Chapter 1

SEARCHING LARGE TEXT COLLECTIONS

Ricardo Baeza-Yates\(^1\)
Alistair Moffat\(^2\)
and Gonzalo Navarro\(^1\)

1 Dept. of Computer Science
Universidad de Chile
Blanco Encalada 2120,
Santiago, Chile
{rbaeza,gnarro}@dcc.uchile.cl

2 Dept. Computer Science and Software Engineering
The University of Melbourne
Victoria 3010, Australia
alistair@cs.mu.oz.au

Abstract In this chapter we present the main data structures and algorithms for searching large text collections. We emphasize inverted files, the most used index, but also review suffix arrays, which are useful in a number of specialized applications. We also cover parallel and distributed implementations of these two structures. As an example, we show how mechanisms based upon inverted files can be used to index and search the Web.

1. INTRODUCTION

The amount of textual information available in electronic form is growing at a staggering rate. The best example of this growth is the World-Wide Web, which is estimated to provide access to at least three terabytes of text (that is, three million megabytes). Even in commercial and private hands text collection sizes which were unimaginable a few years ago are common now, and the challenge is to efficiently search those text collections to find the documents or pages that are of interest to the user. Given that brute force scanning of a large document collection is not a viable option except for occasional one-off queries, some kind of index structure is necessary. Indeed, there is nothing novel at all in the notion of an index, and most of us make regular use of them in libraries, books, and telephone directories. In Sections 2. and 3. we describe inverted files and suffix arrays respectively, which are the two standard mechanisms for providing such indexes.

After a query is resolved, we still have the problem of determining which documents are more relevant for the user. That problem depends on the information retrieval (IR) model chosen by the system. Although an introduction to one of the main IR models is given in Section 2., that problem and other problems related to retrieval evaluation, query languages and operations, user interfaces and feedback, and so on, are beyond the scope of this chapter. We refer the reader to (10, 57, 90)
for a full exposition of IR and further algorithmic details, as well as related topics such as digital and conventional library systems.

The number of users searching the Web, online bibliographical systems, or digital libraries is also increasing, which adds extra pressure to the efficiency requirements of information servers. In addition, increasingly sophisticated search requirements imply more costly searching algorithms, and even with clever implementation heuristics and fast hardware there is a limit to what can be achieved. One way to extend this limit is the use of parallel and distributed computers. Processors can be added to a system as the requirements increase, resulting in an almost unlimited capacity to cope with more users posing more complex queries on larger text collections. An overview of the issues relevant to parallel and distributed computations for information retrieval is given in Section 4., together with the solutions that are currently proposed.

Finally, using as a case study one of the most topical applications of text retrieval, Section 5, describes the techniques involved in the design of Web search engines, and discusses the particular challenges that arise in web searching.

2. INVERTED FILES

The standard structure used to provide content-based access to large document collections is the inverted file index (10, 32, 45, 90). This section describes the components of an inverted file index, shows how they can be economically stored, and then describes the querying process that allows the index to be used to satisfy a user’s information need.

2.1 BASIC CONCEPTS

An inverted file index consists of two principal components: a vocabulary of the index terms that appear in the collection, together with, for each term, some associated facts; and in a separate structure, for each term an inverted list that records the locations in the collection at which that term appears. For example, if the term cat appears in the first, third, and fourth documents of a collection, then the vocabulary for the collection contains a record for cat that, amongst other things, records the number of documents that contains it and the location on disk of its inverted list; and an inverted list is stored that, as a minimum, records the document numbers 1, 3, 4.

Table 1.1 gives a more complete example of a document collection and the vocabulary and inverted lists that result. This structure clearly provides rudimentary query support, since a query cat AND rat designed to retrieve the documents that contain both of the terms cat and rat can

<table>
<thead>
<tr>
<th>Document Collection</th>
<th>Vocabulary</th>
<th>Inverted lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doc. Text</td>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>1 fat rat eat cat</td>
<td>bat 3</td>
<td>(2, 5, 6)</td>
</tr>
<tr>
<td>2 fat rat eat bat</td>
<td>cat 3</td>
<td>(1, 3, 4)</td>
</tr>
<tr>
<td>3 mat cat eat rat</td>
<td>eat 6</td>
<td>(1, 2, 3, 5, 6, 8)</td>
</tr>
<tr>
<td>4 mat cat fat cat</td>
<td>fat 4</td>
<td>(1, 2, 4, 5)</td>
</tr>
<tr>
<td>5 fat rat eat fat mat bat</td>
<td>mat 4</td>
<td>(3, 4, 5, 7)</td>
</tr>
<tr>
<td>6 bat eat rat</td>
<td>rat 7</td>
<td>(1, 2, 3, 5, 6, 7, 8)</td>
</tr>
<tr>
<td>7 rat sat mat</td>
<td>sat 1</td>
<td>(7)</td>
</tr>
<tr>
<td>8 rat eat rat eat rat eat rat</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
be resolved by searching the vocabulary for the two terms, fetching their respective inverted lists \( \langle 1, 3, 4 \rangle \) and \( \langle 1, 2, 3, 5, 6, 7, 8 \rangle \), and then taking the intersection of the sets of document numbers to obtain \( \langle 1, 3 \rangle \). Disjunctive queries (using the operator or) and negations (using a not operator) can be similarly handled, taking respectively the union of inverted lists, and the complement. Queries formed using and, or, and not are known as Boolean queries, since each document in the collection either is or is not an answer to the query.

Although they are straightforward to process, Boolean queries are not the preferred querying mechanism for large document collections. Instead, ranked queries are used. In a ranked query a real-valued similarity score is calculated between the query and each of the documents in the collection, and the top \( r \) ranked documents are presented as the answers, with \( r \) being a parameter set by the user. Provided a suitable heuristic is used, ranked queries provide considerably more flexibility than do Boolean queries, since the formalism of a precise query language is avoided and it becomes much more natural for a user to simply type a list of terms that describe their information need without consideration for the operators that should group them. For example, the intent of the query: indexing, searching, querying, techniques, algorithms, methods, processes, large, big, gigabyte, megabyte, document, text, collection, database, repository, archive is abundantly clear, yet it probably would not be if parentheses and Boolean operators were to be inserted. The flexibility offered to users in selecting a value of \( r \) is also of considerable benefit – they can set \( r \) to be low, and view a short answer list that may have high precision but low recall, or they can increase \( r \) to obtain higher recall at possibly the expense of lower precision.

The desire to use ranked queries does, however, raise the issue of how to calculate the similarity score. One standard mechanism that has been in use in various forms for more than 30 years is the vector space model, of which the cosine rule is an example (81). If term \( t \) appears in document \( d \) a total of \( f_{d,t} \) times, and at least once in \( f_t \) of the \( N \) documents in the collection, then one instantiation of the cosine rule calculates the similarity \( S_{q,d} \) between query \( q \) and document \( d \) as:

\[
S_{q,d} = \frac{\left( \sum_{t \in q} w_{q,t} \cdot w_{q,t} \right)}{(W_d \cdot W_q)}
\]

\[
w_{q,t} = 1 + \log_r (N/f_t)
\]

\[
r_{q,t} = 1 + \log_r f_{q,t}
\]

\[
W_d = \left( \sum_{t} w_{d,t}^2 \right)^{1/2}
\]

\[
W_q = \left( \sum_{t} w_{q,t}^2 \right)^{1/2}
\]

The quantity \( S_{q,d} \) lies between zero and one (with one indicating maximal similarity) and corresponds to the cosine of the angle in \( n \)-dimensional space between a query vector described by \( \langle w_{q,t} \rangle \) and a document vector \( \langle w_{d,t} \rangle \), where \( n \) is the number of distinct terms in the collection. (Note that there are many alternative formulations that have been proposed (80, 93).)

Experimental studies (see, for example, the proceedings of the sequence of TREC Text Retrieval conferences (44)) have shown that formulations such as the one shown above yield relatively good retrieval effectiveness, and on queries ranging from 10 to 50 or more terms, and a collection of approximately 2 GB of English text, approximately half of the first \( r = 100 \) documents can be expected to be relevant to the query. (Of course, it must be noted that performance is highly variable, and for very specific queries with only a handful of answers in the collection, precision at \( r = 100 \) cannot be this high.)

Further gains in retrieval effectiveness (that is, changes that either boost precision, the fraction of retrieved documents that are relevant, or boost recall, the fraction of relevant documents that are retrieved) can be obtained by a number of additional heuristics. One, known as pivoting, biases the document weights \( W_d \) so as to obtain what are determined experimentally to be "fairer" normalization values (83). A second technique is thesaural expansion – searching for each query term in a thesaurus (and possibly thereby also disambiguating homonyms) and extending the query by including all of the possible alternatives for each term. A third technique that has yielded improved retrieval effectiveness is relevance feedback. In automatic feedback systems, the query
is evaluated initially with a small pool depth $r$, and one or more top ranked documents used as the seed for a second query that is taken deeper with a larger $r$ to actually generate answers. The rationale behind this approach is that the highest-scoring document is probably relevant, and inclusion of some or all of its terms in an extended query will hence probably be of benefit. Manual feedback systems allow the user to intervene and distinguish between documents that are relevant, and should be added to the query in a positive sense, and those that are irrelevant, and should be added to the query in a negative sense. After the user views a subset of highly ranked documents the query is reevaluated, using modified term weights $w_t$ that incorporate the positive and negative feedback.

A fourth technique that has been used to enhance effectiveness – with mixed results, it should be noted – is the inclusion of phrases as query terms. The discussion above implicitly assumed that both documents and queries were bags of words. In fact they are not, and word ordering can be critical. Consider, for example, the queries cat eat rat and rat eat cat applied to the example collection shown in Table 1.1 – one would reasonably expect that the two queries would return different documents in the top-ranked position in the answer list. This intuition has been confirmed by experiments, and regarding word pairs and triples as the index “terms” used to compose documents and queries has led to significant improvements in retrieval effectiveness in simple retrieval models, but only small gains when other sophisticated techniques have already been used (61).

Alternatives to the document-based retrieval model described are also possible. One alternative is a locality-based model, which scores word locations rather than documents themselves, summing contributions from any nearby query terms, with the contribution decreasing as the number of intervening words increases (28). This mechanism has the desirable side effect of allowing enhanced document presentation, as the document viewer can be initiated at the exact location that triggered the high similarity score, rather than at the beginning of the document. When documents may be many tens of kilobytes long, this is a distinct advantage.

Another approach to the same problems is passage retrieval, in which the document source is broken into (possibly overlapping) windows of text of roughly the same size, with indexing and querying performed on the basis of these passages (49).

## 2.2 INDEX REPRESENTATION

Having described the overall process of ranked information retrieval, we now turn to some of the technical detail. In this subsection we examine the inverted index, and show that a document-level index of the type illustrated in Table 1.1 requires a remarkably small overhead as a fraction of the indexed text, just $5\text{--}10\%$ for typical collections and typical documents. The next subsection then shows how these compact indexes are used to allow evaluation of ranked queries; and the third subsection considers the problem of actually constructing the inverted index for a large text.

In generating the example of Table 1.1 it was natural to arrange the document numbers in each inverted list into increasing order, and this same ordering can be used to provide economical storage, by recording the difference or $d$-gap between consecutive document numbers. For example, in Table 1.1 the inverted list for the term cat would be represented as the sequence of $d$-gaps $\langle 1, 2, 1 \rangle$. Then, because the sum of the $f_t$ different $d$-gaps that occur in a list of $f_t$ document identifiers cannot exceed $N$, the number of documents in the collection, it is possible to code the integer $d$-gaps in a manner that provides short codewords for the many small $d$-gaps, and long codewords for the relatively infrequent large $d$-gap values.

A number of suitable codes are described in detail in (90), here we briefly describe one of these – the Golomb code (33, 36). Golomb codes are particularly suited to integers distributed according to a geometric probability distribution, in which a gap of length $x$ occurs with probability $Pr(x) = (1 - p)^{x-1} p$ for some fixed probability $p$. This distribution may be assumed if, for example,
the $f_t$ documents that contain a given term $t$ are a random subset of the $N$ documents in the collection. The Golomb code for an integer $x \geq 1$ is then given by a unary code for the quotient $q = (x - 1) \text{ div } b$ (that is, $q - 1$ one-bits followed by a zero-bit) and a $\log_2 b$-bit binary code for the remainder $r = (x - 1) \text{ mod } b$. The simplest situation is when $b$ is a power of two, and in this case the codes are also known as Rice codes. Table 1.2 shows examples of codewords when $b = 4$ and $b = 8$. Note that the gaps in the codewords are shown only to allow the reader to discern the break between unary and binary parts, and are not present in the actual codes. When $b$ is not a power of two adjustments are made to the binary part so as to make some codewords one bit shorter and not waste any combinations.

Table 1.2 Example Rice/Golomb codewords.

<table>
<thead>
<tr>
<th>Gap $x$</th>
<th>Codeword, $b = 4$</th>
<th>Codeword, $b = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 00</td>
<td>0 000</td>
</tr>
<tr>
<td>2</td>
<td>0 01</td>
<td>0 001</td>
</tr>
<tr>
<td>3</td>
<td>0 10</td>
<td>0 010</td>
</tr>
<tr>
<td>4</td>
<td>0 11</td>
<td>0 011</td>
</tr>
<tr>
<td>10</td>
<td>110 01</td>
<td>10 001</td>
</tr>
<tr>
<td>20</td>
<td>11110 11</td>
<td>110 011</td>
</tr>
<tr>
<td>30</td>
<td>11111110 01</td>
<td>1110 101</td>
</tr>
<tr>
<td>40</td>
<td>1111111110 11</td>
<td>11110 111</td>
</tr>
</tbody>
</table>

As can be seen in Table 1.2, small values of $b$ give shorter codewords for small gaps $x$ than do large values of $b$, but achieve this at the expense of generating longer codewords for large values of $x$ than do large-$b$ codes. Indeed, $b$ should be chosen as a function of $p$, the probability governing the geometric distribution. A good first-order approximation is that

$$b = \frac{\ln 2}{p} \approx 0.69 \cdot \frac{N}{f_t}$$

where, as before, $N$ is the number of documents in the collection and $f_t$ is the number of them that contain (one or more occurrences of) the term $t$.

Using Golomb codes, and with the value of $b$ set locally for each inverted list (so that common terms with long inverted lists use small values of $b$), typical document-level indexes occupy about 6 bits per pointer (90). And, since each term in a document of a few kilobytes typically appears approximately twice on average, the number of pointers to be stored in the inverted file (given by $\sum_{i=1}^{n} f_i$ for a collection of $n$ distinct terms) is about half the number of words in the collection. Including white space, each word accounts for about 5–6 bytes in a source document, so one pointer is stored for every approximately 10–12 bytes of source text, and the cost of storing that pointer is around 6 bits.

To this cost must be added the bits required to store the $f_{dt}$ component — the within-document frequency value necessary for ranked queries. For example, the inverted list for cat in the collection of Table 1.1 must be extended to

$\{(1, 1), (2, 1), (1, 2)\}$,

with, for example, the last pair indicating that cat appears twice in the document whose number (d-gap) is one greater than the previous document containing it. The cost of adding the $f_{dt}$ values is small, and even using a unary code accounts for at most one bit per term appearance, or, using the same estimated values, 2 bits per pointer. Hence, for large collections consisting of documents each a few kilobytes long, one byte of index information is required for each 10–12 bytes of source
text, a very modest overhead. Use of more sophisticated compression techniques allows the cost to be reduced below 7 bits per pointer (90). It is also worth noting that these estimated costs are all-inclusive, and even frequent words such as the and it, and numbers such as 28 and 1999 are indexed. Indeed, by employing a suitable compression regime the cost of storing inverted lists for common words is dramatically reduced compared to naive storage methods, and little additional saving is achieved by using a stop list to prevent them from being indexed.

The final cost is storage for the vocabulary. For a large collection this is a minor overhead. For example, a 1 GB collection will contain approximately 200 million words, but almost certainly will contain fewer than one million distinct terms, particularly if case-folding and stemming are used to map document words onto their lexicographic roots. Even allowing 12-16 bytes per distinct term (to store the term itself, plus the corresponding \( f_d \) value, plus a pointer to its inverted list) the vocabulary contributes only a further approximately 1% of the source text.

While the discussion in this section has been focussed on representing inverted lists as a sequence of \( d \)-gaps, other organizations for the inverted lists are also possible. Persin et al. (71) describe a frequency-sorted index organization that retains the same information, but sorts the pairs in each list into decreasing order of within-document frequency \( f_{dt} \). The advantage of this approach is that if only the most “important” documents for each term are to be processed, and importance can be gauged by the magnitude of the \( f_{dt} \) value in conjunction with the term weight \( w_t \), then only a prefix of each inverted list need be processed.

Another alternative index representation is the blocked index described by Anh and Moffat (4). Rather than store \( d \)-gaps that are relative to the previous document number, this index representation stores gaps relative to document numbers at fixed intervals. It is then possible to identify the document number corresponding to a certain pointer without fully processing all of the previous codewords. Moreover, the codes that are processed are done so in a byte-by-byte sense rather than a bit-by-bit sense, and so considerable reductions in computation times can be achieved.

A further variant of inverted file indexing worthy of note is block addressing. Block addressing was first proposed in a system called Glimpse (59). The text is logically divided in blocks, and the index does not point to exact word positions, but only to the blocks where the word appears. Space is saved because there are fewer blocks than text positions (and hence the pointers are shorter), and also because all the occurrences of a given word in a single text block are referenced only once. In simplest form, each file in a collection of files is considered to be a block, and in large collections, multiple files may be placed into the same logical block.

Searching in a block addressing index is similar to searching in a full inverted one. The pattern is searched in the vocabulary and a list of blocks where the pattern appears is retrieved. However, to obtain the exact pattern positions in the text, a sequential search over the qualifying blocks becomes necessary, since the stored index is very coarse-grained. In a sense, the system makes use of exhaustive search, but with a filter-index that allows some or most of the blocks to be avoided. That is, the reduction in index space requirements is obtained at the expense of higher search costs.

Block addressing was analyzed by Baeza-Yates et al. (9), who showed both analytically and experimentally that a block addressing index can achieve both sublinear space overhead and sublinear query time, whereas inverted indexes pointing to words or documents achieve only the second goal.

### 2.3 QUERY PROCESSING

Given a document-level index augmented by within-document \( f_{dt} \) values, how then should a ranked query over a set of terms be evaluated? And how much more costly is it to evaluate a ranked query than to evaluate a Boolean query? These two issues are considered in this subsection.

As for Boolean queries, ranked queries are evaluated by processing inverted lists. The standard mechanism maintains an accumulator variable \( A_d \) for each document \( d \), in which a partial similarity
score between $d$ and the query $q$ is built up. As each inverted list is processed, a contribution is added to $A_d$ for each document $d$ that appears in that inverted list. This process is summarized in the following pseudo-code.

```
process_ranked_query(q, r) {
  for each document $d \in \{1, \ldots, N\}$ do
    set $A_d \leftarrow 0$
    for each term $t \in q$ do {
      calculate $w_t$ and $w_{q,t}$
      for each document $d \in$ the inverted list for term $t$ do {
        calculate $w_{d,t}$
        set $A_d \leftarrow A_d + w_{d,t} \cdot w_{q,t}$
      }
    }
  for each document $d \in \{1, \ldots, N\}$ do
    calculate and return the values of $d$ corresponding to
    the $r$ largest values of $A_d$
}
```

For small collections this mechanism is straightforward in both implementation and execution. It requires that the document length normalization values $W_d$ be precomputed and available in a file, but this is readily done at index creation time.

For large collections with millions of documents there are a number of potential problems with this approach. One obvious issue is that care needs to be taken with the final extraction of the $r$ largest accumulator values, and a partial sort (rather than a full sort of all $N$ values) should be performed. Suitable techniques are discussed by Witten et al. (90, Chapter 4).

Another drawback is the need for memory space to support the accumulators. For a collection of $N$ documents either an array $A$ of $N$ floating point values is required, or, if it can be assumed that most of the accumulators remain unused throughout the calculation, a dynamic structure must be manipulated. Unfortunately, in typical ranked queries of 20–30 words, a substantial fraction of the documents in the collection will contain at least one of the query terms, and result in non-zero accumulators. Moreover, the need for accumulators is on a per-query basis, and sharing between users is not possible. For example, if a system is being designed to allow (say) 100 simultaneous users access to (say) a collection of 10 million documents, use of an array data structure (with four bytes per accumulator) results in a memory consumption of $4 \times 10^7$ bytes $\times 100 = 4$ gigabytes, a rather large amount by any standards.

The final issue to be considered is not unrelated to the memory problem, and that is the cost of performing floating point arithmetic for every value in what are potentially very long inverted lists. If a user inadvertently uses the word ‘the’ in a query – and, given that we regard the query as simply being another piece of text, there is no reason why the user should penalized if they do use such words – a small similarity contribution might need to be calculated and added to the accumulator of every document in the collection.

To sidestep these latter two problems a number of pruning strategies have been devised (4, 20, 63, 71). Query terms are processed in decreasing order of weight $w_t$ (which is generally in increasing order of term frequency $f_t$), and heuristic rules are applied to determine whether or not a document should be allowed an accumulator variable, and whether or not any more of this inverted list, or any more inverted lists at all, should be processed.

The use of such strategies – coupled in some cases with frequency-sorted inverted lists (71) – has been shown experimentally to allow greatly accelerated query processing speeds, within
substantially less memory resources, and usually at negligible (sometimes zero) cost in terms of degraded retrieval effectiveness. Any large-scale system being designed for high throughput rates will need to make use of some kind of pruning mechanism, even if it is as simple as only processing a subset of the query terms. Indeed, the pruning thresholds can be set on a query-by-query basis, and users willing to pay extra for more resource consumption, or users making use of the service late at night at off-peak rates, might be given more generous thresholds than budget-minded and sleep-minded users.

Reorganizing the list representation to facilitate the evaluation of ranked queries does, of course, impact upon Boolean query evaluation. Frequency-sorted indexes lead to more complex Boolean query processing. On the other hand, the mechanisms used by Anh and Moffat (4) in their blocked inverted lists also accelerate Boolean query handling.

2.4 INDEX CONSTRUCTION

The implementer of a system based upon inverted files will also face a further vexing question: how to construct the index? Writing a cross-reference-generator program is a standard exercise in second-year data structures and algorithms subjects, but the obvious solutions break down when applied to large volumes of data – they use in-memory data structures of size comparable to or larger than the volume of text being processed, and access those data structures in a non-sequential manner.

Efficient methods have been developed that allow more economical index construction (62, 90). Unsurprisingly, they also rely on compression of the various components to reduce the volume of data that must be stored in main memory; and reorganize the operations required so as to make use of sequential disk-based processing. In this section we briefly sketch one mechanism that has been used to index (or invert) multi-gigabyte texts within a few tens of megabytes of main memory and only a small amount of extra disk space above and beyond the space required by the final compressed inverted index. This process is sketched in the pseudo-code that follows.

```pseudocode
create_inverted_index(T) {
  while text remains in T do {
    fill the in-memory buffer M with (d, t, fd) triples
    sort M by the t component of each triple
    write M to disk in compressed form as a run
  }
  read the first block of each of the runs into memory
  while at least one of the runs remains unemptied do {
    append the (d, t, fd) triple with the smallest t, d to the current output block,
    recompressing on the fly
    if the output block is full, write it to a vacant slot on disk
  }
  permute the blocks of the index to bring them into order
  release any free space at the end of the index
}
```

The basis of the method is a process that creates a file of (d, t, fd) triples in document number order (that is, by increasing d) and then reorders that file into term order (increasing t) using a combination of in-memory sorting and in-place multi-way external merging. The sorting phase is then followed by an in-place permutation of the fixed-length blocks that collectively comprise the compressed inverted file (62). We now examine these steps one by one.

First, the text is read in document order and parsed into terms. A bounded amount of memory is set aside as a buffer to collect (d, t, fd) triples, and every time that array is full it is sorted using
Quicksort, and written to disk in a compressed format. Each partial index — or run — is padded to be a multiple of a fixed block length so that all of the runs stored on disk are block-aligned.

Once all of the text has been processed in this way, the resultant runs are combined in a multiway merge. Just one block of each run is resident in memory at any given time, and so the memory requirement is modest. As the merge proceeds, output blocks are produced and written back to disk to any available slots within the file. No padding is required in the output blocks, meaning that the writing process tends to consume block slots at a slower rate than they are freed by the reading process, and a vacant slot is almost always available somewhere. The compression also tends to improve between input and output stages, as a result of more data for each term being available, further contributing to this effect. And in any case, if a vacant slot is not available at the time one is needed, that block can always be written at the end of the file.

Finally, once all of the runs have been exhausted, the index is complete, but with its blocks permuted throughout the file space allocated to the index rather than stored sequentially, and with unused block slots interspersed. An in-place permutation is then used to reorder the blocks. The permutation requires that each block of the index — typically a kilobyte or so — be read once and written once in a non-sequential order, and so is disk-intensive, but only for a relatively short time. At the conclusion of the block reordering any now-unused space at the end of the file is released.

This discussion has assumed that the collection is static. While this assumption is often valid — for example, when an information collection is to be distributed on a CD-ROM — dynamic collections must also be managed. In a dynamic collection the updates can be of two varieties: existing documents might be edited and perhaps deleted; and new documents might be added as the collection grows.

In the latter of these two cases, while it is tempting to develop a process that inserts single documents as they become available, the resource costs of doing so are very high. Consider, for example, the process of adding a document. As many as several hundred inverted lists must be fetched from disk, extended by a few bits or bytes, and then written back. Even discounting the complexity associated with managing a large set of bit-variable records, the sheer cost of reading and writing inverted lists means that the per-document disk operations required by this process might consume many seconds, severely limiting the rate at which documents can be added. Instead, updates should be batched into a small stop press collection that is searched exhaustively during querying. Periodically the stop press is indexed using the mechanisms described above, and then the index for the main collection and the index for the stop press combined using a sequential merge. Only by batching updates in this way can the per-document costs be kept modest.

The more complex update operations are edits and deletes on existing documents, since they involve contraction and expansion in the middle of inverted file entries. One pragmatic solution to this problem is to simply append the revised version of the document as if it were new, and retain the old version with a flag marking it as defunct. This approach automatically creates a record of all alterations to a document, which, with disk space being relatively cheap, may be an attractive additional benefit.

### 2.5 OTHER ISSUES

In the earlier parts of this section we have focussed on the index, and shown how it can be compressed and used to resolve queries. The text of the document collection should also, of course, be stored in compressed form if the collection is to be maintained economically. However, standard utilities such as gzip cannot be used, as they code each symbol of the file in a context established by the aggregate of all of the preceding symbols, meaning that it would not be possible to decode individual documents when they are required. Instead, semi-static compression mechanisms are used (90), in which a preliminary pass is made to accumulate symbol frequencies, and then a second pass made to actually compress the text with respect to those frequencies. During this second pass
the bit addresses at which the documents commence are noted in an index file, which is used to facilitate random-access decoding during query processing.

A variety of compression models can be used in this semi-static mode. One attractive option is word-based parsing, in which the text is broken into a sequence of words, and the words are regarded as being emitted from a zero-order Markov model (63, 90, 92). While the compression effectiveness attained with a zero-order word-based model is not as good as can be achieved with higher order character-based models, the word-based model has two distinct advantages: the loss in effectiveness that results from coupling the model with Huffman coding rather than arithmetic coding is small, and Huffman coding is considerably faster; and the memory space required during decoding operations can also be managed in a useful manner (64).

A further useful side-effect of the word-based model is that if the set of codewords is calculated using radix-256 or radix-128 codeletters, fast compressed-text searching is possible (65). Use of a radix-256 code and a spaceless words model allows compression effectiveness only slightly inferior to a radix-2 code, and, because the codewords are all byte-aligned, allows pattern matching on words and phrases by simply compressing them too, and using any standard pattern matching software. This mechanism does, however, allow false matches, since it is impossible to know whether any particular byte in the compressed text is the first byte of a codeword. This alignment problem is solved with the use of a radix-128 code, and storing each codeletter in an 8-bit byte. Setting the top bit in the first byte of each codeword then allows straightforward identification of word alignments, at a cost of using $8/7 = 14\%$ more space than previously.

For typical document collections storing ASCII English text, the word-based model compresses to at least as 25\% of the original size can be achieved (65, 90) (better than Ziv-Lempel techniques), with document access times essentially unaffected (92). That is, a complete indexed retrieval system can be constructed in 30-35\% of the space required by the original source text.

In closing this section on inverted file indexing it is also relevant to comment on one other type of indexing that enjoyed a period of support in the 1980s - signature files. In a signature file index a probabilistic hash-filter is used to support conjunctive Boolean queries, with operations performed on bit slices. As we have seen, however, ranked queries are the preferred retrieval mechanism, and signature files do not readily support them; and even if the only queries to be performed are conjunctive Boolean, it is not clear that signature files enjoy any space or speed advantage over compressed inverted files (94).

3. SUFFIX ARRAYS

Inverted indexes are appropriate for typical text retrieval queries on natural language texts. That is, they assume that the text is composed of words; that it follows some statistical properties that ensure, for example, that the vocabulary is not very large and that the most frequent words are rarely queried; that the queries aim at retrieving whole words or phrases; that if more complex patterns are searched for (say, regular expressions) then they are to be matched against whole words or at least subwords; and that the user is not interested in more complex queries. Under those restrictions, inverted indexes offer a data structure well suited to the problem to be solved, are cheap to build and space-economical, and allow ranking techniques to be used in order to identify the documents most relevant to a given query.

The suffix array is a data structure aimed at a different type of search. It does not require that the text be composed of natural language words, and instead handles any sequences of symbols. It is also able to perform more complex queries at the pattern matching level including simple words and phrases, regular expressions, and in the presence of errors. Suffix arrays can also find the longest repeated string in a text, and even compare the whole text against itself to find interesting auto-repetitions with and without errors.
The suffix array can be applied to classical information retrieval, but it is less adequate than inverted indexes: it is more expensive to build and maintain, powerful in aspects that are marginal in classical scenarios (for example, flexible pattern matching) and less powerful in crucial aspects (such as classical relevance ranking). The real interest for suffix arrays is in other applications, including genetic databases (7, 37, 58) (where the texts are DNA or protein sequences), intrusion detection (where the text is a sequence of events along the time), oriental languages (66) (where word delimiters are not so clear), agglutinating languages (where words are long and carry a more complex meaning, as in Finnish), and linguistic analysis of the text statistics (38) (to detect plagiarism, for instance).

We start by presenting the suffix array structure together with the strategy for searching simple patterns. Then we consider the problem of constructing suffix arrays. Finally, we discuss more complex queries. For simplicity we have used natural language examples, and it should be noted throughout that suffix arrays can be used on any sequence of symbols.

3.1 STRUCTURE

Consider the text example shown in Figure 1.1. We have selected a number of index points in the text. Index points are the text positions that the suffix array will be able to retrieve. In applications where there are no words it is common to consider that all symbols (in the example, characters) are index points, while if we consider only word beginnings, as in the example, we obtain a functionality similar to inverted indexes. Indeed, with word indexing we can choose to omit stop-words if we wish.

```
to be at the beach or to be at work, that is the real question $
```

```
1 4 7 10 14 29 38 43 46 50 55
```

<table>
<thead>
<tr>
<th>Index points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: to be at the beach or to be at work, that is the real question</td>
</tr>
<tr>
<td>4: be at the beach or to be at work, that is the real question</td>
</tr>
<tr>
<td>7: at the beach or to be at work, that is the real question</td>
</tr>
<tr>
<td>10: the beach or to be at work, that is the real question</td>
</tr>
<tr>
<td>......</td>
</tr>
<tr>
<td>55: question</td>
</tr>
</tbody>
</table>

*Figure 1.1* A text example, the selected index points and the resulting suffixes.

Each index point defines a text suffix, that is, a string which starts at the index point and continues until the end of the text. In the figure we have listed the suffixes that are defined by the index points. Notice that the end of the text is signaled by a special character “$”, which is smaller than all the other characters.

The suffix array is, in essence, an array recording the positions of all of the index points, ordered lexicographically by the text suffix they point to. Figure 1.2 shows the suffix array of our example.

The goal of this organization is to permit binary searching the query in the suffix array rather than sequentially searching it in the text. The key idea is that we look for text substrings that start at index points, and any such substring is a prefix of a text suffix.

Imagine that we are interested in finding the occurrences of the word “be” in the text. Since the index points are lexicographically sorted in the suffix array, all the text suffixes that start with “be” are placed in a continuous range of positions in the suffix array. Both extremes of this range of positions can be binary searched in the suffix array.

The extremes are obtained as follows. First, we search for the pattern “be$”. The binary search starts at position 8 in the array. The suffix pointed to is “question$” (index point 55), which
is lexicographically greater than our search pattern and therefore we consider array position 4. This time the suffix is “be at...” (index point 26), which is still greater than our pattern. We then consider array position 2, which points to “at work,...” (index point 29), which is lexicographically smaller than our pattern. The binary search continues until we determine that the first suffix larger than or equal to our pattern is at suffix array position 3. To look for the right extreme of the suffix array we perform a similar search for the smallest pattern that is larger than any suffix starting with “be...”. This pattern is “bf$”, whose search ends at suffix array position 6. Therefore all the occurrences of “be” in the text are in suffix array positions 3 to 5. The corresponding text positions are 4, 26 and 14.

Notice that searching for the occurrences of the pattern “be” is not the same as searching for the word “be”. For example, we have retrieved “beach” as well. Suffix arrays do not consider words, and therefore they find any occurrence of our string. Moreover, if we had indexed all text positions it would also have found the string “be” inside words. If we want to retrieve the word “be” we need to search for the pattern “be” (one space at the end), and filter the text to convert all separators (like comma) to spaces. This filtering can be done on the fly, without need to modify the text.

Note also that we need to access the text at search time to compare the query. Moreover, the accesses to the text are essentially at random. A first consequence is that the text must be available at search time, unlike most inverted index scenarios. A second, more serious consequence, is that in practical situations, in which it may impossible to fit either of the suffix array or the text in main memory, the random accesses will be to secondary memory, and so will be very expensive.

Supra-indexes have been proposed to alleviate this problem (6). A supra-index over a suffix array is obtained by a sampling process: the suffix array is logically divided in blocks of $b$ entries, and the first entry of each block is selected for the supra-index. For each selected entry, the first $\ell$ characters of the corresponding suffix are stored in the supra-index. The result is a coarser version of the suffix array which, by appropriate selection of $b$ and $\ell$, fits in main memory. Figure 1.3 shows a supra-index built for our example.

Notice that searching for the occurrences of the pattern “be” is not the same as searching for the word “be”. For example, we have retrieved “beach” as well. Suffix arrays do not consider words, and therefore they find any occurrence of our string. Moreover, if we had indexed all text positions it would also have found the string “be” inside words. If we want to retrieve the word “be” we need to search for the pattern “be” (one space at the end), and filter the text to convert all separators (like comma) to spaces. This filtering can be done on the fly, without need to modify the text.

Note also that we need to access the text at search time to compare the query. Moreover, the accesses to the text are essentially at random. A first consequence is that the text must be available at search time, unlike most inverted index scenarios. A second, more serious consequence, is that in practical situations, in which it may impossible to fit either of the suffix array or the text in main memory, the random accesses will be to secondary memory, and so will be very expensive.

Supra-indexes have been proposed to alleviate this problem (6). A supra-index over a suffix array is obtained by a sampling process: the suffix array is logically divided in blocks of $b$ entries, and the first entry of each block is selected for the supra-index. For each selected entry, the first $\ell$ characters of the corresponding suffix are stored in the supra-index. The result is a coarser version of the suffix array which, by appropriate selection of $b$ and $\ell$, fits in main memory. Figure 1.3 shows a supra-index built for our example.
The binary search now starts in the supra-index in main memory. At the end, two blocks of the suffix array are identified as holding the initial and final positions of the answer range (in fact, since the suffixes of the supra-index are limited at $\ell$ characters, two blocks may need to be considered for each extreme in the worst case). Only after these bounds have been identified is the disk accessed. The relevant blocks of the suffix array are brought into main memory and the binary searches are completed with the initial algorithm. In the example, the first block (1 to 3) is read to determine the left extreme, and the second block (4 to 6) for the right extreme. The search inside the blocks still needs to access the text at random disk positions.

Most of the accesses to the suffix array on disk have been eliminated, as well as some of the random accesses to the text on disk. In practice this reduces search time to 25% to 30% of its original cost. Note that we have assumed that the final block of the suffix array fits in main memory. In huge texts it is possible that the combined goal of having the supra-index of $n\ell/b$ characters and a suffix array block of $b$ entries in memory cannot be achieved for any $b$. In this case a supra-supra-index can be used, leading to a hierarchical scheme much like prefix B-trees (12).

The expensive part of the search is still the random accesses to the text in the final search in the suffix array block. In this case, deviating from standard binary search may be better. Binary search is the optimal strategy when accessing all the cells of the array has the same cost. In our case, each cell of the suffix array has a different access cost which is not fixed but depends on the current position of the disk head, that is, on the last element accessed in the suffix array block. For instance, we may prefer to access the suffix array at position $b/3$ instead of $b/2$ because the disk head is currently much closer to the text pointed by position $b/3$. The optimal search strategy can be precomputed at suffix array construction time and stored together with each suffix array block at little space penalty. Reductions of 60% over the original search times are reported (68).

If we are indexing natural language, an interesting choice is that the supra-index does not divide the suffix array at regular intervals and takes $\ell$ characters, but that it cuts the blocks where the first word of the suffix pointed changes and stores the word in the supra-index (11). Figure 1.4 illustrates this arrangement. Since the supra-index becomes similar to the text vocabulary, the result is much like a full inverted index, where the suffix array blocks correspond to the lists of occurrences for each word. The main difference is that the lists of occurrences are not sorted by text positions but lexicographically. In this scheme, simple words can be searched using the supra-index only (the suffix array block is directly the answer) without accessing the text. Searching phrases is simpler with this scheme than with inverted indexes.

![Figure 1.4](image-url)

Figure 1.4: Our suffix array and a word supra-index. This time the arrows need to be stored.

With regard to space requirements, the suffix array requires a pointer for each selected index point of the text. Considering that pointers take four bytes, this means that four times the text size is required to index all the text positions, while if indexing the beginnings of non-stopwords the
structure needs 30% to 40% of the text size, still rather more than the equivalent document-level compressed inverted file. The extra space required by supra-indexes is negligible.

### 3.2 CONSTRUCTION

The easiest way to build a suffix array is via an indirect sort of the pointers. That is, first the index points of the text are identified and stored (in increasing text position order) in the suffix array, and then any sorting algorithm is applied on the suffix array. In this sorting process, two suffix array positions are compared by considering the lexicographical relationship between the text suffixes indexed by the two pointers.

Therefore, a suffix array on a text of $n$ index points can be built in $O(n \log n)$ comparisons, where each comparison compares two strings. The cost of a random string comparison is $O(1)$ at the beginning of the sorting process (since on average $\sigma/(\sigma-1)$ comparisons are necessary to differentiate two random text positions, where $\sigma$ is the alphabet size). However, as the sorting process continues, strings that are increasingly similar are compared. On average, two text suffixes that are neighbors in the final suffix array share $O(\log n)$ common letters (86), which shows that the cost in terms of number of character comparisons is between $O(n \log n)$ and $O(n \log^2 n)$. Moreover, in some special cases where, for instance, a complete document is repeated in the collection, the performance can be very bad because very long common substrings exist. The worst case is to build the suffix array for a text like “aaaaaaaa...”, which takes $O(n^2 \log n)$ time. A possible solution to this problem is to weaken the structure so that only the first $\ell$ characters of the suffixes are considered. In this case the worst case cost can be made $O(n \ell \log n)$, and we can search for patterns of length up to $\ell$.

A different construction algorithm which builds the complete suffix array (not the weaker $\ell$-limited version) in $O(n \log n)$ worst case time and $O(n \log \log n)$ on average is presented by Manber and Myers (58). The idea is to perform a sequence of bucket sorts, sorting the suffixes according to longer and longer initial prefixes. At the beginning of the $i$-th iteration the suffixes are already sorted by their first $2^i$ letters, and at the end of that iteration are sorted by their first $2^i$ letters. Hence, in $\lceil \log_2 n \rceil$ iterations the array is totally sorted. Moreover, since on average $O(\log n)$ initial characters are necessary to distinguish between neighboring suffixes of a text, on average only $O(\log \log n)$ iterations are necessary. The complexity follows from the fact that each bucket sort takes $O(n)$ time.

The algorithm starts by sorting the suffixes according to their first letter. Since there are only $\sigma$ possible different values, the bucket sort is simple. At iteration $i > 0$, we refine the buckets obtained at iteration $i - 1$. The sort for the first $2^i$ letters uses the fact that all the suffixes are already sorted by their first $2^{i-1}$ letters. In particular, if two index points $j_1$ and $j_2$ are currently in the same bucket, then they share their first $2^{i-1}$ symbols, and their first $2^i$ symbols can be written as $xy$ and $xz$, where $|x| = |y| = |z| = 2^{i-1}$. Comparing them is the same as comparing $y$ with $z$. And because these are of length $2^{i-1}$, they are already sorted somewhere, since they also start suffixes. Their relative order depends on the buckets in which the index points ($j_1 + 2^{i-1}$ and $j_2 + 2^{i-1}$) are in this iteration.

In particular, if $j_1 + 2^{i-1}$ is currently in the first bucket of the suffix array, then $j_1$ should, after the next iteration, be in the first sub-bucket of its current bucket. This idea is used to refine all the current buckets in a linear pass over the suffix array. We refer the reader to the original article (58) for the remaining details. The method can be adapted to the case where not all the letters are index points, simply by replacing the first iteration of the algorithm with an ad-hoc sort.

Faster algorithms have been presented recently (46, 55, 79). These are hybrids between bucket sorting (as in the Manber and Myers algorithm) and standard sorting algorithms. Despite the fact that they have the same asymptotic time requirements, in practice the new methods are many times faster.
Sorting Large Suffix Arrays. The previous methods work well if the text and the suffix array fit in main memory. However, when this is not the case they suffer from the same problem identified above with respect to the search algorithms: the text is accessed in an essentially random manner on disk. And even if a sorting algorithm designed for secondary memory is used, the problem of random text accesses via the pointers remains.

The following algorithm (38) builds a suffix array without random access to the text or suffix array. The text is cut in blocks and the suffix array is incrementally built for the blocks. That is, at iteration \( i \) the algorithm obtains the suffix array for the first \( i \) blocks of the text. The block size is selected so that the suffix array for one text block can be built in main memory using the previous algorithms.

The construction of the first block is trivial. At iteration \( i > 0 \), we have already on disk the suffix array of the first \( i-1 \) text blocks. We bring the \( i \)-th text block into memory and build its suffix array. The problem now is to merge the new small suffix array with the large suffix array for the previous text. A classical merge for two sorted sequences cannot be used because it would imply comparing text positions from the first \( i-1 \) text blocks, which are on disk. However, the \( i \)-th text block is in memory and the idea of the merge can be carried out after all. If we are able to determine, for each consecutive pair of positions in the current small suffix array, how many (consecutive) positions of the large suffix array lie between them, then we can perform the merge without accessing any text. The problem is then how to compute those counters that tell, in other words, how many suffixes of the first \( i-1 \) text blocks lie between each pair of positions of the small suffix array.

The key idea is that the \( i-1 \) text blocks can be used for this purpose rather than their corresponding suffix array. We simply traverse the first \( i-1 \) text blocks sequentially, and search each text suffix in the small suffix array. When the location of that suffix has been determined – that is, that it lies between two given positions – the corresponding counter is incremented. And to search in the small suffix array we just need the \( i \)-th text block, which is in memory.

Overall, the process for the \( i \)-th step is: (1) build the small suffix array for the \( i \)-th text block; (2) initialize the counters; (3) read all the first \( i-1 \) text blocks and search each of their suffixes in the small suffix array, incrementing for each the corresponding counter; (4) use the counters to merge the small suffix array and the large suffix array representing the first \( i-1 \) text blocks.

If the text is of size \( n \) and we have \( O(M) \) main memory, \( O((n/M)^2) \) blocks are processed, for a total CPU time of \( O(n^2 \log M/M) \). This is greater than the \( O(n \log n) \) complexity of the Manber and Myers algorithm, but is much faster in practice because all of the operations are either sequential, or carried out in memory. Other algorithms and a comparison of them, including the algorithm just presented, can be found in (25).

### 3.3 Complex Searching

Although we have explained only how to search for simple queries, suffix arrays can do much more. Suffix arrays are closely related to suffix trees (5). A suffix tree is a trie data structure built on the suffixes of the text, and the suffix array is obtained by collecting the trie leaves in order. A suffix tree can be built in \( O(n) \) time and takes \( O(n) \) space. Simple strings of length \( m \) can be searched in \( O(m) \) time. Regular expressions can be searched in \( O(n^\alpha) \) average time, where \( 0 < \alpha < 1 \) is a constant dependent on the regular expression (8). The algorithm backtracks in the trie looking for text substrings that can match the regular expression. It is possible to adapt the search to allow a limited number of errors (for example, missing characters, extra characters, or altered characters) in the text occurrences at \( O(\exp(m)) \) or \( O(n^\alpha) \) time (67, 88). The suffix tree has been used in many different applications, including finding the longest common auto-repetition in the text, and matching the whole text against itself with or without errors (5, 7, 38, 39).
The main drawback of suffix trees is that they take 20 to 60 times the text size, which considerably limits their applicability. On the other hand, suffix arrays can simulate any algorithm designed on suffix trees at an $O(\log n)$ extra time penalty, a rather modest overhead for most applications. The key idea of the simulation is that each node in a suffix tree correspond to an interval in the corresponding suffix array, and hence any movement in the suffix tree can simulated by two binary searches in the suffix array. And the benefit that arises from this tradeoff is clear - a suffix array requires just 4 times the memory space of the text being indexed, even if every character of the text is an index point.

4. PARALLEL AND DISTRIBUTED INDEXES

Parallel or distributed processing can be used for two independent tasks: index construction and querying. In some cases, like suffix arrays, it is possible that the bottleneck is in the construction rather than in the querying process. In other cases, the construction is not the problem, but the index has to be distributed to speed up queries, whose performance may be critical to the success of the system.

We start by giving a quick overview of the efficiency measures in parallel computing. Then, we identify three different formats in which an index may be distributed. This is independent on whether the index is generated sequentially or in parallel. Some of the formats can also be queried sequentially or in parallel, although others are specifically designed for a parallel environment. Later, we discuss how to generate and query each type of index, sequentially or in parallel, and analyze the advantages and disadvantages of each choice.

4.1 EFFICIENCY MODELS

The main difference between “parallel” and “distributed” computing is that in the first the processors can share the data and in the second they cannot. More specifically, there is a moderate to high transmission cost to move data from one processor to another, which ranges from a cost similar to that of accessing secondary storage (3) (in a local ATM switch, for instance) to several seconds (in the Internet, for example). We present our algorithms at a high level trying to address both scenarios, but we are careful in pointing out the key factors that make a parallel or a distributed implementation preferable in each case.

The obvious measure for the efficiency of a parallelized index construction is related to the total elapsed time. This can be expressed with a formula called speedup, defined as the time taken by the best sequential algorithm divided by the time taken by the parallel algorithm. The speedup is optimal when it reaches $P$, the number of processors.

When it comes to measure the efficiency of a parallel querying scheme, two possible alternatives exist. A first one, called throughput, is related to the amount of queries answered per time unit and in the ideal case it reaches $P$ times that of a sequential algorithm. A second one, called response time, is the amount of time taken to solve a query and ideally reaches $1/P$ of that of a sequential algorithm. For instance, if two processors take queries from different users and solve them sequentially, the throughput doubles but the response time for each user does not change since their query has been solved sequentially. If, on the other hand, both processors cooperate to solve each query, the response time may be halved, but the overhead costs necessary to divide the query and integrate the solutions may reduce the throughput. Which of the two measures is more important largely depends on the application. Increasing the throughput may be more important when there are many inexpensive queries per second and therefore the sequential response time is acceptable. Reducing response time may more important if there are a small number of expensive queries to be processed and the goal is to reduce the time for each.
4.2 INDEX DISTRIBUTION

In a distributed environment, an index can be stored in three formats, illustrated in Figure 1.5.

![Figure 1.5](image)

**Figure 1.5** Three ways to distribute an index. A centralized index is not partitioned, a document partitioning splits the index by documents, and a lexicographical partitioning splits the index by lexicon.

**Centralized index:** this is the conventional arrangement in which the index is stored in a single location.

**Document partitioned index:** the text collection is distributed across a set of machines, and a local index is separately built for the subcollection maintained at each machine. Each index is totally independent from the rest.

**Lexicographically partitioned index:** the text collection is distributed, but the index is global and each machine is responsible for completely handling the queries that lie in some subset of the vocabulary. Each machine knows which of the other machines is responsible for each subset of the query terms. For simplicity we assume that the sets are lexicographical ranges, but other partitions are possible.

Even a centralized index can be searched in parallel, as we see later. The second choice is clearly the simplest way to take advantage of the presence of many processors to answer queries. The most challenging architecture is the third one, in which there is a single index distributed across all the machines.

An important kind of distributed index arises in loosely coupled heterogeneous databases that for efficiency, formatting or administrative reasons cannot cooperate at index construction time. In this case the only scheme that can be applied is document partitioning.

4.3 PARALLEL INDEX GENERATION

The easiest case of parallel index generation is that of document partitioned indexes, since each processor indexes its local text using a sequential algorithm.

A centralized index, on the other hand, can be generated in parallel as well. Normally, the process of producing a centralized index is just one step over that of producing a lexicographically distributed index. We just have to concatenate the lists of occurrences stored in the different processors to obtain the global list of occurrences. In the case of a suffix array this corresponds to concatenating the local suffix arrays obtained.

The most complex algorithms are those to generate lexicographically distributed indexes. We divide them according to the type of index that is to be produced.
Inverted Indexes. We describe an algorithm presented in (77, 78). We start with the text distributed across the machines. Each processor obtains the vocabulary of its local text. Once this is done the processors engage in a merging process to obtain the complete vocabulary and the global information necessary to distribute it across the processors. The merging of the vocabulary is done by pairwise merging: first each processor numbered $2i + 1$ sends its vocabulary to that numbered $2i$, then each processor numbered $4i + 2$ sends its vocabulary to that numbered $4i$, and so on until a single processor holds all the vocabulary. This is possible because the vocabulary is small.

Once a processor holds the whole vocabulary it assigns a subset of the words to each other, and broadcasts that information. Now each process knows which lists of occurrences must be stored at each other processor, so they obtain the occurrences from the text and send them to the final owners. There are different choices on when to send those lists, that range from sending each occurrence as it is found to building the local inverted file and then distributing the lists.

Suffix Arrays. The construction of distributed suffix arrays is essentially a distributed sorting algorithm. From the local texts each processor builds a local suffix array and the problem is then how to obtain a globally sorted suffix array from the locally sorted pieces. Different algorithms based in different sorting algorithms (mergesort and variants of quicksort) are proposed in (52, 53, 69).

The main complication of the suffix array construction on top of the classical sorting problem is that two suffixes cannot be directly compared since they are pointers to possibly remote text. Using a general sorting scheme and making remote requests each time a comparison is made is impractical. This is generally solved by storing the first characters of each suffix together with the pointer. If the suffixes are long enough this suffices to solve most of the comparisons locally and only a few of them need remote access to the text. Moreover, the suffixes can be kept in the final index to aid in the querying process, as explained in the next section. Of course there is a space-time tradeoff: longer suffixes reduce the number of comparisons but increase the storage required, and they also pose higher communication overheads.

An algorithm to build suffix arrays which is not based in classical sorting is presented in (51). The algorithm is a parallelization of the original sequential Manber and Myers algorithm to build the suffix array (58).

4.4 PARALLEL QUERYING

The querying process used is, of course, determined by the index arrangement. The next three subsections describe the mechanisms possible for parallel querying in each of the three index models described above.

Centralized Index. Parallelism can be used even with a centralized index. The simplest way is to have a threaded database server where many processes answer queries independently. Even if there is a single processor to run the threads the throughput can be increased by parallelizing the CPU and I/O tasks. In general, however, handling text indexes is an I/O intensive task, and therefore the number of I/O channels and disks must increase in accordance to the intended gain in throughput. Consideration must also be given to the extent to which multi-threading disrupts the caching and buffering operations on single processor machines. Running too many threads can have a disruptive effect, resulting in a high proportion of cache misses and a reduction in overall throughput.

On the other hand, if response times are to be reduced, parallelizing the processing of each individual query is necessary. A query may consist of a number of different words or patterns that must be searched for and then later combined via Boolean operations or ranking. The set of
patterns to search can be partitioned among the processors, so that each one accesses the shared index to retrieve the relevant lists of term appearances (be it from an inverted list or a suffix array).

Once the lists of occurrences have been obtained, the most common combining operations are set unions and intersections (for Boolean operators) and sorting (for ranking or suffix arrays). A considerable literature on basic parallel algorithms for set manipulation and sorting has been developed (see, for example, [47, 75]), and many of those techniques are applicable.

In a distributed scenario the communication costs are too high to use a really centralized index. The obvious alternative is replication: there is an exact replica of the index in each machine. Queries can be distributed dynamically according to the workload. Although, again, this increases the throughput of the whole system, the response time does not change. Partitioning the query among processors in a distributed environment is precisely the goal of the two distributed querying schemes that are described next.

**Document Partitioned Index.** The basic strategy in a document partitioned index is to broadcast every query to every processor. Each processor solves the query for its local text, using its local index. Once the query is solved they send back their results to the originating processor, which collates the results and delivers a set of answers to the user.

This scheme is extremely simple to implement and parallelizes the query very well: not only the throughput increases but also the query time is reduced. There are, however, two reasons why perfect speedup in response time cannot be achieved. The first is that the query time using indexes is normally sublinear. That is, if \( n \) is the global text size and \( f(n) \) is the time to answer a query in such text, then \( f(n) = o(n) \), and this immediately implies that \( f(n/P) = \omega(f(n)/P) \). In other words, halving the text does not mean that the query time will halve too. The second reason is that there is an overhead in transmitting and assembling the results.

Assembling the results may be simple or complex depending on the type of query. Since the documents or text positions delivered correspond to disjoint portions of the space, Boolean queries require no more than the concatenation of the individual result sets. The problem is more complex with ranked queries (29). First, there is a performance problem because only a few documents transmitted by each processor will actually qualify after being compared to the rest, an effect that worsens as the number of processors grows. A possible solution to the problem is a lazy scheme where the querying processor asks progressively fewer and fewer results of the processors, stopping when it has enough to answer the query. Another solution is that the processors locally remove the documents that will probably not qualify after a global comparison, therefore transmitting \( O(R/P) \) results, where the final list must have \( R \) elements. This is a heuristic and may yield incorrect results, especially if one processor has results much more relevant than the rest.

This is related to a more serious problem, which is the consistency of the ranking. Each processor has ranked the query according to its parameters of local term frequency and inverse document frequency. The result is not the same if a global ranking is applied. One solution is, at indexing time, to compute and distribute the global frequencies of the terms to all the processors. This scheme, however, is less than satisfactory because the processors are no longer independent in administering their local text collection. In particular, it becomes a hybrid with a lexicographically partitioned index. Another solution is to construct a central vocabulary (rather than a central index) after each of the separate indexes has been constructed; another is to add a further synchronization step at query time in which each processor is first polled to determine local term frequencies and weights for the query terms, and then global query term weights are broadcast back to each of processors so as to ensure that all ranking scores are compatible (29).

Finally, we consider the best way to partition the documents across the machines, when there is a choice. The simplest alternative and one of the best in practice is a random distribution. This ameliorates the problem of ranking inconsistencies, because the global term frequencies should be
similar to the local ones. The workloads also tend to be uniform, since none of the machines will be consistently more loaded than others simply because they hold all of the more popular documents.

The opposite choice is a domain-driven partition: a subject or domain is assigned to each machine and all the documents relevant to that domain are administered by that machine. This makes the relevance ranking very different and does not guarantee a uniform workload. Moreover, even when the workload is uniform the response time is normally not improved because most of the work for a given query is done by a single machine. This distribution, however, is sometimes chosen for administrative reasons or to avoid broadcasting every query to every processor. In fact, this design gets closer to a lexicographically partitioned index, which is covered in the next section.

**Lexicographically Partitioned Index.** In this type of index one processor takes full responsibility of the processing of each query term, and the set of processors involved must either later cooperate somehow to perform the ranking or Boolean operations, or transfer the partial results back to the querying processor for collation.

The machine that originates the query knows, for each pattern of the query, which machines are responsible for the involved terms. Therefore, the query is sent only to the responsible machines. In the simplest scenario, those machines send back their result to the originating processor, which is responsible for assembling the results. The assembly is more complex now, as it can involve performing Boolean operations or ranking among the lists.

Simple queries, such as single words and patterns posed to inverted indexes or suffix arrays, may require no further processing. This gives maximum throughput gain, but the response time is unaltered. On the other hand, in complex queries with large intermediate results the most costly part of the process may be the assembly of the occurrences. It is possible that the processors cooperate in this part too, by merging, intersecting or sorting the lists pairwise in parallel instead of giving all the work to the originating processor. For example, if a conjunctive query with four terms was split among processors 1 to 4, then after obtaining the lists processors 1 and 2 can intersect their lists while processors 3 and 4 do the same, and only then the final processor intersects the two remaining lists.

Globally consistent ranking is not a problem in this type of index, because the processor that solves a query has global information about it. To obtain the global ranked results, the processors send back the lists of documents ranked for the term(s) they have solved. A global merging process is carried out, quite similar to set union but taking care of producing the highest ranked documents first. This is more difficult than for a document partitioned index because each document may appear in many lists (if it was selected by many query terms), very high in some lists and very low in others. In a document partitioned index each document appears in one list and producing the ranked result is trivial. The technique of lazy merging is more difficult to apply and less efficient in a lexicographically partitioned index.

A specific problem of suffix arrays with this scheme is that they need to access the text in order to answer queries. Since the text pointed by the array is not necessarily local, accessing remote text becomes necessary. This problem can be alleviated by storing supra-indexes or the first characters of all the suffixes, but in general some remote text has to be accessed to complete the answer. Other indexes that have problems with this distribution strategy are variations on the inverted index that need direct access to the text, for instance to solve phrase queries.

Finally, notice that the querying scheme for the lexicographically partitioned index is similar to the design shown to partition a query in a centralized index, as each processor takes care of some patterns of the query.
It is clear that if there is only one machine, be it with one or many processors, the only choice is a centralized index, where the sole design decision is how to distribute the index and text across the disk array to improve performance. Queries are solved atomically or are split into components depending on whether maximizing throughput or minimizing response time is important. The only situation in which a document-partitioned index might be used is when the single collection becomes physically too large to be managed, either because of storage constraints on specific devices (memory or disk), or because of the cost of constructing a monolithic index.

In a distributed environment, however, we have presented three choices of partitioning the index: replicated, partitioned by documents and partitioned by lexicon. Replication is the best alternative in terms of performance, as it can behave like any of the other two. Of course, replication is only applicable if the collection is small enough to fit in a single machine. However, some replication can be included in the other two designs to improve the efficiency and make the system fault tolerant.

Of the remaining two alternatives, it is not immediately clear which is the best choice, and studies have reached different conclusions under alternative cost scenarios – Tomasic et al. (87) show that document partitioning performs better on large conjunctive queries where the terms do not distribute uniformly in the query, and Ribeiro-Neto et al. (76) show that lexicographical partitioning is better when the terms distribute uniformly in the queries. This is clear, since if the queries do not distribute uniformly across the documents the document partitioning scheme will suffer a degradation as a consequence of reduced parallelization.

First consider single term queries. A document partitioning index does not achieve optimum throughput, because the local times are not simply the global times divided by $P$ and because of the effort to assemble the final list of results. However, it will improve the response time because the work to answer the query is effectively divided among many processors. A lexicographically partitioned index, on the other hand, achieves optimum throughput if the queries distribute uniformly in the vocabulary, because there is virtually no overhead (unless direct access to remote text is necessary, as explained): the machine that takes care of the query does all the job. Response times, however, are not improved because a single machine is solving each query. Therefore, the question about which partitioning scheme is better boils down to a question of which efficiency measure is important for the application. For many cheap queries the lexicographical partition is probably better, while for few costly queries the choice should be document partition.

Extra considerations are necessary for more complex queries, however. Consider the case of a conjunction of many simple terms, whose final result is much smaller than the list of occurrences of each individual term. In this case a document partitioned index performs much better. The reason is that all the intersections are carried out locally, and it only transmits the final results corresponding to its subset of the text collection. The lexicographically partitioned index, on the other hand, has to intersect long lists which are not in the same machine, so no matter how the intersections are done, long occurrence lists must be transmitted. A similar, although less critical problem, arises with disjunctive queries, since the document partitioned index does not transmit redundant information across the network and the lexicographical partitioned index may transmit many times the same documents because different query terms appear on them.

Finally, we consider ranking. A first problem of document partitioned indexes is the possibility of inconsistent ranking. As explained, one solution is to have a global vocabulary, which is indeed a hybrid solution between a document and a lexicographical partition. Apart from this issue, ranking is simpler and more efficient in a document partitioned index if there are many terms involved, otherwise the situation is the same as for a single term query.

To summarize, a document partitioned index seems to be the best choice in many different scenarios. Enriching it with global inverse document frequency information is a good choice to ensure correct ranking, although it becomes more complex to administer. On the other hand, a
lexicalographically partitioned index is a better solution when the queries are simple (for example, single words) and hence cheap to solve, and the critical issue is to solve many queries per second. This case is very important, since it comprises a large fraction of most Web querying applications.

5. CASE STUDY: SEARCHING THE WEB

The World-Wide Web (WWW, or simply Web) was created at the end of the 1980s (13), and no one who was present at its conception could possibly have imagined its impact just ten years later. The amount of textual data alone is estimated in the order of at least three terabytes, and other media, such as images, audio, and video, are also available, using even more space than the text portion. Thus, the Web can be seen as a very large, unstructured but ubiquitous database. This triggers the need for efficient tools to manage, retrieve, and filter information. Similar problems are also becoming important in large Intranets or in text mining applications.

In this chapter we focus on text, because although there are techniques to search for images and other non-textual data, they cannot be applied (yet) in large scale. While searching Web pages, one implicit problem is that logically a document could be just part of a Web page (for example, a summary of an article in the set of summaries for a magazine issue) or split across many Web pages (for example, a section of an article). However, this cannot be easily known and we have to use the physical division as document space. We also emphasize syntactic search. That is, we search for Web documents that have user-specified words or patterns in their text. Words or patterns may or may not reflect the intrinsic semantics of the text. An alternative approach to syntactic search is to do a natural language analysis of the text. Although the techniques to preprocess natural language and extract the text semantics are not new, they are not yet very effective and are costly for large amounts of data. In addition, in most cases they are only effective with well structured text, a thesaurus, and other contextual information.

There are basically three different forms of searching the Web. Two of them are well known and are used frequently by most users. The first is to use search engines that index a portion of the Web documents as a full-text database. The second is to use Web directories, which classify selected Web documents by subject. The third, not yet widely available, is to exploit the hyperlink structure present in Web documents. The scope of this chapter is related to the first case. We refer the reader to Baeza-Yates and Ribeiro-Neto (10) for information on other techniques.

We now mention the main problems posed by the Web. We can divide them in two classes: problems with the data itself and problems regarding the user and his interaction with the retrieval system. The problems related to the data are:

- Distributed data: due to the intrinsic nature of the Web, data spans over many computers and platforms. These computers are interconnected with no predefined topology and the available bandwidth and reliability on the network interconnections vary widely.

- High percentage of volatile data: due to Internet dynamics, new computers and data can be added or removed easily (it is estimated that 40% of the Web changes every month (48)). We also have dangling links and relocation problems when domain or file names change or disappear. In addition, many pages are generated on demand, and might be missed by automatic software.

- Large volume: the exponential growth of the Web poses scaling issues that are difficult to cope with. Current estimations of the number of Web pages is at least 800 million (56).

- Unstructured and redundant data: most people say that the Web is a distributed hypertext. However, this is not exactly so. Any hypertext has a conceptual model behind it, which organizes and adds consistency to the data and the hyperlinks. That is hardly true in the Web, even for individual documents. In addition, each HTML page is not well structured
and some people use the term *semi-structured data*. Moreover, much Web data is repeated (mirrors or copies) or very similar. Approximately 20% of Web pages are (near) duplicates (18, 82). Semantic redundancy can be even larger.

- **Quality of data**: the Web can be considered as a new publishing media. However, there is, in most cases, no editorial process. So, data can be even false, invalid (for example, because it is too old), poorly written or, typically, with many errors from different sources (typos, grammatical mistakes, OCR errors, and so on). In particular, around 1 out of every 200 common words and 1 out of every 3 foreign surnames, have at least one typo.

- **Heterogeneous data**: in addition of having to deal with multiple media types and hence with multiple formats, we also have different languages and, what is worse, different alphabets, some of them very large (for example, Chinese or Japanese Kanji).

The second class of problems are those faced by the user during the interaction with the retrieval system. There are basically two problems: (1) how to specify a query and (2) how to interpret the answer provided by the system. Without taking into account the semantic content of a document, it is not easy to precisely specify a query, unless it is very simple. Further, even if the user is able to pose the query, the answer might be a thousand Web pages. How do we handle a large answer? How do we rank the documents? How do we select the documents that really are of interest to the user? In addition, a single document could be large. How do we browse efficiently in large documents? These problems are still not solved completely (10).

Next, we cover different architectures of retrieval systems that model the Web as a full-text database. One main difference between standard IR systems and the Web is that, in the Web, all queries must be answered without accessing the text (that is, only the indexes are available). To do otherwise would require either storing locally a copy of all of the indexed Web pages (too expensive) or accessing remote pages through the network at query time (too slow). This difference has an impact in the indexing and searching algorithms, as well as in the query languages made available.

Most indexes use variants of the inverted file already mentioned before. Some search engines eliminate stopwords to reduce the size of the index, but if the index is stored using the techniques summarized earlier, the actual saving is relatively small. Also, it is important to remember that a logical view of the text is indexed. Normalization operations may include removal of punctuation and multiple spaces and folding of uppercase to lowercase letters. To give the user some idea about each document retrieved, the index is complemented with a short description of each Web page (creation date, size, the title and the first lines or a few headings are typical). Assuming that 500 bytes are required to store the URL and the description of each Web page, we need 50GB to store the description for 100 million pages. As the user initially receives only a subset of the complete answer of each query, the search engine usually keeps the whole answer set in memory for a short time, to avoid having to recompute it if the user asks for more documents.

By using the compression techniques already discussed, the index size can be reduced to under 10% of the text (90). A query is answered by doing a binary search on the vocabulary of the inverted file. If we are searching multiple words, the results have to be combined to generate the final answer. This step is efficient if each word is relatively rare. Another possibility is to compute the complete answer while the user requests more Web pages, using a lazy evaluation scheme.

5.1 **CENTRALIZED ARCHITECTURE**

Most search engines use a centralized crawler-indexer architecture. Crawlers are programs (software agents) that traverse the Web sending new or updated pages to a main server where they are indexed. Crawlers are also called robots, spiders, wanderers, walkers, and knowbots. In spite of their name, a crawler does not actually move to and run on remote machines. Rather, the crawler
runs on a local system and sends requests to remote Web servers. The index is used in a centralized fashion to answer queries submitted from different places in the Web. Figure 1.6 shows the software architecture of a search engine based on the AltaVista architecture (2). It has two parts: one that deals with the users, consisting of the user interface and the query engine and another that consists of the crawler and indexer modules.

![](image)

*Figure 1.6 Typical crawler-indexer architecture.*

The main problem faced by this architecture is the gathering of the data, because of the highly dynamic nature of the Web, the saturated communication links, and the high load at Web servers. Another important problem is the volume of the data. In fact, the crawler-indexer architecture may not be able to cope with Web growth in the near future. Particularly important is good load balancing between the different activities of a search engine, internally (answering queries and indexing) and externally (crawling).

The largest search engines, considering Web coverage in January of 2000, were Fast (31), AltaVista (2), Northern Light (70), Google (40), and Snap (84), in that order. According to recent studies, these engines cover less than 30% of all Web pages (56).

### 5.2 DISTRIBUTED ARCHITECTURE

There are several variants of the crawler-indexer architecture. Among them, the most important is Harvest (16). Harvest uses a distributed architecture to gather and distribute data, which is more efficient than the crawler architecture. The main drawback is that Harvest requires the coordination of several Web servers.

The Harvest distributed approach addresses several of the problems of the crawler-indexer architecture, such as: (1) Web servers receive request from different crawlers, increasing their load; (2) Web traffic increases because crawlers retrieve entire objects, but most of their content is discarded; and (3) information is gathered independently by each crawler, without coordination between all the search engines.

To solve these problems, Harvest introduces two main elements: gatherers and brokers. A gatherer collects and extracts indexing information from one or more Web servers. Gathering times are defined by the system and are periodic (that is, there are harvesting times as the name of the system suggests). A broker provides the indexing mechanism and the query interface to the data gathered. Brokers retrieve information from one or more gatherers or other brokers, updating
their indexes incrementally. Depending on the configuration of gatherers and brokers, different improvements on server load and network traffic can be achieved. For example, a gatherer can run on a Web server, generating no external traffic for that server. Also, a gatherer can send information to several brokers, avoiding work repetition. Brokers can also filter information and send it to other brokers. This design allows the sharing of work and information in a very flexible and generic manner. An example of the Harvest architecture is shown in Figure (16).

One of the goals of Harvest is to build topic-specific brokers, focusing the index contents and avoiding many of the vocabulary and scaling problems of generic indexes. Harvest includes a distinguished broker that allows other brokers to register information about gatherers and brokers. This is useful to search for an appropriate broker or gatherer when building a new system. The Harvest architecture also provides replicators and object caches. A replicator can be used to replicate servers, enhancing user-base scalability. For example, the registration broker can be replicated in different geographic regions to allow faster access. Replication can also be used to divide the gathering process among many Web servers. Finally, the object cache reduces network and server load, as well as response latency when accessing Web pages. More details on the system can be found in (16).

5.3 CRAWLING THE WEB

In this section we discuss how to crawl the Web, as there are several techniques. The simplest is to start with a set of URLs and from there extract other URLs which are followed recursively in a breadth-first or depth-first fashion. For that reason, search engines allow users to submit top Web sites that will be added to the URL set. A variation is to start with a set of popular URLs, because we can expect that they have information frequently requested. Both cases work well for one crawler, but it is difficult to coordinate several crawlers to avoid visiting the same page more than once. Another technique is to partition the Web using country codes or Internet names, and assign one or more robots to each partition, and explore each partition exhaustively.

Considering how the Web is traversed, the index of a search engine is analogous to the stars in the sky. What we see never did exist, as the light has traveled different distances to reach our eye. Web pages referenced in the index were also explored at different dates and they may not exist any
more. Nevertheless, when we retrieve a page, we obtain the actual content of it. How fresh are the Web pages referenced in an index? The pages will be from one day to two months old. For that reason, most search engines show in the answer the date when the page was indexed. The percentage of invalid links stored in search engines vary from 2 to 9%. User submitted pages are usually crawled after a few days or weeks. Starting there, some engines traverse the whole Web site, while others select just a sample of pages or pages up to a certain depth. Non-submitted pages will wait from weeks up to a couple of months to be detected. There are some engines that learn the change frequency of a page and visit it accordingly (26). They may also crawl more frequently popular pages (for example, pages having many links pointing to them). Overall, the current fastest crawlers are able to traverse up to several million Web pages per day using many processes.

The order in which the URLs are traversed is important. As already mentioned, the links in a Web page can be traversed breadth first or depth first. Using a breadth first policy, we first look at all the pages linked by the current page, and so on. This matches well Web sites that are structured by related topics. On the other hand, the coverage will be wide but shallow and a Web server can be bombarded with many rapid requests. In the depth first case, we follow the first link of a page and we do the same on that page until we cannot go deeper, returning recursively. This provides a narrow but deep traversal. Only recently, some research on this problem has appeared (24), showing that good ordering schemes can make a difference if crawling better pages first.

6. CONCLUDING REMARKS

There are many research problems in searching large text collections. Many of them are related to the Web, which is certainly the application which currently enjoys the highest profile. Some important trends are:

- **Distributed architectures**: New distributed schemes to traverse and search the Web must be devised to cope with its growth. This will have an impact on current crawling and indexing techniques, as well as caching techniques for the Web. Currently it remains unclear whether server capacity or network bandwidth will be the bottleneck limiting performance.

- **Indexing**: Which is the best logical view for the text? What should be indexed? How can good compression schemes be exploited to achieve fast searching and reduced network traffic? And how can word lists, URL tables, and so on be efficiently and effectively compressed and updated?

- **Searching**: How to cope with syntactical uncertainty in the user query and with typing errors in databases with low quality control such as the Web? How can we allow for more flexible pattern searching while keeping at the same time search efficiency and user friendliness in the query language? How can we index to search for flexible patterns? How does this affect the precision and recall of the search?

- **Duplicated data**: Good mechanisms to detect and eliminate repeated Web pages (or pages that are syntactically very similar) are needed. Initial approaches are based on resemblance measures using document fingerprints (18, 19). This is related to an important problem in databases: finding similar objects.

An important issue to be settled in the future is a standard protocol to query search engines. One proposal for such a protocol is STARTS (41), which allows users to choose the best sources for querying, evaluate the query at those sources, and merge the query results. This protocol would make easier to build metasearchers, but at the same time that outcome could also be given as a reason for not having a standard, to prevent metasearchers from profiting from the work done by search engines and Web directories. This is a particular case of the federated searching problem.
from heterogeneous sources, as it is called in the database community (74). Federated searching is a problem already studied in the case of the Web, including discovery and ranking of sources (22, 42, 91). These issues are also very important for digital libraries (73) and visualization (1). A related topic is metadata standards for the Web. XML helps (30, 35, 50), but semantic integration is still needed.

Hyperlinks can also be used to infer information about the Web. Although this is not exactly searching the Web, it is an important trend called Web mining. Traditionally, Web mining had been focused on text mining, that is, extracting information from Web pages. However, the hyperlink structure can be exploited to obtain useful information. For example, the ParaSite system (85) uses hyperlink information to find pages that have moved, related pages, and personal Web pages. Hyperlinks can also be used for ranking, and has also been used to find communities and similar pages (34, 54). Other results on exploiting hyperlink structure can be found in (21, 60, 72). Further improvements in this problem include Web document clustering (18, 23, 89) (already mentioned), connectivity services (for example, asking which Web pages point to a given page (14), automatic link generation (43), extracting information (15, 17), and so on.

More information on searching and crawling the Web can also be found in the chapter of this book by Andrei Broder and Monika Hezinger, titled *Algorithmic Aspects of Information Retrieval on the Web*. That chapter also covers the main challenges: ranking, duplicate filtering, and search-by-example.
References


