information cannot be usefully factored out and because byte addresses are used rather than word-offset addresses. If the base text is large, the suffix array itself can be indexed and partial strings stored in index to avoid needing to access the base text. For very large texts, the pointers can be managed hierarchically. However, these techniques add considerable complexity to the building and searching processes.

Another drawback is that there is no equivalent of ranked querying. All but simple stemming regimes are also problematic. On the other hand, suffix arrays offer increased string searching functionality compared to inverted files—complex patterns, such as wild-characters, can be handled. The principal strength of suffix arrays is that they greatly accelerate `grep`-style pattern matching, swapping decreased time for increased space.

Worth noting is that much of the functionality offered by a suffix array can be achieved in the context of an inverted file by indexing the vocabulary either with a suffix array spanning the vocabulary strings themselves or by a secondary inverted index making use of the character bigrams or trigrams that comprise the vocabulary terms.

*Signature Files.* Signature files are a probabilistic indexing method. For each term, $s$ bits are set in a *signature* of $w$ bits. The term descriptors for the terms that appear in each document are superimposed (i.e., OR'ed together) to obtain a document descriptor, and the document descriptors are then stored in a *signature file* of size $wN$ bits for $N$ documents.

To query on a term, the query descriptor for that term is formed; all of the documents whose document descriptors are a superset of the query descriptor are fetched; and the *false matches* are separated from the *true matches* with only the latter returned to the user as answers.

Assuming that all records are the same length, that is, then contain the same number of distinct terms; all terms occurring in over 5% of records are stopped; and there is on average just one false match per single-term query, then the index size is about 20% of the text. With an unstopped index, signatures must be much wider, giving a 40%-of-text index.

Variability in document length significantly complicates these calculations. With a fixed-length signature, more bits are set for long documents than short. These documents are then more likely to be selected as false matches, and because they are long, the cost of false-match checking is further increased. To get a false-match volume of one average document-length, a signature file needs to be much larger than the indicative sizes given here.

The simplest organization for a signature file is a sequence of signatures, which implies that the whole index is processed in response to each query. A range of alternative organizations of two kinds have been proposed. One is bitslicing in which the signature file, viewed as a matrix, is transposed in which case it is only necessary to fetch a small number of slices—perhaps 10 to 15—in response to a query. Each slice has one bit per stored record, and so, for the `Web` collection, would contain 1.5MB of data.



المنارة للاستشارات
www.manaraa.com

Another alternative organization is the use of descriptors in which signatures are partitioned according to a small number of selected bits and only matching partitions are fetched for each query. We are unaware of any practical demonstration of the merits of such descriptor schemes. Bitslicing, however, has been explored experimentally, including a range of variations designed to reduce costs. For example, slice length can be dramatically reduced with a small increase in the number of slices fetched, although costs remain linear in the number of stored documents.

Even for the relatively unsophisticated task of Boolean retrieval, signature files have several crucial defects. One is the need to eliminate false matches. For a given signature width, the number of false matches is linear in the collection size and hence, as the number of indexed documents grows, the number of documents unnecessarily fetched increases. For any reasonable parameter settings, this cost must dominate. And the longer a document, the higher the likelihood that it will be retrieved as a false match. Second, signature file indexes are large compared to compressed inverted files and are no easier to construct or maintain. Third, they require more disk accesses for short queries—exactly the kind that is commonplace in Web searching. And fourth, they are even more inefficient if Boolean OR and NOT operators are permitted in addition to AND.

More critically, there is no sensible way of using signature files for handling ranked queries or for identifying phrases. That is, there are no situations in which signature files are the method of choice for text indexing even for exact-match Boolean style searching.

## 12. BEYOND AN ELEMENTARY SEARCH ENGINE

We have outlined indexing and query evaluation algorithms that can be used in a practical search engine. These algorithms can be used to index large volumes of data and provide rapid response to user queries. However, in specific searching applications, further improvements in performance are available. In this section, we briefly review some of these options.

*Crawling.* Data acquisition is a key activity in dynamic systems. In unusual cases, the revised content will be delivered as it is generated, and we need do no more than index it. For example, a commercial arrangement might result in the daily delivery of updated catalog entries for an online store.

More usually, however, the owners of data change it without regard for whether or not it is indexed. *Crawling* is the process of seeking out changed data and feeding it back to the retrieval system for incorporation into updated indexes. In principle, crawling is easy, a set of seed URLs is used as the basis for the search, and those pages are fetched one by one. Each hyperlink within those pages is extracted and, if it has not been explored yet, appended to a queue of pending pages. Eventually, all pages reachable from the seed set will have been accessed, and the process can be repeated.

However, there are many subtle issues that require attention if a crawler is to be efficient. First, not all pages are of the same importance, and, while crawling the CNN Web site at hourly intervals might be appropriate, crawling a set of University pages probably is not. Indeed, the volume of data available might mean that the crawl simply does not finish before it is necessary to start it again. In this case, a strategy for prioritizing page visits and aborting the crawl when only low-priority pages remain may be required.

Second, not all pages are edited at the same rate, and editing may follow a regular pattern. An adaptive crawler might be able to significantly reduce crawl volume by identifying the update strategy that applies to pages and learning which pages are almost never modified.



المنارة للاستشارات
www.manaraa.com

Third, there are crawling traps that need to be anticipated and avoided—unintentional traps such as recursive script-based Web pages that implement calendars with a "next month" link are just one example. Fourth, the crawler needs to be sensitive to the requirements of the site it is visiting and must stagger its requests so as to avoid flooding it over a short period of time. Fifth, the crawler must be alert to the problem of duplicate pages and mirrored sites. And, finally, crawling is expensive because it involves either payment per gigabyte of data fetched (for some connection modes) whether it is useful or not, or paying for a high-capacity connection to the Internet so that the necessary bandwidth can be achieved.

*Caching.* In many contexts, the same queries and query terms recur. For example, many queries to the Web search engines are topical, and queries to a site-specific search engine may also be clustered. (Many of the queries on the Web site of one of our universities contain the query term `results`. Over half of these queries are posed on the two single days after first and second semester on which student grades are released.) A search engine can take advantage of this behavior by caching.

There are several kinds of information that can be cached. The inverted lists that are fetched in response to a query can be explicitly kept in memory. Alternatively, if these lists are maintained in disk blocks, the operating system is likely to cache them automatically on the basis of most-recently-used. As was noted earlier, compression helps by increasing the impact of this caching.

It is tempting to store the vocabulary in memory because doing so means that a disk access is avoided for every query term. However, if the vocabulary is large, keeping it in memory reduces the space available for caching of other information and may not be beneficial overall. Relying on the operating system swapping pages of vocabulary information is almost certainly a poor decision, as the distribution of hot query terms among pages is unlikely to be clustered in a useful way. An alternative is to manage the vocabulary in a B-tree as discussed earlier and to explicitly cache recently-accessed or frequently-accessed query terms in a separate small table.

Perhaps the most effective caching is of answer lists. If queries recur, it makes sense to store their answer lists rather than recompute them. Even keeping them on disk is effective as one short answer set can be fetched much more rapidly than can multiple inverted lists. For a typical search engine, most users will only view the first $r$ answers, but it may be effective to keep the next $r$ answers handy for the small percentage of cases in which they are requested.

Another form of caching is phrase indexing. Indexing all phrases is impractical, but using term-based inverted lists to identify which documents contain a phrase is expensive. The cost of phrase query processing could be significantly reduced if inverted lists are explicitly maintained for common term combinations. That is, the users themselves, via their querying behavior, can guide the evaluation as to which phrases might be worth explicitly storing when the index is next rebuilt.

One of the major bottlenecks of query evaluation is the need to fetch the documents that are presented in the answer lists; most search engines return not only a document identifier but a corresponding document summary or *snippet*. Typically these summaries are based on the query and thus cannot be computed ahead of time. Saving complete answer pages, including all of the required snippets, can thus be a dramatic saving.

*Pre-Ordering.* In the context of Web retrieval, techniques such as Google's PageRank can be used to determine a static score for each page to complement the dynamic score calculated by whatever similarity function is being used. The final rank ordering is then a blend of the static score which can be regarded as an a priori probability ordering that a page is relevant to queries in general, and a dynamic score which reflects the probability that a page is relevant to this particular query.



المنارة للاستشارات
www.manaraa.com

Static rankings are only useful in contexts where additional nontextual information can be used such as (in the case of PageRank) the link structure. The PageRank of a Web page is based on the link structure of the Web. Each page is assigned a score that simulates the actions of a random Web surfer, someone who either with probability $p$ is equally likely to follow any of the links out of the page they are currently on, or with probability $1 - p$ jumps to a new page chosen at random. An iterative computation can be used to compute the long-term probability that such a user is visiting any particular page and that probability is then used to set the PageRank. The effect is that pages with many in-links tend to be assigned high PageRank values, especially if the pages that host those links themselves have a high PageRank; pages with low in-link counts, or in-links from only relatively improbable pages, are themselves deemed to be improbable as answers.

HITS is a similar technique which scores pages as both *authorities* and *hubs*, again using a matrix computation based on the connectivity graph established by the Web's hyperlinks. Loosely, a good hub is one that contains links to many good authorities, and a good authority is one that is linked to from many good hubs. Hubs are useful information aggregations and often provide some broad categorization of a whole topic, whereas authorities tend to provide detailed information about a narrower facet of a topic. Both may be useful to queriers.

Pages might also be assigned static scores contributions based on how similar they are to an amalgam of previous queries, or how often users of the search service have clicked through them to read their underlying content, or how deep they are in the directory structure of that particular site.

Where static weights are available from some source such as PageRank or HITS, they can potentially be used to eliminate some of the cost of ranking. For example, suppose that inverted lists are maintained in decreasing PageRank order. Then a top-$r$ ranking could be rapidly computed by taking the Boolean intersection of the inverted lists corresponding to the query terms, terminating as soon as $r$ documents containing all of the query terms have been found. These would be presented in decreasing PageRank order. If the user then requests more documents, presumably because the top $r$ are unsatisfactory, then the system simply computes the top $2r$, and discards the first $r$ of them.

*User Feedback.* Relevance feedback has been widely explored in the context of information retrieval research. In explicit feedback systems, users identify the answers that are of value (and perhaps others that are not), and this information is incorporated into a revised query, to improve the overall quality of the ranking. Much of this research assumes that queries are independent, and, in practical systems, it has proven difficult to gather relevance assessments from users.

However, in a system processing large numbers of queries, it is straightforward to identify which answers users choose to view. The action of a user clicking on a link can be interpreted as a vote for a document. Such voting can be used to alter the static weights of documents and thus their ordering in the ranking generated for subsequent queries even if the subsequent queries differ from the one that triggered the click though.

*Index Terms.* In structured documents, such as those stored as HTML, different emphasis can be placed on terms that appear in different parts of the document. For example, greater weight might be placed on terms that appear in a document's `<title>` tags or as part of an `<h1>` heading.

Another important difference between Web searching and more general document retrieval is that the anchor text associated with a link in one page is in many cases an accurate summary of what the target page is about. Links are usually manually



المنارة للاستشارات
www.manaraa.com

inserted, and unless they say "click here",[1] represent a succinct assessment to a human author as to the content of the target page. Indexing the anchor text as if it were part of the target page—perhaps even as if it were a title, or heading—can significantly improve retrieval effectiveness in Web search applications.

Another useful technique in Web searching is indexing the URL string itself. For example, the page at `www.qantas.com` shows the word Qantas as an image rather than a textual title and unless the URL were parsed into terms and indexed with the document, would risk not being searchable. Page importance can also be biased according to the length of the URL that accesses it on the assumption that URLs with fewer slashes lead to more important pages.

Finally, it is worth noting that the `<alt>` text associated with images is a useful indication of what the image content is, and indexing of `<alt>` text is part of the Google image search mechanism.

*Manual Intervention.* If some queries are particularly common, then a reasonable strategy for improving perceived performance is to manually alter the behavior of these queries. For example, it might make sense for the term `Quantas` to be manually associated with the Qantas page, given the high incidence of confusion over the exact name. By the same token, if the query `CNN` does not lead directly to `www.cnn.com` as a result of the ranking heuristic, then direct manipulation to make it so could well be appropriate. One of the most common queries in the search engine of one of our institutions is the single word `library`; sadly, the home page of the library is only the fourteenth highest ranked answer.

## 13. BIBLIOGRAPHY

The material presented in the previous sections is based on an extensive literature spanning more than 40 years. This section provides a high-level critical overview of that literature and can be regarded as further reading that augments the tutorial.

Our examination of past work is through the prism of current interests rather than a complete historical study. For example, in the first decades of work on information retrieval, both Boolean and ranked queries were widely investigated but, in the last twenty years, most work on search has focused on ranking; the research on Boolean querying is increasingly of historical interest only and is largely outside our scope. Note, too, that terminology has changed since the 1980s. In many older papers, *retrieval* was taken to mean Boolean matching. Scoring and ranking of matches was often described as *nearest neighbor searching*. Some retrieval papers focused on cluster retrieval, an activity that now receives relatively little attention.

Within the narrow area of inverted file indexing for text, this bibliography is reasonably complete. However, some minor papers have been omitted on grounds such as lack of availability (in particular, technical reports and theses); and, where a preliminary paper has later been expanded and republished, the preliminary paper is not cited. In the noncore areas, such as signature files, the bibliography is far from exhaustive. In these cases, we have focused on papers with innovations in index structures or query evaluation algorithms.

Some related topics are not considered. We do not explore vocabulary representations such as hash tables and B-trees or in-memory text search structures such as suffix arrays; and we do not discuss use of text indexes in other applications or special-purpose

[1] In late 2005, for the query `click here` Google returns the Adobe Acrobat Reader home page, the Macromedia Flash player download page, and the Apple QuickTime download page as the three top-ranked answers because of the hundreds of thousands of sites that point at the these pages using the anchor text "to obtain a copy, click here".



www.manaraa.com

text indexes such as those used for plagiarism detection. Nor do we discuss structured document retrieval, or use of inverted files for relations or other data. The purpose of this bibliography is to review and categorize the literature on free-text indexing and ranked query evaluation.

*Books and Overviews.* In contrast to areas such as database systems, there have been relatively few textbooks on information retrieval or text search. Only three recent books have substantial coverage of algorithms and data structures for information retrieval, those by Frakes and Baeza-Yates [1992], Grossman and Frieder [2004], and, in particular, Witten et al. [1999]. The implementation described in the tutorial is similar to the methods presented by Witten et al. [1999] and, more briefly, in a chapter of Bertino et al. [1997]. However, several of the extensions and methods summarized here were developed since those texts were written.

Baeza-Yates et al. [2002] include an overview of index representation and query evaluation as well as topics such as string searching, suffix arrays, and issues for distributed processing. Frakes and Baeza-Yates [1992] and Salton [1968] provide useful background, but they do not consider current approaches. Faloutsos and Oard [1995] give an introduction to the field of text searching in a technical report. Arusu et al. [2001] review text retrieval in the wider context of Web search, including issues such as distribution and crawling.

There are several older textbooks on information retrieval and retrieval systems (but with only limited material on indexes and indexing) including Salton [1971, 1981]; Lancaster and Fayen [1973]; van Rijsbergen [1979]; and Salton and McGill [1983]. For an overview of information retrieval research through approximately 1995, see Sparck Jones and Willett [1997], an extensive, annotated compilation of many classic information retrieval papers. A text from this same era that does consider indexing and compression—but not ranking—is Heaps [1978] which discusses topics such as implementation issues for search of bibliographic records; in this context, Heaps [1978] reports that index size is expected to be about 70% of data size. For a general introduction to information retrieval including an overview of indexing, see Baeza-Yates and Ribeiro-Neto [1999]. Kobayashi and Takeda [2000] review information retrieval and search on the Web. Voorhees and Harman [2005] is an overview and history of the TREC project. Zobel et al. [1996] discuss criteria against which an indexing method might be assessed.

Books that consider relevant compression techniques include Witten et al. [1999], Moffat and Turpin [2002], Salomon [2000], and Sayood [2000]. Only the first of these discusses the application of compression to indexing.

*Text Search and Information Retrieval.* Segesta and Reid-Green [2002] briefly review a 1953–8 implementation of associative text indexing using punch cards and magnetic tape. The indexes were, in modern terms, inverted lists represented as bitmaps; manual searching was reportedly easier than search via the computer. Ivie [1966] describes a range of ways similarity computations might be performed, and Matthew and Thomson [1967] describe a successful implementation of a tape-based inverted index in which answers are ranked according to user-supplied term importance. Salton [1968] describes a rich system that makes use of syntactic analysis, thesauri, dictionaries, and the vector space model to search tapes for matches. Salton [1972] gives reasons why—with the machines, algorithms, and collections of the time—inverted files are not always practical. Haskin [1980] reports that "space estimates for storing [an inverted] index range from 50% to 300% of the space needed to store the text itself", a quotation that continues to be used to justify research into alternative indexing methods despite the fact that it refers to a primitive form of inverted file with storage of redundant



المنارة للاستشارات
www.manaraa.com

information and no compression. The 50%–300% figure is originally due to Bird et al. [1978] who report that the inverted file sizes of a range of then current systems "range in size from .5 to 3 times the size" of the indexed data. The exact structure of the inverted indexes in these systems is not given, but each posting appears to contain a search key. Thus this widely-cited figure is based on structures quite different from inverted files as understood today.

Bird et al. [1978] also report the cost of a search on a full-text database: typically $20 to $30 per query, and sometimes as high as $100. Assuming 2005 commodity hardware pricing; a cost multiple of ten to allow for data gathering, software purchases, and maintenance; and (very conservatively) one second per search; then a typical query today might cost 0.03 cents, around 100,000 times less. That is, on average the cost has halved every eighteen months, not even taking into account that the volume of data being searched has probably grown by a factor of 1,000 or more during the same period, while the cost of accessing disk, then as now a key bottleneck in query processing, has fallen by a factor of no more than 100. Bird et al. [1977] state that "it has become obvious to many [researchers in the] text retrieval disciplines that new hardware devices are needed; continued reliance on software ingenuity will probably not produce a spectacular breakthrough". The gains of the last three decades have not confirmed their pessimistic assessment.

Harper [1982] tested four commercial systems (ASSASSIN, BASIS, STAIRS, and STATUS) and notes that all use a form of inverted file; for the 62MB collection used in the experiments, the block-oriented index of STATUS was estimated to require around 45MB. A categorization and partial bibliography of pre-1983 papers on indexing and search is given by Eastman [1983]; this paper demonstrates that a wide range of indexing methods were under consideration at the time. A corresponding survey by Faloutsos [1985a] also considers a range of techniques. Faloutsos [1985a] reports that inversion is "adopted in most of the commercial systems", but has significant disadvantages, and that, "for the office environment", the signature method is the most promising. These conclusions have been invalidated by subsequent work.

Cluster retrieval was another alternative that is now little investigated. In some older textbooks, cluster searching is regarded as the most promising technique for search. However, Voorhees [1986] compared cluster and inverted file search and found that inverted file search is more efficient. Salton [1989] wrote that "inverted-file search is substantially faster ... it is likely that inverted file searches will always remain faster than cluster searches, no matter how refined a cluster organization is available". Can [1994] proposed a combination of cluster search and inverted file search, arguing that it is advantageous to organize inverted list entries by cluster, but interest in cluster-based searching has greatly declined since the 1980s; there is little evidence that clustering provides a viable alternative to inverted indexes.

Early work on inverted files for text used methods developed for databases such as that of Bayer and McCreight [1972], Cardenas [1975], and McDonell [1977]. In these approaches, the cost of update was a key concern, leading to representations of inverted lists in sequences of blocks. The focus on reducing update costs thus led to increased space consumption and slower query evaluation. Stellhorn [1977] considered inverted files designed for text, arguing for the use of special-purpose hardware to allow rapid merging; the arguments do not apply to current systems.

Much of the work on inverted indexes assumes that lists are directly managed via a file system, an approach that allows control over fragmentation and use of special-purpose free-space management algorithms. Zobel et al. [1993b] consider layout of lists on disk and management of short lists with blocks, showing that their hybrid approach can lead to low fragmentation.



المنارة للاستشارات
www.manaraa.com

Alternatively, relational database systems can be used as file systems, for example, an inverted list can be stored as a contiguous blob in a record. Such an approach is explored by, among others, Brown et al. [1994], Melnik et al. [2001], and Vasanthakumar et al. [1996], who show that it simplifies implementation. There are costs, however; in addition to the extra layer of management that is introduced by use of a database system for file management, this organization removes flexibility such as the ability to retrieve lists in stages and to ensure contiguous storage. Grabs et al. [2001] consider whether a distributed relational database provides advantages for information retrieval compared to a monolithic relational database, but the generality of the results is unclear due to use of a collection that fits in memory and lack of explanation of query evaluation techniques. The potential advantages during update are considered by Brown et al. [1994], but this method assumes segmentation of lists into 8-kilobyte blocks and does not allow efficient query processing.

Standard relational indexing techniques are an alternative to indexes designed for text. Numerous approaches of this kind have been described in which an inverted list for a term is represented typically by a series of tuples, one per document containing the term. Such implementations are orders of magnitude slower than text-specific designs, and we do not explore this literature of unsuccessful methods. Also, there have been many descriptions of special-purpose implementations; most of these are of at best limited interest.

*Similarity Measures.* Improvement in similarity measurement is a key aim of information retrieval research and, over the years, has been the subject of hundreds of papers in the annual SIGIR Information Retrieval Conference. A survey of even the principal contributions is beyond the scope of this article, and we simply note a small number of key papers.

The most effective similarity formulations have been based on models of documents, queries, and the retrieval process. One such approach is the vector space model, formally explained by Salton et al. [1975] but in use much earlier. The first reference to the cosine measure that we know of is by Salton [1962] who considers "the vectors of document properties as vectors in $m$-space, and [takes] as a distance function the cosine of the angle between each vector pair".[2] Other early work is by Ivie [1966] who also considered similarity functions.

The cosine measure—which has consistently been the most successful similarity measure described under the vector space model—has undergone continuing refinement, in particular with regard to term weighting. Arguably, the most recent significant innovation was the introduction of document-length pivoting [Singhal et al. 1996] which addresses the issue that document length and likelihood of relevance are correlated. A range of other extensions of this type, for instance, introduction of a tunable constant through the use of training data, have followed.

For much of the history of information retrieval, the principal alternative to the vector space model has been probabilistic models. Statistical approaches to information retrieval were articulated during the early days of document computing; for example, Luhn [1957] wrote that

> a standard program could serve to direct the machine to compare the question [with] the documents of the collection. Since an identical match is highly improbable, this process would be carried out on a statistical basis by asking for a given degree of similarity.

[2]We thank Michael Lesk for locating and forwarding to us this key early information. Also worth noting is that the Salton Award, presented every three years for research in the area of information retrieval, is named after Gerard Salton [1927–1995], the pioneering researcher who penned this statement.



المنارة للاستشارات
www.manaraa.com

The probabilistic model's basis is the probability ranking principle, explored by Maron and Kuhns [1960] and later formalized by Robertson [1977]. Similarly, Edmundson and Wyllys [1961] wrote:

> In preparation for the widespread use of automatic scanners which will read documents and transmit their contents to other machines for analysis, this report presents a new concept in automatic analysis: the relative-frequency approach to measuring the significance of words, word groups, and sentences.

Many of the underpinnings of probabilistic information retrieval are summarized by Sparck Jones et al. [2000] who give a detailed derivation of the probabilistic similarity measure often referred to as the Okapi measure or BM25 [Robertson et al. 1994].

A recent innovation is the use of language models, introduced by Ponte and Croft [1998], where similarity is based on the kinds of word probability estimates used in statistical modeling [Manning and Schütze 1999] and text compression [Bell et al. 1990]. Many of the key issues in language models are examined in the papers collected by Croft and Lafferty [2003]. Language modeling is an active area of research.

We are aware of no recent surveys or overviews of similarity measurement. Harman [1992] lists a range of similarity measures. Salton and Buckley [1988b] surveyed different retrieval mechanisms and introduced notation to describe them; Zobel and Moffat [1998] combined a range of measures into an orthogonal framework and compared them on several different test collections.

*Phrase Querying.* An aspect of query evaluation that has attracted only a little published research is the algorithmic problem of ranking with phrases. Lee et al. [1996] compare inverted files and signature files for keyword-based retrieval of structured documents. Williams et al. [1999] describe an index organization for word pairs, allowing phrase queries of arbitrary length. Williams et al. [2004] show that a small, partial phrase index can be used in conjunction with a conventional index and a phrase cache to achieve most of the speed gains attained by explicit word-pair indexes at much lower space overhead.

*Index Construction.* Although the problem of index construction for text search has similarities to both standard index construction and external sorting, there are particular features of text indexes that mean that special-purpose algorithms can be more efficient than a general-purpose algorithm.

There are several alternative approaches [Witten et al. 1999, Chapter 5]. Fox and Lee [1991] proposes a two-pass method in which the index is structured during one pass through the data and constructed during multiple second passes. This method requires little overhead space but substantial overhead time for the repeated passes over the data.

Many authors, well into the 1990s, appear to have assumed that indexes would be constructed with the methods used for relational databases. Others assumed that an index should be built by generating tuples representing list entries in occurrence order and sorting them into term order. Harman and Candela [1990] and Rogers et al. [1995] explored a method based on writing temporary list entries to disk which, while faster than the simplistic sorting methods, is still impractical. Moffat [1992] describes a scheme that assumes that the entire compressed index can fit into main memory. Moffat and Bell [1995] describe a process that makes use of all of compression, in-place on-disk sorting, and one-pass processing to build a compressed index within only a moderate amount of memory in excess of that required by the vocabulary. Heinz and Zobel [2003] experimentally examine several techniques and demonstrate that a straightforward approach based on building and merging runs in limited memory is, in practice, the most efficient.



المنارة للاستشارات
www.manaraa.com

*Index Maintenance.* Cutting and Pedersen's [1990] work was one of the first examinations of the particular problems posed by update of text indexes, including sequential management of list entries, and makes practical recommendations that are still relevant such as the need to make updates in batches rather than term-at-a-time or document-at-a-time. Brown et al. [1994], as noted earlier, use a relational system to manage list entries as they are incrementally modified. Motzkin [1994] describes a method based on storing list entries in a B-tree, giving analytical results, but the small blocks of list entries that result are likely to lead to inefficient query evaluation. Burkowski [1990] considers the problem of free-space management in the context of an index organized for fast updates; while the issues raised are interesting, the solution is inapplicable to current implementations of indexing.

Lester et al. [2006] experimentally compare different approaches to update, finding that, for large numbers of new documents, in-place update is costly compared to merging old and new indexes. Tomasic et al. [1994] and Shoens et al. [1994] evaluate incremental update strategies on synthetic data, considering a range of approaches to management of space allocation and free lists. Clarke et al. [1994] and Clarke and Cormack [1995] consider the costs of batch update strategies in the context of a method for updating in stages. Lester et al. [2005] and, in a case of independent discovery, Büttcher and Clarke [2005] describe the intermittent merging approach in which the index is partitioned into a sequence of fragments of increasing size. This strategy reduces both the underlying complexity and the in-practice costs.

Barbará et al. [1996] consider caching strategies for reducing the costs of updates and queries, arguing afresh that buffering of updates is likely to significantly reduce costs. Much of the paper considers locking strategies, an operation that is neglected (or viewed as unnecessary) in many text retrieval applications. Shieh and Chung [2005] propose an over-allocation strategy designed to balance space wastage against the cost of too-frequent list relocation. The results in this paper and others indicate that some form of over-allocation is essential. Lim et al. [2003] reconsider an assumption widely made in older work, that document update or deletion is rare, noting that this assumption is false for Web data. Using a simple index representation, they show that update costs can be reduced through use of structured rather than ordinal word positions.

*Distributed Information Retrieval.* There is a wide literature on distributed information retrieval. Many of the papers in this area concern topics such as database selection and result fusion rather than index structures or algorithms for query evaluation. However, there are numerous exceptions. Most of the algorithm-oriented papers on distributed information retrieval concern proposals for specific architectures and measurement or estimation of efficiency under the architecture.

Stanfill et al. [1989, 1999] describe how an index might be distributed among a large number of tightly coupled low-power parallel processors. The underlying index design is relatively simplistic, and it is not clear that the predicted speedup, based on extrapolating to collections thousands of times larger than those tested, could have been achieved in practice. Cringean et al. [1990] explore similar issues with similar limitations in a transputer network. A related design was explored by Reddaway [1991]. Salton and Buckley [1988a] critically evaluate these approaches. Couvreur et al. [1994] compare the suitability of different index types (inverted files, signature files, and special-purpose hardware) for parallel text search for both Boolean and wildcard queries.

Harman et al. [1991] describe an implementation based on document distribution across servers with separate collections, used as a library information system. This system was successfully used in practice. Macleod et al. [1987] use simulation to compare approaches to distribution, including having documents on one server and the



المنارة للاستشارات
www.manaraa.com

index on another, as well as the more usual approaches to division of workload via partitioning and replication. They found distribution to be advantageous but under assumptions that do not correspond to current systems such as low workload and Boolean querying. Martin et al. [1990] found via simulation that caching of material at client or server can improve performance. Again, many of the assumptions are not applicable to current systems, but one key finding—explicit caching is superior to relying on the operating system's caching—remains valid. Martin and Russell [1991], however, question whether client-side caching is of value.

Cahoon and McKinley [1996] and Cahoon et al. [2000] describe a distributed system built on the Inquery server, showing (primarily by simulation) that distributing documents among servers can improve response until the client becomes choked. Large gains in performance were obtained by reducing the numbers of answers required. Lu and McKinley [2003] report substantial further gains with carefully targeted replication, an alternative or enhancement to caching in distributed systems.

Riberto-Neto et al. [1999] explore alternative ways of building a term-distributed static index in a distributed environment, showing that it is best to distribute inverted lists during construction of runs. This means that the vocabulary distribution must be known in advance.

Barroso et al. [2003] provide an overview of the Google architecture. In this system, the index is distributed document-wise among several servers, giving a server cluster. Documents are stored on separate machines. An alternative document-distributed architecture is explored by Hawking [1997] with experiments showing that it scales well. Another document-distributed architecture is explored by Melnik et al. [2001]. de Kretser et al. [1998] outline how queries on a document-distributed system might be evaluated, considering communication costs and correspondence to a monolithic system, arguing that a key optimization is to minimize the number of handshakes between clients and servers.

Xi et al. [2002a, 2002b] describe an approach in which each node contains a fixed fraction of each inverted list. Whether this approach is in practice significantly different to document distribution is unclear; their experiments indicated small improvements over a short sequence of queries and used the assumption that lists have to be completely processed during query evaluation.

Jeong and Omiecinski [1995] compare document distribution to term distribution, finding experimentally (but on artificial data) that document distribution is likely to be superior in practice. MacFarlane et al. [2000], using a large data set, a small number of real queries, and a single serialized client, fount that document partitioning is superior. Tomasic and García-Molina [1993] compare distributed index organizations, using simulation, and found that document distribution is superior to term distribution for Boolean queries. Tomasic and García-Molina [1996] undertook a similar investigation, using bibliographic records instead of full-text documents, and found that distribution without replication is unhelpful. Ribeiro-Neto and Barbosa [1998] compare term and document distribution with simulations and found that term distribution is likely to outperform document distribution. Badue et al. [2001] compare the same approaches, finding experimentally with artificial queries that term distribution is clearly superior. Cacheda et al. [2004] compare distribution with replication, finding the latter to be more efficient, using simulation, artificial queries, and extrapolation; due to simplifying assumptions such as that the servers are memoryless, the results are unrealistic. However, this is an active area of research; at the moment there is no clear best design, and choices between approaches depend on a range of machine and application specifics.

More recently, Moffa et al. [2005] describe a term-oriented system in which partial answers are transferred between servers rather than inverted lists. Their results show



المنارة للاستشارات
www.manaraa.com

that the revised method is competitive with document distribution in terms of query throughput but has problems with load balancing and is thus not as scalable.

The contrasts between these many papers highlight the difficulties in experimental or simulated work of this kind: high throughput can be achieved at the expense of poor response time to individual queries; caching of inverted lists and answer sets affects different architectures in different ways; if indexes are smaller than available memory, timings may be highly unrealistic; the relative importance of the component costs changes dramatically with the volume of data to be indexed; and artificial queries are likely to be unrealistic. These kinds of problems lead to question marks over much of this work. In some papers, for example, it is assumed that the distribution of query terms is the same as that of document terms, leading to the conclusion that long lists must be processed in full for typical queries. Failure to model caching effects leads to drastic inaccuracies; experiments with small numbers of queries, even if realistic, say little about behavior with large numbers of queries where repetition is observed.

Comparisons are meaningful only if both implementations are of high quality or if, improbably, in our view, simulations are fully realistic without simplifying assumptions. Moffat and Zobel [2004] comment on some of these issues.

*Efficient Index Representations.* The majority of the representations used in index compression are based on integer coding methods, in particular those of Golomb [1966], Gallager and Van Voorhis [1975], Elias [1975], and Rice [1979].

Early approaches to index compression were based on interpreting inverted lists as bitmaps. (Most of these schemes did not consider in-document frequencies but, in hindsight, incorporation of these frequencies would be straightforward.) Jakobsson [1978] gives a compression method based on interpreting bit-vectors as sequences of $k$-bit symbols each of which can be allocated a Huffman code. Fraenkel and Klein [1985] explore variants of this scheme. Bookstein and Klein [1992] and Choueka et al. [1986] describe a refinement in which an initial index is used to eliminate blocks in which all bits are zero. All of these schemes, at best, simply approximate the Golomb code and its excellent behavior when the appearances of each term are random in the set of documents. Other refinements are described by Bookstein and Klein [1991a, 1991b, 1991c].

Schuegraf [1976] described another variant based on bitmap compression, using analysis, to demonstrate that, in principle, large savings were available. However, it is not clear that this method is significantly different from the other bitmap-based approaches discussed previously.

Choueka et al. [1987] extended the bitmap approach to allow determination of whether words occur in proximity to each other. Compression of a word-level index is explored by Choueka et al. [1988], using a range of purpose-designed representations for tuples representing word positions. Some of the later methods can be viewed as generalizations, with greater compression effectiveness, of this approach. Klein et al. [1989] briefly outline an organization for an inverted index based on these tuples in which word positions are described as coordinates of document, paragraph, and sentence number in a static database, allowing space savings compared to an uncompressed representation.

Prior to much of this work, Teuhola [1978] proposed use of Golomb-like codes for compressing bit vectors, a principled approach that achieves better compression than many of the alternatives described since.

The compressed index representation outlined in Section 8 was developed in a sequence of papers in the early 1990s [Witten et al. 1991, Moffat and Zobel 1992a, 1992b; Zobal et al. 1992; Bell et al. 1993]. Refinements cover including additional structure to accelerate processing [Moffat and Zobel 1996; Anh and Moffat 1998]; exploiting any



www.manaraa.com

clustering that may be available [Moffat and Stuiver 2000; Blandford and Blelloch 2002; Silvestri et al. 2004]; exploration of bytewise codes [Williams and Zobel 1999; Scholer et al. 2002; Brisaboa et al. 2003; Culpepper and Moffat 2005]; exploration of fixed binary codes [Anh and Moffat 2005]; suppression of unimportant index information [Carmel et al. 2001]; and experimental comparison of alternative representations [Scholer et al. 2002; Trotman 2003]. Much of the current research in the area of efficient indexing and query evaluation is based on these papers.

Linoff and Stanfill [1993] describe an approach based on bitwise codes that, in the best case, are equivalent to an Elias gamma code. The compression achieved is poor. Shieh et al. [2003] propose reorganization of lists into trees to allow fast Boolean and ranked querying. It seems unlikely that this approach is practical as the reorganization appears to introduce large numbers of pointers. Use of Bayesian models is explored by Bookstein et al. [2000]; the results, on common terms in small collections, are not comparable to those in other work. Shieh et al. [2003] describe a similarity-based reordering of the stored documents to reduce median gap values and thus improve compression with Elias codes. Such techniques may be of value for static collections.

Faloutsos and Jagadish [1992] analyze search and construction times for an uncompressed index that is a mix of inverted lists and bitmaps (without word frequencies), and consider practical issues such as bitmap update. Such methods are superseded by the implementation described in the tutorial, as a posting list with Elias or Golomb coding is almost always shorter than the corresponding bitmap regardless of term frequency.

*Limiting Memory Requirements.* Early implementations of ranked query evaluation assumed an array of accumulators, one for each document in the collection, as explained, for example, by Buckley and Lewit [1985]. Perry and Willett [1983] survey ranking methods based on inverted files. They report that, prior to about 1980, while some papers described systems that directly ranked documents, the "great majority of . . . interactive document retrieval services are based on partial match, Boolean search procedures" [Perry and Willett 1983]. Noreault et al. [1977] showed that inverted files can be used to support ranking with only slightly increased costs compared to Boolean querying.

For a query of many terms, a key optimization to standard ranking methods is to avoid processing of inverted lists corresponding to words of low weight. Smeaton and van Rijsbergen [1981] and Buckley and Lewit [1985] describe early schemes of this kind, using term statistics to determine when the set of top documents cannot be altered by further index processing (though their order might change). Lucarella [1988] and Wong and Lee [1993] describe further variants to these schemes. For short queries, however, these approaches cannot be effective, and the overhead of dynamically maintaining the necessary statistics during query evaluation is high. Perhaps most seriously, as collection size grows, the difference in score between highly-ranked documents becomes small, and the termination condition is unlikely to be met until it is too late to achieve significant savings.

An alternative stopping condition was described by Moffat and Zobel [1996]. In this approach, the total number of accumulators was fixed. Only the document identifiers corresponding to the first inverted lists processed could be included. Once the limit in the number of accumulators is reached, query processing can stop—giving the final set of documents, but possibly in a poor order—or can continue, updating existing accumulators but not adding new ones. Persin et al. [1996] described a thresholding method for throttling the number of accumulators created, without specifying a precise limit. The impact-sorted indexes of Anh et al. [2001] and Anh and Moffat [2002] also support early query termination, trading effectiveness for efficiency. Lester et al. [2005] described a thresholding method designed for Web-style queries where accumulators



المنارة للاستشارات
www.manaraa.com

are both created and discarded as each list is processed. With the growth in the size of typical collections, all practical implementations now use some method for limiting numbers of accumulators.

In uncompressed inverted lists, a common assumption in work on query evaluation prior to 1993 or so, the document weights could be incorporated into the inverted lists. With the compression methods described in the early 1990s, it was much cheaper to store frequencies (which are small integers, whereas normalized frequencies are floating point numbers drawn from a large domain). Moffat et al. [1994] showed that document weights could be stored in a few bits each; the accuracy of, for instance, 32 bits is unnecessary given the existing approximations inherent in ranking. As the array of document weights needs to be randomly accessed, this can greatly reduce the memory required for a search engine.

Turtle and Flood [1995] argue that document-ordered query processing in which lists are merged rather than processed in turn in term-ordered query processing, may be more efficient in practical multiuser environments. Kaszkiel et al. [1999] compare term-ordered and document-ordered processing for whole documents and for passages and find that document-ordered processing can be effective if some query terms are rare. However, the distinction between the methods is unclear in the context of limited numbers of accumulators. Strohman et al. [2005] carry out further experiments with a document-ordered processing regime.

A space-saving approach is to trade off index-processing cost against document-retrieval cost. Signature files exemplify this technique. As the amount of information held about each document is reduced, the likelihood of fetching a document by mistake grows. In such an approach, it is necessary to postprocess documents to ensure that they do in fact match the query—a cost that in a large collection must easily dominate all other components of query processing. Other approaches of this kind were proposed by Manber and Wu [1994], Baeza-Yates and Navarro [2000], and Navarro et al. [2000] in which words are indexed by approximate location, such as a group of documents, which, after index processing, could then be exhaustively processed.

A radically different approach to query evaluation is described by de Kretser and Moffat [1999], Clarke and Cormack [2000], and Clarke et al. [2000]. In this approach, word positions are stored in inverted lists and documents are ranked according to the proximity of the query terms: a document that has a region that is dense with query terms is highly ranked. This approach can be very effective, particularly on long or poorly defined documents. Passage retrieval, in which the document collection is broken into fixed-size passages for the purpose of similarity checking, has also had some success [Kaszkiel et al. 1999; Zobel et al. 1995].

*Reducing Retrieval Costs.* Perhaps the single most effective heuristic for reducing query evaluation costs is to reorder inverted lists according to frequency or contribution. Until the mid-1990s, most of the query evaluation schemes were based either on the assumption that inverted lists were sorted by document identifier, or made no use of an ordering assumption at all. Wong and Lee [1993] describe a scheme in which lists are sorted by decreasing frequency. This initial description of the approach was marred by the assumption that inverted lists would consist of short blocks of document identifiers (leading to high retrieval costs) but allowed interleaved processing of inverted lists. It is the first description of this sorting principle which gives substantial savings.

Persin et al. [1996] proposed that inverted lists be sorted by in-document frequency. As in the method described by Wong and Lee [1993], query evaluation then processes the front part of each inverted list; how much is processed is determined by parameters reflecting system load and memory availability. A practical refinement to this approach was proposed by Brown [1995] who suggested that, for each list of sufficient length,



المنارة للاستشارات
www.manaraa.com

the list begin with a fixed-length block of document identifiers in which the term is frequent. An alternative is to store normalized frequencies in the inverted lists as described by Hawking [1998], Anh et al. [2001], and Anh and Moffat [2002]. To avoid loss of compression efficiency in this impact-ordering approach, the resulting absolute term weights are quantized. Large efficiency gains can be obtained through this method which is also suitable for evaluating queries within a fixed time bound.

Carmel et al. [2001] propose the static pruning of inverted lists, discarding at index construction time entries that are heuristically judged to be unlikely to affect ranking. While successful at reducing index size, the method is less effective and less efficient than the dynamic pruning methods. Garcia et al. [2004] describe another ordering variant where, within each inverted list, documents are ordered according to the number of times the document is highly ranked by a training query. The savings yielded are not as high as for impact-ordering and large numbers of training queries are required.

Caching reduces the cost of accessing disk. The impact of answer and list caches is explored by Saraiva et al. [2001] who use a large collection and a query log to show that both kinds of cache contribute to dramatically improved throughput. Lempel and Moran [2005] give theoretical bounds for the benefits of caching. Technical aspects of cache implementation are considered by Shieh et al. [2003] who show that, unsurprisingly, the costs of running a retrieval system can be significantly reduced by the use of caches; experiments with simulated queries are used to compare caching schemes, but the results (which depend on query distribution, a small collection, and specific hardware) seem unlikely to generalize. Jónsson et al. [1998] show that cache replacement policy has a significant effect on performance, while Lempel and Moran [2004] demonstrate efficiency gains (in the context of a distributed system) by prefetching and anticipating events such as users proceeding to the next page of answers. In this context, as illustrated by the work of Spink et al. [2001] and Spink and Xu [2000] on query logs, even though query evaluation can in principle focus on the rarest of the query terms provided, terms in typical queries occur in up to 1% of the indexed documents. Thus the inverted lists of query terms can be expected to be at least a megabyte in a large text collection and caching is likely to lead to significant savings. Lempel and Moran [2003] consider answer caching strategies in practice, using a trace of seven million real-life queries from a search engine log.

*Web Searching.* Brin and Page [1998] describe the PageRank mechanism, in effect a query-independent similarity measure, that led them to develop the first version of the Google search engine. Other aspects of the Google search engine are presented by Barroso et al. [2003]. The HITS algorithm is due to Kleinberg [1999].

Other simpler metrics for assigning static weights to pages include counting the in-degree and the out-degree of pages and various techniques based on statistics of the page, for example, the ratio of normal text to other content. Many of the weighting factors used by commercial search engines are proprietary and the subject of considerable speculation on the part of both consumers of the services they provide and also the spammers who seek to manipulate those services.

*Other Approaches to Indexing.* There are many papers on exhaustive string search but only de Kretser and Moffat [2004] consider how to rank exhaustively. Suffix arrays can be used for a wide range of searching tasks, including the Boolean retrieval more usually undertaken using inverted files. Baeza-Yates and Ribeiro-Neto [1999] and Baeza-Yates et al. [2002] provide an introduction to the literature in the area. Note that suffix arrays provide exact-match searching; there is no concept of ranking or of similarity scoring. On the other hand, some of the flexibility of string searching can be achieved if the



للاستشارات المنارة
www.manaraa.com

vocabulary, rather than the underlying set of documents, is searched for possible query terms [Zobel et al. 1993a]

There is a wide literature on signature files, dating back to the 1970s. Many of the papers in the area are principally concerned with the organization of the files and do not consider (at least, not in significant detail) how they would be applied to the problem of Boolean or ranked text search. Only a few papers concern the problem of ranking using signature file indexes, in particular Croft and Savino [1988] and Lee et al. [1995]. The schemes described in these papers are not competitive with inverted file implementations.

Zobel et al. [1998] give a detailed comparison of inverted files and signature files. Using analysis and experiments with data sets of under a gigabyte, it was concluded that signature files were highly inefficient as a Boolean retrieval mechanism for text. Since that time, the size of typical collections has greatly increased; extrapolation of the cost components of those experiments suggests that, for today's collections, signature files are relatively even less efficient. When inverted files can find matches in 100GB in less than a tenth of a second, there is little reason to use a signature file that requires seconds to search a gigabyte. Prior to the investigation of Zobel et al. [1998], Couvreur et al. [1994] compared inverted files to signature files in the context of a parallel system and found signature files to be lacking.

A focus of signature file research has been methods for rearranging signatures to reduce costs. Most of these methods propose, in effect, division of the file to reduce the length of bit vectors by a fixed ratio. Under these schemes, typically, the cost of query resolution remains linear to collection size, and there is no amelioration of the problem of false matches. Examples include work by Bookstein and Klein [1990], Ciaccia et al. [1996], Kocberbera and Can [1997], Kent et al. [1990], and Zezula et al. [1991]. It is not clear that any of these schemes are of value in practice. For example, Kocberbera and Can [1997] note a problem with scaling and removed the longest documents in order to reduce false matches to predicted levels.

A problem in much of the signature file work is the assumption of the independent probability of a bit being set. See, for example, Faloutsos [1985a, 1985b] who analyzes false drop probabilities on this assumption for different signature approaches; see also Ciaccia and Zezula [1993] and Lee and Leng [1989]. Sacks-Davis et al. [1987], in contrast, explicitly relate the probability of a bit being set to word probabilities, while continuing to assume that documents contain a fixed number of words.

*Nothing Is Truly New.* We close our article with a salutary quote, drawn from an article also published in ACM Computing Surveys [Severance and Carlis 1977]:

> When compared to a sequentially organized database, all [of] the preceding [variant index] organizations have several disadvantages: the software required for retrieval is much more complex; file retrieval generally requires at least one secondary memory access for each record retrieved; the space occupied by pointers constitutes a storage overhead (typically, 10% to 70% of total space requirements); and finally, the management and maintenance of data for a large set of lists represent a second-order database design problem whose solution will critically affect the total system's performance. Nevertheless, when rapid retrieval of small subsets of records from a large database is an essential system requirement, one of these file structures must be employed. The inverted list structure is generally preferred.

Current research directions in inverted files often relate to evaluation methods for the innovative similarity metrics that continue to appear. However, the core principles have become well established. Efficient ranking requires compact inverted lists, limited accumulators, effective pruning strategies, and distribution. In combination, these techniques allow ranking to be applied to collections of any likely size or characteristics.



المنارة للاستشارات
www.manaraa.com

## ACKNOWLEDGMENTS

We thank the many collaborators, assistants, and students who have contributed to our work on indexing. Jamie Callan, Bruce Croft, Donna Harman, Mike Lesk, and Ellen Voorhees helped us identify some of the early work in the area.

## REFERENCES

ANH, V. N., DE KRETSER, O., AND MOFFAT, A. 2001. Vector-space ranking with effective early termination. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. New Orleans, LA. 35–42.

ANH, V. N. AND MOFFAT, A. 1998. Compressed inverted files with reduced decoding overheads. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Melbourne, Australia. 291–298.

ANH, V. N. AND MOFFAT, A. 2002. Impact transformation: Effective and efficient web retrieval. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Tampere, Finland. 3–10.

ANH, V. N. AND MOFFAT, A. 2005. Inverted index compression using word-aligned binary codes. *Kluwer International Journal of Information Retrieval 8*, 1, 151–166.

ARUSU, A., CHO, J., GARCIA-MOLINA, H., PAEPCKE, A., AND RAGHAVAN, S. 2001. Searching the Web. *ACM Trans. Internet Technol. 1*, 1, 2–43.

BADUE, C., BAEZA-YATES, R., RIBEIRO-NETO, B., AND ZIVIANI, N. 2001. Distributed query processing using partitioned inverted files. In *Proceedings of String Processing and Information Retrieval Symposium*, Laguna de San Rafael, Chile. G. Navarro, Ed. IEEE Computer Society, 10–20.

BAEZA-YATES, R., MOFFAT, A., AND NAVARRO, G. 2002. Searching large text collections. In *Handbook of Massive Data Sets*, J. Abello, P. Pardalos, and M. Resende, Eds. Kluwer Academic Publishers, Boston, MA. 195–244.

BAEZA-YATES, R. AND RIBEIRO-NETO, B. 1999. *Modern Information Retrieval*. ACM Press, New York, NY.

BAEZA-YATES, R. A. AND NAVARRO, G. 2000. Block addressing indices for approximate text retrieval. *J. Amer. Soc. Inform. Science 51*, 1, 69–82.

BARBARÁ, D., MEHROTRA, S., AND VALLABHANENI, P. 1996. The Gold text indexing engine. In *Proceedings of IEEE International Conference on Data Engineering*, New Orleans, LA. S. Y. W. Su, Ed. IEEE Computer Society Press, Los Alamitos, CA. 172–179.

BARROSO, L. A., DEAN, J., AND HÖLZLE, U. 2003. Web search for a planet: The Google cluster architecture. *IEEE Micro 23*, 2 (April), 22–28.

BAYER, R. AND MCCREIGHT, R. 1972. Organization and maintenance of large ordered indexes. *Acta Informatica 1*, 173–189.

BEAULIEU, M., BAEZA-YATES, R., MYAENG, S. H., AND JÄRVELIN, K., EDS. 2002. *Proceedings of the 25th Annual International ACM SIGIR Conf. on Research and Development in Information Retrieval*. Tampere, Finland, ACM Press.

BELL, T. C., CLEARY, J. G., AND WITTEN, I. H. 1990. *Text Compression*. Prentice-Hall, Englewood Cliffs, NJ.

BELL, T. C., MOFFAT, A., NEVILL-MANNING, C. G., WITTEN, I. H., AND ZOBEL, J. 1993. Data compression in full-text retrieval systems. *J. Amer. Soc. Inform. Science 44*, 9 (Oct.), 508–531.

BERTINO, E., OOI, B. C., SACKS-DAVIS, R., TAN, K.-L., ZOBEL, J., SHIDLOVSKY, B., AND CATANIA, B. 1997. *Indexing Techniques for Advanced Database Systems*. Kluwer Academic Publishers, Boston, MA.

BIRD, R. M., NEWSBAUM, J. B., AND TREFFTZS, J. L. 1978. Text file inversion: An evaluation. In *Proceedings of the 4th Workshop on Computer Architecture for Non-Numeric Processing*. Blue Mountain Lake, NY, ACM Press, 42–50.

BIRD, R. M., TU, J. C., AND WORTHY, R. M. 1977. Associative/parallel processors for searching very large textual data bases. In *Proceedings of the 3rd Non-Numeric Workshop*. Syracuse, NY, ACM Press, 8–16.

BLANDFORD, D. AND BLELLOCH, G. 2002. Index compression through document reordering. In *Proceedings of the IEEE Data Compression Conference*, Snowbird, UT, J. A. Storer and M. Cohn, Eds. IEEE Computer Society Press, Los Alamitos, CA. 342–351.

BOOKSTEIN, A. AND KLEIN, S. T. 1990. Using bitmaps for medium sized information retrieval systems. *Inform. Proces. Manag. 26*, 525–533.

BOOKSTEIN, A. AND KLEIN, S. T. 1991a. Compression of correlated bit-vectors. *Inform. Syst. 16*, 4, 387–400.



www.manaraa.com

BOOKSTEIN, A. AND KLEIN, S. T. 1991b. Flexible compression for bitmap sets. In *Proceedings of the IEEE Data Compression Conference*, Snowbird, UT, J. Storer and M. Cohn, Eds. IEEE Computer Society Press, Los Alamitos, CA. 402–410.

BOOKSTEIN, A. AND KLEIN, S. T. 1991c. Generative models for bitmap sets with compression applications. In *Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Chicago, IL. A. Bookstein, Y. Chiaramella, G. Salton, and V. V. Raghavan, Eds. ACM Press, 63–71.

BOOKSTEIN, A. AND KLEIN, S. T. 1992. Models of bitmap generation: A systematic approach to bitmap compression. *Inform. Proc. Manag. 28*, 6, 735–748.

BOOKSTEIN, A., KLEIN, S. T., AND RAITA, T. 2000. Simple Bayesian model for bitmap compression. *Kluwer Int. J. Inform. Retriev. 1*, 4, 315–328.

BRIN, S. AND PAGE, L. 1998. The anatomy of a large-scale hypertextual Web search engine. *Comput. Netw. ISDN Syst. 30*, 1–7, 107–117.

BRISABOA, N. R., FARIÑA, A., NAVARRO, G., AND ESTELLER, M. F. 2003. $(S, C)$-dense coding: An optimized compression code for natural language text databases. In *Proceedings of String Processing and Information Retrieval Symposium*, Manaus, Brazil, M. A. Nascimento, Ed. Springer, 122–136.

BROWN, E. W. 1995. Fast evaluation of structured queries for information retrieval. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seattle, WA, E. A. Fox, P. Ingwersen, and R. Fidel, Eds. ACM Press, 30–38.

BROWN, E. W., CALLAN, J. P., AND CROFT, W. B. 1994. Fast incremental indexing for full-text information retrieval. In *Proceedings of the International Conference on Very Large Databases*, Santiago, Chile, J. B. Bocca, M. Jarke, and C. Zaniolo, Eds. Morgan Kaufmann, 192–202.

BROWN, E. W., CALLAN, J. P., CROFT, W. B., ELIOT, J., AND MOSS, B. 1994. Supporting full-text information retrieval with a persistent object store. In *Proceedings of the International Conference on Advances in Database Technology (EDBT)*, Cambridge, UK, M. Jarke, J. A. B. Jr., and K. G. Jeffery, Eds. Springer, 365–378.

BUCKLEY, C. AND LEWIT, A. F. 1985. Optimisation of inverted vector searches. In *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Montreal, Canada, 97–110.

BURKOWSKI, F. J. 1990. Surrogate subsets: a free space management strategy for the index of a text retrieval system. In *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Brussels, Belgium, 211–226.

BÜTTCHER, S. AND CLARKE, C. L. A. 2005. Indexing time vs. query time trade-offs in dynamic information retrieval systems. In *Proceedings of the International Conference on Information and Knowledge Management*, Bremen, Germany, A. Chowdhury, N. Fuhr, M. Ronthaler, H.-J. Schek, and W. Teiken, Eds. ACM Press, 317–318.

CACHEDA, F., PLACHOURAS, V., AND OUNIS, I. 2004. Performance analysis of distributed architectures to index one terabyte of text. In *Proceedings of the European Conference on IR Research*, Sunderland, UK, S. McDonald and J. Tait, Eds. 395–408. Lecture Notes in Computer Science, Springer, vol. 2997.

CAHOON, B. AND MCKINLEY, K. S. 1996. Performance evaluation of a distributed architecture for information retrieval. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Zurich, Switzerland, 110–118.

CAHOON, B., MCKINLEY, K. S., AND LU, Z. 2000. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Trans. Inform. Syst. 18*, 1 (Jan.) 1–43.

CAN, F. 1994. On the efficiency of best-match cluster searches. *Inform. Proc. Manag. 30*, 3, 343–361.

CARDENAS, A. 1975. Analysis and performance of inverted data base structures. *Comm. ACM 18*, 5, 253–263.

CARMEL, D., COHEN, D., FAGIN, R., FARCHI, E., HERSCOVICI, M., MAAREK, Y. S., AND SOFFER, A. 2001. Static index pruning for information retrieval systems. In *Proceedings of the 24th Annual International Conference on Research and Development in Information Retrieval*. New Orleans, LA. 43–50.

CHOUEKA, Y., FRAENKEL, A., KLEIN, S., AND SEGAL, E. 1987. Improved techniques for processing queries in full-text systems. In *Proceedings of the 10th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New Orleans, LA, C. T. Yu and C. J. V. Rijsbergen, Eds. ACM Press, 306–315.

CHOUEKA, Y., FRAENKEL, A. S., AND KLEIN, S. T. 1988. Compression of concordances in full-text retrieval systems. In *Proceedings of the 11th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Grenoble, France, Y. Chiaramella, Ed. ACM Press, 597–612.

CHOUEKA, Y., FRAENKEL, A. S., KLEIN, S. T., AND SEGAL, E. 1986. Improved hierarchical bit-vector compression in document retrieval systems. In *Proceedings of the 9th Annual International*



المنارة للاستشارات
www.manaraa.com

*ACM SIGIR Conference on Research and Development in Information Retrieval*. Pisa, Italy, 88–97.

Ciaccia, P., Tiberio, P., and Zezula, P. 1996. Declustering of key-based partitioned signature files. *ACM Trans. Datab. Syst. 21*, 3 (Sept.), 295–338.

Ciaccia, P. and Zezula, P. 1993. Estimating accesses in partitioned signature file organizations. *ACM Trans. Inform. Syst. 11*, 2, 133–142.

Clarke, C. L. A. and Cormack, G. V. 1995. Dynamic inverted indexes for a distributed full-text retrieval system. Tech. rep. MT-95-01, Department of Computer Science, University of Waterloo, Waterloo, Canada.

Clarke, C. L. A. and Cormack, G. V. 2000. Shortest-substring retrieval and ranking. *ACM Trans. Inform. Syst. 18*, 1, 44–78.

Clarke, C. L. A., Cormack, G. V., and Burkowski, F. J. 1994. Fast inverted indexes with on-line update. Tech. rep. CS-94-40, Department of Computer Science, University of Waterloo, Waterloo, Canada.

Clarke, C. L. A., Cormack, G. V., and Tudhope, E. A. 2000. Relevance ranking for one to three term queries. *Inform. Proc. Manage. 36*, 2, 291–311.

Couvreur, T. R., Benzel, R. N., Miller, S. F., Zeitler, D. N., Lee, D. L., Singhal, M., Shivaratri, N., and Wong, W. Y. P. 1994. An analysis of performance and cost factors in searching large text databases using parallel search systems. *J. Amer. Soc. Inform. Science 45*, 7, 443–464.

Cringean, J. K., England, R., Manson, G. A., and Willett, P. 1990. Parallel text searching in serial files using a processor farm. In *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Brussels, Belgium, 429–453.

Croft, W. B., Harper, D. J., Kraft, D. H., and Zobel, J., Eds. 2001. *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New Orleans, LA. ACM Press.

Croft, W. B. and Lafferty, J. 2003. *Language Modeling for Information Retrieval*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Croft, W. B., Moffat, A., van Rijsbergen, C. J., Wilkinson, R., and Zobel, J., Eds. 1998. *Proceedings of the 21th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Melbourne, Australia. ACM Press.

Croft, W. B. and Savino, P. 1988. Implementing ranking strategies using text signatures. *ACM Trans. Office Inform. Syst. 6*, 1, 42–62.

Culpepper, J. S. and Moffat, A. 2005. Enhanced byte codes with restricted prefix properties. In *Proceedings of the 12th International Symposium on String Processing and Information Retrieval*. Buenos Aires, Argentina, M. P. Consens and G. Navarro, Eds. Lecture Notes in Computer Science, vol. 3772, Springer, 1–12.

Cutting, D. and Pedersen, J. 1990. Optimisations for dynamic inverted index maintenance. In *Proceedings of the ACM-SIGIR International Conference on Research and Development in Information Retrieval*. Brussels, Belgium, J.-L. Vidick, Ed. ACM Press, 405–411.

de Kretser, O. and Moffat, A. 1999. Effective document presentation with a locality-based similarity heuristic. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. San Francisco, CA. 113–120.

de Kretser, O. and Moffat, A. 2004. Seft: A search engine for text. *Softw.—Prac. Exper. 34*, 10 (Aug.), 1011–1023.

de Kretser, O., Moffat, A., Shimmin, T., and Zobel, J. 1998. Methodologies for distributed information retrieval. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*. Amsterdam, The Netherlands. M. P. Papazoglou, M. Takizawa, B. Krämer, and S. Chanson, Eds. IEEE Computer Society Press, Los Alamitos, CA. 66–73.

Eastman, C. 1983. Current practice in the evaluation of multikey search algorithms. In *Proceedings of the 6th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Washington DC. J. J. Kuehn, Ed. ACM Press. 197–204.

Edmundson, H. P. and Wyllys, R. E. 1961. Automatic abstracting and indexing—survey and recommendations. *Comm. ACM 4*, 5 (May), 226–234.

Elias, P. 1975. Universal codeword sets and representations of the integers. *IEEE Trans. Inform. Theory IT-21*, 2 (March), 194–203.

Faloutsos, C. 1985a. Access methods for text. *Comput. Surv. 17*, 1, 49–74.

Faloutsos, C. 1985b. Signature files: Design and performance comparison of some signature extraction methods. In *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Montreal, Canada, 63–82.



المنارة للاستشارات
www.manaraa.com

FALOUTSOS, C. AND JAGADISH, H. V. 1992. Hybrid index organizations for text databases. In *Proceedings of the International Conference on Extending Database Technology*, Vienna, Austria, A. Pirotte, C. Delobel, and G. Gottlob, Eds. Lecture Notes in Computer Science, vol. 580, Springer, 310–327.

FALOUTSOS, C. AND OARD, D. W. 1995. A survey of information retrieval and filtering methods. Tech. rep., University of Maryland Institute for Advanced Computer Studies Report, University of Maryland at College Park, MD.

FOX, E. A. AND LEE, W. C. 1991. FAST-INV: A fast algorithm for building large inverted files. Tech. rep. TR 91–10, Virginia Polytechnic Institute and State University, Blacksburg, VA.

FRAENKEL, A. S. AND KLEIN, S. T. 1985. Novel compression of sparse bit-strings—Preliminary report. In *Combinatorial Algorithms on Words, Volume 12*, A. Apostolico and Z. Galil, Eds. NATO ASI Series F. Springer, Berlin, Germany, 169–183.

FRAKES, W. B. AND BAEZA-YATES, R., EDS. 1992. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Englewood Cliffs, NJ.

FREI, H.-P., HARMAN, D., SCHÄUBLE, P., AND WILKINSON, R., EDS. 1996. *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Zurich, Switzerland. ACM Press.

GALLAGER, R. G. AND VAN VOORHIS, D. C. 1975. Optimal source codes for geometrically distributed integer alphabets. *IEEE Trans. Inform. Theory IT–21*, 2 (March), 228–230.

GARCIA, S., WILLIAMS, H. E., AND CANNANE, A. 2004. Access-ordered indexes. In *Proceedings of the Australasian Computer Science Conference*, Dunedin, New Zealand. V. Estivill-Castro, Ed. Australian Computer Society, 7–14.

GOLOMB, S. W. 1966. Run-length encodings. *IEEE Trans. Inform. Theory IT–12*, 3 (July), 399–401.

GRABS, T., BÖHM, K., AND SCHEK, H.-J. 2001. PowerDB-IR: Information retrieval on top of a database cluster. In *Proceedings of the CIKM International Conference on Information and Knowledge Management*, Atlanta, GA. H. Paques, L. Liu, and D. Grossman, Eds. ACM Press, 411–418.

GROSSMAN, D. A. AND FRIEDER, O. 2004. *Information Retrieval: Algorithms and Heuristics*, 2nd Ed. Kluwer Academic Publishers, Dordrecht, The Netherlands.

HARMAN, D., MCCOY, W., TOENSE, R., AND CANDELA, G. 1991. Prototyping a distributed information retrieval system using statistical ranking. *Inform. Proc. Manag. 27*, 5, 449–460.

HARMAN, D. K. 1992. Ranking algorithms. In *Information Retrieval: Data Structures and Algorithms*. W. B. Frakes and R. Baeza-Yates, Eds. Prentice Hall, Englewood Cliffs, NJ. 362–392.

HARMAN, D. K. AND CANDELA, G. 1990. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *J. Amer. Soc. Inform. Science 41*, 8 (Aug.), 581–589.

HARPER, D. J. 1982. An evaluation of four information storage and retrieval packages. Tech. rep. 7, CSIRO Division of Computing Research, Canberra, Australia.

HASKIN, R. L. 1980. Hardware for searching very large text databases. In *Proceedings of the Workshop on Computer Architecture for Non-Numeric Processing*, Pacific Grove, CA, ACM Press, 49–56.

HAWKING, D. 1997. Scalable text retrieval for large digital libraries. In *Proceedings of the European Conference on Research and Advanced Technology for Digital Libraries*, Pisa, Italy. C. Thanos, Ed. Lecture Notes in Computer Science, vol. 1324. Springer, 127–145.

HAWKING, D. 1998. Efficiency/effectiveness trade-offs in query processing. *ACM SIGIR Forum 32*, 2, 16–22.

HEAPS, H. S. 1978. *Information Retrieval, Computational and Theoretical Aspects*. Academic Press, New York, NY.

HEARST, M., GEY, F., AND TONG, R., EDS. 1999. *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, San Francisco. CA, ACM Press.

HEINZ, S. AND ZOBEL, J. 2003. Efficient single-pass index construction for text databases. *J. Amer. Soc. Inform. Science Techn. 54*, 8, 713–729.

IVIE, E. L. 1966. Search procedure based on measures of relatedness between documents. Ph.D. thesis. MIT, Cambridge, MA.

JAKOBSSON, M. 1978. Huffman coding in bit-vector compression. *Inform. Pro. Let. 7*, 6 (Oct.) 304–307.

JEONG, B. S. AND OMIECINSKI, E. 1995. Inverted file partitioning schemes in multiple disk systems. *IEEE Tran. Parall. Distrib. Syst. 6*, 2, 142–153.

JÓNSSON, B. T., FRANKLIN, M. J., AND SRIVASTAVA, D. 1998. Interaction of query evaluation and buffer management for information retrieval. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, Seattle, WA. ACM Press, 118–129.

KASZKIEL, M., ZOBEL, J., AND SACKS-DAVIS, R. 1999. Efficient passage ranking for document databases. *ACM Trans. Inform. Syst. 17*, 4 (Oct.), 406–439.



المنارة للاستشارات
www.manaraa.com

KENT, A. J., SACKS-DAVIS, R., AND RAMAMOHANARAO, K. 1990. A signature file scheme based on multiple organisations for indexing very large text databases. *J. Amer. Soc. Inform. Science 41*, 7, 508–534.

KLEIN, S. T., BOOKSTEIN, A., AND DEERWESTER, S. 1989. Storing text retrieval systems on CD-ROM: Compression and encryption considerations. *ACM Trans. Office Inform. Syst. 7*, 3, 230–245.

KLEINBERG, J. M. 1999. Authoritative sources in a hyper-linked environment. *J. ACM 46*, 5, 604–632.

KOBAYASHI, M. AND TAKEDA, K. 2000. Information retrieval on the web. *Comput. Surv. 32*, 2, 144–173.

KOCBERBERA, S. AND CAN, F. 1997. Vertical framing of superimposed signature files using partial evaluation of queries. *Inform. Proc. Manag. 33*, 3, 353–376.

LANCASTER, F. W. AND FAYEN, E. G. 1973. *Information Retrieval OnLine*. Melville, Los Angeles, CA.

LEE, D. L., KIM, Y. M., AND PATEL, G. 1995. Efficient signature file methods for text retrieval. *IEEE Tran. Knowl. Data Eng. 7*, 3 (June) 423–435.

LEE, D. L. AND LENG, C.-W. 1989. Partitioned signature files: Design issues and performance evaluation. *ACM Trans. Inform. Syst. 7*, 2, 158–180.

LEE, Y. K., YOO, S. J., YOON, K., AND BERRA, P. B. 1996. Index structures for structured documents. In *Proceedings of the ACM Digital Libraries*, Bethesda, MD, E. A. Fox and G. Marchionini, Eds. ACM Press, 91–99.

LEMPEL, R. AND MORAN, S. 2003. Predictive caching and prefetching of query results in search engines. In *Proceedings of the World Wide Web Conference*. Budapest, Hungary. ACM Press, 19–28.

LEMPEL, R. AND MORAN, S. 2004. Optimal result prefetching in web search engines with segmented indices. *ACM Trans. Internet Techn. 4*, 1, 31–59.

LEMPEL, R. AND MORAN, S. 2005. Competitive caching of query results in search engines. *Theoret. Comput. Science 324*, 253–271.

LESTER, N., MOFFAT, A., WEBBER, W., AND ZOBEL, J. 2005. Space-limited ranked query evaluation using adaptive pruning. In *Proceedings of the 6th International Conference on Web Informations Systems*. Lecture Notes in Computer Science, vol. 3806, Springer, 470–477.

LESTER, N., MOFFAT, A., AND ZOBEL, J. 2005. Fast on-line index construction by geometric partitioning. In *Proceedings of the CIKM International Conference on Information and Knowledge Management*, Bremen, Germany, A. Chowdhury, N. Fuhr, M. Ronthaler, H.-J. Schek, and W. Teiken, Eds. ACM Press, 776–783.

LESTER, N., ZOBEL, J., AND WILLIAMS, H. E. 2006. Efficient online index maintenance for text retrieval systems. *Inform. Proce. Manag.* To appear.

LIM, L., WANG, M., PADMANABHAN, S., VITTER, J. S., AND AGARWAL, R. 2003. Dynamic maintenance of web indexes using landmarks. In *Proceedings of the World-Wide Web Conference*, Budapest, Hungary. Y.-F. R. Chen, L. Kovács, and S. Lawrence, Eds. ACM Press, 102–111.

LINOFF, G. AND STANFILL, C. 1993. Compression of indexes with full positional information in very large text databases. In *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Pittsburg, PA. R. Korfhage, E. Rasmussen, and P. Willett, Eds. ACM Press, 88–97.

LU, Z. AND MCKINLEY, K. S. 2003. Partial collection replication for information retrieval. *Kluwer Int. J. Inform. Retrie. 6*, 2, 159–198.

LUCARELLA, D. 1988. A document retrieval system based upon nearest neighbour searching. *J. Inform. Science 14*, 25–33.

LUHN, H. P. 1957. A statistical approach to mechanised encoding and searching of library information. *IBM J. Resear. Develop. 1*, 309–317.

MACFARLANE, A., MCCANN, J. A., AND ROBERTSON, S. E. 2000. Parallel search using partitioned inverted files. In *Proceedings of the String Processing and Information Retrieval Symposium*. A Coruña, Spain. P. de la Fuente, Ed. IEEE Computer Society Press, Los Alamitos, CA. 209–220.

MACLEOD, I. A., MARTIN, T. P., NORDIN, B., AND PHILLIPS, J. R. 1987. Strategies for building distributed information retrieval systems. *Inform. Proc. Manag. 23*, 6, 511–528.

MANBER, U. AND WU, S. 1994. GLIMPSE: a tool to search through entire file systems. In *USENIX Winter Technical Conference*. San Francisco CA. USENIX Association, Berkeley, CA. 23–32.

MANNING, C. D. AND SCHÜTZE, H. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press Cambridge, MA.

MARON, M. E. AND KUHNS, J. L. 1960. On relevance, probabilistic indexing and information retrieval. *J. ACM 7*, 3, 216–244.

MARTIN, T. P., MACLEOD, I. A., RUSSELL, J. I., LEESE, K., AND FOSTER, B. 1990. A case study of caching strategies for a distributed full text retrieval system. *Inform. Proc. Manag. 26*, 2, 227–247.



المنارة للاستشارات
www.manaraa.com

MARTIN, T. P. AND RUSSELL, J. I. 1991. Data caching strategies for distributed full text retrieval systems. *Inform. Syst. 16*, 1, 1–11.

MATTHEW, F. W. AND THOMSON, L. 1967. Weighted term search: a computer program for an inverted coordinate search on magnetic tape. *J. Chem. Document. 7*, 1, 49–56.

MCDONELL, K. J. 1977. An inverted index implementation. *Comput. J. 20*, 1, 116–123.

MELNIK, S., RAGHAVAN, S., YANG, B., AND GARCIA-MOLINA, H. 2001. Building a distributed full-text index for the web. *ACM Trans. Inform. Syst. 19*, 3, 217–241.

MOFFAT, A. 1992. Economical inversion of large text files. *Comput. Syst. 5*, 2, 125–139.

MOFFAT, A. AND BELL, T. A. H. 1995. In-situ generation of compressed inverted files. *J. Ame. Soci. Inform. Science 46*, 7 (Aug.) 537–550.

MOFFAT, A. AND STUIVER, L. 2000. Binary interpolative coding for effective index compression. *Kluwer Int. J. Inform. Retriev. 3*, 1 (July) 25–47.

MOFFAT, A. AND TURPIN, A. 2002. *Compression and Coding Algorithms*. Kluwer Academic Publishers, Boston, MA.

MOFFAT, A., WEBBER, W., ZOBEL, J., AND BAEZA-YATES, R. 2005. A pipelined architecture for distributed text query evaluation. Submitted for publication.

MOFFAT, A. AND ZOBEL, J. 1992a. Coding for compression in full-text retrieval systems. In *Proceedings of the IEEE Data Compression Conference*, Snowbird, UT, J. A. Storer and M. Cohn, Eds. IEEE Computer Society Press, Los Alamitos, CA, 72–81.

MOFFAT, A. AND ZOBEL, J. 1992b. Parameterised compression for sparse bitmaps. In *Proceedings of the 5th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Copenhagen, Denmaaank, N. J. Belkin, P. Ingwersen, and A. M. Pejtersen, Eds. ACM Press, 274–285.

MOFFAT, A. AND ZOBEL, J. 1996. Self-indexing inverted files for fast text retrieval. *ACM Trans. Informa. Syst. 14*, 4 (Oct.) 349–379.

MOFFAT, A. AND ZOBEL, J. 2004. What does it mean to "measure performance"? In *Proceedings of the 5th International Conference on Web Informations Systems*, Brisbane, Australia. X. Zhou, S. Su, M. P. Papazoglou, M. E. Owlowska, and K. Jeffrey, Eds. Lecture Notes in Computer Science, vol. 3306. Springer, 1–12.

MOFFAT, A., ZOBEL, J., AND SACKS-DAVIS, R. 1994. Memory efficient ranking. *Inform. Proc. Manag. 30*, 6 (Nov.) 733–744.

MOTZKIN, D. 1994. On high performance of updates within an efficient document retrieval system. *Inform. Proc. Manag. 30*, 1, 93–118.

NAVARRO, G., DE MOURA, E., NEUBERT, M., ZIVIANI, N., AND BAEZA-YATES, R. 2000. Adding compression to block addressing inverted indexes. *Kluwer Int. J. Inform. Retriev. 3*, 1, 49–77.

NOREAULT, T., KOLL, M., AND MCGILL, M. J. 1977. Automatic ranked output from Boolean searches in SIRE. *J. Amer, Soc. Inform. Science 28*, 333–339.

PERRY, S. A. AND WILLETT, P. 1983. A review of the use of inverted files for best match searching in information retrieval systems. *J. Inform. Science 6*, 59–66.

PERSIN, M., ZOBEL, J., AND SACKS-DAVIS, R. 1996. Filtered document retrieval with frequency-sorted indexes. *J. Amer. Soc. Inform. Science 47*, 10, 749–764.

PONTE, J. M. AND CROFT, W. B. 1998. A language modelling approach to information retrieval. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Melbourne, Australia, 275–281.

RABITTI, F., Ed. 1986. In *Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Pisa, Italy, ACM Press.

REDDAWAY, S. F. 1991. High speed text retrieval from large databases on a massively parallel processor. *Inform. Proc. Manag. 27*, 4, 311–316.

RIBEIRO-NETO, B. AND BARBOSA, R. 1998. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the ACM Digital Libraries*, Pittsburgh, PA, I. Witten, R. Akscyn, and F. M. S. III, Eds. ACM Press, 182–190.

RIBERTO-NETO, B., DE MOURA, E. S., NEUBERT, M. S., AND ZIVIANI, N. 1999. Efficient distributed algorithms to build inverted files. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. San Francisco, CA, 105–112.

RICE, R. F. 1979. Some practical universal noiseless coding techniques. Tech. Rep. 79–22, Jet Propulsion Laboratory, Pasadena, CA.

ROBERTSON, S. E. 1977. The probability ranking principle in IR. *J. Document. 33*, 4 (Dec.) 294–304.

ROBERTSON, S. E., WALKER, S., JONES, S., HANCOCK-BEAULIEU, M. M., AND GATFORD, M. 1994. Okapi at TREC-3.



المنارة للاستشارات
www.manaraa.com

In *Overview of the 3rd TREC Text REtrieval Conference*, Gaithersburg, MD, D. Harman, Ed. NIST, NIST Special Publication 500-226.

ROGERS, W., CANDELA, G., AND HARMAN, D. 1995. Space and time improvements for indexing in information retrieval. In *Proceedings of the Annual Symposium on Document Analysis and Information Retrieval*, Las Vegas, NV, L. Spitz and D. D. Lewis, Eds.

SACKS-DAVIS, R., KENT, A. J., AND RAMAMOHANARAO, K. 1987. Multi-key access methods based on superimposed coding techniques. *ACM Trans. Datab. Syst. 12*, 4, 655–696.

SALOMON, D. 2000. *Data Compression: The Complete Reference*, 2nd Ed. Springer, Berlin, Germany.

SALTON, G. 1962. The use of citations as an aid to automatic content analysis. Tech. Rep. ISR-2, Section III, Harvard Computation Laboratory, Cambridge, MA.

SALTON, G. 1968. *Automatic Index Organization and Retrieval*. McGraw-Hill, New York, NY.

SALTON, G., Ed. 1971. *The SMART Retrieval System: Experiments in Automatic Document Processing*. Prentice-Hall, Englewood Cliffs, NJ.

SALTON, G. 1972. Dynamic document processing. *Comm. ACM 15*, 7, 658–668.

SALTON, G. 1989. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, Reading, MA.

SALTON, G. AND BUCKLEY, C. 1988a. Parallel text search methods. *Comm. ACM 31*, 2 (Feb.) 202–215.

SALTON, G. AND BUCKLEY, C. 1988b. Term-weighting approaches in automatic text retrieval. *Inform. Proc. Manag. 24*, 5, 513–523.

SALTON, G. AND MCGILL, M. J. 1983. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, NY.

SALTON, G., WONG, A., AND WANG, C. S. 1975. A vector space model for automatic indexing. *Comm. ACM 18*, 11 (Nov.) 613–620.

SARAIVA, P. C., DE MOURA, E. S., ZIVIANI, N., FONSECA, R., MEIRA, W., MURTA, C., AND RIBEIRO-NETO, B. 2001. Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. New Orleans, LA. 51–58.

SAYOOD, K. 2000. *Introduction to Data Compression* 2nd Ed. Morgan Kaufmann, San Francisco, CA.

SCHOLER, F., WILLIAMS, H. E., YIANNIS, J., AND ZOBEL, J. 2002. Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Tampere, Finland. 222–229.

SCHUEGRAF, E. J. 1976. Compression of large inverted files with hyperbolic term distribution. *Inform. Proc. Manag. 12*, 377–384.

SEGESTA, J. AND REID-GREEN, K. 2002. Harley Tillitt and computerized library searching. *IEEE Ann. History Comput. 24*, 3 (Sept.) 23–34.

SEVERANCE, D. G. AND CARLIS, J. V. 1977. A practical approach to selecting record access paths. *Comput. Surv. 9*, 4, 259–272.

SHIEH, W.-Y., CHEN, T.-F., AND CHUNG, C.-P. 2003. A tree-based inverted file for fast ranked-document retrieval. In *Proceedings of the International Conference on Information and Knowledge Engineering*. Las Vegas, NV. H. R. Arabnia, Ed. CSREA Press, 64–69.

SHIEH, W.-Y., CHEN, T.-F., SHANN, J. J.-J., AND CHUNG, C.-P. 2003. Inverted file compression through document identifier reassignment. *Inform. Proc. Manag. 39*, 1, 117–131.

SHIEH, W.-Y. AND CHUNG, C.-P. 2005. A statistics-based approach to incrementally update inverted files. *Inform. Proc. Manag. 41*, 2, 275–288.

SHIEH, W.-Y., SHANN, J. J.-J., AND CHUNG, C.-P. 2003. An inverted file cache for fast information retrieval. *J. Inform. Science Eng. 19*, 4, 681–695.

SHOENS, K., TOMASIC, A., AND GARCÍA-MOLINA, H. 1994. Synthetic workload performance analysis of incremental updates. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Dublin, Ireland, W. B. Croft and C. J. van Rijsbergen, Eds. ACM Press, 329–338.

SILVESTRI, F., ORLANDO, S., AND PEREGO, R. 2004. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Sheffield, England, M. Sanderson, K. Järvelin, J. Allan, and P. Bruza, Eds. ACM Press, 305–312.

SINGHAL, A., BUCKLEY, C., AND MITRA, M. 1996. Pivoted document length normalization. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Zurich, Switzerland, 21–29.



المنارة للاستشارات
www.manaraa.com

SMEATON, A. AND VAN RIJSBERGEN, C. J. 1981. The nearest neighbour problem in information retrieval. In *Proceedings of the 4th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Oakland, CA, C. J. Crouch, Ed. ACM Press, 83–87.

SPARCK JONES, K., WALKER, S., AND ROBERTSON, S. E. 2000. A probabilistic model of information retrieval: development and comparative experiments. parts 1&2. *Inform. Proc. Manag. 36*, 6, 779–840.

SPARCK JONES, K. AND WILLETT, P., Eds. 1997. *Readings in Information Retrieval*. Academic Press/Morgan Kaufmann, San Francisco, CA.

SPINK, A., WOLFRAM, D., JANSEN, B. J., AND SARACEVIC, T. 2001. Searching the Web: The public and their queries. *J. Amer. Soci. Inform. Science 52*, 3, 226–234.

SPINK, A. AND XU, J. L. 2000. Selected results from a large study of web searching: The Excite study. *Inform. Resear.—Int. Electron. J. 6*, 1.

STANFILL, C. 1990. Partitioned posting files: a parallel inverted file structure for information retrieval. In *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Brussels, Belgium. 413–428.

STANFILL, C., THAU, R., AND WALTZ, D. 1989. A parallel indexed algorithm for information retrieval. In *Proceedings of the 12th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Cambridge, MA, , N. J. Belkin and C. J. van Rijsbergen, Eds. ACM Press, 88–97.

STELLHORN, W. H. 1977. An inverted file processor for information retrieval. *IEEE Trans. Comput. 26*, 12, 1258–1267.

STROHMAN, T., TURTLE, H., AND CROFT, W. B. 2005. Optimization strategies for complex queries. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Salvador, Brazil, G. Marchionini, A. Moffat, J. Tate, R. Baeza-Yates, and N. Ziviani, Eds. ACM Press, 219–225.

TAGUE, J., Ed. 1985. *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Montreal, Canada, ACM Press.

TEUHOLA, J. 1978. A compression method for clustered bit-vectors. *Inform. Proc. Lett. 7*, 6 (Oct.) 308–311.

TOMASIC, A. AND GARCÍA-MOLINA, H. 1993. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proceedings of the International Conference on Parallel and Distributed Information Systems.* San Diego, CA, M. J. Carey and P. Valduriez, Eds. IEEE Computer Society Press, 8–17.

TOMASIC, A. AND GARCÍA-MOLINA, H. 1996. Performance issues in distributed shared-nothing information-retrieval systems. *Inform. Proc. Manag. 32*, 6, 647–665.

TOMASIC, A., GARCÍA-MOLINA, H., AND SHOENS, K. 1994. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*. Minneapolis, MA, R. T. Snodgrass and M. Winslett, Eds. ACM Press, 289–300.

TROTMAN, A. 2003. Compressing inverted files. *Kluwer Int. J. Inform. Retriev. 6*, 5–19.

TURTLE, H. AND FLOOD, J. 1995. Query evaluation: strategies and optimizations. *Inform. Proc. Manag. 31*, 1 (Nov.), 831–850.

VAN RIJSBERGEN, C. J. 1979. *Informat. Retriev.*, 2nd Ed. Butterworths, London, UK.

VASANTHAKUMAR, S. R., CALLAN, J. P., AND CROFT, W. B. 1996. Integrating INQUERY with an RDBMS to support text retrieval. *Bull. Techn. Comm. Data Eng. 19*, 1, 24–33.

VIDICK, J. L., Ed. 1990. *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Brussels, Belgium. ACM Press,

VOORHEES, E. M. 1986. The efficiency of inverted index and cluster searches. In *Proceedings of the 9th Annual ACM SIGIR Conference on Research and Development in Information Retrieval*. Pisa, Italy. 164–174.

VOORHEES, E. M. AND HARMAN, D. K. 2005. *TREC: Experiment and evaluation in information retrieval*. MIT Press, Cambridge, MA.

WILLIAMS, H. E. AND ZOBEL, J. 1999. Compressing integers for fast file access. *Comput. J. 42*, 3, 193–201.

WILLIAMS, H. E., ZOBEL, J., AND ANDERSON, P. 1999. What's next? Index structures for efficient phrase querying. In *Proceedings of the Australasian Database Conference.* Auckland, New Zealand. M. Orlowska, Ed. Australian Computer Society, 141–152.

WILLIAMS, H. E., ZOBEL, J., AND BAHLE, D. 2004. Fast phrase querying with combined indexes. *ACM Trans. Inform. Syst. 22*, 4, 573–594.

WITTEN, I. H., BELL, T. C., AND NEVILL, C. G. 1991. Models for compression in full-text retrieval systems. In *Proceedings of the IEEE Data Compression Conference*. Snowbird, UT, J. A. Storer and J. H. Reif, Eds. IEEE Computer Society Press, Los Alamitos, CA. 23–32.



المنارة للاستشارات
www.manaraa.com

Witten, I. H., Moffat, A., and Bell, T. C. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd Ed. Morgan Kaufmann, San Francisco, CA.

Wong, W. Y. P. and Lee, D. K. 1993. Implementations of partial document ranking using inverted files. *Inform. Proc. Manag. 29*, 5 (Sept.), 647–669.

Xi, W., Sornil, O., and Fox, E. A. 2002a. Hybrid partition inverted files for large-scale digital libraries. In *Proceedings of the Digital Library: IT Opportunities and Challenges in the New Millennium*. Beijing Library Press, Beijing, China, 404–418.

Xi, W., Sornil, O., Luo, M., and Fox, E. A. 2002b. Hybrid partition inverted files: Experimental validation. In *Proceedings of the European Conference on Research and Advanced Technology for Digital Libraries*. Rome, Italy, M. Agosti and C. Thanos, Eds. Lecture Notes in Computer Science, vol. 2458, Springer, 422–413.

Zezula, P., Rabitti, F., and Tiberio, P. 1991. Dynamic partitioning of signature files. *ACM Tran. Inform. Syst. 9*, 4 (Oct.), 336–369.

Zobel, J. and Moffat, A. 1998. Exploring the similarity space. *SIGIR Forum 32*, 1, 18–34.

Zobel, J., Moffat, A., and Ramamohanarao, K. 1996. Guidelines for presentation and comparison of indexing techniques. *SIGMOD Record 25*, 3 (Oct.), 10–15.

Zobel, J., Moffat, A., and Ramamohanarao, K. 1998. Inverted files versus signature files for text indexing. *ACM Trans. Datab. Syst. 23*, 4 (Dec.), 453–490.

Zobel, J., Moffat, A., and Sacks-Davis, R. 1992. An efficient indexing technique for full-text database systems. In *Proc. VLDB Int. Conf. on Very Large Databases*, L.-Y. Yuan, Ed. Morgan Kaufmann, Vancouver, 352–362.

Zobel, J., Moffat, A., and Sacks-Davis, R. 1993a. Searching large lexicons for partially specified terms using compressed inverted files. In *Proceedings of the International Conference on Very Large Databases*. Dublin, Ireland, R. Agrawal, S. Baker, and D. Bell, Eds. Morgan Kaufmann, 290–301.

Zobel, J., Moffat, A., and Sacks-Davis, R. 1993b. Storage management for files of dynamic records. In *Proceedings of the Australasian Database Conference.* Brisbane, Australia, 26–38.

Zobel, J., Moffat, A., Wilkinson, R., and Sacks-Davis, R. 1995. Efficient retrieval of partial documents. *Inform. Proc. Manag. 31,* 3, 361–377.


Received September 2004; revised November 2005; accepted March 2006



المنارة للاستشارات

www.manaraa.com