Very Large Scale Information Retrieval

David Hawking

CSIRO Mathematical and Information Sciences, Canberra, Australia

Abstract. This chapter is based on a series of five lectures presented at the EL-SNET TesTia Summer School held in Chios, Greece in July, 2000. The material has been updated in August 2001 and, at the suggestion of the students, some explanatory diagrams which were at the time drawn on the whiteboard have been included in more polished form.

The scale of electronic document collections has grown dramatically in recent decades. Test collections of the 1960s and 70s (such as Cranfield [9]) contained thousands of documents; the initial TREC collection of 1991 [21] reached almost a million; and the collections indexed by current Web search engines contain approximately a billion.

Information Retrieval (IR) has been associated from its beginning with the analogy of "looking for a needle in a haystack." Extending this metaphor to very large scale, we see that the haystack is now big enough to cover Australia! Furthermore, enthusiastic farmers have filled it with every possible type of item, including many which are very similar to needles but which are not what the searcher wanted. Most items include instructions on how to go directly to other items, but often the instructions are misleading or out of date.

Now there are not only needles but sewing machines, business cards for tailors, needle exchange services, needle-sharpening services, a sewing technology futures exchange, catalogues of needles available for sale or hire and directories of where to find needles within the haystack.

Unfortunately, cunning businesspeople have inserted items which look identical to needles but which turn out to be pictures of naked women or advertisements for get-rich-quick schemes.

Millions of searchers arrive each day to search and they do so with the expectation that they will find what they want within less than two seconds. Some of them are very demanding; they are looking for a particular individual needle and they will not be satisfied unless they find it first. Others want to find as many different needles as they can. Some just want to get an overview of the types of needles which are "out there". A few start looking for needles but when they find one, realise that what they really wanted was a can-opener or an air-ticket to Hawaii!

This chapter attempts to cover the changes which occur when document collections and searcher populations become very large. It addresses the major engineering challenges imposed by very large scale search (particularly on the World Wide Web), outlines parallel and distributed models and canvasses the problem of how to evaluate the effectiveness of very large scale retrieval.

S. Renals, G. Grefenstette (Eds.): Text- and Speech-Triggered Info. Access, LNAI 2705, pp. 106-144, 2003.

Table 1. Examples of different types of Information Retrieval (IR) application.

Category	Description	Example Task
Ad hoc retrieval	Find "relevant" documents in a fixed collection.	*Find documents which tell me about investment strategies.*
Question answering	Extracting answers from retrieved documents.	*Who is the prime minister of Australia?*
Directory lookup	Navigating to a specific Web page.	*Where is the ELSNET home page?*
Selective dissemination of information.	Monitor an incoming stream of documents for ones which match a stored profile.	*Send me any new information on high-tech companies.*
Document Clustering	Automatically grouping similar documents.	*Find the natural groupings in this set of scientific publications.*
Document Categorisation	Assigning pre-defined category labels to a set of documents	*Classify incoming books according to their Dewey decimal category.*
Document Synthesis	Extracting information from multiple retrieved documents	*Construct a personalised travel guide for my visit to Athens in July, 2000.*
Database Lookup	Extracting records from a structured database.	*Find books where author = Hawking and year = 2001*

1 Introduction to Information Retrieval (IR)

Before considering the special issues associated with very large scale, it is important to have an understanding of the fundamentals of IR. A recent text book in the area is [3].

1.1 Types of IR Application

Table 1 lists a number of different types of electronic information processing activity which may be considered to fall under the IR umbrella. Among these applications, the inclusion of database lookup may be considered a little controversial, because the database and information retrieval fields of research are traditionally distinct.

Database research generally deals with highly structured data and with issues of simultaneous update, transaction logging, access authorisation and recovery after failure. The types of queries which can be supported by a given relational database are determined by the database schema and queries have a precisely defined, certain answer set[1]. Any uncertainty which may have been present in the original data has been removed during data entry.

[1] For simplicity of exposition, let us ignore the fact that many modern database systems include text retrieval facilities such as free text fields, relevance ranking and approximate textual matching.

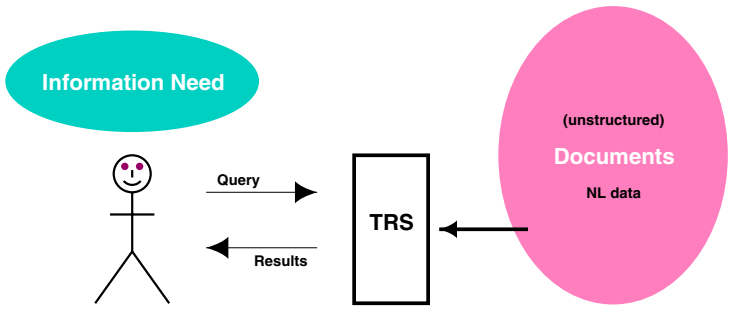

Fig. 1. The ad hoc text retrieval model. A searcher with a particular information need submits a query to a Text Retrieval System. The query is processed against a document collection, whose contents may be considered to be static, and a set of results is returned to the searcher.

Topic A fully-detailed written description of a searcher's information need. (As an researcher might write down for a research assistant.)

Query What the searcher actually types to the retrieval system in order to try to satisfy their information need. Queries are usually very much shorter than the topics to which they correspond.

Search term The textual elements of a query, such as words, phrases, word prefixes etc. The list of acceptable search types depends upon the particular retrieval system.

Boolean retrieval system One which takes a query containing logical operators such as AND, NOT and OR and produces an unranked answer set containing all documents which match the query expression.

Ranked retrieval system One which takes a query and ranks documents on the basis of a computed similarity or relevance score.

Fig. 2. Terminology. Definitions of some jargon used in the text.

By contrast, IR research generally deals with unstructured (or semi-structured) text or multimedia documents and often considers collections to be read-only, thereby avoiding the need to consider updates. However, this simplification is compensated for by uncertainty as to what constitutes the set of right answers. Modern retrieval systems tend to rank documents in decreasing order of estimated likelihood of relevance. Whether or not a document is actually relevant can only be determined subjectively, by a human judge. Judgments vary from person to person and may also depend upon the judge's state of mind at the time.

Space prevents treatment of all the Table 1 IR applications here. Accordingly, this chapter concentrates entirely on the ad hoc retrieval application, particularly in the context of Web[2] search. There are important issues of scale in other IR applications such as

[2] Here, the capitalised W is used to mean the World Wide Web as opposed to an arbitrary hyperlinked web of documents.

clustering but ad hoc retrieval on the Web reaches very large scales indeed and is used daily by millions of people.

1.2 Ad Hoc Retrieval

Figure 1 shows the basic model of ad hoc document retrieval. In its purest form, a stream of incoming queries is processed against a fixed set of documents, the inverse of the *selective dissemination of information* case, where a stream of incoming documents is processed against a fixed set of queries (see Table 1). Figure 2 defines some expressions which will be used in the following discussion.

Web search engines are now the most heavily used ad hoc retrieval service but ad hoc retrieval systems are also found on individual websites, in commercial information services such as Dialog and Lexis-Nexis and on informational CD-ROMs such as encyclopaedias.

Results from some ad hoc retrieval systems are in the form of an unranked set comprising all documents matching a specified criterion. Queries to such a system are usually Boolean (eg. `ELSNet AND "Summer School" AND (Chios OR Greece)` and the systems are often described as *Boolean retrieval systems*. Unskilled searchers often have trouble understanding Boolean queries. (Does the query `cat AND dog` mean "I want documents which mention both words", or does it mean, "I want documents that contain cat AND I want documents that contain dog"?)

The recent trend has been in favour of *ranked retrieval systems* in which queries are treated as *bags of words*. This means that there are no operators and that the order of query words isn't important. For example, `Summer Chios School`. In a ranked retrieval system, the result set is sorted in order of decreasing estimated relevance to the query. Relevance estimates are made by combining weights of the query features in a document. In the simplest case the query features are the query words and the weight assigned to a query word in a document may depend upon the number of occurrences in this document, the length of the document and the number of other documents containing this word.

The basic elements of a query, such as words, phrases and part words are usually referred to as *terms*.

In reality, ranked retrieval systems often have Boolean aspects. Often, only documents which are members of the set which would result from the dysjunction of all the query terms (`Summer OR Chios OR School)` are eligible to be ranked. In fact, some popular search engines restrict rankings to documents which are members of the set which would result from the conjunction of all the query terms (i.e. `Summer AND Chios AND School)`. From here on, only ranked retrieval systems will be considered.

Searchers in an ad hoc retrieval system are concerned with various important dimensions of a ranked retrieval service:

1. Does it present results in a useful way?
2. Does it respond quickly enough?
3. Does it rank documents in sensible order?

The first of these questions is largely independent of the scale of the retrieval problem, and comes under the area of Human-Computer Interaction (HCI), but questions two and

three are particularly important in the area of very large scale retrieval. Past IR research has mostly focused on question 3.

A series of collaborative experiments in ad hoc retrieval has been carried out since 1991 under the auspices of the Text REtrieval Conference, TREC [41]. The TREC ad hoc test collections now comprise about two million government and newspaper documents, along with 500 topics and corresponding relevance judgments.

```
<num> Number: 261
<title> Topic: Threat posed by Fissionable Material

<desc> Description:
Does the availability of fissionable material in the
former states of the Soviet Union and its susceptibility
to theft, pose a real and growing threat that terrorist
groups/terrorist states will acquire such
material and be able to construct nuclear weapons?

<narr> Narrative:
Under the terms of the strategic disarmament treaty with
the U.S., the states of the former Soviet Union have been
dismantling 2000 warheads each year.  From each warhead a
shiny sphere of plutonium is extracted.  These spheres,
called ``pits'', are the elemental cores of a bomb.  In addition,
other forms of plutonium are scattered over the former Soviet
Union in institutes, laboratories, plants, shipyards and
power stations.  Disgruntled employees, who are often underpaid
or paid irregularly have access to the plutonium.  This worries
leaders in other countries.  Enriched uranium, an alternate fuel,
is harder to come by because it is stored in well-guarded military
facilities, but it is easier to turn into a bomb.  The Russians
have denied that it came through or from their country, but German
authorities believe that it did.  Any item which speaks to failures
in the safeguarding of nuclear material or to black-market operations
in nuclear material, or to efforts of terrorist groups or terrorist
states to acquire such material would be relevant.
</top>
```

Fig. 3. An example of a TREC topic. The narrative in this case is longer than average.

TREC distinguishes between *topics*, which are structured, detailed, English language statements of a searcher's information need and *queries*, possibly expressed in a system-specific query language, which are sent to the retrieval system in an attempt to find documents matching the underlying information need. Figures 3 and 4 show a sample TREC topic and queries which might correspond to the same information need.

Figure 5 illustrates the test collection approach to information retrieval evaluation. A realistic information need is recorded, e.g. as a TREC topic, and a corresponding

```
A. Threat posed by Fissionable Material

B. [threat* danger*] [fissionable plutonium uranium U238]
   [USSR Soviet]

C. (threat OR danger) AND (plutonium OR uranium OR fissionable OR U238)
   AND (USSR OR Soviet)
```

Fig. 4. Examples of different queries derived from the example topic in the preceding figure.

query is fed to the IRS (Information Retrieval System). The query may be generated by automatic processing of the topic description or it may be manually generated by either the originator of the search or by a search intermediary.

The IRS processes the query with respect to a collection of documents and generates a list of results. On the right hand side of the diagram a group of relevance assessors takes the specified information need and assesses whether documents from the collection are relevant to the topic or not. These judgments are then used by an *evaluation package* to evaluate the quality of the ranked results returned by the IRS and to generate performance measurements by which this IRS can be compared with others. For such comparisons to be meaningful, a large number of topics (usually 50 or more) must be used to average out topic-specific variations.

Judging Issues. If the document collection contains more than a few thousand documents, it is not feasible to judge each document in the collection. TREC addresses this issue by using a technique known as *pooling* in which the union of the sets of documents retrieved by a broad and diverse range of retrieval systems forms the pool of documents to be judged. Documents not in the pool are assumed to be irrelevant. Zobel [61] has shown that although the TREC collections do include unjudged relevant documents, these have a very small effect on system comparisons made using TREC.

Voorhees [57] has shown that although agreement between different assessors is far from perfect, that system comparisons are remarkably stable across judgment sets prepared by different assessors.

Measures. The measures used to compare systems are almost always variants of precision and recall. Looking at the documents retrieved at a particular point in the ranking, *precision* is the proportion of retrieved documents which are relevant and *recall* is the proportion of all relevant documents in the collection which have been retrieved. In Web search, searchers are typically more concerned with the precision of the results on the first one or two result pages than with recall. Consequently, *precision at n documents retrieved* or P@*n*, where *n* is typically 10 or 20, is a useful measure.

It is usual in TREC to plot precision against recall to give a full picture of the performance characteristics of the retrieval system. Example precision-recall curves are shown in Figure 6. In TREC, systems are often compared using the single number

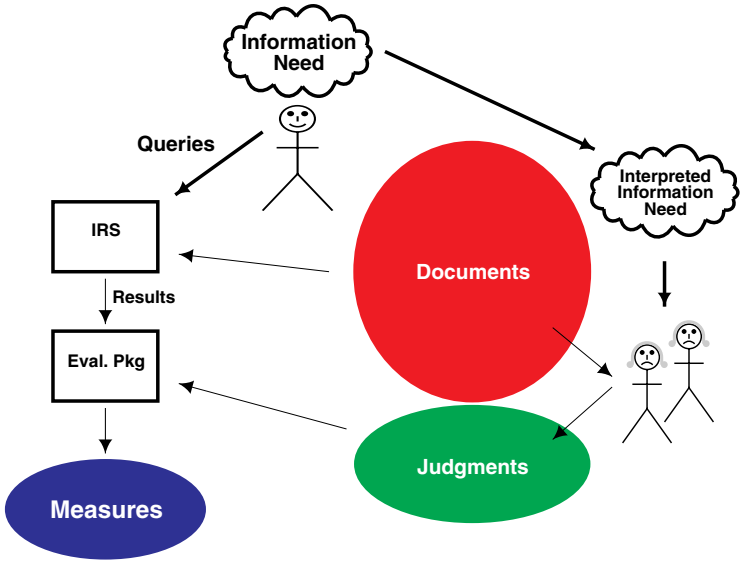

Fig. 5. The TREC retrieval evaluation paradigm.

measure *mean average precision* which takes into account aspects of both precision and recall. *Average precision* corresponds to the area under the precision recall curve and can be computed by summing the precisions at each point in the ranking where a relevant document was retrieved[3] and dividing by the number of known relevant documents for the topic. A mean is then taken of the average precisions recorded across a large number of topics[4].

1.3 Multi-media Retrieval

Ideally, a retrieval system would not be restricted to the text domain and would be able to retrieve documents containing information in the form of images, sounds, video, music and perhaps even tastes and smells. Many fascinating issues arise in non-textual retrieval, such as how to express queries and how to match queries against documents.

However, retrieval in the textual domain is quite fascinating in itself and more than sufficient to fill five lectures! Please note that documents in non-textual media may often be retrieved effectively by applying text retrieval techniques to captions, transcripts, catalogue entries, metadata records and in other descriptive information. For example, a GIF or JPEG image referenced in a Web document may include useful descriptive information in the name of the file, and in the alternate-text field which many Web authors provide for the benefit of blind people or people with non-graphical browsers. For example:

[3] Note that recall only changes when a relevant document is retrieved.
[4] Please note that other authors sometimes use different definitions of the term average precision.

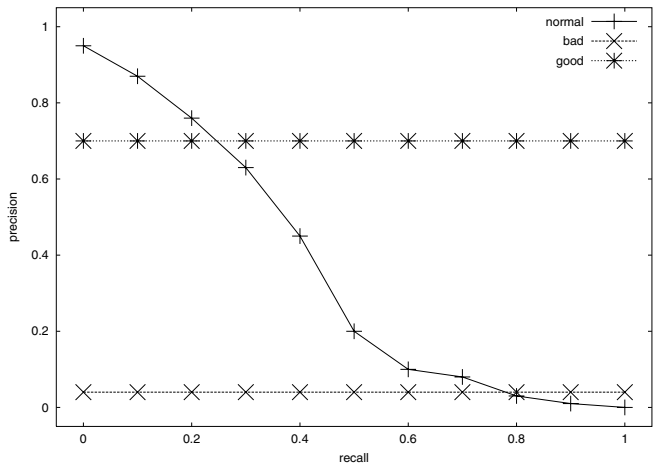

Fig. 6. Precision-recall curves. The horizontal line at the top shows the performance of an unrealistically good retrieval system and the horizontal line at the bottom shows the performance of a very poor system. The third line is more typical of real ad hoc retrieval, showing that discrimination between relevant and irrelevant is initially very good but falls with increasing recall, becoming almost random at very high recall levels.

'`<img src="keystaff.gif" alt="Key Asian studies staff at the ANU">`'. Some search engines provide "image search" services based on this type of information.

Some also allow retrieval of multi-media web pages on the basis of the *anchor text* of hyper links which refer to them. The following example shows an HTML link whose target is a JPEG file and whose anchor text is "The Mayor welcomes students to Chios." The anchor text is highlighed when displayed by a Web browser and you click on it to make the browser display the target.

```
<a href="x.y.z/Welcome-to-Chios.jpg">The Mayor welcomes students
to Chios.</a>
```

1.4 Cross-Language Retrieval

Given the very wide range of linguistic backgrounds represented at the Summer School, I am very sorry that only a tiny part of this chapter can relate to the topic of cross-language or multi-lingual retrieval. Cross-language retrieval means that queries phrased in one language may retrieve documents written in another.

In the past, the TREC conference has included special interest tracks on Chinese and Spanish retrieval and also spawned a cross-language track involving English, French, German and Italian, which has now gained its independence as the European based CLEF initiative [8].

There are many challenging issues in cross-language retrieval and also in retrieval of documents in the searcher's native language from within a multi-lingual collection. These problems are rapidly becoming more important as the once-supreme dominance of English as the language of the Web is eroded.

The lowest level problem is that of the character set. The ASCII character-set is inadequate for even European languages. The ISO 8859 series of standards extends 8-bit character sets to permit the representation of European accented letters and additional letters. However, 8 bits are insufficient to accommodate the additional characters needed in languages such as Arabic, Thai, Japanese and Chinese. Unicode standards [54] encompass 16 and 32 bit character formats to address this problem, however, Unicode has been by no means universally adopted. My understanding is that most Japanese electronic text is actually encoded in EUC, JIS, or Shift-JIS formats.

Another problem is cross language polysemy. The word *sale* means "reduced-price selling" in English, "dirty" in French and, I think, "salt" in Italian. Even a sequence of words may have meaning in multiple languages. For example, *la chair sale* might mean "dirty flesh" in French whereas *LA chair sale* could refer to a discount furniture sale in Los Angeles.

Text retrieval systems operating in a multi-lingual environment must recognise the use of different character sets and detect the language being used. To complicate matters, more than one character set and more than one language may be used in the same document [31]. Systems performing cross-language retrieval need to incorporate translation facilities for queries.

1.5 How Do Text Retrieval Systems Work?

Text retrieval systems based solely on statistical analysis of patterns of term occurrences within documents consistently perform well on TREC ad hoc tasks. A *term* is the basic indexable unit, such as a word, word-stem or phrase, from which queries and documents are constructed. For retrieval purposes, both documents and queries can be considered to be sequences of terms. In what follows, *term* can usually be interpreted as *word*.

Surprisingly, on TREC ad hoc tasks, systems using natural language processing (NLP) techniques such as word-sense disambiguation and part of speech tagging have not managed to outperform the best statistical systems[5].

Text Retrieval Models. Over the years, a number of information retrieval models have been proposed to estimate document relevance based on the statistics of term occurrences. The most prominent are the Vector Space Model, exemplified in the SMART retrieval system from Cornell University [47] and the Probabilistic Models , exemplified in the Okapi retrieval system [45] from City University, London and the Inquery system [1] from the University of Massachussets.

In practice, when implemented, there is relatively little difference between these models. All are based on the following simple heuristics:

1. The more occurrences of a query term in a document, the more likely it is that the document is relevant.

[5] However, NLP processing has come into its own in the TREC question-answering track. [56]

2. A long document containing the same number of occurrences of a query term as a short one is less likely to be relevant.
3. The more documents in the collection which contain a query term, the less weight should be attached to it in determining relevance.

The Okapi BM25 weighting function [45] is a very well known mathematical formulation of these heuristics:

$$w_t = q_t \times tf_d \times \frac{\log(\frac{N-n+0.5}{n+0.5})}{2 \times (0.25 + 0.75 \times \frac{dl}{avdl}) + tf_d} \quad . \tag{1}$$

where w_t is the relevance weight assigned to a document due to query term t, q_t is the weight attached to the term by the query, tf_d is the number of times t occurs in the document, N is the total number of documents, n is the number of documents containing at least one occurrence of t, dl is the length of the document and $avdl$ is the average document length.

Retrieval models based on lexical proximity of term occurrences have been proposed ([26] [7] [14] but have not been widely adopted. They arise from an additional heuristic:

4 Occurrences of multiple query words within close lexical proximity are more significant than isolated occurrences.

```
1. Foreach document
     Set document score to zero.
2. Foreach query term
     Foreach document containing the query term
       Compute the relevance contribution.
       Add the contribution to this document's score.
3. Sort documents into descending order of score.
```

Fig. 7. The basic IR ranking algorithm.

A Simple Ranking Algorithm. Figure 7 shows a very simple algorithm for producing a ranked list of documents using a relevance formula like Okapi BM25. In order to actually implement it, there are a number of lexical issues to resolve:

1. Should *stopwords* such as `of`, `the` and `and` be considered as words?
2. Should words be represented in the form in which they appear, or should they be *stemmed*? Stemming means that different forms of the same word are represented as a common stem or root. For example `run`, `running`, `ran`, `runs`, `runner` and so on might all be represented as `run`.

3. Should letters be *case folded*? i.e. should upper case letters be converted to lower case, so that `The` and `the` are treated as the same?
4. What exactly should constitute a term? Is 2001 a word? What about B52 or anti-social?
5. Are there areas of text which should be excluded from consideration? For example, HTML comments and tags?

Stemming and case folding generally increase recall and may sometimes improve precision. However, they can also dramatically reduce precision. For example, the query word `Hawking` would be stemmed and case-folded to `hawk` and is likely to match other English family names such as `Hawke`, `Hawker`, `Hawkins` and `Hawkes` as well as ordinary words such as `hawk` (a bird) and `hawker` (a door-to-door salesperson). Similarly, the acronym `IT` would be case-folded so as to be indistinguishable from a common pronoun.

Retrieval systems differ in the way they handle these lexical issues.

Data Structures for Text Retrieval. A *full text scanning* implementation of the algorithm shown in Figure 7 can be written very quickly and easily in a language like `perl` provided that the document collection is not too large and there is no requirement for query processing speed. Full text scanning means that the full text of each document in the collection is scanned for query terms, once for each query term (inner loop in Step 2.)

However, query processing speed is usually very important and the collections discussed in this chapter are very large. The data structure most commonly used to speed up Step 2 of the algorithm is the *inverted file index*, diagrammed in Figure 8.

An inverted file speeds up processing by keeping lists of the documents in which each term occurs. These lists are called *postings lists* for reasons which will be explained in Section 1.5. In the inverted file shown in Figure 8, each posting in the postings list contains both a document number and the corresponding *tf* value (how many times the term occurred in the document) for use in the Okapi formula.

¿From the example we can see that the word `oboe` occurs three times in document 2, once in document 7, twice in document 11 and so on. The document table shown in the bottom right allows us to match up document numbers to real documents and also records information about the document such as length (for use in the Okapi formula), a checksum (CRC) of the content and a snippet (small sample) of text to be displayed when presenting query results.

Efficient lookup of the term dictionary is essential to achieve fast query processing. The term dictionary shown in the figure is sorted into lexicographic order to permit binary searching.

Building an Inverted File – The Old Way. The first step of the original method for building inverted files was to scan the text of the documents and to append a *posting* to the end of the postings file each time a term was encountered. Each posting consists of a $(document-id, term-id)$ pair. Understanding what postings are and how to generate the postings file is important to make sense of what follows. Readers are advised to work through the example documents in Figure 9 and be sure they understand.

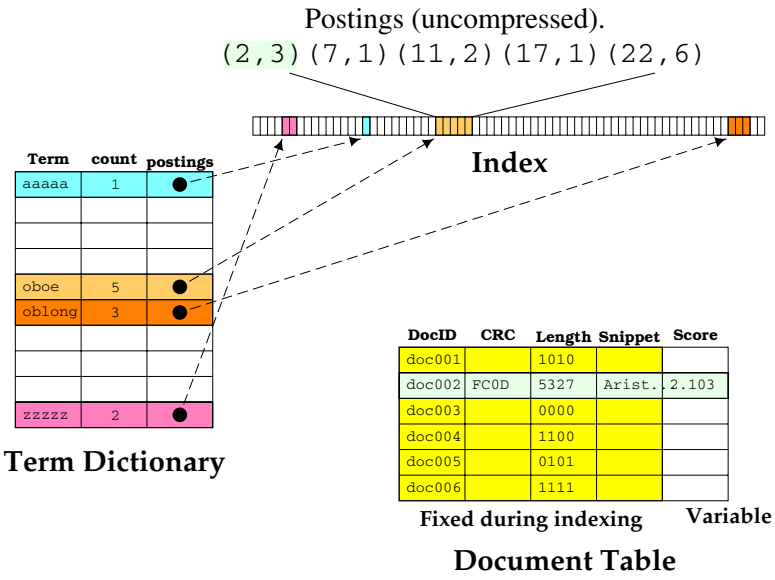

Fig. 8. Key IR data structures: Term dictionary, document table and inverted file index.

It should be obvious that the postings generated as described in the preceding paragraph must initially be emitted in document order. The postings file can subsequently be *inverted* by sorting the postings using $term - id$ as the primary key and $document - id$ as the secondary. Study the inverted file shown in Figure 9 to be sure you understand what it represents. The first posting relates to term 0 (`a`) which occurs only in document 2. This is the complete postings list for that term. Next there are two postings for term 1 (`ate`) and they appear in order of the documents in which they occur.

A post-processing step is needed to convert the sorted postings into the form shown in Figure 8. First, note that it is not necessary to record the term number in each posting – that information is implicitly recorded in the term dictionary. Second, note that whenever the same term occurs $tf > 1$ times in the same document, the sort described in the preceding paragraph will produce a consecutive sequence of tf identical postings. The post-processing step replaces every sequence of tf identical postings with a single (docid,tf) posting and fills in the offset field in the document table.

The final result is the document table and inverted file as shown in Figure 9.

Processing Queries Using an Inverted File. To confirm your understanding of inverted files, work through the query processing example in Figure 10. Don't bother to compute Okapi scores, just count how many query terms are present in each document. Look up each query term in the term dictionary and use the offset (entry in the third column) to find where the postings for the term start in the inverted file. Then add one to the score of each document mentioned in the postings for the term.

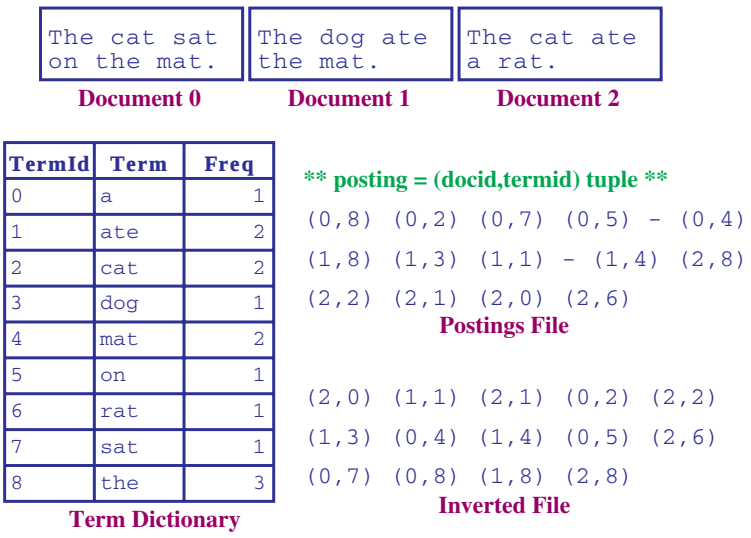

Fig. 9. Raw postings and inverted file index for a collection of three tiny documents.

More Sophisticated Inverted Files. The primitive form of inverted file shown in Figure 9 contains considerable redundancy due to the encoding (often repeated) of the $term-id$. This information doesn't need to be stored, as it may easily be deduced from the term dictionary. If phrases or proximity operators must be supported, it is usual to replace the $term-id$ with the position of the term occurrences within the document. On the other hand, if position information is not needed, multiple postings for the same $(document-id, term-id)$ pair can be replaced with a single $(document-id, tf)$ pair.

Section 4 describes more efficient ways of building an inverted file and also covers another type of posting which can reduce query processing time.

1.6 Relevance Feedback

When searching for documents relevant to a topic, the initial query generated by the searcher is usually far from optimal. A widely used technique for improving it is based on *relevance feedback*, which works as follows: The initial query is processed and documents returned high in the ranking are judged by the searcher. The retrieval system then attempts to identify terms which usefully discriminate between the known relevant documents and the rest of the collection. These new terms are added to the query with appropriate weights and the augmented query is run to produce the final result list.

Interestingly, the same process can also be made to work in the absence of human judgments by assuming that the top 10 or so documents are relevant whether they actually are or not. This technique is often called *pseudo relevance feedback* and, in TREC ad hoc tasks, it has been shown to significantly boost average retrieval effec-

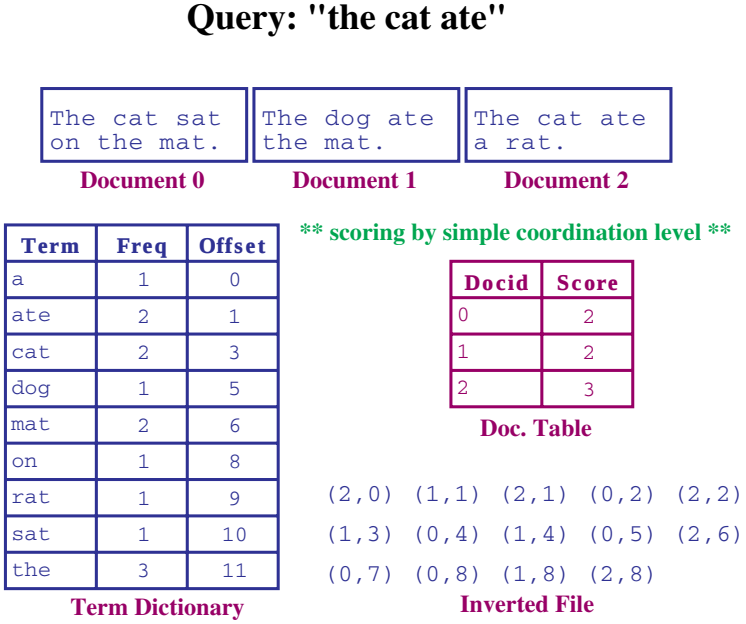

Fig. 10. A query processing example using the document collection and inverted file from the previous figure. For simplicity, relevance scores are simply a count of how many of the query terms were present in the document.

tiveness. Sometimes it causes harm but usually there is an improvement of some kind. The question for us, is whether it works on a very large question and whether it can be implemented efficiently.

1.7 Scaling Up over the Last Two Decades

The reason that it is important to consider very large scale information retrieval is that over the last 20 years there has been stupendous growth both in the scale of text document collections and in the cost-effectiveness of computing resources. At the time of the Summer School I calculated that the "bang-per-buck" ratio for computer CPUs had increased by a factor of about 200,000 or 5.3 orders of magnitude over that period. The comparable figures for random-access memory (RAM) and disk were 4.7 and 4.3 orders of magnitude respectively. Even more dramatic has been the growth in computer networks which were almost non-existent in 1980.

In 1980, IR researchers were still working with test collections comprising only a couple of megabytes, i.e. a few thousand documents. By comparison the VLC2 collection [25] first distributed in 1998 contains 100 gigabytes of data (18.5 million documents) and represents an increase of 4.7 orders of magnitude.

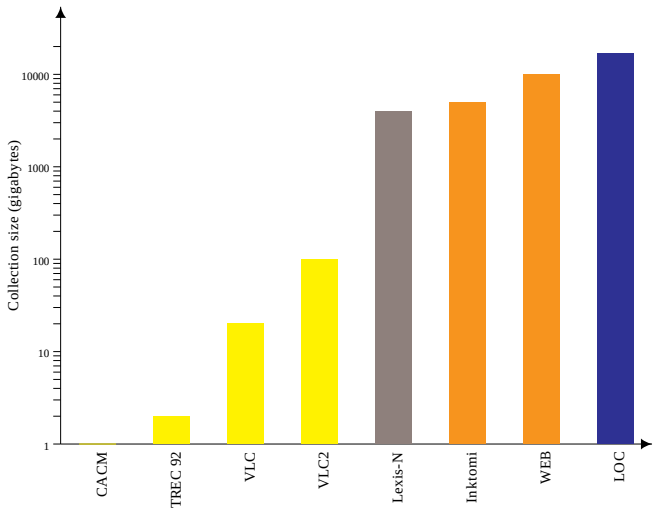

Fig. 11. Approximate sizes (in gigabytes. One gigabyte is approximately the amount of text in one thousand books) of various text collections. The barely discernable bar at the extreme left represents the collection of Communications of the ACM abstracts which was a commonly used test collection. At the far right, the 17 million volumes held by the U.S. Library of Congress represent slightly more data than indexed by Web search engines in 2000. Lexis-Nexis is a commercial document service.

At the time of the summer school, several public Web search engines were indexing of the order of 500 million pages or about 5 terabytes of text.

2 Introduction to the World Wide Web

The dramatic increase in importance of very large scale text retrieval has been almost entirely due to the advent and growth of the Web. In 2001 millions of ordinary people each day submit queries to be processed over "the entire Web"[6]. Web search is very large scale both in terms of the collection size and in terms of query volume. Engines like Alta Vista, Inktomi, FAST and Google are believed to handle loads in excess of one thousand queries per second.

Figure 12 shows a number of ways in which the Web differs from traditional electronic document collections. In some cases, the differences provide opportunities to improve retrieval effectiveness. In others, they represent additional hurdles to be overcome.

Figure 13 shows the components of a typical web search system. The indexer and query processor components may correspond quite closely to a traditional text retrieval

[6] In reality it makes little sense to talk of the entire Web, as the size of the Web is made boundless by the presence of automatic page generators.

Hyperlinks. Web pages are routinely hyperlinked. Sometimes this is to provide access from a stand-alone Web page to other Web resources but in other cases Web pages are merely components of a hyperdocument and the links are used to define its structure. Traditional documents sometimes contain links (e.g. citations from within a scientific article) but on the Web, these links can be followed instantaneously.

Dynamic nature. The document collection represented by the Web changes with time. Web pages are frequently created and destroyed and often change their content.

Viewpoint dependence. Discovery of Web pages is normally done by recursively following links from a set of starting points. Consequently, the set of Web pages which are visible depends upon the access rights of the observer, the set of start points and the range of page formats from which the observer can extract and follow links.

Multimedia. It is common for Web pages to include or to link to images, sound clips and videos.

Multiple languages. Very many languages are represented among the documents of the Web.

Multiple formats. Text documents published via the Web are seldom in plain text format. The majority use HTML (HyperText Markup Language) but many use PDF (Portable Document Format), PostScript, or Microsoft Word.

Duplicate documents. The Web provides mechanisms by which duplicate documents are systematically created. Due to hostname, directory name and filename aliasing, the same Web page may have many different URLs. (A URL, [55] or Uniform Resource Locator is the standard way of referencing a web page. For example, the URL `http://www.youruni.edu.au/physics/PH2040/index.html` might be equivalent to `http://youruni.edu.au/subjects/PH2040/index.html`. Another means by which duplicate documents are created is through the technique of *mirroring*. To save download cost and time and to reduce server load, the content of a popular Web site may be replicated or mirrored on other web servers around the world.

SPAM. The Web is heavily used for commercial purposes and many commercial operators attempt to increase traffic to their sites by fooling Web search engines. The most naive SPAM merely inserts large numbers of deceptive keywords which are invisible to the user. For example, inserting "Who wants to be a millionaire" many times into a pornographic Web page, but in white text on a white background.

Fig. 12. Features which distinguish the Web from conventional document collections.

system. In web search, the spider (sometimes called a crawler or robot) is necessary in order to identify the documents which form part of the "collection". The decrypter might be part of either kind of system, depending upon goals or requirements.

Spidering or crawling is the process of discovering Web pages to index by recursively following links from a set of seed pages. Spiders must be programmed to adhere to the following etiquette:

1. Respect the `robots.txt` protocol [36]. Web publishers may place a `robots.txt` in the root of their Web directory to specify which pages should and should not be accessed.
2. Ensure that individual Web servers are not overloaded. A typical rule of thumb is that requests to a particular server should not be sent more than once per second. However, Web servers vary greatly in their ability to handle load and their responsiveness to requests depends upon the amount of load from other sources.

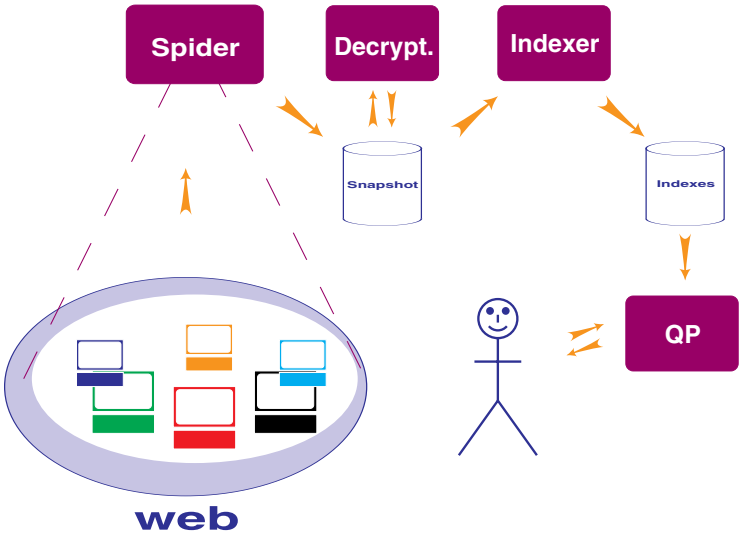

Fig. 13. The components of a simple web search engine. The spider discovers Web pages to index by recursively following links from a set of seed pages. The output of the spider is a kind of snapshot of the visible part of the Web. Note that the snapshot may take weeks or months to build up. Not all search engines include a decrypter, but if included its job is to extract indexable text from binary or compressed formats such as Microsoft Word and PDF. The indexer builds an inverted file index from the documents in the decrypted snapshot. Finally, the index is used by the query processor to process incoming queries.

Consequently some spiders vary the length of delay they insert between successive requests based on observed response times from the server.

3. Ensure that elements of the Internet infrastructure are not overloaded. Even if the spider shows appropriate politeness to each individual server, it can still overload a network link if it simultaneously accesses many servers in the same region of the network. Andrei Broder, Chief Scientist at Alta Vista reports that the Alta Vista spider is easily capable of soaking up the entire bandwidth of the network connection to countries as large as Spain.

Spiders implement various policy decisions about which types of web resource will be fetched. For example, one spider may decide to fetch HTML and plaintext pages only whereas another may also fetch XML and PDF pages as well as JPEG and GIF images. File types may be determined using MIME-type information supplied by the Web server or by the URL suffix (eg. `.htm`). Unfortunately, both sources of information are frequently inaccurate. Consequently, it is advisable to confirm the file type by looking at the head of the file.

Other Spidering Issues. Implementers of Web spiders face a range of major challenges caused by the pathological nature of large parts of the Web. Web servers are frequently guilty of supplying misleading or inaccurate information. Many Web site constructors deliberately or inadvertently set up spider traps. Others set up automatic scripts which generate infinite sequences of pages with trivially different content. Some Web authors include unprintable characters, spaces and newlines in the URLs of their Web pages.

Readers are referred to [29] for further discussion of spidering issues and information on how to build a spider.

3 Properties of Very Large Collections

The major impacts of very large collection size are on efficiency rather than effectiveness.

3.1 Collection Size and Speed/Efficiency

A larger collection obviously requires more disk space for the documents themselves and for associated data structures.

Vocabulary size A large English dictionary contains of the order of 100,000 entries. A naive person might assume that the vocabulary size for a collection would stop growing once this number had been reached. However, a profusion of typographical errors, acronyms, codes (such as message identifiers and car registration numbers), new words, headword variants, proper nouns and foreign words mean that, even after 100,000 different words have been found, the vocabulary size continues to grow at a rate of something like one new word per thousand words of additional text. Depending upon the definition of a word, the number of distinct indexable words in the VLC2 collection is something like ten million! In other words, 99% of distinct words in the collection are not dictionary headwords.

A very large vocabulary increases the time taken to look up a word both during indexing and while processing queries. It also increases the size of the term dictionary and consequently the demands on memory space.

Increased number of occurrences of common terms. As a document collection grows, the number of occurrences of common words is likely to increase in proportion. This means that postings lists for common terms will be longer, increasing processing time during both indexing and query processing. The inverted file also grows in proportion to the size of the collection. File size limits imposed by the operating system may be exceeded, increasing implementation complexity.

Increased number of documents. An increase in the number of documents in the collection results in an increase in the size of the document table. If the Okapi BM25 scoring function shown in Equation 1 were used and the document table were represented as shown in Figure 8 a serious memory residency issue might arise from pattern of accesses to the document length information.

Many more matches for a query. A larger collection is likely to result in proportionately more documents containing each of the query terms. This raises memory residency issues when recording document scores and may non-linearly increase the cost of the final sort.

3.2 Effectiveness and Collection Size

It is fairly intuitive that a very narrowly specified query is more likely to find an answer within a large collection than within a much smaller one. This would obviously be true if the small collection were a subset of the big one. In general, when looking for a particular document, that document is more likely to be a member of a large collection than a small one.

When the query is broad enough that there are many answers within a small collection, would you expect retrieval effectiveness to be greater within a small collection or within a superset of it? You might think that retrieval would be easier because there are more right answers. Alternatively, you might expect it to be harder because there are also an increased number of documents which share features with the relevant documents but which are not actually relevant.

Signal detection theory [52] predicts that precision at fixed cutoff (e.g. precision at *n* documents retrieved) will be lower in a sample collection. It predicts that there will be a smaller number of documents in the high-scoring range where the difference between the signal distribution and the noise distribution, and consequently the probability of relevance, is greatest. These predictions have been borne out empirically in the TREC Very Large Collection track, where all participants observed a decline in precision at 20 documents retrieved when processing a set of queries over a 10% sample of the 20 gigabyte VLC collection[28]. See Table 2.

Table 2.

Group	Baseline	VLC	Ratio
City	0.320	0.515	1.61
ATT	0.348	0.530	1.52
ACSys	0.356	0.509	1.43
UMass	0.387	0.505	1.31
IBMg	0.275	0.361	1.31
Waterloo	0.498	0.643	1.29
IBMs	0.271	0.348	1.28

The expected increase in early precision when querying a very large collection of documents could form the basis of an optimisation technique in which only part of a large collection were actually processed. This might achieve acceptable effectiveness for a large proportion of queries but would seriously harm others. It is unclear whether this optimisation is used in practical Web search.

3.3 Exercise 1 – Characterising Search Engines

Take a comparative look at three or four of the following search engines:
www.metacrawler.com, www.google.com, www.euroseek.com, www.altavista.com

www.thunderstone.com, www.fast.com, www.teoma.com, www.northernlight.com www.hotbot.com, www.LookSmart.com, www.go.com.

Try to answer the following questions:

```
1. How good is the result presentation:
 - How many answers are displayed on first screen?
 - How good are the displayed summaries?
 - How easy is it to find help?

2. Does the engine use stemming?
3. Does the engine eliminate stopwords?
4. Is the engine case sensitive?
5. Does the engine support phrases?
6. Does the engine assume term conjunction? (AND)
```

For the following queries:

```
1. Chios
2. ELSNet Summer Courses 2000
3. Aareschlucht
4. who is the current Greek prime minister?
5. the The
6. "to be or not to be"
7. "David Hawking"
```

look at the result lists and determine the rank of the first useful answer. (Give up after ten results.)

This is not a very good evaluation experiment because assessment is not blind, there aren't enough test queries and the measure employed may not be sufficiently stable. In Section 7 more rigorous evaluations are presented.

4 Efficiency Techniques

Efficiency differs from *speed* or *throughput* in that it is expressed relative to the resources employed. It is an imprecise measure of the amount of work achieved by a retrieval system, using a given amount of hardware. Efficiency is increased if queries are processed or text is indexed faster, without upgrading the hardware. Alternatively, efficiency has increased if the same throughput is achieved by a smaller machine configuration.

Two classes of technique are used to improve the efficiency of a retrieval system. Techniques of the first kind are *lossy* in that they may materially affect the quality of results obtained by taking shortcuts in the query evaluation or indexing process. Computational optimisations and engineering improvements which increase the speed of indexing or query processing without changing the results make up the second class.

This section proposes some general efficiency advice and then covers efficiency aspects of each of the spider, indexer and query processor components of the Web search system diagrammed in Figure 13 on page 122.

To give an idea of the relative time required for each of the processes, the intranet search engine at the Australian National University takes about two days to spider the whole site, a few hours to decrypt non-HTML documents, and about an hour to index the snapshot. It processes typical queries in a fraction of a second.

4.1 General Advice

It is important to choose efficient algorithms and data structures. For example, an $O(n^2)$ sorting algorithm applied to a list of one million search results may require 50,000 times as many comparisons as an $O(n \log n)$ one.

It is also crucially important to implement algorithms and data structures in a way which makes minimises the number of accesses to slower levels of memory. In certain circumstances, it may be advantageous to use an algorithm with a slower theoretical running time in order to make better use of faster memory.

To illustrate how enormous are the speed differences between different levels of memory, consider a 1.5 gHz Intel Pentium IV CPU with 512 megabytes of RAM and a 7200 r.p.m disk. A disk like this has an average rotational latency of 4.2 ms (milliseconds) and a typical seek latency of 5 ms. Consequently, when a disk read request is issued which cannot be satisfied from buffers or caches, a delay of about 9ms ensues. During this period of time, something like 180 megabytes could be transferred from (RAMBUS) RAM to the CPU and the CPU could execute about 13.5 million instructions from its on-chip cache. It is clearly of crucial importance to ensure high cache hit rates and to minimize disk accesses.

Most modern operating systems run programs in *virtual memory*. In other words, program code and data structures are assigned to addresses in an imaginary address space without regard to the limited size of primary memory (RAM) and the need to share it with other programs or processes. The operating system divides the virtual address space into *pages* (often about 4 kilobytes in size). At a particular point in the execution of a program, some of the pages will be represented in primary memory, others will be represented only on disk and some may not yet have been created. As execution proceeds, reference may be made to an address in a page which is not resident in primary memory, causing a *page fault*. When this happens some pages in primary memory may be written out to disk and replaced by others from the disk which are known or predicted to be needed by the computation.

Virtual memory operates efficiently provided that page faults occur infrequently. It can degenerate into extreme inefficiency (known as *page thrashing*)if this is not the case. During page thrashing the retrieval process is forced to operate at disk speed rather than primary memory or CPU speed. Frequent page faults will occur when the pattern of memory references is not localised. In indexing or query processing this could occur if random accesses were made into a file or data structure which is larger than the available primary memory.

An example of where a data structure re-organisation could improve memory reference locality is the document table in Figure 8 on 117. Consider the memory access pattern caused by processing queries using the simple algorithm shown in Figure 7. In Step 1, the score field only of every row in the table is accessed sequentially. In Step 2, each successive query term is associated with a sweep through the table which accesses

the document length of each document which contains the term and updates the score. Step 3 accesses all the scores.

It is not until the results are prepared for display to the searcher that the `docid`, `crc` and `snippet` fields are accessed at all. Furthermore, during result display only a small number of the rows in the table are accessed.

The presence of `docid`, `crc` and `snippet` in amongst the score and length information reduces the locality of memory references in Steps 1, 2 and 3. It increases the number of virtual memory pages which must be loaded to perform these steps. Reference locality could be significantly improved by splitting the table such that scores and lengths were in one table and `docid`s, `crc`s and `snippet`s in another.

4.2 Compression

Compressing data structures is another way to improve locality of reference, during spidering, index building and query processing. For example, rather than representing a document length as a 32-bit integer, it could be represented in a smaller number of bits. Compression of URLs during spidering is discussed in [29]. Very effective methods exist for compressing lists of postings and you are referred to *Managing Gigabytes* [60] for a detailed treatment.

In addition to improving memory reference locality, compression may significantly reduce the amount of disk space required to store the raw text and the various index files. It also reduces I/O transfer times from disk at the expense of additional CPU time to decompress postings.

4.3 Spidering

Section 2 explained the basic operation of a spider and outlined the politeness constraints under which spiders should operate.

Network Costs. A major motivation for efficiency in spidering is the cost of network traffic. If a billion pages, averaging 12.5 kbytes each, are spidered from Australia, where network traffic charges are of the order of 80 euros per gigabyte, the total cost will be one million euros!

Network traffic can be reduced by ensuring that excessively large files are truncated or not fetched at all and that binary files are detected and truncated.

Further reductions in cost can be achieved by detecting infrequently updated or infrequently accessed parts of the Web and spidering them less frequently.

Incremental Spidering. A basic spider fetches every page it encounters. An incremental spider tries to fetch only those pages which have changed since they were last fetched. Potentially, a great deal of network traffic can be eliminated by this means but the technique only works if Web servers supply accurate information such as last modified date, size, or checksum.

Another issue to deal with is how to detect pages in the snapshot which have been removed from the Web.

Multi-threading. It is not feasible for a spider to scan the entire Web if, due to requirements of etiquette, it accesses only one page per second. At that rate, at most 86,400 pages can be fetched per day and it would take more than 31 years to collect a billion.

An obvious solution is multi-threading. A hashing function can be used to assign each distinct Web server to a particular parallel thread. Each thread inserts the appropriate politeness delay between successive requests, and each can operate independently of the others without risking etiquette violations. Large scale spiders may make use of thousands of parallel threads, possibly spread across multiple systems.

URL Storage. A spider must maintain two lists of URLs: a) a *frontier* of URLs still to be fetched, and b) a *cache* of URLs already encountered. In simplest form, the frontier can be a straight-forward queue but it may be priority-ordered to enable the most useful pages to be fetched first [32]. To save memory, it can reference URLs in the cache rather than repeating the strings.

The cache must be capable of very rapid lookup and insertion. Every URL encountered in every page scanned must be looked up in the cache. If found, no action is required. Otherwise, a new entry must be made in both the cache and the frontier. When a URL from the frontier is selected for fetching, it is removed from the frontier.

In a multi-threaded spiderer, there should be a frontier for each thread to avoid the need for scanning to find the next URL to be processed by a thread. The cache may also be divided across threads.

The amount of memory required to store all the URLs in the cache is potentially huge. If there are a billion URLs and the average length of a URL is 50 characters, the amount of space required in a naive implementation exceeds 50 gigabytes! This is too large to fit in memory and careful organisation is needed to ensure that most lookups can be satisfied with few or no disk accesses. Compression techniques can be used to reduce the storage required for URLs.

Detection of Duplicate Pages and Mirror Sites. The Web provides two ways by which duplicate content or near-duplicate content can be created. The first is aliasing of hostnames, directory names and files in which there multiple URLs refer to exactly the same page on the same machine. The second is mirroring, where a popular Web site is replicated on other hosts to improve responsiveness and cut network traffic costs. The content of pages on a mirror site may be slightly different to those on the original due to the addition of a site label or date or to version differences.

Exact duplicates can be detected with very small error rate using checksums, but checksums must be efficiently computed and another efficient lookup structure with up to a billion entries must be created. Detection of mirror sites (and deciding what to do about them) is less straight forward and the reader is referred to [4] for details.

4.4 Indexing

Some systems impose a limit, say 64 kilobytes, on how much of a document they will index. Words occurring after that limit will not be indexed. This reduces not only index size but also indexing time and eventually the processing time for some queries. Some

important information will be lost, but often there is enough information in the head of a document to accurately characterise it.

Inverted File Postprocessing. Considerable computational savings can be effected if the postings in an inverted file index contain relevance-contribution information rather than raw term frequencies. To understand this, consider the Okapi BM25 formula in Equation 1 and notice that the only query dependent variable is q_t. For every possible (term, document) pair, the values of all other variables are known once the indexer has finished scanning the collection. Either during indexing or, more simply, in a postprocessing step, the tf_d values in the inverted file (as in e.g. 8) can be replaced by the values obtained by pre-evaluating the bulk of Equation 1. These values would normally be computed as floating point numbers but, if desired, they can be quantised and represented in a more space-efficient way with a small cost in accuracy.

The benefit at query time of pre-computed relevance contributions is considerable. Not only is the number of arithmetic operations, including a logarithm, reduced, but the need to randomly access the table of document lengths is averted. If physical memeory is small, the effect of the latter may be dramatic.

Index Pruning. Having pre-computed relevance contributions for each (term, document) pair as described in the immediately preceding section, it is possible to sort the postings for a term into order of decreasing contribution and to truncate the tail of the postings list at the point where the contribution becomes so small to be unlikely to significantly affect the final ranking. The truncation condition can be tuned to achieve the desired balance between speed and effectiveness.

This is a lossy technique because information is being discarded. There may be rare cases where effectiveness is harmed, but there is evidence [2, 33] that usually it is not.

More Efficient Index Building. In Web search, fast query processing is much more important than fast indexing because hundreds of millions of queries may be processed in the interval between successive index builds.

However, use of efficient indexing algorithms and appropriate data structures is worthwhile: a) to increase the amount of text which can be indexed on a given hardware configuration, and b) to allow rapid response to changes in the collection.

The traditional method for building inverted files was described in Section 1.5 and Figure 9. The major flaw of this algorithm is the potentially very time consuming and disk-space intensive external (i.e. disk based) sort of the postings.

Moffat et al [40] have proposed various efficient schemes for sorting postings including methods which require no additional disk space. However, the following method avoids sorting altogether and is quite fast in practice. Similar ideas are presented in [15, chapter 3].

The basic idea is that multiple passes are made over the text collection. The first pass does not write postings but merely builds up a term dictionary including occurrence counts for each term. Subsequent passes are responsible for re-scanning the input and writing the inverted file.

At the end of the first pass, it is possible to compute the size of the inverted file and the offset within the file of the postings list for each term. After this has been done a file of the necessary size is created.

If disk space is not excessively tight, the first pass can also write a tokenised version of the input, to avoid the relatively expensive lexical scanning of the raw text.

For convenience and efficiency during the output passes, the inverted file, or part of it, is memory mapped using the virtual memory capabilities available in most modern operating systems[7]. Once the file is mapped, it can be treated as an array and accessed using normal array subscripting.

As each virtual memory page of the inverted file is accessed, it will be read into memory. Depending upon how much memory is available, this may result in a less recently accessed page being written out to disk and removed from memory.

If sufficient primary memory is available to accommodate the entire inverted file, only one additional pass is needed because there will be no unnecessary virtual memory activity. The tokenised input is rescanned and a posting for each indexable term encountered is written in the appropriate spot in the postings file. Then a pointer associated with this term's entry in the term dictionary is incremented to indicate where the next posting for this term should be placed.

As you can probably see, the pattern of accesses to the inverted file is highly random. If the inverted file is significantly larger than the available primary memory, there will be a high probability that each access will generate a page fault. This would cause the speed of the process to drop from memory speed to disk speed, possibly causing indexing time to grow from hours to days or weeks!

A solution presented in [22] is to divide the inverted file into a number of equal sized *windows* where each window is approximately the size of available physical memory, and to write each window in a separate pass through the tokenised text. During each pass the entired tokenised file is read but term references corresponding to postings lying outside the currently memory mapped window are ignored. Consequently, accesses to the inverted file are restricted to the memory-resident window and speed is restored. A large amount of additional disk i/o is generated by the need to repeatedly rescan the tokenised input but: a) sequential access to the disk is far more efficient than random access, and b) the tokenised form of the input can be a lot smaller than the original text.

Compression of postings can also have a highly beneficial effect on writing of the inverted file by significantly reducing the number of output passes required.

Efficient Lexical Scanning. During the first indexing pass, attention to a number of engineering issues can make a large difference to the amount of time required for the first pass and also beneficially affect subsequent passes and query processing.

Choosing a data structure for the term dictionary which supports rapid insertions as well as rapid lookups is essential. The best choices are probably a hash table or a trie [35]. For a large collection, the memory space occupied by the term dictionary will be considerable. It must be kept memory resident because accesses will be random. A hash table should be designed to minimise the frequency and cost of collisions.

[7] e.g. the `mmap()` call in Unix or Linux.

Stemming can be applied either during indexing or at query processing time. In my opinion, it is not a good idea to stem words during indexing, because stemming actually discards information which could be useful during query processing. However, stemming does reduce the size of the term dictionary and many retrieval systems create stemmed indexes. If stemming is performed during indexing, a great deal of time may be saved by using a second trie or hash table to translate words to their corresponding stems rather than calling a stemming function. For example, the public domain Porter stemming function [15] takes 17 microseconds per call on a Sun Ultra-1 machine. In a 100 gigabyte collection, approximately 7 billion word occurrences need to be stemmed, adding about 33 hours to (i.e. more or less doubling) the indexing time.

The actual lexical scanning code needs careful attention. It should be implementable as a finite state machine [17, 30] (coded by hand or using a lexical scanner generator like `flex`). Its running time should be linear with the length of the input text. It is important to design the finite state machine in such a way as to reject parts of the text which do not need to be indexed. Doing so will reduce the size of the term dictionary, the tokenised input file and the inverted file.

Examples of parts of documents which should normally be rejected include random message-identifiers, HTML or XML tags, HTTP headers, binary data or text written in languages the system is not designed to accept. If non-textual data is not rejected, accidental sequences of letters will be recognized as words, increasing data structure sizes and possibly reducing precision for certain queries.

4.5 Query Processing

One of the best ways to speed up query processing is to avoid processing the query at all. Many current search engines do this by caching the results of queries, sometimes on a machine dedicated to the task. However, although some queries are repeated very often a large percentage are submitted only once [49]. Consequently, it is necessary to optimize the query processing machinery.

Early Termination of Query Processing. Another lossy optimisation technique involves processing the query terms in order of decreasing importance, until some stopping condition is satisfied. The importance of a query term must take into account both the weight assigned to it by the query (through repetition or explicit weight setting) and its discrimination power within the collection. The latter could be estimated by the highest relevance score contribution found in its postings list or more simply by its inverse document frequency (i.e. the reciprocal of the number of documents in which it occurs.)..

The stopping condition could be expressed in terms of a fixed number of terms to process, a fixed time limit (CPU or elapsed), or a more sophisticated dynamic termination condition.

Whether or not the index has been pruned as described in Section 4.4, processing of postings in a contribution-sorted postings list can be terminated early, when it is determined that subsequent postings are unlikely to have any effect.

Optimisation of Document Scoring. In the document table shown in Figure 8 there is a score accumulator for every document in the collection. An alternative scheme is to limit the number of document score accumulators to some arbitrary number, thus reducing: a) memory usage, b) time taken to zero scores, and c) time taken to sort results. On the other side of the ledger, an additional computation is required to locate the accumulator assigned to a particular document. This can be done using a hash table.

Used in conjunction with both query term re-ordering and ordering of term postings by relevance score contribution, this scheme can save time with negligible harm to effectiveness.

```
1. Foreach document accumulator
     Set accumulator to zero.
2. Sort query terms into order of decreasing importance
3. Foreach query term
     Foreach posting for the query term
       a. find the accumulator allocated to the document
       b. if no accumulator has been allocated, try to allocate a new one
       c. if an accumulator is now assigned, add the relevance contribution
       d. Exit inner loop if next posting may be neglected
     Exit outer loop if the stopping condition is satisfied.
4. Sort document accumulators into descending order of score.
```

Fig. 14. An optimised IR ranking algorithm. It is assumed that postings contain pre-computed relevance contributions and that postings lists have been sorted in decreasing order of contribution.

A more efficient query processing algorithm based on pre-computed relevance contributions, sorted postings lists and limited score accumulators is shown in Figure 14. Note that because of the ordering of query terms and the ordering of postings for a query term, document accumulators are allocated preferentially to the best terms and to the best documents for those terms. Once the limit on the number of accumulators is reached, postings referencing documents which have no accumulator assigned are simply ignored.

Efficient Ranking. Once document scores have been calculated in response to a query, the task of ranking involves sorting all the non-zero document accumulators and keeping track of the associated documents.

A very widely used sorting algorithm is quicksort [35] whose average running time is $O(n\log(n))$, but $O(n^2)$ in the worst case. If there are a million numbers to sort, $n^2 = 10^{12}$, a factor of 5×10^4 slower than $n\log(n) = 2 \times 10^7$. Unix `qsort()` exhibits worst case behaviour when values are equal, which could happen with certain relevance scoring functions or when relevance scores are quantised to a small number of distinct values.

Three options are available to avoid this problem:

1. Avoid the need for a final sort by maintaining a *heap* [60, 35] of top scores during score update.
2. Introduce a secondary sorting key, such as document number, to ensure that the comparison function used by `qsort()` never signals equality.
3. Use *radix sort* [35] rather than quicksort. Radix sort can potentially sort a million numbers in less than two passes, but requires that the numbers to be sorted be quantised into a small number of distinct values (typically between 1000 and 1,000,000).

Efficient Result Presentation. Once a ranking has been generated internally, it must be presented to the searcher in a useful format. This typically involves looking up the document table to find the document identifier and snippet (or canned summary). Accesses into the document table are of course random and, in the worst case, presenting n results could require n disk accesses, another case where memory residency is important and compression could help.

Better quality result listings replace the snippet or canned summary with a *query-biased* summary [53] showing the context of query words within the document. It is possible to generate query biased summaries on the fly, opening files and running a summariser program for each document. However, under heavy query load, this would be unacceptably slow and it would be more efficient to make use of the tokenised input file created during indexing to identify the context of query term occurrences.

4.6 Processing Phrases and Proximity Operators

If the query language supports proximity operators such as `near` or `followed by`, for example `ELSNet near summer`, postings in the inverted file must specify the word position within the document where the term occurred. One way is to leave the primary inverted file in the format shown in Figure 8 and to use an additional file to record the term positions. For efficiency, this file can include skip records which allow positions which cannot possibly form part of the desired proximity relation to be skipped over rather than examined one by one.

Computing the proximity relation for two terms requires intersecting the two postings lists. This can be done efficiently by examining the positions for the term with the lowest *df* occurrence by occurrence and then skipping through the position list for the second term to find the occurrences which form part of the desired relation. Note that the position lists can be compressed even though the skip records consume some additional information.

In this model, phrases can be computed as `followed by` relations with a one-word proximity limit. Common two-word phrases can be processed even more efficiently by building a phrase cache. The first time a phrase is encountered it is entered into a phrase dictionary associated with the cache, possibly replacing a phrase which hasn't been accessed recently. Then, a postings list created by evaluating the phrase using the proximity operation, is associated with the new entry. Subsequent references to the same phrase will be served from the cache.

A further alternative for phrases is to record the term identifier for the following term with each term position in the positions file, as proposed by Williams [59].

4.7 Relevance Feedback

Pseudo relevance feedback has proven quite effective in the context of TREC ad hoc retrieval. However, very few large scale text retrieval systems implement it. This is probably because of the computational expense entailed in doing so. Moreover, recent evidence [50] suggests that less benefit may arise from relevance feedback in a Web context.

Assuming that relevance feedback is to be used, the Vector Space model of retrieval allows for cheaper relevance feedback, using the Rocchio [46] than does Okapi. In the Okapi model of relevance feedback it may be necessary to return to the raw text of the top ranked documents, build term tables for those documents and to thereby identify terms whose occurrence densities in the relevant text is higher than for the text as a whole. The Robertson term selection value [44] is used to pick the best terms to add to the query.

5 Use of Parallelism in IR

Parallel computing hardware has been used extensively to increase the data handling and/or query handling capacity of text retrieval systems.

5.1 Types of Parallelism

Stanfill and colleagues [51] and Reddaway [43] have described the use of *SIMD* (Single Instruction Multiple Data, or data parallel) machines in text retrieval applications. However, these machines are no longer common.

A number of early search engines made use of *SMP* (Symmetric Multi-Processing) machines such as up-market DEC (later Compaq) Alpha machines. In these machines, a number of processors share a single large memory. However, systems of this type are quite expensive.

In the last few years, the *MIMD* (Multiple Instruction Multiple Data) model of parallelism, implemented as a cluster of PCs (COP), has become the dominant search engine architecture. Inktomi, FAST and Google are all understood to use it. Figure 15 shows a typical arrangement. Usually, each node in an n node cluster is responsible for $1/n$ of the collection. This is called document-id partitioning. [39] Each query is broadcast to all nodes in the cluster and each of them processes the query over the index for the piece of the collection for which they are responsible. The nodes may need to communicate with each other to exchange global statistical information such as *df* values. They definitely need to communicate with each other to form a merged ranking of the top t documents.

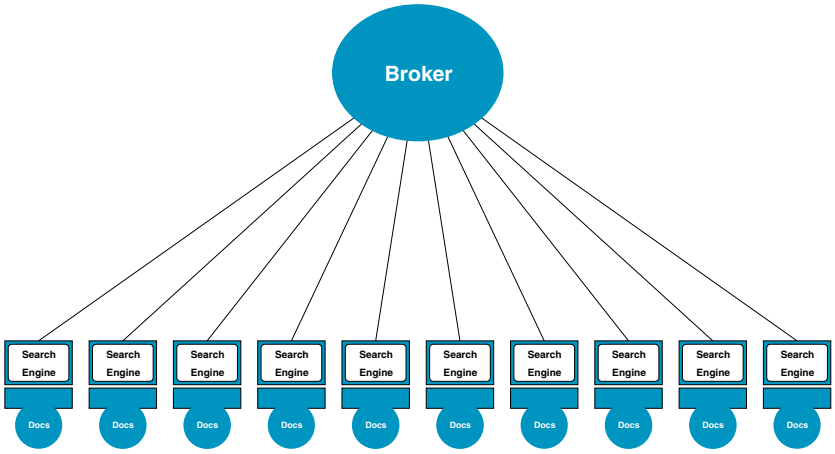

Fig. 15. The cluster of PCs (COP) model.

5.2 Parallel Efficiency and Scalability

The *parallel efficiency* of a retrieval system on a COP can be computed as $\frac{t_{one-node} \times 100}{t_{n-nodes} \times n}\%$. An ideal system in which there is no penalty for a parallel as opposed to a sequential algorithm will show a parallel efficiency of 100%. The parallel efficiency of a real system will fall short of perfection due to three possible causes: a) sequential parts of the algorithm, b) communication overhead, and c) load imbalance.

It is easy to obtain high parallel efficiency for text search on a COP provided that: a) there is sufficient data to ensure that the (parallelisable) work of actually processing the query dominates the sequential parts of broadcasting the query and merging results, b) care is taken to minimize the amount of communication, c) each node has a similar amount of similar data.

Another term often used in the context of parallel computing is that of *scalability*. A search system can be considered to be scalable either if the time taken to process a query is directly proportional to the number of nodes in the cluster. i.e. doubling the number of nodes halves the time taken to process queries. An alternative definition of a scalable system is a system where query processing time remains constant despite growth in data, provided that the number of nodes in the cluster is proportionately increased.

5.3 Replication to Increase Throughput

If a given retrieval system, be it a single PC, an SMP server or a COP, can process q queries per second, it is possible to increase query handling to mq queries per second, for arbitrary m, by replicating the system m times.

Vendors such as CISCO and Layer Four sell network devices to which all m query processing systems can be connected and made to look like a single very fast system

with a single network address. The network device allows for systems being added or taken off-line and automatically bypasses systems which crash.

Provided that the capacity of the network device is not exceeded, the parallel efficiency of this type of parallelism is effectively 100%.

5.4 Real Web Search Hardware

If a single PC can efficiently process queries over a collection of 10 million Web pages, a cluster of 100 PCs will be needed to deal with a collection of one billion pages. This represents a large investment in hardware. Search engine companies have a strong motivation to try to avoid using a cluster of this size to evaluate every single incoming query. One obvious solution is to cache the results of the most commonly submitted queries and to dedicate a single PC to intercepting these queries and supplying canned answers. (See Section 4.5.) Something like one third of incoming queries can be handled in this way, resulting in large scale savings in hardware required.

A considerable investment in hardware is needed to operate a large-scale Web search engine. Google, whose indexes cover nearly a billion Web pages, and whose query rate is of the order of 140 million queries per day, is understood to use cheap Intel Celeron PCs. However, at last count around 12,000 such PCs were deployed!

5.5 Exercise 2 – Search Engine Economics

The InfoGurgle company operates a search engine which is funded entirely by advertising revenue. InfoGurgle technology is based on low cost PC hardware. One InfoGurgle PC is capable of processing queries over only 10 million web pages, but PCs may be clustered to handle larger amounts of data. In addition to the search PCs, there are a number of PCs dedicated to serving cached answers to common queries. The InfoGurgle spider works by completely respidering the entire Web each time.

The following are the budget and operating estimates for the forthcoming year.

```
Size of index: 1 billion pages
Average Web page size: 10 kbytes
Average size of InfoGurgle results page: 15 kbytes
Revenue per query: 0.25 cents
Number of queries per day: 20 million average, 50 million peak.
Time taken to fully process a query:  0.2 sec average.
Time taken to process a cached query: 0.001 sec.
Proportion of query load processed from cache: 35\%
Cost per standard PC: 300 Euro (annual lease cost)
Network charges: 30 Euro per gigabyte
Budget for spidering: 1.2 million Euros
Fixed costs (eg. salaries, rent, Ferrari lease): 2 million Euros.

Q1: How much does it cost (in network charges) to re-spider once?

Q2: What interval must there be between spider runs to stay within
    budget?
```

```
Q3: How many search PCs are needed to cope with the uncached query
    portion of peak load?

Q4: How many cached-query PCs are needed to cope with the cached
    query portion of peak load?

Q5: What is the cost of the query processing hardware?

Q6: What is the expected total revenue?

Q7: What will be InfoGurgle's profit or loss this year?

Q8: What would be InfoGurgle's profit/loss if the following
    measures were adopted?

    a. Use larger and more expensive (3000 Euro p.a.) cached query
       PCs to increase the percentage of queries handled from
       cache to 50%, while retaining current response time.

    b. Introducing a query optimisation which speeds query
       processing to 0.15 sec.

Q9: If the spidering budget were reduced to the point necessary for
    financial break-even, how often would spidering occur?

Q10: What motivation is there for InfoGurgle to:
    a. Improve the quality of its search results?
    b. Update its index more frequently?
```

6 Distributed Information Retrieval

Centralised web search engines which operate purely as shown in Figure 13 and as described above, are unable to index all the information published via the Web. Apart from the fact that the Web is infinite due to the presence of automatic content generators, spiders are unable to index Web *dark matter*.

Dark matter is content which is published via the HTTP protocol from a server on the Internet but which can't be fetched by a particular spider due to password protection, IP-address or DNS-domain restriction, `robots.txt` exclusion, or because the page is not reachable by that spider by following links.

An alternative model of search which can potentially avoid these problems is *meta-search* or distributed information retrieval, shown in Figure 17. Most current meta-searchers such as MetaCrawler, ProFusion and SavvySearch address the alleged problem [38] that centralised search engines only index a small fraction of the Web by broadcasting queries to a selection (often ten or twelve) of centralised search engines and merge the results into a single list.

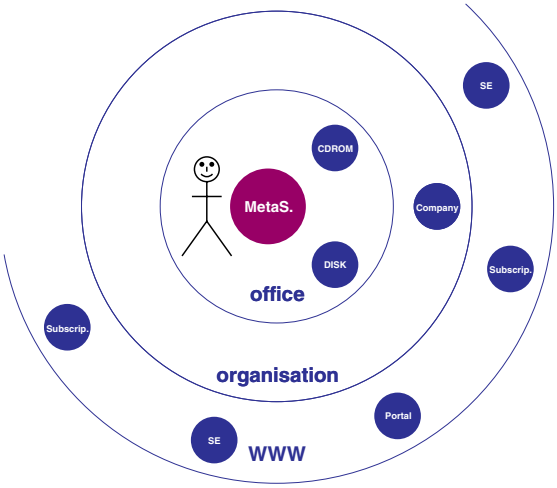

Fig. 16. The range of different information sources available to a modern information worker.

An alternative model uses the search broker to aggregate results obtained from a large number of local search engines operating on individual sites or groups of sites across the Web. Local search engines are potentially able to index more, or all, of the local content and may not have to obey `robots.txt`. Interesting examples of sites operating local search services include current news sites (such as www.msnbc.com), and the PubMed index of medical abstracts.

Figure 16 shows that a modern worker in their office has access to a large number of different information sources. An ideal distributed information retrieval system might provide a unified search service over all of them.

To do so, it would need to solve four key problems:

Server identification and characterisation It is a non-trivial matter to identify all the potentially useful search services available and to gather useful information about them – what types of documents they index, how many documents, how effective is the search algorithm they employ.

Server selection Using knowledge of the available servers, what would constitute an appropriate server subset for processing this query. It may be undesirable to forward the query to all servers because of network and computational costs and because some servers may charge money for each query processed. There is also a possibility that search quality may be improved by restricting the search to the most appropriate servers.

Query Translation Different search engines support different query syntax and implement different semantics. Consequently, queries submitted to the broker must be translated for some engines.

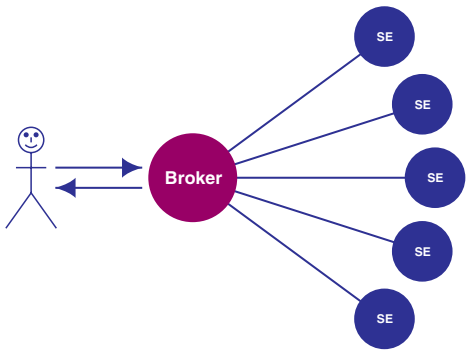

Fig. 17. The architecture of a metasearcher.

Result Merging Combining several results lists into a single merged list is more difficult than it sounds. Merging based on reported relevance scores is problematic because scores returned by different algorithms (or even the same algorithm working on different collections) are not in general comparable. Even worse, scores are often not reported. It is usually possible to merge on the basis of ranks but the highest ranked document from one search server may be inferior to the lowest supplied by another. In general, best results are obtained by downloading all the documents and running a high quality relevance scoring function over the resulting pool of documents.

6.1 Further Reading

Space does not permit a full treatment of the field of distributed information retrieval. As a substitute you may wish to read research papers in the following areas:

- **Combining centralised Web search engines:** [48, 16]
- **Fusion of partitioned collections** [6, 58, 20, 13, 42]
- **Metasearching using cooperating servers** [19, 27, 34]
- **Metasearching by downloading** [5, 37, 10]

7 Evaluation of Web Search Quality

Figure 5 shows an evaluation paradigm for standard retrieval systems. This paradigm must be interpreted and refined if it is to be applied to the evaluation of public Web search engines. When evaluating Web search engines from across the Web, it is not possible to isolate the indexing/ranking process from spidering (and decrypting). The quality of results returned must depend upon all of these components. If one or more of the desired answers to a search failed to be found by the spider, they will not be in the collection and cannot be returned as a search result. Similarly, if a required answer

document is in PDF format, the spider must be able to find it and the decrypter must be able to extract its text content for the search to have any chance of success.

Furthermore, there is no standardised, stable test collection. Rather, it is necessary to treat the whole Web as the test collection. Because the Web is dynamic, relevance or quality judgments are not re-usable. Two detailed studies of search engine performance discuss in detail the various methodological questions associated with public search engine evaluation. [18, 23]

One of the key issues in Web search evaluation is that there are in fact many different types of search. Was the searcher trying to buy something on the Web? Were they looking for the homepage of a person or organisation? Did they need background information for a newspaper article they were writing? Did they need up-to-date information about the latest terrorist attack? Were they trying to find the most popular fan sites for the latest pop culture hero? Were they instead conducting an exhaustive search for every Web page that mentions their name?

Evaluation of each different mode of search potentially may require mode-specific: a) judging criteria, b) number of results judged, and c) measures to be reported. Not only that, but it is fairly clear that optimal ranking algorithms are search-mode dependent. [12, 50]

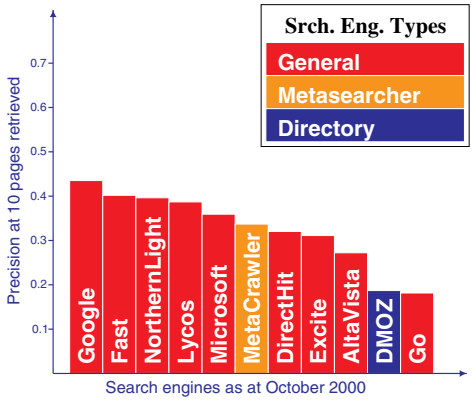

Fig. 18. Comparison of public search engines on the basis of their ability to find documents relevant to a topic. Judges were asked to judge result pages were as either relevant or irrelevant. A relevant page was required to a) be on the topic and b) to contribute some additional information not supplied by the question. Judging was blind and results from all engines were pooled prior to judging. Fifty-four queries were used, taken from search engine query logs. An example is: 'thalidomide and multiple sclerosis'.

Figures 18 - 20 show the results of more recent evaluations I have conducted with my colleagues for: a) topic relevance, b) online service location and c) homepage finding modes of search. More detailed reports of these studies are to be found in [24] and [11].

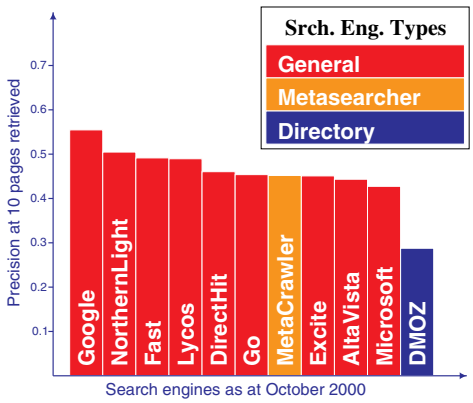

Fig. 19. Comparison of public search engines on the basis of their ability to find online services. Judges were asked to judge whether result pages were useful. A useful page was required to provide direct access to the desired service. Judging was blind and results from all engines were pooled prior to judging. One hundred and six queries were used, taken from search engine query logs. An example is: 'where can i buy power tools online?'

References

1. J. Allan, J. Callan, M. Sanderson, J. Xu, and S.Wegmann. INQUERY and TREC-7. In *Proceedings of TREC-7*, November 1998. NIST special publication 500-242, trec.nist.gov/pubs/trec7/t7_proceedings.html.
2. Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. Vector-space ranking with effective early termination. In *Proceedings of ACM SIGIR'01*, pages 35–42, New Orleans, LA, 2001.
3. Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. ACM Press/Addison-Wesley, New York, 1999.
4. Krishna Bharat and Andrei Broder. Mirror, mirror on the web: a study of host pairs with replicated content, 1999. www8.org/w8-papers/4c-server/mirror/mirror.html.
5. J. Callan, M. Connell, and A. Du. Automatic discovery of language models for text databases. In *Proceedings of ACM SIGMOD'99*, pages 479–490, New York, 1999.
6. James P. Callan, Zihong Lu, and W. Bruce Croft. Searching distributed collections with inference networks. In *Proceedings of ACM SIGIR'95*, pages 12–20, 1995.
7. Charles L.A. Clarke and Gordon V. Cormack. Shortest-substring retrieval and ranking. *ACM Transactions on Information Systems*, 18(1), 44-78 2000.
8. Cross Language Evaluation Forum webpage. www.iei.pi.cnr.it/DELOS/CLEF/. accessed 25 Sep 2001.
9. Cyril Cleverdon. The Cranfield tests on index language devices. In Karen Sparck Jones and Peter Willett, editors, *Readings in Information Retrieval*, pages 47–59. Morgan Kauffman, San Francisco, 1997. (Reprinted from Aslib Proceedings, 19, 173-192).
10. Nick Craswell, Peter Bailey, and David Hawking. Server selection on the world wide web. In *Proceedings of the ACM Digital Libraries Conference, San Antonio, Texas*, pages 37–46. ACM Press, New York, June 2000.

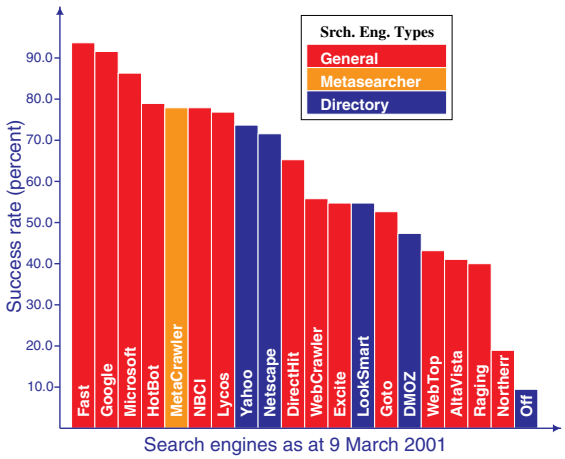

Fig. 20. Comparison of public search engines on the basis of their ability to find airline home pages. Queries were 100 names of airlines listed in the IATA (International Air Transport Association) member list. The correct answer for each query was the official homepage as listed in the members page. Manual judging of results was only necessary to identify aliases of the correct answer. For example `www.qantas.com` and `www.qantas.com.au/index.html` may reference the same page. The measure used was *success rate* – the proportion of cases in which the right answer (or an alias) was found in the top ten results.

11. Nick Craswell, David Hawking, and Kathleen Griffiths. Which search engine is best at finding airline site home pages? Technical Report 2001/45, CSIRO Mathematical and Information Sciences, 2001. www.ted.cmis.csiro.au/ nickc/pubs/airlines.pdf.
12. Nick Craswell, David Hawking, and Stephen Robertson. Effective site finding using link anchor information. In *Proceedings of ACM SIGIR 2001*, pages 250–257, New Orleans, 2001. www.ted.cmis.csiro.au/nickc/pubs/sigir01.pdf.
13. Nick Craswell, David Hawking, and Paul Thistlewaite. Merging results from isolated search engines. In John Roddick, editor, *Proceedings of the 10th Australasian Database Conference, Auckland, NZ*, pages 189–200. Springer-Verlag, January 1999. www.ted.vic.cmis.csiro.au/ nickc/pubs/adc99.ps.gz.
14. Owen de Kretser and Alistair Moffat. Effective document presentation with a locality based similarity heuristic. In *Proceedings of ACM SIGIR'99*, pages 113–120, Berkeley, CA, 1999.
15. William B. Frakes and Ricardo Baeza-Yates, editors. *Information Retrieval. Data Structures and Algorithms*. Prentice Hall, Upper Saddle River NJ, 1992.
16. Susan Gauch and Guijun Wang. Information fusion with ProFusion. In *Proceedings of WebNet '96: The First World Conference of the Web Society*, pages 174–179, October 1996. Also at www.designlab.ukans.edu/ProFusion.html.
17. Arthur Gill. *Introduction to the theory of finite-state machines*. McGraw-Hill, New York, 1962.
18. Michael Gordon and Praveen Pathak. Finding information on the World Wide Web: The retrieval effectiveness of search engines. *Information Processing and Management*, 35(2):141–180, March 1999.

19. Luis Gravano, Kevin Chang, Hector Garcia-Molina, Carl Lagoze, and Andreas Paepcke. STARTS - Stanford protocol proposal for internet retrieval and search. www-db.stanford.edu/ gravano/starts.html, January 1997.
20. Luis Gravano and Hector Garcia-Molina. Generalising GlOSS to vector-space databases and broker hierarchies. In *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, 1996. Morgan Kaufmann, San Francisco CA. Also Stanford technical report CS-TN-95-21.
21. D. K. Harman, editor. *Proceedings of TREC-1*, November 1992. NIST special publication 500-207.
22. David Hawking. Scalable text retrieval for large digital libraries. In Carol Peters and Costatino Thanos, editors, *Proceedings of the First European Conference on Digital Libraries*, volume 1324 of *Lecture Notes in Computer Science*, pages 127–146, Pisa, Italy, September 1997. Springer, Berlin.
23. David Hawking, Nick Craswell, Peter Bailey, and Kathleen Griffiths. Measuring the quality of public search engines, 2000. pastimeánuédubáu/TAR/Search_Engines_Conf/.
24. David Hawking, Nick Craswell, and Kathleen Griffiths. Which search engine is best at finding online services? In *WWW10 Poster Proceedings*, May 2001. www.ted.cmis.csiro.au/ dave/www10poster.pdf.
25. David Hawking, Nick Craswell, and Paul Thistlewaite. Overview of TREC-7 Very Large Collection Track. In *Proceedings of TREC-7*, pages 91–104, November 1998. NIST special publication 500-242, trec.nist.gov/pubs/trec7/t7_proceedings.html.
26. David Hawking and Paul Thistlewaite. Relevance weighting using distance between term occurrences. Technical Report TR-CS-96-08, Department of Computer Science, The Australian National University, cs.anu.edu.au/techreports/1996/index.html, 1996.
27. David Hawking and Paul Thistlewaite. Methods for information server selection. *ACM Transactions on Information Systems.*, 17(1):40–76, 1999.
28. David Hawking, Paul Thistlewaite, and Donna Harman. Scaling up the TREC Collection. *Information Retrieval*, 1(1):115–137, 1999.
29. Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. www.research.compaq.com/SRC/mercator/papers/www/paper.html, 1999. Accessed 25 Sep 2001.
30. John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley, Reading, MA, 1969.
31. Alis Technology Inc. ?'qué? the system for identification of language and character encoding. www.alis.com/castil/silc/. Accessed 25 Sep 2001.
32. H. Garcia-Molina J. Cho and L. Page. Efficient crawling through URL ordering. Efficient crawling through url ordering. In *Proceedings of WWW7*, pages 161–172, Brisbane, Australia, 1998. www7.scu.edu.au/programme/fullpapers/1919/com1919.htm.
33. Fergus Kelledy. *Query Space Reduction in Information Retrieval*. PhD thesis, Dublin City University, Dublin, Eire, 1997. Also available at www.compapp.dcu.ie/asmeaton/FK-thesis/.
34. Steven T. Kirsch. Distributed search patent. U.S. Patent 5,659,732, August 1997. Infoseek Corporation. software.infoseek.com/patents/dist_search/patents.htm.
35. Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley, Reading MA, 1973.
36. Martijn Koster. The web robots pages. info.webcrawler.com/mak/projects/robots/robots.html.
37. Steve Lawrence and C. Lee Giles. Inquirus, the NECI meta search engine. In *Proceedings of WWW7*, pages 95–105, 1998.
38. Steve Lawrence and C. Lee Giles. Accessibility of information on the web. *Nature*, 400:107–109, 8 July 1999.
39. A. MacFarlane, S.E. Robertson, and J.A. McCann. Parallel computing in information retrieval - an updated review. *Journal of Documentation*, 53(3):274–315, June 1997.

40. Alistair Moffat and Timothy A. H. Bell. *In situ* generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550, 1995.
41. National Institute of Standards and Technology. TREC home page. trec.nist.gov/, 1997.
42. Allison L. Powell, James C. French, Jamie Callan, Margaret Connell, and Charles L. Viles. The impact of database selection on distributed searching. In *Proceedings of ACM SIGIR'00*, pages 232–239, Athens, Greece, 2000.
43. S. F. Reddaway. High speed text retrieval from large databases on a massively parallel processor. *Information Processing and Management*, 27(4):311–316, 1991.
44. S. E. Robertson. On term selection for query expansion. *Journal of Documentation*, 46(4):359–364, 1990.
45. S. E. Robertson, S. Walker, M.M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *Proceedings of TREC-3*, November 1994. NIST special publication 500-225.
46. J.J. Rocchio, Jr. Relevance feedback in information retrieval. In Gerard Salton, editor, *The SMART Retrieval System – experiments in automatic document processing*, page chapter 14, Englewood Cliffs, NJ, 1971. Prentice-Hall.
47. Gerard Salton. *The SMART Retrieval System – experiments in automatic document processing*. Prentice-Hall, Englewood Cliffs, NJ, 1971.
48. E. Selberg and O. Etzioni. Multi-service search and comparison using the meta-crawler. In *Proceedings of WWW4*, Boston MA, December 1995.
49. Craig Silverstein, Monika Henzinger, Hannes Marais, and Michael Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, 1999. Previously available as Digital Systems Research Center TR 1998-014 at www.research.digital.com/SRC.
50. Amit Singhal and Marcin Kaszkiel. A case study in web search using TREC algorithms. In *Proceedings of WWW10*, pages 708–716, Hong Kong, 2001. www.www10.org/cdrom/papers/pdf/p317.pdf.
51. Craig Stanfill and Brewster Kahle. Parallel free text search on the Connection Machine system. *Communications of the ACM*, 29(12):1229–1239, December 1986.
52. John A. Swets. Information retrieval systems. *Science*, 141(3577):245–250, July 1963.
53. Anastasios Tombros and Mark Sanderson. Advantages of query biased summaries in information retrieval. In *Proceedings of SIGIR'98*, pages 2–10, Melbourne, Australia, August 1998.
54. Unicode home page. www.unicode.org. accessed 25 Sep 2001.
55. RFC1738: Uniform resource locators (URL). www.w3.org/Addressing/rfc1738.txt. Accessed 25 Sep 2001.
56. Ellen Voorhees. Overview of the TREC-9 question answering track. In *Proceedings of TREC-9*, November 2000. trec.nist.gov/pubs/trec9/papers/qa_overview.pdf.
57. Ellen M. Voorhees. Variations in relevance judgments and the measurement of retrieval effectiveness. In *Proceedings of SIGIR'98*, pages 315–323, Melbourne, Australia, August 1998.
58. Ellen M. Voorhees, Narendra K. Gupta, and Ben Johnson-Laird. Learning collection fusion strategies. In *Proceedings of ACM SIGIR'95*, pages 172–179, 1995.
59. Hugh E. Williams, Justin Zobel, and Phil Anderson. What's next? efficient structures for phrase querying. In John Roddick, editor, *Proceedings of the Tenth Australasian Database Conference, Auckland, NZ*, pages 141–152, Singapore, 1999. Springer Verlag.
60. Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes*. Morgan Kaufmann, San Francisco, 2nd ed. edition, 1999.
61. Justin Zobel. How reliable are the results of large-scale information retrieval experiments? In *Proceedings of ACM SIGIR'98*, Melbourne, Australia, August 1998.