# CS 721
# “Information Retrieval Systems”

## Chapter 4

## *Index Construction*

*This material is adapted from the book:*

Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze,
*Introduction to Information Retrieval*, Cambridge University Press. 2008.

http://www-csli.stanford.edu/~hinrich/information-retrieval-book.html

# 4. Index construction

## 4.1 Hardware basics

- When building an IR system, many decisions are based on the characteristics of the computer hardware on which the system runs.
- Performance characteristics typical of systems in 2007 are shown in Table 4.1.
- A list of hardware basics that we need to motivate IR system design follows.

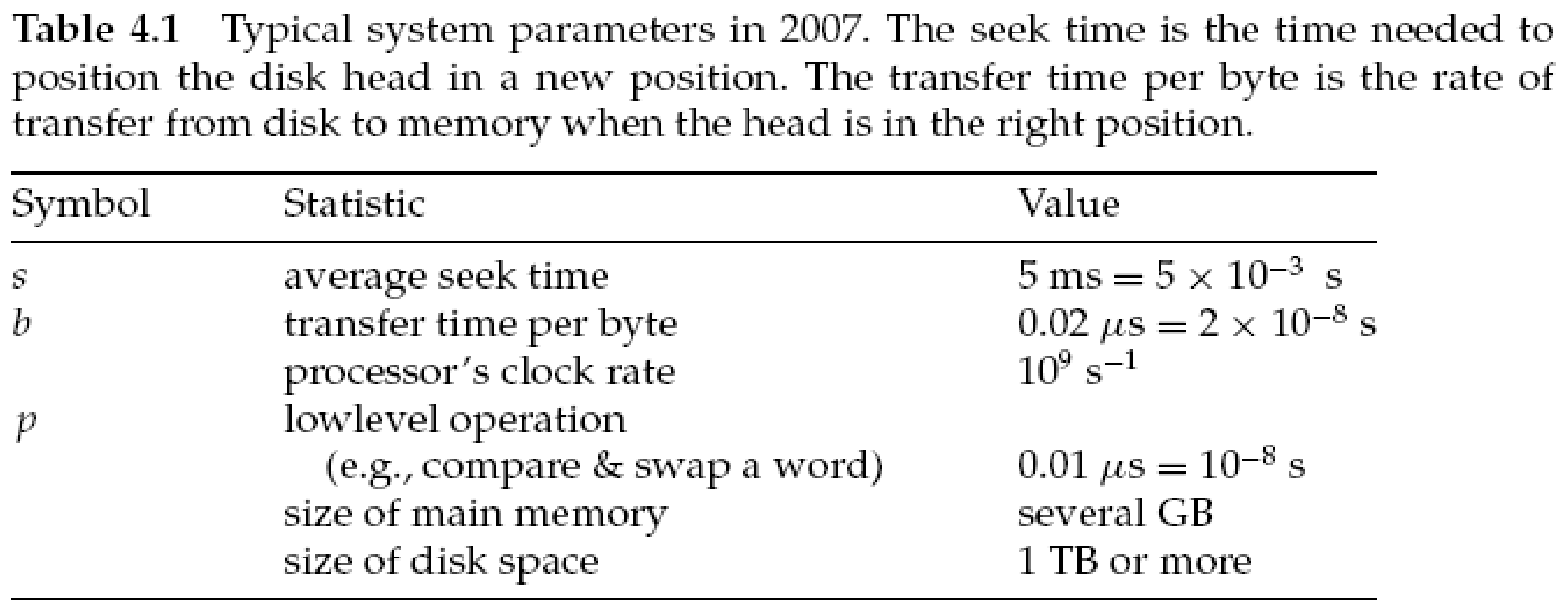


**Table 4.1** Typical system parameters in 2007. The seek time is the time needed to position the disk head in a new position. The transfer time per byte is the rate of transfer from disk to memory when the head is in the right position.

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

- *Access to data in memory is much faster than access to data on disk.*
  - We want to keep as much data as possible in memory especially those data that we need to access frequently (*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*.
  - 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.

- Operating systems generally read and write entire blocks.
  - Block sizes of 8, 16, 32, and 64 KB are common.

- 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 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.

## 4.2 Blocked sort-based indexing

- The basic steps in constructing a non-positional index are:
  1. We first make a pass through the collection assembling all term – docID pairs.
  2. We then sort the pairs with the term as the dominant key and docID as the secondary key.
  3. Finally, we organize the docIDs for each term into a postings list and compute statistics like term and document frequency.

- To make index construction more efficient, we represent terms as termIDs instead of strings termID, 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.

- We work with the *Reuters-RCV1* collection as a model collection,
  - a collection with roughly 1 GB of text.
  - It consists of about 800,000 documents.
  - covers a wide range of international topics, including politics, business, sports, and science.
  - has 100 million tokens.
    - Collecting all <termID, docID> pairs of the collection using 4 bytes each for termID and docID therefore requires 0.8 GB of storage.

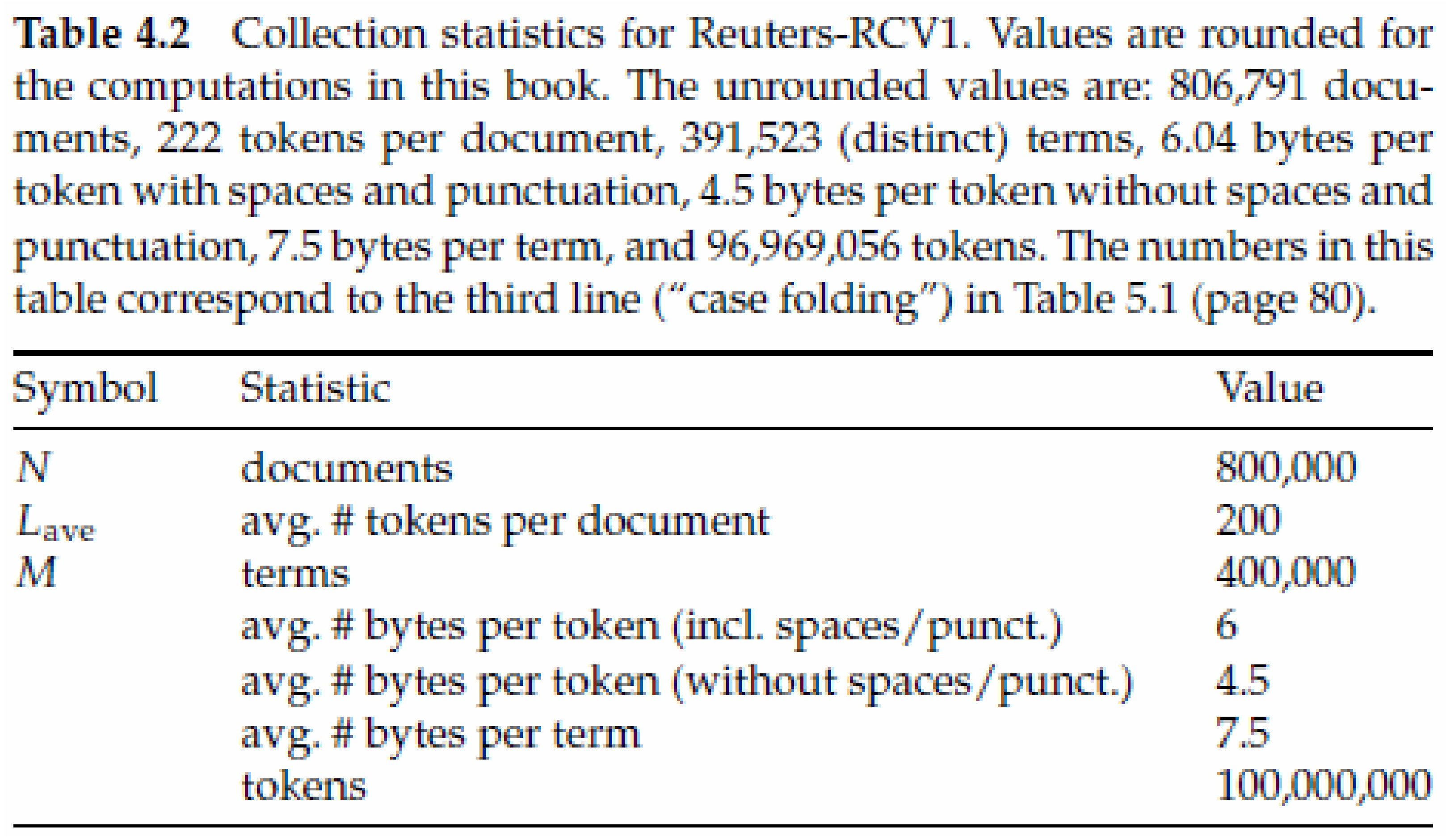
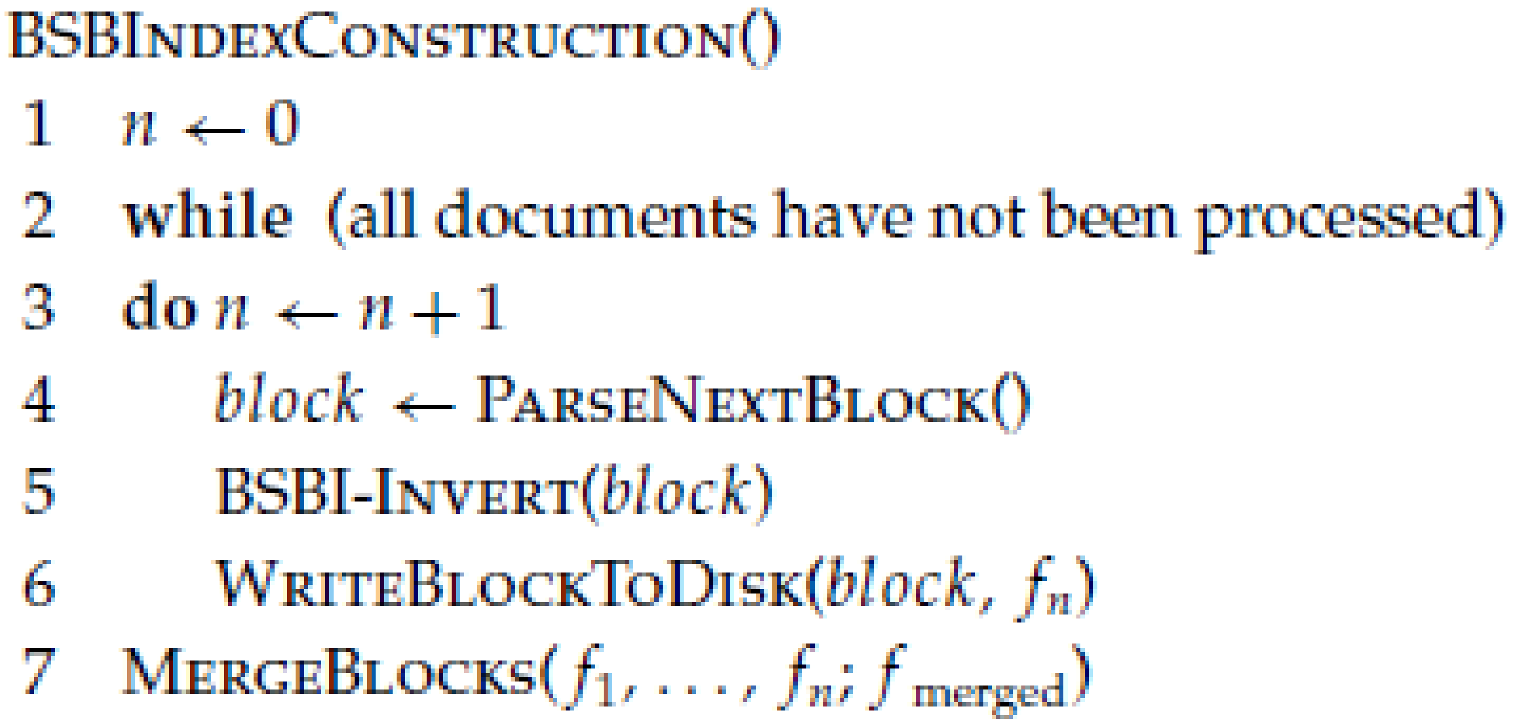


**Table 4.2** Collection statistics for Reuters-RCV1. Values are rounded for the computations in this book. The unrounded values are: 806,791 documents, 222 tokens per document, 391,523 (distinct) terms, 6.04 bytes per token with spaces and punctuation, 4.5 bytes per token without spaces and punctuation, 7.5 bytes per term, and 96,969,056 tokens. The numbers in this table correspond to the third line ("case folding") in Table 5.1 (page 80).

| Symbol | Statistic | Value |
|---|---|---|
| $N$ | documents | 800,000 |
| $L_{\text{ave}}$ | 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 |
| | tokens | 100,000,000 |

- The *blocked sort-based indexing algorithm* (*BSBI)* is shown in Figure 4.2.

- BSBI
  - segments the collection into parts of equal size,
  - sorts the <termID, docID> pairs of each part in memory,
  - stores intermediate sorted results on disk, and
  - merges all intermediate results into the final index.



```
BSBIINDEXCONSTRUCTION()
1  $n \leftarrow 0$
2  while (all documents have not been processed)
3  do $n \leftarrow n + 1$
4        $block \leftarrow$ PARSENEXTBLOCK()
5        BSBI-INVERT($block$)
6        WRITEBLOCKTODISK($block$, $f_n$)
7  MERGEBLOCKS($f_1, \ldots, f_n; f_{\text{merged}}$)
```

**Figure 4.2** Blocked sort-based indexing. The algorithm stores inverted blocks in files *f*1, . . ., *fn* and the merged index in *f* merged.

- The algorithm parses documents into <termID, docID> pairs and accumulates the pairs in memory until a block of a fixed size is full.

- 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 the algorithm 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.
- the algorithm simultaneously merges the ten blocks into one large merged index.
- An example with two blocks is shown in Figure 4.3, where we use $d_i$ to denote the *ith* 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.

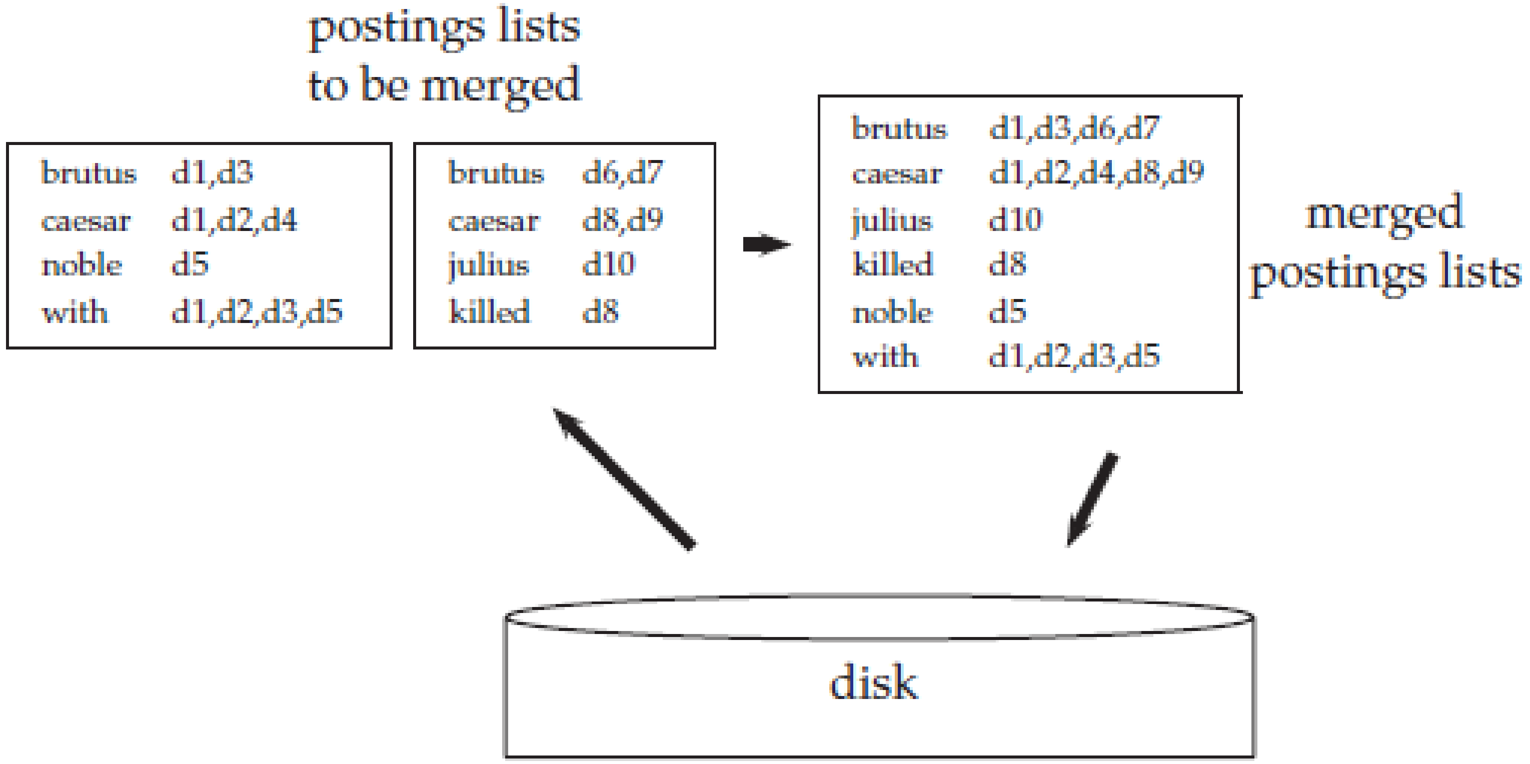


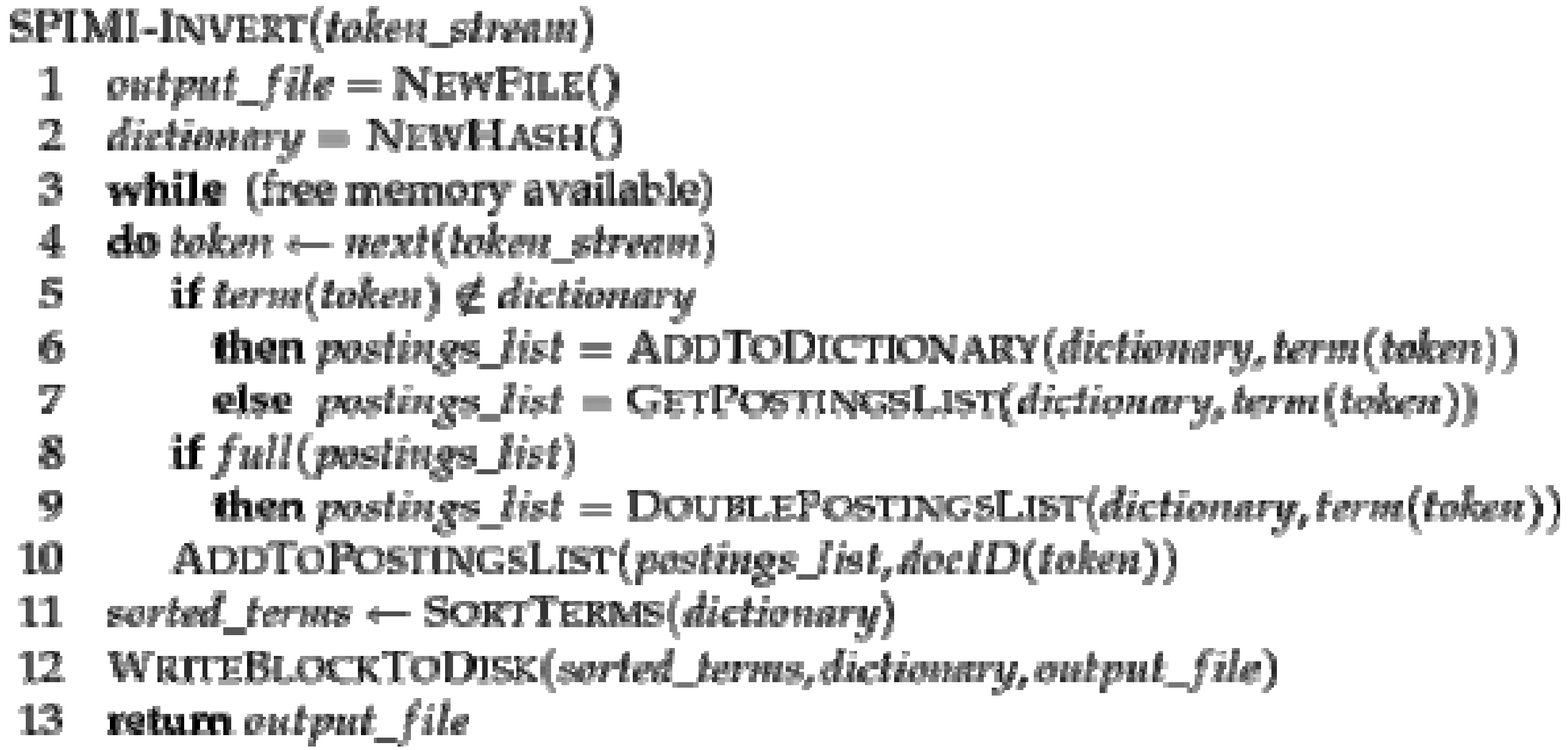
**Figure 4.3** Merging in blocked sort-based indexing. Two blocks (“postings lists to be merged”) are loaded from disk into memory, merged in memory and written back to disk.

## 4.3 Single-pass in-memory indexing

- *BSBI* 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* (*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 4.4 .
- SPIMI-INVERT is called repeatedly on the token stream until the entire collection has been processed.
- 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, and a new postings list is created (line 6). The call in line 7 returns this postings list for subsequent occurrences of the term.



```
SPIMI-INVERT(token_stream)
output_file = NEWFILE()
dictionary = NEWHASH()
while (free memory available)
do token ← next(token_stream)
    if term(token) $\notin$ dictionary
      then postings_list = ADDTODICTIONARY(dictionary, term(token))
      else postings_list = GETPOSTINGSLIST(dictionary, term(token))
    if full(postings_list)
      then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
    ADDTOPOSTINGSLIST(postings_list, docID(token))
sorted_terms ← SORTTERMS(dictionary)
WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
return output_file
```

**Figure 4.4 :**Inversion of a block in single-pass in-memory indexing

- 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 are 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* (no sorting required), and
  – it *saves memory* (we keep track of the term a postings list belongs to, so the termIDs of postings need not be stored.)

- We allocate space for a short postings list initially and double the space each time it is full (lines 8-9).

- When memory has been exhausted, we write the index of the block 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.

- Each call of SPIMI-INVERT writes a block to disk, just as in BSBI. The last step of SPIMI 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.

## 4.4 Distributed indexing

- Collections are often so large that we cannot perform index construction efficiently on a single machine. This is particularly true of the www for which we need large computer *clusters* to construct any reasonably sized web index.
- Web search engines 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*.
- Most large search engines prefer a *document-partitioned index* (which can be easily generated from a term-partitioned index).

- The distributed index construction method 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.

- One requirement for robust distributed indexing is 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.

- 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 4.5 and an example on a collection consisting of two documents is shown in Figure 4.6 .

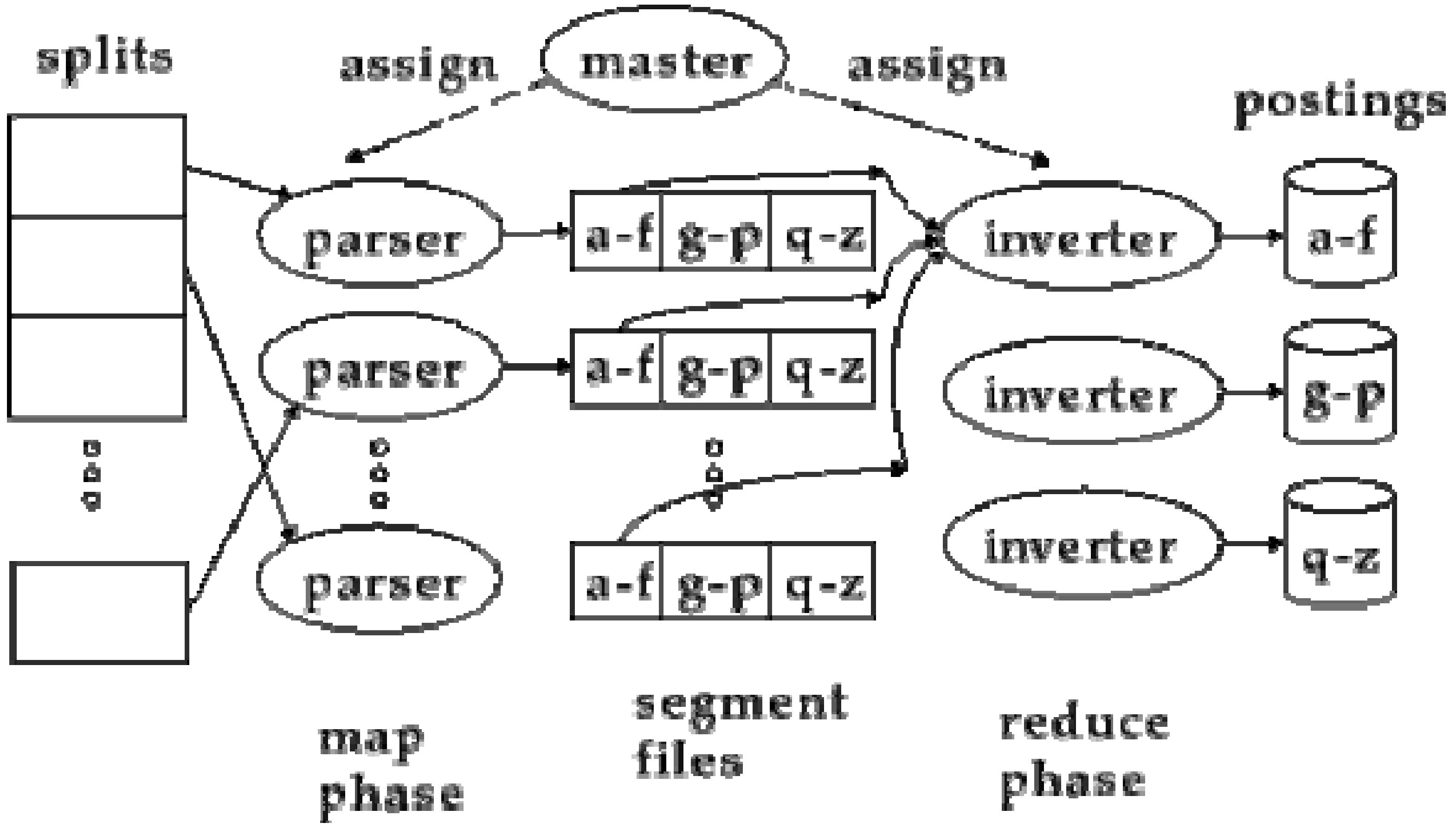
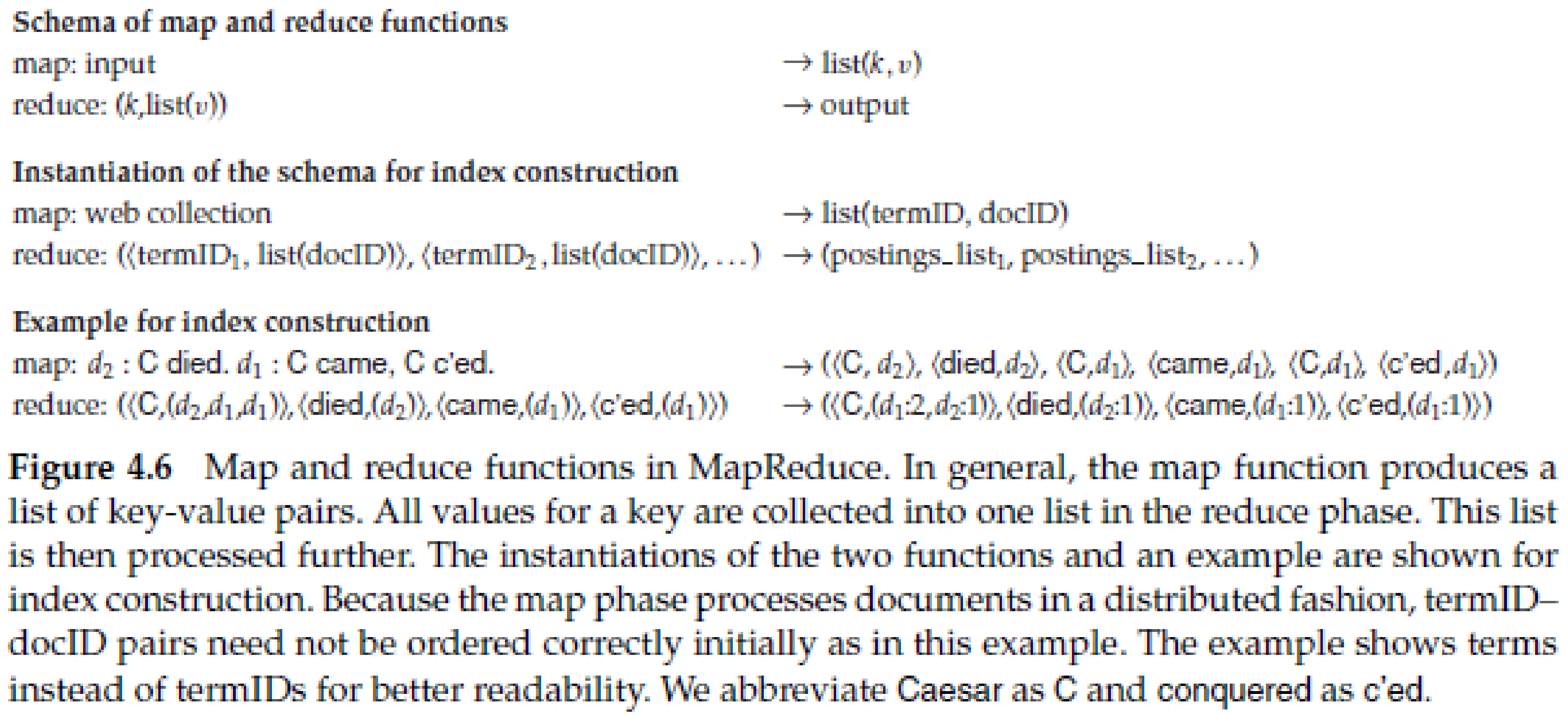


**Figure 4.5 :**An example of distributed indexing with MapReduce.



**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 4.6** 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.

- First, the input data is splits 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 64 MB are good sizes in distributed indexing.

- Splits are not pre-assigned 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.

- MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of *key-value pairs*.

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

- Each parser writes its output to local intermediate files, the *segment files* (shown as a-f, g-p, q-z in Figure 4.5 ).

- 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 4.5 , the term partitions are according to first letter: a-f, g-p, q-z, and *j = 3* .

- 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.

- 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 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.

- Finally, the list of values is sorted for each key and written to the final sorted postings list ("postings" in the figure).

## 4.5 Dynamic indexing

- 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*.

- 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.