MODUL TOPIK DALAM INFORMATION RETRIEVAL (CMA 102)

MODUL PERTEMUAN 14
Index Construction

DISUSUN OLEH
Dr. Fransiskus Adikara, S.Kom, MMSI

UNIVERSITAS ESA UNGGUL
2019

INDEX CONSTRUCTION

A. Kemampuan Akhir Yang Diharapkan

After reading this session, you will be able to answer the following questions:

1. Two index construction algorithms: BSBI (simple) and SPIMI (more realistic).
2. Distribution index construction: MapReduce?
3. Dynamic index construction: How to keep the index up-to-date as the collection changes?

B. Uraian dan Contoh

1.1. Hardware basics

When building an information retrieval (IR) system, many decisions are based on the characteristics of the computer hardware on which the system runs. We therefore begin this chapter with a brief review of computer hardware. Performance characteristics typical of systems in 2007 are shown in Table 1.1.

► **Table 1.1** Typical system parameters in 2007. The seek time is the time needed to position the disk head in a new position. The transfer time per byte is the rate of transfer from disk to memory when the head is in the right position.

Symbol	Statistic	Value
s	average seek time	5 ms $= 5 \times 10^{-3}$ s
b	transfer time per byte	0.02 μs $= 2 \times 10^{-8}$ s
	processor's clock rate	10^9 s^{-1}
p	lowlevel operation (e.g., compare & swap a word)	0.01 μs $= 10^{-8}$ s
	size of main memory	several GB
	size of disk space	1 TB or more

A list of hardware basics that we need in this book to motivate IR system design follows.

- Access to data in memory is much faster than access to data on disk. It takes a few clock cycles (perhaps 5×10^{-9} seconds) to access a byte in memory, but much longer to transfer it from disk (about 2×10^{-8} seconds). Consequently, we want to keep as much data as possible in memory, especially those data that we need to access frequently. We call the technique of keeping frequently used disk data in main memory *caching*. CACHING
- When doing a disk read or write, it takes a while for the disk head to move to the part of the disk where the data are located. This time is called the *seek time* and it averages 5 ms for typical disks. SEEK TIME No data are being transferred during the seek. To maximize data transfer rates, chunks of data that will be read together should therefore be stored contiguously on disk. For example, using the numbers in Table 1.1 it may take as little as 0.2 seconds to transfer 10 megabytes (MB) from disk to memory if it is stored as one chunk, but up to $0.2 + 100 \times (5 \times 10{-}3) = 0.7$ seconds if it is stored in 100 noncontiguous chunks because we need to move the disk head up to 100 times.

- Operating systems generally read and write entire blocks. Thus, reading a single byte from disk can take as much time as reading the entire block. Block sizes of 8, 16, 32, and 64 kilobytes (KB) are common. We call the part of main memory where a block being read or written is stored a *buffer*. BUFFER
- Data transfers from disk to memory are handled by the system bus, not by the processor. This means that the processor is available to process data during disk I/O. We can exploit this fact to speed up data transfers by storing compressed data on disk. Assuming an efficient decompression algorithm, the total time of reading and then decompressing compressed data is usually less than reading uncompressed data.
- Servers used in IR systems typically have several gigabytes (GB) of main memory, sometimes tens of GB. Available disk space is several orders of magnitude larger.

1.2. Blocked sort-based indexing

We first make a pass through the collection assembling all term–docID pairs. We then sort the pairs with the term as the dominant key and docID as the secondary key. Finally, we organize the docIDs for each term into a postings 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.

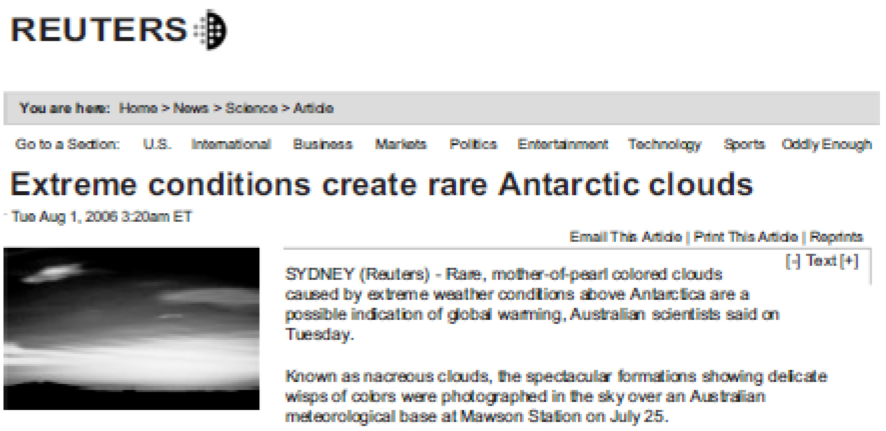
To make index construction more efficient, we represent terms as termIDs, where each *termID* is a unique serial number. We can build the mapping from terms to termIDs on the fly while we are processing the collection; or, in a two-pass approach, we compile the vocabulary in the first pass and construct the inverted index in the second pass. The index construction algorithms described in this chapter all do a single pass through the data. TERMID

REUTERS-RCV1 We work with the *Reuters-RCV1* collection as our model collection in this chapter, a collection with roughly 1 GB of text. It consists of about 800,000 documents that were sent over the Reuters newswire during a 1-year period between August 20, 1996, and August 19, 1997. A typical document is shown in Figure 1.1, but note that we ignore multimedia information like images in this book and are only concerned with text. Reuters-RCV1 covers a wide range of international topics, including politics, business, sports, and (as in this example) science. Some key statistics of the collection are shown in Table 1.2.

Reuters-RCV1 has 100million tokens. Collecting all termID–docID pairs of the collection using 4 bytes each for termID and docID therefore requires 0.8 GB of storage. Typical collections today are often one or two orders of magnitude larger than Reuters-RCV1. You can easily see how such collections overwhelm even large computers if we try to sort their termID–docID pairs in memory. If the size of the intermediate files during index construction is within a small factor of available memory; however, the postings file of many large collections cannot fit into memory even after compression.

► **Table 1.2** Collection statistics for Reuters-RCV1. Values are rounded for the computations in this book. The unrounded values are: 806,791 documents, 222 tokens per document, 391,523 (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, and 96,969,056 tokens.

Symbol	Statistic	Value
N	documents	800,000
L_{ave}	avg. # tokens per document	200
M	terms	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term	7.5
T	tokens	100,000,000


REUTERS

You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enough

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints

[-] Text [+]

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

► **Figure 1.1** Document from the Reuters newswire.

EXTERNAL SORTING ALGORITHM

BLOCKED SORT-BASED INDEXING ALGORITHM

With main memory insufficient, we need to use an *external sorting algorithm*, that is, one that uses disk. For acceptable speed, the central requirement of such an algorithm is that it minimize the number of random disk seeks during sorting – sequential disk reads are far faster than seeks as we explained in Section 1.1. One solution is the *blocked sort-based indexing algorithm* or *BSBI* in Figure 1.2. BSBI (i) segments the collection into parts of equal size, (ii) sorts the termID–docID pairs of each part in memory, (iii) stores intermediate sorted results on disk, and (iv) merges all intermediate results into the final index.

INVERSION

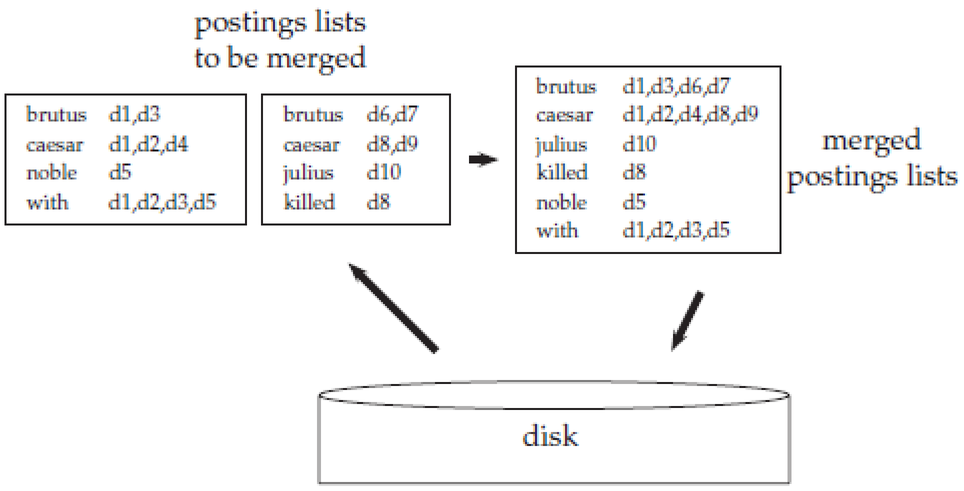
POSTING

The algorithm parses documents into termID–docID pairs and accumulates the pairs in memory until a block of a fixed size is full (PARSENEXTBLOCK in Figure 1.2). 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. *Inversion* involves two steps. First, we sort the termID–docID pairs. Next, we collect all termID–docID pairs with the same termID into a postings list, where a *posting* is simply a docID. 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 termID–docID pairs into memory, we end up with ten blocks, each an inverted index of one part of the collection.

```
BSBINDEXCONSTRUCTION()
$n \leftarrow 0$
while (all documents have not been processed)
do $n \leftarrow n + 1$
    $block \leftarrow$ PARSENEXTBLOCK()
    BSBI-INVERT($block$)
    WRITEBLOCKTODISK($block, f_n$)
MERGEBLOCKS($f_1, \ldots, f_n; f_{merged}$)
```

▶ **Figure 1.2** Blocked sort-based indexing. The algorithm stores inverted blocks in files $f_1, \ldots, f_n$ and the merged index in f_{merged}.

In the final step, the algorithm simultaneously merges the ten blocks into one large merged index. An example with two blocks is shown in Figure 1.3, where we use d_i to denote the i^{th} document of the collection. To do the merging, we open all block files simultaneously, and maintain small read buffers for the ten 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 postings lists for this termID are read and merged, and the merged list is written back to disk. Each read buffer is refilled from its file when necessary.



▶ **Figure 1.3** Merging in blocked sort-based indexing. Two blocks ("postings lists to be merged") are loaded from disk into memory, merged in memory ("merged postings lists") and written back to disk. We show terms instead of termIDs for better readability.

How expensive is BSBI? Its time complexity is $\Theta(T \log T)$ because the step with the highest time complexity is sorting and T is an upper bound for the number of items we must sort (i.e., the number of termID–docID pairs). But the actual indexing time is usually dominated by the time it takes to parse the documents (PARSENEXTBLOCK) and to do the final merge (MERGEBLOCKS).

Notice 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, 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.

Exercise 1.1

If we need $T \log_2 T$ comparisons (where T is the number of termID–docID pairs) and two disk seeks for each comparison, how much time would index construction for Reuters-RCV1 take if we used disk instead of memory for storage and an unoptimized sorting algorithm (i.e., not an external sorting algorithm)? Use the system parameters in Table 1.1.

Exercise 1.2 [⋆]

How would you create the dictionary in blocked sort-based indexing on the fly to avoid an extra pass through the data?

1.3. Single-pass in-memory indexing

SINGLE-PASS IN-MEMORY INDEXING

Blocked sort-based indexing has excellent scaling properties, but it needs a data structure for mapping terms to termIDs. For very large collections, this data structure does not fit into memory. A more scalable alternative is *single-pass in-memory indexing* or *SPIMI*. SPIMI uses terms instead of termIDs, writes each block's dictionary to disk, and then starts a new dictionary for the next block. SPIMI can index collections of any size as long as there is enough disk space available.

The SPIMI algorithm is shown in Figure 1.4. The part of the algorithm that parses documents and turns them into a stream of term–docID pairs, which we call *tokens* here, has been omitted. SPIMI-INVERT is called repeatedly on the token stream until the entire collection has been processed.

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5      if term(token) ∉ dictionary
 6         then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7         else  postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8      if full(postings_list)
 9         then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10      ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

► **Figure 1.4** Inversion of a block in single-pass in-memory indexing

Tokens are processed one by one (line 4) during each successive call of SPIMI-INVERT. When a term occurs for the first time, it is added to the dictionary (best implemented as a hash), and a new postings list is created (line 6). The call in line 7 returns this postings list for subsequent occurrences of the term.

A difference between BSBI and SPIMI is that SPIMI adds a posting directly to its postings list (line 10). Instead of first collecting all termID–docID pairs and then sorting them(as we did in BSBI), each postings list is dynamic (i.e., its size is adjusted as it grows) and it is immediately available to collect postings. This has two advantages: It is faster because there is no sorting required, and it saves memory because we keep track of the term a postings list belongs to, so the termIDs of postings need not be stored. As a result, the blocks that individual calls of SPIMI-INVERT can process are much larger and the index construction process as a whole is more efficient.

Because we do not know how large the postings list of a term will be when we first encounter it, we allocate space for a short postings list initially and double the space each time it is full (lines 8–9). This means that some memory is wasted, which counteracts the memory savings from the omission of termIDs in intermediate data structures. However, the overall memory requirements for the dynamically constructed index of a block in SPIMI are still lower than in BSBI.

When memory has been exhausted, we write the index of the block (which consists of the dictionary and the postings lists) to disk (line 12). We have to sort the terms (line 11) before doing this because we want to write postings lists in lexicographic order to facilitate the final merging step. If each block's postings lists were written in unsorted order, merging blocks could not be accomplished by a simple linear scan through each block.

Each call of SPIMI-INVERT writes a block to disk, just as in BSBI. The last step of SPIMI (corresponding to line 7 in Figure 1.2; not shown in Figure 1.4) is then to merge the blocks into the final inverted index.

In addition to constructing a new dictionary structure for each block and eliminating the expensive sorting step, SPIMI has a third important component: compression. Both the postings and the dictionary terms can be stored compactly on disk if we employ compression. Compression increases the efficiency of the algorithm further because we can process even larger blocks, and because the individual blocks require less space on disk.

The time complexity of SPIMI is $\Theta(T)$ because no sorting of tokens is required and all operations are at most linear in the size of the collection.

1.4. Distributed indexing

Collections are often so large that we cannot perform index construction efficiently 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 a distributed 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).

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. Although 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

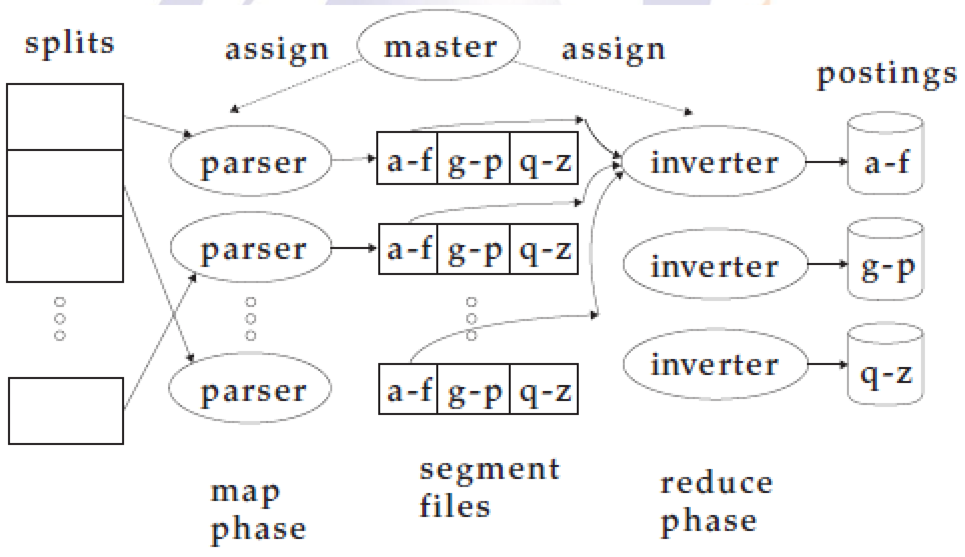
MASTER NODE

– reassign. A *master node* directs the process of assigning and reassigning tasks to individual worker nodes.

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 of MapReduce are shown in Figure 1.5 and an example on a collection consisting of two documents is shown in Figure 1.6. First, the input data, in our

SPLITS

case a collection of web pages, are split into *n splits* where the size of the split is chosen to ensure that the work can be distributed evenly (chunks should not be too large) and efficiently (the total number of chunks we need to manage should not be too large); 16 or 64MB are good sizes in distributed indexing. Splits are not 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.



► **Figure 1.5** An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).

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). 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 do 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 key-value pairs. This is the same parsing task we also encountered in BSBI and

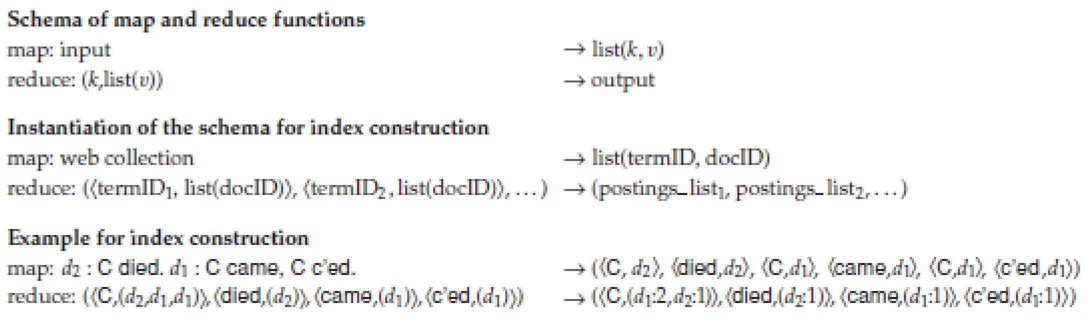
PARSER SPIMI, and we therefore call the machines that execute the map phase *parsers*.

SEGMENT FILE Each parser writes its output to local intermediate files, the *segment files* (shown as $\boxed{\text{a-f}}\boxed{\text{g-p}}\boxed{\text{q-z}}$ in Figure 1.5).

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* term partitions and having the parsers write key value pairs for each term partition into a separate segment file. In Figure 1.5, the term partitions are according to first letter: a–f, g–p, q–z, and *j* = 3. (We chose these key ranges for ease of exposition. In general, key ranges need not correspond to contiguous terms or termIDs.) The term partitions are defined by the person who operates the indexing system. The parsers then write corresponding segment files, one for each term partition. Each term partition thus corresponds to *r* segments files, where *r* is the number of parsers. For instance, Figure 1.5 shows three a–f segment files of the a–f partition, corresponding to the three parsers shown in the figure.

Collecting all values (here: docIDs) for a given key (here: termID) into one list is

INVERTER the task of the *inverters* in the reduce phase. The master assigns each term partition to a different inverter – and, as in the case of parsers, reassigns term partitions in case of failing or slow inverters. Each term partition (corresponding to *r* segment files, one on each parser) is processed by one inverter. We assume here that segment files are of a size that a single machine can handle. Finally, the list of values is sorted for each key and written to the final sorted postings list ("postings" in the figure). (Note that postings in Figure 1.6 include term frequencies, whereas each posting in the other sections of this chapter is simply a docID without term frequency information.) The data flow is shown for a–f in Figure 1.5. This completes the construction of the inverted index.



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(termID, docID)
reduce: ($\langle$termID$_1$, list(docID)$\rangle$, $\langle$termID$_2$, list(docID)$\rangle$, ...)	$\rightarrow$ (postings_list$_1$, postings_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 1.6** 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 processed further. The instantiations of the two functions and an example are shown for index construction. Because the map phase processes documents in a distributed fashion, termID–docID pairs 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.

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 its 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., of the *r* segment files of the a–f partition). Each segment file only requires one sequential read because 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 1.6 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 run in sequence. In fact, this was the design of the Google indexing system in 2004. What we describe in this section corresponds to only one of five to ten MapReduce operations in that indexing system. Another MapReduce operation transforms 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 semiautomatic method for splitting index construction into smaller tasks, it can scale to almost arbitrarily large collections, given computer clusters of sufficient size.

Exercise 1.3
For $n = 15$ splits, $r = 10$ segments, and $j = 3$ term partitions, how long would distributed index creation take for Reuters-RCV1 in a MapReduce architecture? Base your assumptions about cluster machines on Table 1.1.

1.5. 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 are modified frequently with documents being added, deleted, and updated. This means that new terms need to be added to the dictionary, and postings 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 – and 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 postings list as a separate file, then the merge simply consists of extending each postings list of the main index by the corresponding postings 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 requires up to M_{ave} disk seeks, where M_{ave} is the average size of the vocabulary of documents in the collection. With an auxiliary index, we only put additional load on the disk when we merge auxiliary and main indexes.

Unfortunately, the one-file-per-postings-list scheme is infeasible because most file systems cannot efficiently handle very large numbers of files. The simplest alternative is to store the index as one large file, that is, as a concatenation of all postings lists. In reality, we often choose a compromise between the two extremes. To simplify the discussion, we choose the simple option of storing the index as one large file here.

```
LMERGEADDTOKEN(indexes, Z_0, token)
 1  Z_0 ← MERGE(Z_0, {token})
 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 LMERGEADDTOKEN(indexes, Z_0, GETNEXTTOKEN())
```

► **Figure 1.7** Logarithmic merging. Each token (termID,docID) is initially added to in-memory index Z_0 by LMERGEADDTOKEN. LOGARITHMICMERGE initializes Z_0 and *indexes*.

In this scheme, we process each posting ⌊T/n⌋ times because we touch it during each of ⌊T/n⌋ merges where n is the size of the auxiliary index and T the total number of postings. Thus, the overall time complexity is $\Theta(T^2/n)$. (We neglect the representation of terms here and consider only the docIDs. For the purpose of time complexity, a postings list is simply a list of docIDs.)

LOGARITHMIC MERGING

We can do better than $\Theta(T^2/n)$ by introducing $\log_2(T/n)$ 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 processed only once on each level. This scheme is called *logarithmic merging* (Figure 1.7). As before, up to n postings are accumulated in an in-memory auxiliary index, which we call Z_0. When the limit n is reached, the $2^0 \times n$ postings in Z_0 are transferred to a new index I_0 that 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. Readers familiar with the binomial heap data structure will recognize its similarity with the structure of the inverted indexes in logarithmic
merging.

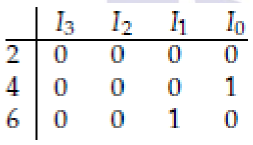
Overall index construction time is $\Theta(T \log(T/n))$ because each posting is processed only once on each of the $\log(T/n)$ levels. We trade this efficiency gain for a slowdown of query processing; we now need to merge results from $\log(T/n)$ indexes as opposed to just two (the main and auxiliary indexes). As in the auxiliary index scheme, we still need to merge very large indexes occasionally (which slows down the search system during the merge), but this happens less frequently and the indexes involved in a merge on average are smaller.

Having multiple indexes complicates the maintenance of collection-wide statistics. For example, it affects the spelling correction algorithm 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. In fact, all aspects of an IR system – index maintenance, query processing, distribution, and so on – 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.

Exercise 1.4

For $n = 2$ and $1 \leq T \leq 30$, perform a step-by-step simulation of the algorithm in Figure 1.7. Create a table that shows, for each point in time at which $T = 2 * k$ tokens have been processed ($1 \leq k \leq 15$), which of the three indexes $I_0, \ldots, I_3$ are in use. The first three lines of the table are given below.



	I_3	I_2	I_1	I_0
2	0	0	0	0
4	0	0	0	1
6	0	0	1	0

1.6. Other types of indexes

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

In the indexes we have considered so far, postings lists are ordered with respect to docID. 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* – as opposed to Boolean – *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 postings 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. In a docID-sorted index, new documents are always inserted at the end of postings lists. In an impact-sorted index, the insertion can occur anywhere, thus complicating the update of the inverted index.

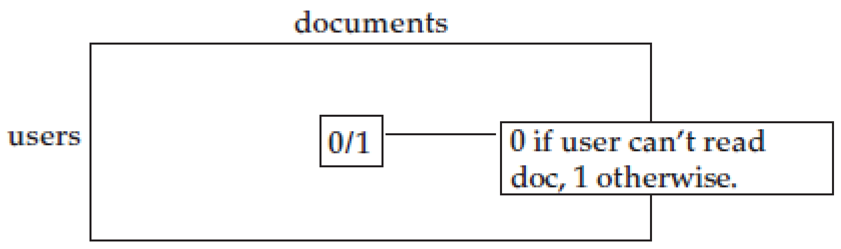
RANKED RETRIEVAL SYSTEMS

SEQURITY

Security is an important consideration for retrieval systems in corporations. A low-level 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' results lists must not contain documents they are barred from opening; 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 information retrieval system by representing each document as the set of users that can access them (Figure 1.8) and then inverting the resulting user-document matrix. The inverted ACL index has, for each user, a "postings 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 postings lists. It also requires the processing of very long postings lists for users with access to large document subsets. User membership is therefore often verified by retrieving access information directly from the file system at query time – even though this slows down retrieval.

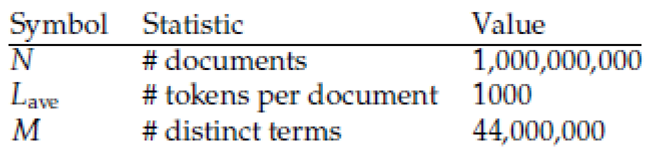


► **Figure 1.8** A user-document matrix for access control lists. Element (*i*, *j*) is 1 if user *i* has access to document *j* and 0 otherwise. During query processing, a user's access postings list is intersected with the results list returned by the text part of the index.

► **Table 1.3** The five steps in constructing an index for Reuters-RCV1 in blocked sort-based indexing. Line numbers refer to Figure 1.2.

	Step	Time
1	reading of collection (line 4)	
2	10 initial sorts of 10^7 records each (line 5)	
3	writing of 10 blocks (line 6)	
4	total disk transfer time for merging (line 7)	
5	time of actual merging (line 7)	
	total	

► **Table 1.4** Collection statistics for a large collection.



Symbol	Statistic	Value
N	# documents	1,000,000,000
L_{ave}	# tokens per document	1000
M	# distinct terms	44,000,000

Exercise 1.5
Can spelling correction compromise document-level security? Consider the case where a spelling correction is based on documents to which the user does not have access.

Exercise 1.6
Total index construction time in blocked sort-based indexing is broken down in Table 1.3. Fill out the time column of the table for Reuters-RCV1 assuming a system with the parameters given in Table 1.1.

Exercise 1.7
Repeat Exercise 1.6 for the larger collection in Table 1.4. Choose a block size that is realistic for current technology (remember that a block should easily fit into main memory). How many blocks do you need?

Exercise 1.8
Assume that machines in MapReduce have 100 GB of disk space each. Assume further that the postings list of the term the has a size of 200 GB. Then the MapReduce algorithm as described cannot be run to construct the index. How would you modify MapReduce so that it can handle this case?

Exercise 1.9
For optimal load balancing, the inverters in MapReduce must get segmented postings files of similar sizes. For a new collection, the distribution of key-value pairs may not be known in advance. How would you solve this problem?

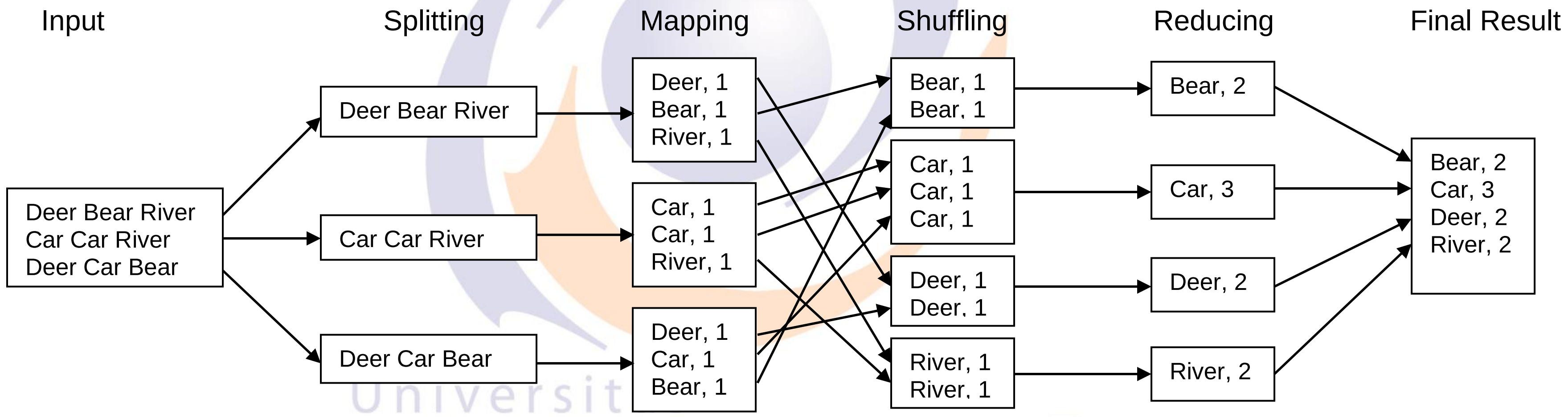
Exercise 1.10
Apply MapReduce to the problem of counting how often each term occurs in a set of files. Specify map and reduce operations for this task. Write down an example along the lines of Figure 1.6.

C. Latihan dan Jawaban

1. Penerapan MapReduce word count process.

   Input :
   - Deer Bear River
   - Car Car River
   - Deer Car Bear

   Output : ...



D. Daftar Pustaka

1. Manning, C. D., Raghavan, P., & Schutze, H. (2008). *Introduction to Information Retrieval.* Cambridge University Press.