4 *Index construction*

In this chapter, we look at how to construct an inverted index. To do this, we essentially have to perform a sort of the postings file. This is non-trivial for the large data sets that are typical in modern information retrieval. We will first introduce block merge indexing, an efficient single-machine algorithm designed for static collections (Section 4.1). For very large collections like the web, indexing has to be *distributed* over large computer clusters with hundreds or thousands of machines (Section 4.2). Collections with frequent changes require *dynamic indexing* so that changes in the collection are immediately reflected in the index (Section 4.3). Finally, some complicating issues that can arise in indexing – such as security and indexes for ranked retrieval – will be covered in Section 4.4.

Other issues are outside the scope of this chapter. Frequently, documents are not on a local file system, but have to be spidered or crawled, as we discuss in Chapter 20. Also, the indexer needs raw text, but documents are encoded in many ways, as we mentioned in Chapter 2. There are interactions between index compression (see Chapter 5) and construction because intermediate posting lists may have to be compressed and decompressed during construction. Finally documents are often encapsulated in varied content management systems, email applications and databases. While most of these applications can be accessed via http, native APIs are usually more efficient. Although we do not discuss these issues, the reader should be aware that building the subsystem that feeds raw text to the indexing process can in itself be a challenging problem.

4.1 Block merge indexing

The basic steps in constructing a non-positional index are depicted in Figures 1.4 and 1.5 (page 8). We first make a pass through the collection assembling all postings (i.e., term-docID pairs). We then sort the entries with the term as the dominant key and docID as the secondary key. Finally, we

► **Figure 4.1** Document from the Reuters newswire.

organize the docIDs for each term into a posting list and compute statistics like term and document frequency. For small collections, all this can be done in memory. In this chapter, we describe methods for large collections that require the use of secondary storage.

To make index construction more efficient, we represent postings as termID-docID pairs (instead of term-docID pairs as we did in Figures 1.4 and 1.5). We can build the mapping from terms to termIDs on the fly while we are processing the collection; or we can adopt a two-pass approach that compiles the lexicon in the first pass and constructs the inverted index in the second pass. The index construction algorithms described in this chapter use on-the-fly construction because it avoids the extra pass through the data and works well if the lexicon is not too large. Section 4.5 gives references to algorithms that can handle very large lexicons.

REUTERS-RCV1 We will work with the *Reuters-RCV1* collection as our model collection in this chapter, a collection with roughly one gigabyte of text. It consists of about 800,000 documents that were sent over the Reuters newswire during a one year period between August 20, 1996, and August 19, 1997. A typical document is shown in Figure 4.1. Reuters-RCV1 covers a wide range of international topics, including politics, business, sports and (as in the example) science. Some key statistics of the collection are shown in Table 4.1.[1]

Reuters-RCV1 has 100 million postings. If we used 8 bytes per posting (4 bytes each for termID and docID), these 100 million postings will require 0.8 gigabyte of storage. Typical collections today are often one or two orders of magnitude larger than Reuters-RCV1. You can easily see how such collec-

1. The numbers in this table correspond to the third line ("case folding") in Table 5.1, page 68. For the definitions of token and type, see Chapter 2, page 20.

symbol	statistic	value
N	documents	800,000
L	avg. # tokens per document	200
M	term types	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term type	7.5
	non-positional postings	100,000,000

▶ **Table 4.1** Collection statistics for Reuters-RCV1. Values are rounded for the computations in this chapter. The unrounded values are: 806,791 documents, 222 tokens per document, 391,523 term types (or distinct terms), 6.04 bytes per token with spaces and punctuation, 4.5 bytes per token without spaces and punctuation, 7.5 bytes per term type and 96,969,056 non-positional postings.

BLOCKMERGEINDEXCONSTRUCTION()
1 $n = 0$
2 **while** (all documents have not been processed)
3 **do** $n = n + 1$
4 $block$ = PARSENEXTBLOCKOFDOCUMENTS()
5 INVERT($block$)
6 WRITETODISK($block, f_n$)
7 MERGEBLOCKS($f_1, \ldots, f_n; f_{\text{merged}}$)

▶ **Figure 4.2** Block merge indexing. The algorithm stores inverted blocks in files $f_1, \ldots, f_n$ and the merged index in f_{merged}.

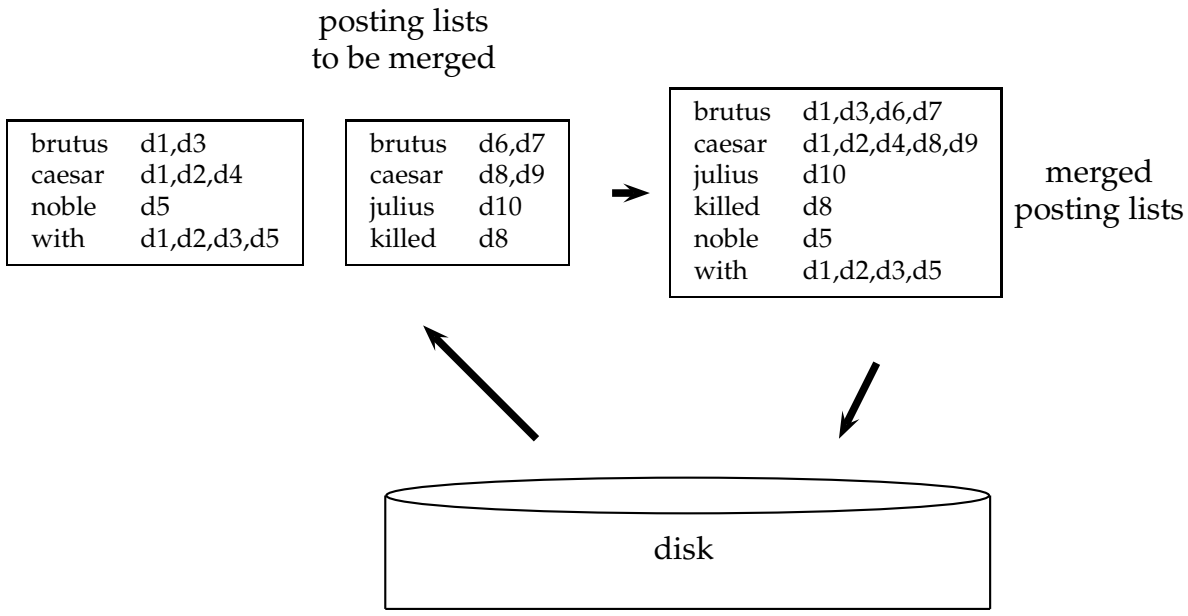
tions will overwhelm even large computers if we tried to sort their postings files in memory. If the size of the postings file is within a small factor of available memory, then the compression techniques introduced in Chapter 5 can help; but the postings file of many large collections cannot fit into memory even after compression.

EXTERNAL SORTING

With main memory insufficient, we need to use an *external sorting* algorithm, i.e., one that uses disk in a way that minimizes random disk seeks during sorting. One solution is the *block merge algorithm* in Figure 4.2, which sorts the postings of parts of the collection in memory, stores intermediate results on disk and then merges all intermediate results into the final index.

BLOCK MERGE ALGORITHM

The first step of the algorithm is to parse documents into postings and accumulate the postings in memory until a block of a fixed size is full (PARSENEXTBLOCKOFDOCUMENTS). We choose the block size to fit comfortably into memory to permit a fast in-memory sort. The block is then inverted and written to disk.



► **Figure 4.3** Merging in block merge indexing. Two blocks ("posting lists to be merged") are loaded from disk into memory, merged in memory ("merged posting lists") and written back to disk. We show terms instead of termIDs for better readability.

Inversion involves two steps. First, we sort the postings. Then all postings with the same termID are collected into a posting list. The result, an inverted index for the block we have just read, is then written to disk. Applying this to Reuters-RCV1 and assuming we can fit 10 million postings into memory, we end up with 10 blocks, each an inverted index of part of the collection.

In the second step, the algorithm simultaneously merges the 10 blocks into one large merged index. An example with two blocks is shown in Figure 4.3. To do this, we open all block files simultaneously, and maintain small read buffers for the 10 blocks we are reading and a write buffer for the final merged index we are writing. In each iteration, we select the lowest termID that has not been processed yet using a priority queue or a similar data structure. All posting lists for this termID are read, merged and the merged list written back to disk. Each read buffer is refilled from its file when necessary.

How expensive is block merge indexing? It is dominated by the time it

takes to parse the documents (PARSENEXTBLOCKOFDOCUMENTS) and to do the final merge (MERGEBLOCKS). Exercise 4.2 asks you to compute the total index construction time for RCV1 that includes these steps as well as inversion and writing the blocks to disk.

The reader will have noticed that Reuters-RCV1 is not particularly large in an age when one or more GB of memory are standard on personal computers. With appropriate compression (Chapter 5), we could have created an inverted index for RCV1 in memory on a not overly beefy server. The techniques we have described are needed, however, for collections that are several orders of magnitude larger.

4.2 Distributed indexing

Collections are often so large that we cannot perform index construction on a single machine. This is particularly true of the World Wide Web for which we need large computer clusters to construct any reasonably sized web index. Web search engines therefore use *distributed indexing* algorithms for index construction. The result of the construction process is an index that is partitioned across several machines – either according to term or according to document. In this section, we describe distributed indexing for a term-partitioned index. Most large search engines prefer a document-partitioned index (which can be easily generated from a term-partitioned index). We discuss this topic further in Section 20.3 (page 369).

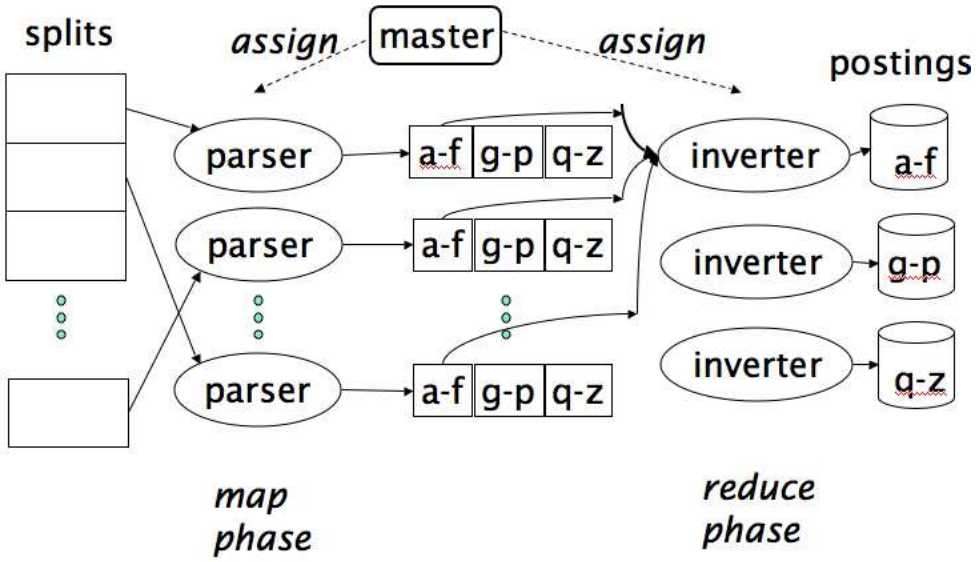
MAPREDUCE

The distributed index construction method we describe in this section is an application of *MapReduce*, a general architecture for distributed computing. MapReduce is designed for large computer clusters. The point of a cluster is to solve large computing problems on cheap commodity machines or *nodes* that are built from standard parts (processor, memory, disk) as opposed to on a supercomputer with specialized hardware. While hundreds or thousands of machines are available in such clusters, individual machines can fail at any time. One requirement for robust distributed indexing is therefore that we divide the work up into chunks that we can easily assign and – in case of failure – reassign. A *master node* directs the process of assigning and reassigning tasks to individual worker nodes.

MASTER NODE

The *map* and *reduce* phases of MapReduce split up the computing job into chunks that standard machines can process in a short time. The various steps are shown in Figure 4.4 and an example for a collection with two documents is shown in Figure 4.5. First, the input data, in our case a collection of web pages, is split into *n splits* where the size of the split is chosen to ensure that the work can be distributed evenly (chunks shouldn't be too large) and efficiently (the total number of chunks we need to manage shouldn't be too large). 16 MB or 64 MB are good sizes in distributed indexing. Splits are not

SPLITS



► **Figure 4.4** An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).

preassigned to machines, but are instead assigned by the master node on an ongoing basis: as a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard due to hardware problems, the split it is working on is simply reassigned to another machine.

KEY-VALUE PAIRS

In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of *key-value pairs*. For indexing, a key-value pair has the form (termID,docID) and, not surprisingly, is nothing other than a posting. In distributed indexing, the mapping from terms to termIDs is also distributed and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms directly (instead of termIDs) for infrequent terms. We will not address this problem here and assume that all nodes share a consistent term → termID mapping.

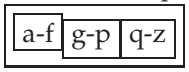
MAP PHASE

The *map phase* of MapReduce consists of mapping splits of the input data to postings or key-value pairs. This is the same parsing task we also encountered in block merge indexing, and we therefore call the machines that

PARSER

execute the map phase *parsers*. Each parser writes its output to three local

SEGMENT FILE

intermediate files, the *segment files* (shown as 

in Figure 4.4).

REDUCE PHASE

For the *reduce phase*, we want all values for a given key to be stored close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into j partitions and having the parsers write key-value pairs for each partition into a separate segment file. In Figure 4.4, the partitions are according to first letter: a–f, g–p, q–z, so $j = 3$. (We chose these key ranges for ease of exposition. In general, key ranges are not contiguous.) The partitions are defined by the person who operates the indexing system (Exercise 4.8). The parsers then write corresponding segment files, one for each partition.

INVERTERS

Collecting all values (here: docIDs) for a given key (here: termID) into one list is the task of the *inverters* in the reduce phase. The master assigns each partition to a different inverter – and, as in the case of parsers, reassigns partitions in case of failing or slow inverters. Each partition (corresponding to k segment files, one on each parser) is processed by one inverter. Finally, the list of values is sorted for each key and written to the final sorted posting list ("postings" in the figure). Note that we assume that all posting lists are of a size that a single machine can handle (see Exercise 4.7). This data flow is shown for "a-f" in Figure 4.4. This completes the construction of the inverted index.

Parsers and inverters are not separate sets of machines. The master identifies idle machines and assigns tasks to them. The same machine can be a parser in the map phase and an inverter in the reduce phase. And there are often other jobs that run in parallel with index construction, so in between being a parser and an inverter a machine might do some crawling or another unrelated task.

To minimize write times before inverters reduce the data, each parser writes the segment files to its *local disk*. In the reduce phase, the master communicates to an inverter the locations of the relevant segment files (e.g., for the a-f partition). Each segment file only requires one sequential read since all data relevant to a particular inverter were written to a single segment file by the parser. This setup minimizes the amount of network traffic needed during indexing.

Figure 4.5 shows the general schema of the MapReduce functions. Input and output are often lists of key-value pairs themselves, so that several MapReduce jobs can be run in sequence. In fact, this was the design of the Google indexing system in 2004. What we have just covered in this section corresponds to only one of 5 to 10 MapReduce operations in that indexing system. Another MapReduce operation transform the term-partitioned index we just created into a document-partitioned one.

MapReduce offers a robust and conceptually simple framework for implementing index construction in a distributed environment. By providing a semi-automatic method for splitting index construction into smaller tasks, it can scale to almost arbitrarily large collections, given computer clusters of

Schema of map and reduce functions		
map:	input	$\rightarrow$ list(k, v)
reduce:	(k,list(v))	$\rightarrow$ output
Instantiation of the schema for index construction		
map:	web collection	$\rightarrow$ list($\langle$termID, docID$\rangle$)
reduce:	($\langle$termID$_1$, list(docID)$\rangle$, $\langle$termID$_2$, list(docID)$\rangle$, ...)	$\rightarrow$ (inverted_list$_1$, inverted_list$_2$, ...)
Example for index construction		
map:	d_2 : C died. d_1 : C came, C c'ed.	$\rightarrow$ ($\langle$C, $d_2\rangle$, $\langle$died,$d_2\rangle$, $\langle$C,$d_1\rangle$, $\langle$came,$d_1\rangle$, $\langle$C,$d_1\rangle$, $\langle$c'ed,$d_1\rangle$
reduce:	($\langle$C,(d_2,d_1,d_1)$\rangle$,$\langle$died,(d_2)$\rangle$,$\langle$came,(d_1)$\rangle$,$\langle$c'ed,(d_1)$\rangle$)	$\rightarrow$ ($\langle$C.,(d_1:2,d_2:1)$\rangle$,$\langle$died,(d_2:1)$\rangle$,$\langle$came,(d_1:1)$\rangle$,$\langle$c'ed,(d_1:1)$\rangle$

► **Figure 4.5** Map and reduce functions in MapReduce. In general, the map function produces a list of key-value pairs. All values for a key are collected into one list in the reduce phase. This list is then further processed. The instantiations of the two functions and an example are shown for index construction. Since the map phase processes documents in a distributed fashion, postings need not be ordered correctly initially as in this example. The example shows terms instead of termIDs for better readability. We abbreviate Caesar as C. and conquered as c'ed.

sufficient size.

4.3 Dynamic indexing

Thus far, we have assumed that the document collection is static. This is fine for collections that change infrequently or never (e.g., the Bible or Shakespeare). But most collections frequently change with documents being added, deleted and updated. This means that new terms need to be added to the dictionary; and posting lists need to be updated for existing terms.

The simplest way to achieve this is to periodically reconstruct the index from scratch. This is a good solution if the number of changes over time is small and a delay in making new documents searchable is acceptable – or if enough resources are available to construct a new index while the old one is still available for querying.

AUXILIARY INDEX

If there is a requirement that new documents be included quickly, one solution is to maintain two indexes: a large main index and a small *auxiliary index* that stores new documents. The auxiliary index is kept in memory. Searches are run across both indexes and results merged. Deletions are stored in an invalidation bit vector. We can then filter out deleted documents before returning the search result. Documents are updated by deleting and reinserting them.

Each time the auxiliary index becomes too large, we merge it into the main index. The cost of this merging operation depends on how we store the index in the file system. If we store each posting list as a separate file, then the

```
LMERGEPOSTING(indexes, Z_0, posting)
 1  Z_0 ← MERGE(Z_0, {posting})
 2  if |Z_0| = n
 3    then for i ← 0 to ∞
 4         do if I_i ∈ indexes
 5               then Z_{i+1} ← MERGE(I_i, Z_i)
 6                    (Z_{i+1} is a temporary index on disk.)
 7                    indexes = indexes − {I_i}
 8               else I_i ← Z_i    (Z_i becomes the permanent index I_i.)
 9                    indexes = indexes ∪ {I_i}
10                    BREAK
11         Z_0 ← ∅

LOGARITHMICMERGE()
1  Z_0 ← ∅    (Z_0 is the in-memory index.)
2  indexes ← ∅
3  while true
4  do LMERGEPOSTING(indexes, Z_0, GETNEXTPOSTING())
```

► **Figure 4.6** Logarithmic merging. Each posting (termID,docID) is initially added to in-memory index Z_0 by LMERGEPOSTING. LOGARITHMICMERGE initializes Z_0 and *indexes*.

merge simply consists of extending each posting list of the main index by the corresponding posting list of the auxiliary index. In this scheme, the reason for keeping the auxiliary index is to reduce the number of disk seeks required over time. Updating each document separately would require M_d disk seeks, where M_d is the size of the vocabulary of the document. With an auxiliary index, we only put additional load on the disk when we merge auxiliary and main indexes.

Unfortunately, the one-file-per-posting-list-scheme is inefficient because most file systems cannot efficiently handle very large numbers of files. The simplest alternative is to store the index as one large file, i.e., as a concatentation of all posting lists. In reality, we will often choose a compromise between the two extremes (Section 4.5). To simplify the discussion, we choose the simple option of storing the index as one large file here.

In this scheme, if the the auxiliary index has n postings and we have indexed a total of T postings so far, then each posting has been processed $\lfloor T/n \rfloor$ times because we had to touch it during each of $\lfloor T/n \rfloor$ merges. Thus, the overall time complexity is $O(T^2/n)$.

We can do better than $O(T^2/n)$ by introducing $O(\log_2 T)$ indexes I_0, I_1,

I_2, ... of size $2^0 \times n$, $2^1 \times n$, $2^2 \times n$ .... Postings percolate up this sequence of indexes and are process only once on each level. This scheme is called *logarithmic merging* (Figure 4.6). As before, up to n postings are accumulated in an in-memory index, which we call Z_0. When the limit n is reached, an index I_0 with $2^0 \times n$ postings is created on disk. The next time Z_0 is full, it is merged with I_0 to create an index Z_1 of size $2^1 \times n$. Then Z_1 is either stored as I_1 – if there isn't already an I_1 – or merged with I_1 into Z_2 if I_1 exists; and so on. We service search requests by querying in-memory Z_0 and all currently valid indexes I_i on disk and merging the results.

LOGARITHMIC MERGING

Overall index construction time is $O(T \log T)$. We trade this efficiency gain for a slow-down of query processing as we now need to merge results from $\log T$ indexes as opposed to just two (the main and auxiliary indexes). As in the auxiliary index scheme, we still need to merge very large indexes occasionally (which will slow down the search system during the merge), but this will happen less frequently and the indexes involved in a merge will on average be smaller.

Having multiple indexes complicates the maintenance of collection-wide statistics. For example, it affects the spell correction algorithm in Section 3.2 (page 43) that selects the corrected alternative with the most hits. With multiple indexes and an invalidation bit vector, the correct number of hits for a term is no longer a simple lookup. And we will see in Chapter 6 that collection-wide statistics are also important in ranking. In fact, all aspects of the system – index maintenance, query processing, distribution – are more complex in logarithmic merging.

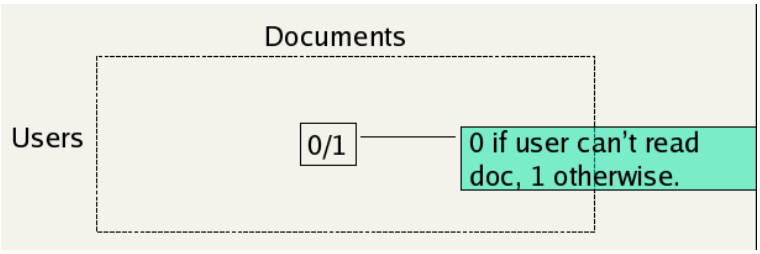
Because of this complexity of dynamic indexing, some large search engines adopt a reconstruction-from-scratch strategy. They do not construct indexes dynamically. Instead, a new index is built from scratch periodically. Query processing is then switched from the new index and the old index is deleted.

4.4 Other types of indexes

This chapter only describes construction of non-positional indexes. Except for the much larger data volume we need to accommodate, the only difference for positional indexes is that (termID, docID, (position1, position2, ...)) triples instead of (termID, docID) pairs have to be sorted and posting lists contain position information in addition to docIDs. With this minor change, the algorithms discussed here can all be applied to positional indexes.

In the indexes we have considered so far, postings occur in document order. As we will see in the next chapter, this is advantageous for compression – instead of docIDs we can compress smaller *gaps* between IDs, thus reducing space requirements for the index. However, this structure for the index is not optimal when we build *ranked* (Chapters 6 and 7) – as opposed to Boolean

RANKED RETRIEVAL



► **Figure 4.7** An inverted user-document matrix for access control lists. Element (i, j) is 1 if user i has access to document j and 0 otherwise. During query processing, a user's access posting list is intersected with the result list returned by the text part of the index.

– retrieval systems. In ranked retrieval, postings are often ordered according to weight or impact, with the highest-weighted postings occurring first. With this organization, scanning of long posting lists during query processing can usually be terminated early when weights have become so small that any further documents can be predicted to be of low similarity to the query (see Chapter 6). In a docID-sorted index, new documents are always inserted at the end of posting lists. In an impact-sorted index, the insertion can occur anywhere, thus complicating the update of the inverted index.

SECURITY

Security is an important consideration for retrieval systems in corporations. The average employee should not be able to find the salary roster of the corporation, but authorized managers need to be able to search for it. Users' result lists must not contain documents they are barred from opening since the very existence of a document can be sensitive information.

ACCESS CONTROL LISTS

User authorization is often mediated through *access control lists* or ACLs. ACLs can be dealt with in an inverted index data structure by representing each document as the set of users that can access them (Figure 4.7) and then inverting the resulting user-document matrix. The inverted ACL index has, for each user, a "posting list" of documents they can access – the user's access list. Search results are then intersected with this list. However, such an index is difficult to maintain when access permissions change – we discussed these difficulties in the context of incremental indexing for regular posting lists in the last section. It also requires the processing of very long posting lists for users with access to large document subsets. User membership is therefore often verified by retrieving access lists directly from the file system at query time – even though this slows down retrieval.

4.5 References and further reading