4 Static Inverted Indices

In this chapter we describe a set of index structures that are suitable for supporting search queries of the type outlined in Chapter 2. We restrict ourselves to the case of static text collections. That is, we assume that we are building an index for a collection that never changes. Index update strategies for dynamic text collections, in which documents can be added to and removed from the collection, are the topic of Chapter 7.

For performance reasons, it may be desirable to keep the index for a text collection completely in main memory. However, in many applications this is not feasible. In file system search, for example, a full-text index for all data stored in the file system can easily consume several gigabytes. Since users will rarely be willing to dedicate the most part of their available memory resources to the search system, it is not possible to keep the entire index in RAM. And even for dedicated index servers used in Web search engines it might be economically sensible to store large portions of the index on disk instead of in RAM, simply because disk space is so much cheaper than RAM. For example, while we are writing this, a gigabyte of RAM costs around $40 (U.S.), whereas a gigabyte of hard drive space costs only about $0.20 (U.S.). This factor-200 price difference, however, is not reflected by the relative performance of the two storage media. For typical index operations, an in-memory index is usually between 10 and 20 times faster than an on-disk index. Hence, when building two equally priced retrieval systems, one storing its index data on disk, the other storing them in main memory, the disk-based system may actually be faster than the RAM-based one (see Bender et al. (2007) for a more in-depth discussion of this and related issues).

The general assumption that will guide us throughout this chapter is that main memory is a scarce resource, either because the search engine has to share it with other processes running on the same system or because it is more economical to store data on disk than in RAM. In our discussion of data structures for inverted indices, we focus on hybrid organizations, in which some parts of the index are kept in main memory, while the majority of the data is stored on disk. We examine a variety of data structures for different parts of the search engine and evaluate their performance through a number of experiments. A performance summary of the computer system used to conduct these experiments can be found in the appendix.

4.1 Index Components and Index Life Cycle

When we discuss the various aspects of inverted indices in this chapter, we look at them from two perspectives: the structural perspective, in which we divide the system into its components and examine aspects of an individual component of the index (e.g., an individual postings list); and the operational perspective, in which we look at different phases in the life cycle of an inverted index and discuss the essential index operations that are carried out in each phase (e.g., processing a search query).

As already mentioned in Chapter 2, the two principal components of an inverted index are the *dictionary* and the *postings lists*. For each term in the text collection, there is a postings list that contains information about the term's occurrences in the collection. The information found in these postings lists is used by the system to process search queries. The dictionary serves as a lookup data structure on top of the postings lists. For every query term in an incoming search query, the search engine first needs to locate the term's postings list before it can start processing the query. It is the job of the dictionary to provide this mapping from terms to the location of their postings lists in the index.

In addition to dictionary and postings lists, search engines often employ various other data structures. Many engines, for instance, maintain a *document map* that, for each document in the index, contains document-specific information, such as the document's URL, its length, PageRank (see Section 15.3.1), and other data. The implementation of these data structures, however, is mostly straightforward and does not require any special attention.

The life cycle of a static inverted index, built for a never-changing text collection, consists of two distinct phases (for a dynamic index the two phases coincide):

1. *Index construction*: The text collection is processed sequentially, one token at a time, and a postings list is built for each term in the collection in an incremental fashion.
2. *Query processing*: The information stored in the index that was built in phase 1 is used to process search queries.

Phase 1 is generally referred to as *indexing time*, while phase 2 is referred to as *query time*. In many respects these two phases are complementary; by performing additional work at indexing time (e.g., precomputing score contributions — see Section 5.1.3), less work needs to be done at query time. In general, however, the two phases are quite different from one another and usually require different sets of algorithms and data structures. Even for subcomponents of the index that are shared by the two phases, such as the search engine's dictionary data structure, it is not uncommon that the specific implementation utilized during index construction is different from the one used at query time.

The flow of this chapter is mainly defined by our bifocal perspective on inverted indices. In the first part of the chapter (Sections 4.2–4.4), we are primarily concerned with the query-time aspects of dictionary and postings lists, looking for data structures that are most suitable for supporting efficient index access and query processing. In the second part (Section 4.5) we focus on aspects of the index construction process and discuss how we can efficiently build the data structures outlined in the first part. We also discuss how the organization of the dictionary and the postings lists needs to be different from the one suggested in the first part of the chapter, if we want to maximize their performance at indexing time.

For the sake of simplicity, we assume throughout this chapter that we are dealing exclusively with *schema-independent* indices. Other types of inverted indices, however, are similar to the schema-independent variant, and the methods discussed in this chapter apply to all of them (see Section 2.1.3 for a list of different types of inverted indices).

4.2 The Dictionary

The *dictionary* is the central data structure that is used to manage the set of terms found in a text collection. It provides a mapping from the set of index terms to the locations of their postings lists. At query time, locating the query terms' postings lists in the index is one of the first operations performed when processing an incoming keyword query. At indexing time, the dictionary's lookup capability allows the search engine to quickly obtain the memory address of the inverted list for each incoming term and to append a new posting at the end of that list.

Dictionary implementations found in search engines usually support the following set of operations:

1. Insert a new entry for term T.
2. Find and return the entry for term T (if present).
3. Find and return the entries for all terms that start with a given prefix P.

When building an index for a text collection, the search engine performs operations of types 1 and 2 to look up incoming terms in the dictionary and to add postings for these terms to the index. After the index has been built, the search engine can process search queries, performing operations of types 2 and 3 to locate the postings lists for all query terms. Although dictionary operations of type 3 are not strictly necessary, they are a useful feature because they allow the search engine to support *prefix queries* of the form "inform∗", matching all documents containing a term that begins with "inform".

Table 4.1 Index sizes for various index types and three example collections, with and without applying index compression techniques. In each case the first number refers to an index in which each component is stored as a simple 32-bit integer, while the second number refers to an index in which each entry is compressed using a byte-aligned encoding method.

	Shakespeare	**TREC45**	**GOV2**
Number of tokens	1.3×10^6	3.0×10^8	4.4×10^{10}
Number of terms	2.3×10^4	1.2×10^6	4.9×10^7
Dictionary (uncompr.)	0.4 MB	24 MB	1046 MB
Docid index	n/a	578 MB/200 MB	37751 MB/12412 MB
Frequency index	n/a	1110 MB/333 MB	73593 MB/21406 MB
Positional index	n/a	2255 MB/739 MB	245538 MB/78819 MB
Schema-ind. index	5.7 MB/2.7 MB	1190 MB/532 MB	173854 MB/63670 MB

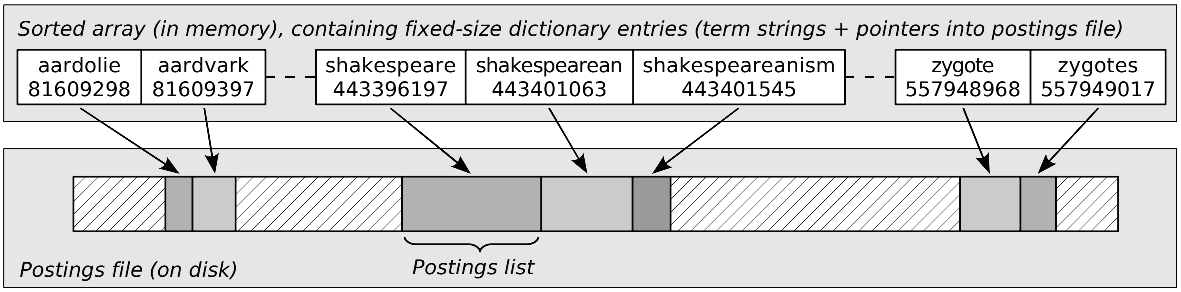

Figure 4.1 Dictionary data structure based on a sorted array (data extracted from a schema-independent index for TREC45). The array contains fixed-size dictionary entries, composed of a zero-terminated string and a pointer into the postings file that indicates the position of the term's postings list.

For a typical natural-language text collection, the dictionary is relatively small compared to the total size of the index. Table 4.1 shows this for the three example collections used in this book. The size of the uncompressed dictionary is only between 0.6% (GOV2) and 7% (Shakespeare) of the size of an uncompressed schema-independent index for the respective collection (the fact that the relative size of the dictionary is smaller for large collections than for small ones follows directly from Zipf's law — see Equation 1.2 on page 16). We therefore assume, at least for now, that the dictionary is small enough to fit completely into main memory.

The two most common ways to realize an in-memory dictionary are:

- A *sort-based* dictionary, in which all terms that appear in the text collection are arranged in a sorted array or in a search tree, in lexicographical (i.e., alphabetical) order, as shown in Figure 4.1. Lookup operations are realized through tree traversal (when using a search tree) or binary search (when using a sorted list).
- A *hash-based* dictionary, in which each index term has a corresponding entry in a hash table. Collisions in the hash table (i.e., two terms are assigned the same hash value) are resolved by means of *chaining* — terms with the same hash value are arranged in a linked list, as shown in Figure 4.2.

Storing the dictionary terms

When implementing the dictionary as a sorted array, it is important that all array entries are of the same size. Otherwise, performing a binary search may be difficult. Unfortunately, this causes some problems. For example, the longest sequence of alphanumeric characters in GOV2 (i.e., the longest term in the collection) is 74,147 bytes long. Obviously, it is not feasible to allocate 74 KB of memory for each term in the dictionary. But even if we ignore such extreme outliers and truncate each term after, say, 20 bytes, we are still wasting precious memory resources. Following the simple tokenization procedure from Section 1.3.2, the average length of a term

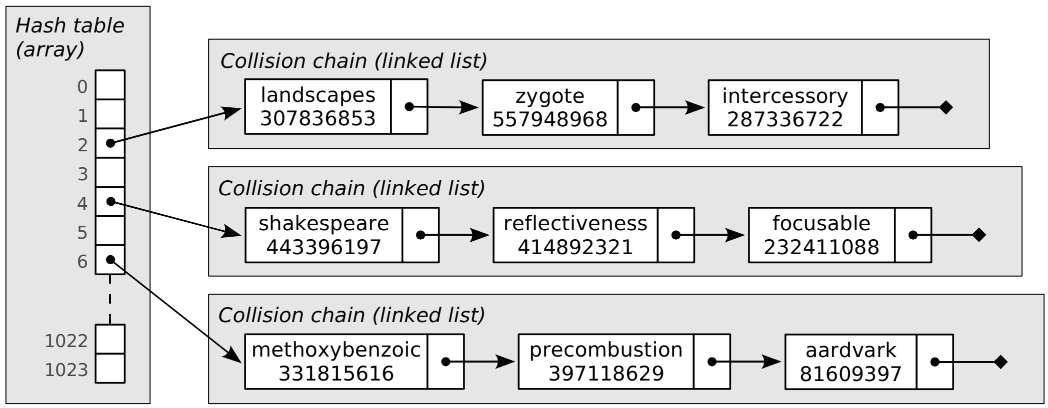

Figure 4.2 Dictionary data structure based on a hash table with $2^{10} = 1024$ entries (data extracted from schema-independent index for TREC45). Terms with the same hash value are arranged in a linked list (*chaining*). Each term descriptor contains the term itself, the position of the term's postings list, and a pointer to the next entry in the linked list.

in GOV2 is 9.2 bytes. Storing each term in a fixed-size memory region of 20 bytes wastes 10.8 bytes per term on average (*internal fragmentation*).

One way to eliminate the internal fragmentation is to not store the index terms themselves in the array, but only pointers to them. For example, the search engine could maintain a *primary* dictionary array, containing 32-bit pointers into a *secondary* array. The secondary array then contains the actual dictionary entries, consisting of the terms themselves and the corresponding pointers into the postings file. This way of organizing the search engine's dictionary data is shown in Figure 4.3. It is sometimes referred to as the *dictionary-as-a-string approach*, because there are no explicit delimiters between two consecutive dictionary entries; the secondary array can be thought of as a long, uninterrupted string.

For the GOV2 collection, the dictionary-as-a-string approach, compared to the dictionary layout shown in Figure 4.1, reduces the dictionary's storage requirements by $10.8 - 4 = 6.8$ bytes per entry. Here the term 4 stems from the pointer overhead in the primary array; the term 10.8 corresponds to the complete elimination of any internal fragmentation.

It is worth pointing out that the term strings stored in the secondary array do not require an explicit termination symbol (e.g., the "\0" character), because the length of each term in the dictionary is implicitly given by the pointers in the primary array. For example, by looking at the pointers for "shakespeare" and "shakespearean" in Figure 4.3, we know that the dictionary entry for "shakespeare" requires $16629970 - 16629951 = 19$ bytes in total: 11 bytes for the term plus 8 bytes for the 64-bit file pointer into the postings file.

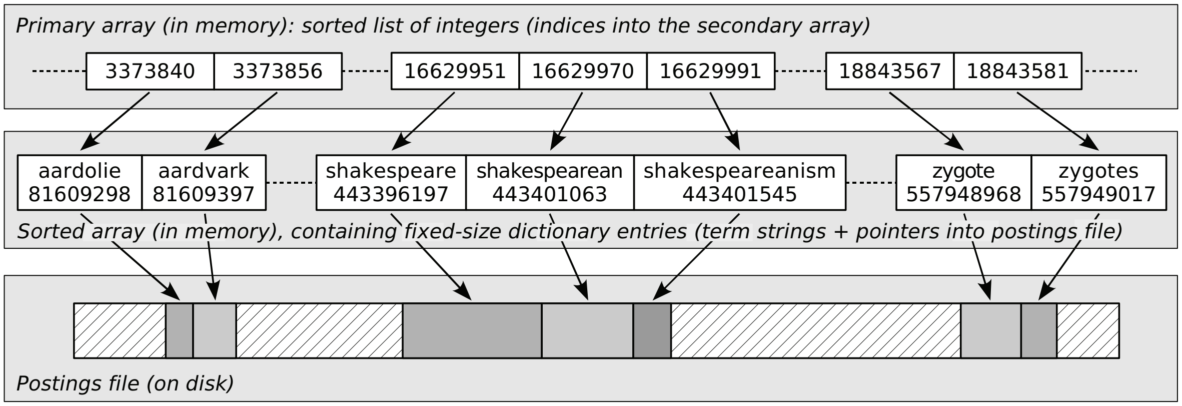

Figure 4.3 Sort-based dictionary data structure with an additional level of indirection (the so-called *dictionary-as-a-string* approach).

Sort-based versus hash-based dictionary

For most applications a hash-based dictionary will be faster than a sort-based implementation, as it does not require a costly binary search or the traversal of a path in a search tree to find the dictionary entry for a given term. The precise speed advantage of a hash-based dictionary over a sort-based dictionary depends on the size of the hash table. If the table is too small, then there will be many collisions, potentially reducing the dictionary's performance substantially. As a rule of thumb, in order to keep the lengths of the collision chains in the hash table small, the size of the table should grow linearly with the number of terms in the dictionary.

Table 4.2 Lookup performance at query time. Average latency of a single-term lookup for a sort-based (shown in Figure 4.3) and a hash-based (shown in Figure 4.2) dictionary implementation. For the hash-based implementation, the size of the hash table (number of array entries) is varied between 2^{18} ($\approx$ 262,000) and 2^{24} ($\approx$ 16.8 million).

	Sorted	**Hashed (2^{18})**	**Hashed (2^{20})**	**Hashed (2^{22})**	**Hashed (2^{24})**
Shakespeare	0.32 μs	0.11 μs	0.13 μs	0.14 μs	0.16 μs
TREC45	1.20 μs	0.53 μs	0.34 μs	0.27 μs	0.25 μs
GOV2	2.79 μs	19.8 μs	5.80 μs	2.23 μs	0.84 μs

Table 4.2 shows the average time needed to find the dictionary entry for a random index term in each of our three example collections. A larger table usually results in a shorter lookup time, except for the *Shakespeare* collection, which is so small (23,000 different terms) that the effect of decreasing the term collisions is outweighed by the less efficient CPU cache utilization. A bigger hash table in this case results in an increased lookup time. Nonetheless, if the table size is chosen properly, a hash-based dictionary is usually at least twice as fast as a sort-based one.

Unfortunately, this speed advantage for single-term dictionary lookups has a drawback: A sort-based dictionary offers efficient support for prefix queries (e.g., "inform∗"). If the dictionary is based on a sorted array, for example, then a prefix query can be realized through two binary search operations, to find the first term T_j and the last term T_k matching the given prefix query, followed by a linear scan of all $k - j + 1$ dictionary entries between T_j and T_k. The total time complexity of this procedure is

$$\Theta(\log(|\mathcal{V}|)) + \Theta(m), \tag{4.1}$$

where $m = k - j + 1$ is the number of terms matching the prefix query and $\mathcal{V}$ is the vocabulary of the search engine.

If prefix queries are to be supported by a hash-based dictionary implementation, then this can be realized only through a linear scan of all terms in the hash table, requiring $\Theta(|\mathcal{V}|)$ string comparisons. It is therefore not unusual that a search engine employs two different dictionary data structures: a hash-based dictionary, used during the index construction process and providing efficient support of operations 1 (insert) and 2 (single-term lookup), and a sort-based dictionary that is created after the index has been built and that provides efficient support of operations 2 (single-term lookup) and 3 (prefix lookup).

The distinction between an indexing-time and a query-time dictionary is further motivated by the fact that support for high-performance single-term dictionary lookups is more important during index construction than during query processing. At query time the overhead associated with finding the dictionary entries for all query terms is negligible (a few microseconds) and is very likely to be outweighed by the other computations that have to be performed while processing a query. At indexing time, however, a dictionary lookup needs to be performed for every token in the text collection — 44 billion lookup operations in the case of GOV2. Thus, the dictionary represents a major bottleneck in the index construction process, and lookups should be as fast as possible.

4.3 Postings Lists

The actual index data, used during query processing and accessed through the search engine's dictionary, is stored in the index's postings lists. Each term's postings list contains information about the term's occurrences in the collection. Depending on the type of the index (docid, frequency, positional, or schema-independent — see Section 2.1.3), a term's postings list contains more or less detailed, and more or less storage-intensive, information. Regardless of the actual type of the index, however, the postings data always constitute the vast majority of all the data in the index. In their entirety, they are therefore usually too large to be stored in main memory and have to be kept on disk. Only during query processing are the query terms' postings lists (or small parts thereof) loaded into memory, on a by-need basis, as required by the query processing routines.

To make the transfer of postings from disk into main memory as efficient as possible, each term's postings list should be stored in a contiguous region of the hard drive. That way, when accessing the list, the number of disk seek operations is minimized. The hard drives of the computer system used in our experiments (summarized in the appendix) can read about half a megabyte of data in the time it takes to perform a single disk seek, so discontiguous postings lists can reduce the system's query performance dramatically.

Random list access: The per-term index

The search engine's list access pattern at query time depends on the type of query being processed. For some queries, postings are accessed in an almost strictly sequential fashion. For other queries it is important that the search engine can carry out efficient random access operations on the postings lists. An example of the latter type is phrase search or — equivalently — conjunctive Boolean search (processing a phrase query on a schema-independent index is essentially the same as resolving a Boolean AND on a docid index).

Recall from Chapter 2 the two main access methods provided by an inverted index: **next** and **prev**, returning the first (or last) occurrence of the given term after (or before) a given index address. Suppose we want to find all occurrences of the phrase "iterative binary search" in GOV2. After we have found out that there is exactly one occurrence of "iterative binary" in the collection, at position [33,399,564,886, 33,399,564,887], a single call to

$$\textbf{next}(\text{“search”},\ 33{,}399{,}564{,}887)$$

will tell us whether the phrase "iterative binary search" appears in the corpus. If the method returns 33,399,564,888, then the answer is yes. Otherwise, the answer is no.

If postings lists are stored in memory, as arrays of integers, then this operation can be performed very efficiently by conducting a binary search (or galloping search — see Section 2.1.2) on the postings array for "search". Since the term "search" appears about 50 million times in GOV2, the binary search requires a total of

$$\lceil \log_2(5 \times 10^7) \rceil = 26$$

random list accesses. For an on-disk postings list, the operation could theoretically be carried out in the same way. However, since a hard disk is not a true random access device, and a disk seek is a very costly operation, such an approach would be prohibitively expensive. With its 26 random disk accesses, a binary search on the on-disk postings list can easily take more than 200 milliseconds, due to seek overhead and rotational latency of the disk platter.

As an alternative, one might consider loading the entire postings list into memory in a single sequential read operation, thereby avoiding the expensive disk seeks. However, this is not a good solution, either. Assuming that each posting requires 8 bytes of disk space, it would take our computer more than 4 seconds to read the term's 50 million postings from disk.

list header

per-term index (5 postings)

TF: 27	239539	242435	248080	255731	281080
239539	239616	239732	239765	240451	242395
242435	242659	243223	243251	245282	247589
248080	248526	248803	249056	254313	254350
255731	256428	264780	271063	272125	279107
281080	281793	284087			

Figure 4.4 Schema-independent postings list for "denmark" (extracted from the Shakespeare collection) with per-term index: one synchronization point for every six postings. The number of synchronization points is implicit from the length of the list: $\lceil 27/6 \rceil = 5$.

In order to provide efficient random access into any given on-disk postings list, each list has to be equipped with an auxiliary data structure, which we refer to as the *per-term index*. This data structure is stored on disk, at the beginning of the respective postings list. It contains a copy of a subset of the postings in the list, for instance, a copy of every 5,000th posting. When accessing the on-disk postings list for a given term T, before performing any actual index operations on the list, the search engine loads T's per-term index into memory. Random-access operations of the type required by the **next** method can then be carried out by performing a binary search on the in-memory array representing T's per-term index (identifying a candidate range of 5,000 postings in the on-disk postings list), followed by loading up to 5,000 postings from the candidate range into memory and then performing a random access operation on those postings.

This approach to random access list operations is sometimes referred to as *self-indexing* (Moffat and Zobel, 1996). The entries in the per-term index are called *synchronization points*. Figure 4.4 shows the postings list for the term "denmark", extracted from the Shakespeare collection, with a per-term index of granularity 6 (i.e., one synchronization point for every six postings in the list). In the figure, a call to **next**(250,000) would first identify the postings block starting with 248,080 as potentially containing the candidate posting. It would then load this block into memory, carry out a binary search on the block, and return 254,313 as the answer. Similarly, in our example for the phrase query "iterative binary search", the random access operation into the list for the term "search" would be realized using only 2 disk seeks and loading a total of about 15,000 postings into memory (10,000 postings for the per-term index and 5,000 postings from the candidate range) — translating into a total execution time of approximately 30 ms.

Choosing the granularity of the per-term index, that is, the number of postings between two synchronization points, represents a trade-off. A greater granularity increases the amount of data between two synchronization points that need to be loaded into memory for every random access operation; a smaller granularity, conversely, increases the size of the per-term index and thus the amount of data read from disk when initializing the postings list (Exercise 4.1 asks you to calculate the optimal granularity for a given list).

In theory it is conceivable that, for a very long postings list containing billions of entries, the optimal per-term index (with a granularity that minimizes the total amount of disk activity) becomes so large that it is no longer feasible to load it completely into memory. In such a situation it is possible to build an index for the per-term index, or even to apply the whole procedure recursively. In the end this leads to a multi-level static B-tree that provides efficient random access into the postings list. In practice, however, such a complicated data structure is rarely necessary. A simple two-level structure, with a single per-term index for each on-disk postings list, is sufficient. The term "the", for instance, the most frequent term in the GOV2 collection, appears roughly 1 billion times in the collection. When stored uncompressed, its postings list in a schema-independent index consumes about 8 billion bytes (8 bytes per posting). Suppose the per-term index for "the" contains one synchronization point for every 20,000 postings. Loading the per-term index with its 50,000 entries into memory requires a single disk seek, followed by a sequential transfer of 400,000 bytes ($\approx$ 4.4 ms). Each random access operation into the term's list requires an additional disk seek, followed by loading 160,000 bytes ($\approx$ 1.7 ms) into RAM. In total, therefore, a single random access operation into the term's postings list requires about 30 ms (two random disk accesses, each taking about 12 ms, plus reading 560,000 bytes from disk). In comparison, adding an additional level of indexing, by building an index for the per-term index, would increase the number of disk seeks required for a single random access to at least three, and would therefore most likely decrease the index's random access performance.

Compared to an implementation that performs a binary search directly on the on-disk postings list, the introduction of the per-term index improves the performance of random access operations quite substantially. The true power of the method, however, lies in the fact that it allows us to store postings of variable length, for example postings of the form (*docid, tf,* ⟨*positions*⟩), and in particular compressed postings. If postings are stored not as fixed-size (e.g., 8-byte) integers, but in compressed form, then a simple binary search is no longer possible. However, by compressing postings in small chunks, where the beginning of each chunk corresponds to a synchronization point in the per-term index, the search engine can provide efficient random access even for compressed postings lists. This application also explains the choice of the term "synchronization point": a synchronization point helps the decoder establish synchrony with the encoder, thus allowing it to start decompressing data at an (almost) arbitrary point within the compressed postings sequence. See Chapter 6 for details on compressed inverted indices.

Prefix queries

If the search engine has to support prefix queries, such as "inform∗", then it is imperative that postings lists be stored in lexicographical order of their respective terms. Consider the GOV2 collection; 4,365 different terms with a total of 67 million occurrences match the prefix query "inform∗". By storing lists in lexicographical order, we ensure that the inverted lists for these 4,365 terms are close to each other in the inverted file and thus close to each other on disk. This decreases the seek distance between the individual lists and leads to better query performance. If lists were stored on disk in some random order, then disk seeks and rotational latency alone would account for almost a minute (4,365 × 12 ms), not taking into account any of the other operations that need to be carried out when processing the query. By arranging the inverted lists in lexicographical order of their respective terms, a query asking for all documents matching "inform∗" can be processed in less than 2 seconds when using a frequency index; with a schema-independent index, the same query takes about 6 seconds. Storing the lists in the inverted file in some predefined order (e.g., lexicographical) is also important for efficient index updates, as discussed in Chapter 7.

A separate positional index

If the search engine is based on a document-centric positional index (containing a docid, a frequency value, and a list of within-document positions for each document that a given term appears in), it is not uncommon to divide the index data into two separate inverted files: one file containing the docid and frequency component of each posting, the other file containing the exact within-document positions. The rationale behind this division is that for many queries — and many scoring functions — access to the positional information is not necessary. By excluding it from the main index, query processing performance can be increased.

4.4 Interleaving Dictionary and Postings Lists

For many text collections the dictionary is small enough to fit into the main memory of a single machine. For large collections, however, containing many millions of different terms, even the collective size of all dictionary entries might be too large to be conveniently stored in RAM. The GOV2 collection, for instance, contains about 49 million distinct terms. The total size of the concatenation of these 49 million terms (if stored as zero-terminated strings) is 482 MB. Now suppose the dictionary data structure used in the search engine is based on a sorted array, as shown in Figure 4.3. Maintaining for each term in the dictionary an additional 32-bit pointer in the primary sorted array and a 64-bit file pointer in the secondary array increases the overall memory consumption by another 12×49 = 588 million bytes (approximately), leading to a total memory requirement of 1046 MB. Therefore, although the GOV2 collection is small enough to be managed by a single machine, the dictionary may be too large to fit into the machine's main memory.

To some extent, this problem can be addressed by employing dictionary compression techniques (discussed in Section 6.4). However, dictionary compression can get us only so far. There exist situations in which the number of distinct terms in the text collection is so enormous that it becomes impossible to store the entire dictionary in main memory, even after compression. Consider, for example, an index in which each postings list represents not an individual term but a term bigram, such as "information retrieval". Such an index is very useful for processing

Table 4.3 Number of unique terms, term bigrams, and trigrams for our three text collections. The number of unique bigrams is much larger than the number of unique terms, by about one order of magnitude.

	Tokens	Unique Words	Unique Bigrams	Unique Trigrams
Shakespeare	1.3×10^6	2.3×10^4	2.9×10^5	6.5×10^5
TREC45	3.0×10^8	1.2×10^6	2.5×10^7	9.4×10^7
GOV2	4.4×10^{10}	4.9×10^7	5.2×10^8	2.3×10^9

phrase queries. Unfortunately, the number of unique bigrams in a text collection is substantially larger than the number of unique terms. Table 4.3 shows that GOV2 contains only about 49 million distinct terms, but 520 million distinct term bigrams. Not surprisingly, if trigrams are to be indexed instead of bigrams, the situation becomes even worse — with 2.3 billion different trigrams in GOV2, it is certainly not feasible to keep the entire dictionary in main memory anymore.

Storing the entire dictionary on disk would satisfy the space requirements but would slow down query processing. Without any further modifications an on-disk dictionary would add at least one extra disk seek per query term, as the search engine would first need to fetch each term's dictionary entry from disk before it could start processing the given query. Thus, a pure on-disk approach is not satisfactory, either.

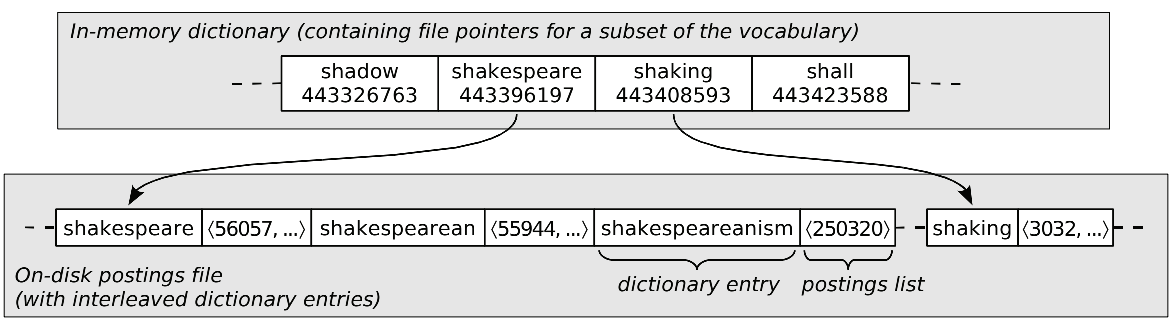

Figure 4.5 Interleaving dictionary and postings lists: Each on-disk inverted list is immediately preceded by the dictionary entry for the respective term. The in-memory dictionary contains entries for only *some* of the terms. In order to find the postings list for "shakespeareanism", a sequential scan of the on-disk data between "shakespeare" and "shaking" is necessary.

A possible solution to this problem is called *dictionary interleaving*, shown in Figure 4.5. In an interleaved dictionary all entries are stored on disk, each entry right before the respective postings list, to allow the search engine to fetch dictionary entry and postings list in one sequential read operation. In addition to the on-disk data, however, copies of *some* dictionary entries

Table 4.4 The impact of dictionary interleaving on a schema-independent index for GOV2 (49.5 million distinct terms). By choosing an index block size B = 16,384 bytes, the number of in-memory dictionary entries can be reduced by over 99%, at the cost of a minor query slowdown: 1 ms per query term.

Index Block Size (in bytes)	1,024	4,096	16,384	65,536	262,144
No. of in-memory dict. entries ($\times 10^6$)	3.01	0.91	0.29	0.10	0.04
Avg. index access latency (in ms)	11.4	11.6	12.3	13.6	14.9

(but not all of them) are kept in memory. When the search engine needs to determine the location of a term's postings list, it first performs a binary search on the sorted list of in-memory dictionary entries, followed by a sequential scan of the data found between two such entries. For the example shown in the figure, a search for "shakespeareanism" would first determine that the term's postings list (if it appears in the index) must be between the lists for "shakespeare" and "shaking". It would then load this index range into memory and scan it in a linear fashion to find the dictionary entry (and thus the postings list) for the term "shakespeareanism".

Dictionary interleaving is very similar to the self-indexing technique from Section 4.3, in the sense that random access disk operations are avoided by reading a little bit of extra data in a sequential manner. Because sequential disk operations are so much faster than random access, this trade-off is usually worthwhile, as long as the additional amount of data transferred from disk into main memory is small. In order to make sure that this is the case, we need to define an upper limit for the amount of data found between each on-disk dictionary entry and the closest preceding in-memory dictionary entry. We call this upper limit the *index block size*. For instance, if it is guaranteed for every term T in the index that the search engine does not need to read more than 1,024 bytes of on-disk data before it reaches T's on-disk dictionary entry, then we say that the index has a block size of 1,024 bytes.

Table 4.4 quantifies the impact that dictionary interleaving has on the memory consumption and the list access performance of the search engine (using GOV2 as a test collection). Without interleaving, the search engine needs to maintain about 49.5 million in-memory dictionary entries and can access the first posting in a random postings list in 11.3 ms on average (random disk seek + rotational latency). Choosing a block size of B = 1,024 bytes, the number of in-memory dictionary entries can be reduced to 3 million. At the same time, the search engine's list access latency (accessing the first posting in a randomly chosen list) increases by only 0.1 ms — a negligible overhead. As we increase the block size, the number of in-memory dictionary entries goes down and the index access latency goes up. But even for a relatively large block size of B = 256 KB, the additional cost — compared with a complete in-memory dictionary — is only a few milliseconds per query term.

Note that the memory consumption of an interleaved dictionary with block size B is quite different from maintaining an in-memory dictionary entry for every B bytes of index data. For example, the total size of the (compressed) schema-independent index for GOV2 is 62 GB. Choosing an index block size of B = 64 KB, however, does not lead to 62 GB / 64 KB $\approx$ 1 million

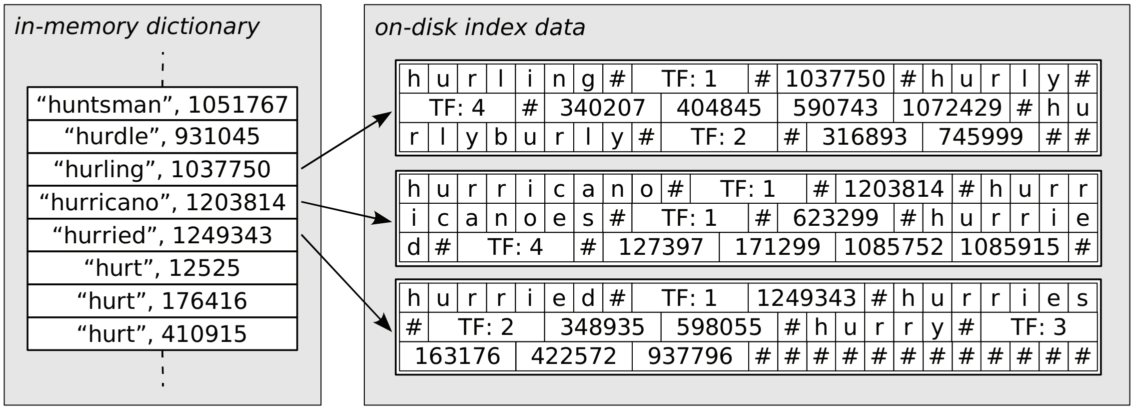

Figure 4.6 Combining dictionary and postings lists. The index is split into blocks of 72 bytes. Each entry of the in-memory dictionary is of the form (*term*, *posting*), indicating the first term and first posting in a given index block. The "#" symbols in the index data are record delimiters that have been inserted for better readability.

dictionary entries, but 10 times less. The reason is that frequent terms, such as "the" and "of", require only a single in-memory dictionary entry each, even though their postings lists each consume far more disk space than 64 KB (the compressed list for "the" consumes about 1 GB).

In practice, a block size between 4 KB and 16 KB is usually sufficient to shrink the in-memory dictionary to an acceptable size, especially if dictionary compression (Section 6.4) is used to decrease the space requirements of the few remaining in-memory dictionary entries. The disk transfer overhead for this range of block sizes is less than 1 ms per query term and is rather unlikely to cause any performance problems.

Dropping the distinction between terms and postings

We may take the dictionary interleaving approach one step further by dropping the distinction between terms and postings altogether, and by thinking of the index data as a sequence of pairs of the form (*term*, *posting*). The on-disk index is then divided into fixed-size index blocks, with each block perhaps containing 64 KB of data. All postings are stored on disk, in alphabetical order of their respective terms. Postings for the same term are stored in increasing order, as before. Each term's dictionary entry is stored on disk, potentially multiple times, so that there is a dictionary entry in every index block that contains at least one posting for the term. The in-memory data structure used to access data in the on-disk index then is a simple array, containing for each index block a pair of the form (*term*, *posting*), where *term* is the first term in the given block, and *posting* is the first posting for *term* in that block.

An example of this new index layout is shown in Figure 4.6 (data taken from a schema-independent index for the Shakespeare collection). In the example, a call to

$$\textbf{next}(\text{“hurried”},\ 1{,}000{,}000)$$

would load the second block shown (starting with “hurricano”) from disk, would search the block for a posting matching the query, and would return the first matching posting (1,085,752). A call to

$$\textbf{next}(\text{“hurricano”},\ 1{,}000{,}000)$$

would load the first block shown (starting with “hurling”), would not find a matching posting in that block, and would then access the second block, returning the posting 1,203,814.

The combined representation of dictionary and postings lists unifies the interleaving method explained above and the self-indexing technique described in Section 4.3 in an elegant way. With this index layout, a random access into an arbitrary term's postings list requires only a single disk seek (we have eliminated the initialization step in which the term's per-term index is loaded into memory). On the downside, however, the total memory consumption of the index is higher than if we employ self-indexing and dictionary interleaving as two independent techniques. A 62-GB index with block size B = 64 KB now in fact requires approximately 1 million in-memory entries.

4.5 Index Construction

In the previous sections of this chapter, you have learned about various components of an inverted index. In conjunction, they can be used to realize efficient access to the contents of the index, even if all postings lists are stored on disk, and even if we don't have enough memory resources to hold a complete dictionary for the index in RAM. We now discuss how to efficiently construct an inverted index for a given text collection.

From an abstract point of view, a text collection can be thought of as a sequence of term occurrences — tuples of the form (*term*, *position*), where *position* is the word count from the beginning of the collection, and *term* is the term that occurs at that position. Figure 4.7 shows a fragment of the Shakespeare collection under this interpretation. Naturally, when a text collection is read in a sequential manner, these tuples are ordered by their second component, the position in the collection. The task of constructing an index is to change the ordering of the tuples so that they are sorted by their first component, the term they refer to (ties are broken by the second component). Once the new order is established, creating a proper index, with all its auxiliary data structures, is a comparatively easy task.

In general, index construction methods can be divided into two categories: in-memory index construction and disk-based index construction. In-memory indexing methods build an index for a given text collection entirely in main memory and can therefore only be used if the collection

Text fragment:

⟨SPEECH⟩
⟨SPEAKER⟩ JULIET ⟨/SPEAKER⟩
⟨LINE⟩ O Romeo, Romeo! wherefore art thou Romeo? ⟨/LINE⟩
⟨LINE⟩ . . .

Original tuple ordering (collection order):

. . ., ("⟨speech⟩", 915487), ("⟨speaker⟩", 915488), ("juliet", 915489),
("⟨/speaker⟩", 915490), ("⟨line⟩", 915491), ("o", 915492), ("romeo", 915493),
("romeo", 915494), ("wherefore", 915495), ("art", 915496), ("thou", 915497),
("romeo", 915498), ("⟨/line⟩", 915499), ("⟨line⟩", 915500), . . .

New tuple ordering (index order):

. . ., ("⟨line⟩", 915491), ("⟨line⟩", 915500), . . .,
("romeo", 915411), ("romeo", 915493), ("romeo", 915494), ("romeo", 915498),
. . ., ("wherefore", 913310), ("wherefore", 915495), ("wherefore", 915849), . . .

Figure 4.7 The index construction process can be thought of as reordering the tuple sequence that constitutes the text collection. Tuples are rearranged from their original *collection order* (sorted by position) to their new *index order* (sorted by term).

is small relative to the amount of available RAM. They do, however, form the basis for more sophisticated, disk-based index construction methods. Disk-based methods can be used to build indices for very large collections, much larger than the available amount of main memory.

As before, we limit ourselves to schema-independent indices because this allows us to ignore some details that need to be taken care of when discussing more complicated index types, such as document-level positional indices, and lets us focus on the essential aspects of the algorithms. The techniques presented, however, can easily be applied to other index types.

4.5.1 In-Memory Index Construction

Let us consider the simplest form of index construction, in which the text collection is small enough for the index to fit entirely into main memory. In order to create an index for such a collection, the indexing process needs to maintain the following data structures:

- a dictionary that allows efficient single-term lookup and insertion operations;
- an extensible (i.e., dynamic) list data structure that is used to store the postings for each term.

Assuming these two data structures are readily available, the index construction procedure is straightforward, as shown in Figure 4.8. If the right data structures are used for dictionary and extensible postings lists, then this method is very efficient, allowing the search engine to build

```
buildIndex (inputTokenizer) ≡
    position ← 0
    while inputTokenizer.hasNext() do
        T ← inputTokenizer.getNext()
        obtain dictionary entry for T; create new entry if necessary
        append new posting position to T's postings list
        position ← position + 1
    sort all dictionary entries in lexicographical order
    for each term T in the dictionary do
        write T's postings list to disk
    write the dictionary to disk
    return
```

Figure 4.8 In-memory index construction algorithm, making use of two abstract data types: in-memory dictionary and extensible postings lists.

an index for the Shakespeare collection in less than one second. Thus, there are two questions that need to be discussed: 1. What data structure should be used for the dictionary? 2. What data structure should be used for the extensible postings lists?

Indexing-time dictionary

The dictionary implementation used during index construction needs to provide efficient support for single-term lookups and term insertions. Data structures that support these kinds of operations are very common and are therefore part of many publicly available programming libraries. For example, the C++ Standard Template Library (STL) published by SGI[1] provides a `map` data structure (binary search tree), and a `hash_map` data structure (variable-size hash table) that can carry out the lookup and insert operations performed during index construction. If you are implementing your own indexing algorithm, you might be tempted to use one of these existing implementations. However, it is not always advisable to do so.

We measured the performance of the STL data structures on a subset of GOV2, indexing the first 10,000 documents in the collection. The results are shown in Table 4.5. At first sight, it seems that the lookup performance of STL's `map` and `hash_map` implementations is sufficient for the purposes of index construction. On average, `map` needs about 630 ns per lookup; `hash_map` is a little faster, requiring 240 ns per lookup operation.

Now suppose we want to index the entire GOV2 collection instead of just a 10,000-document subset. Assuming the lookup performance stays at roughly the same level (an optimistic assumption, since the number of terms in the dictionary will keep increasing), the total time spent on

[1] `www.sgi.com/tech/stl/`

`hash_map` lookup operations, one for each of the 44 billion tokens in GOV, is:

$$44 \times 10^9 \times 240 \text{ ns} \ = \ 10{,}560 \text{ sec} \ \approx \ 3 \text{ hrs.}$$

Given that the fastest publicly available retrieval systems can build an index for the same collection in about 4 hours, including reading the input files, parsing the documents, and writing the (compressed) index to disk, there seems to be room for improvement. This calls for a custom dictionary implementation.

When designing our own dictionary implementation, we should try to optimize its performance for the specific type of data that we are dealing with. We know from Chapter 1 that term occurrences in natural-language text data roughly follow a Zipfian distribution. A key property of such a distribution is that the vast majority of all tokens in the collection correspond to a surprisingly small number of terms. For example, although there are about 50 million different terms in the GOV2 corpus, more than 90% of all term occurrences correspond to one of the 10,000 most frequent terms. Therefore, the bulk of the dictionary's lookup load may be expected to stem from a rather small set of very frequent terms. If the dictionary is based on a hash table, and collisions are resolved by means of chaining (as in Figure 4.2), then it is crucial that the dictionary entries for those frequent terms are kept near the beginning of each chain. This can be realized in two different ways:

1. **The insert-at-back heuristic**
   If a term is relatively frequent, then it is likely to occur very early in the stream of incoming tokens. Conversely, if the first occurrence of a term is encountered rather late, then chances are that the term is not very frequent. Therefore, when adding a new term to an existing chain in the hash table, it should be inserted at the end of the chain. This way, frequent terms, added early on, stay near the front, whereas infrequent terms tend to be found at the end.
2. **The move-to-front heuristic**
   If a term is frequent in the text collection, then it should be at the beginning of its chain in the hash table. Like insert-at-back, the move-to-front heuristic inserts new terms at the end of their respective chain. In addition, however, whenever a term lookup occurs and the term descriptor is not found at the beginning of the chain, it is relocated and moved to the front. This way, when the next lookup for the term takes place, the term's dictionary entry is still at (or near) the beginning of its chain in the hash table.

We evaluated the lookup performance of these two alternatives (using a handcrafted hash table implementation with a fixed number of hash slots in both cases) and compared it against a third alternative that inserts new terms at the beginning of the respective chain (referred to as the *insert-at-front* heuristic). The outcome of the performance measurements is shown in Table 4.5.

The insert-at-back and move-to-front heuristics achieve approximately the performance level (90 ns per lookup for a hash table with 2^{14} slots). Only for a small table size does move-to-front have a slight edge over insert-at-back (20% faster for a table with 2^{10} slots). The insert-at-front

Table 4.5 Indexing the first 10,000 documents of GOV2 (≈ 14 million tokens; 181,334 distinct terms). Average dictionary lookup time per token in microseconds. The rows labeled "Hash table" represent a handcrafted dictionary implementation based on a fixed-size hash table with chaining.

Dictionary Implementation	Lookup Time	String Comparisons
Binary search tree (STL `map`)	0.63 μs per token	18.1 per token
Variable-size hash table (STL `hash_map`)	0.24 μs per token	2.2 per token
Hash table (2^{10} entries, insert-at-front)	6.11 μs per token	140 per token
Hash table (2^{10} entries, insert-at-back)	0.37 μs per token	8.2 per token
Hash table (2^{10} entries, move-to-front)	0.31 μs per token	4.8 per token
Hash table (2^{14} entries, insert-at-front)	0.32 μs per token	10.1 per token
Hash table (2^{14} entries, insert-at-back)	0.09 μs per token	1.5 per token
Hash table (2^{14} entries, move-to-front)	0.09 μs per token	1.3 per token

heuristic, however, exhibits a very poor lookup performance and is between 3 and 20 times slower than move-to-front. To understand the origin of this extreme performance difference, look at the table column labeled "String Comparisons". When storing the 181,344 terms from the 10,000-document subcollection of GOV2 in a hash table with 2^{14} = 16,384 slots, we expect about 11 terms per chain on average. With the insert-at-front heuristic, the dictionary — on average — performs 10.1 string comparisons before it finds the term it is looking for. Because the frequent terms tend to appear early on in the collection, insert-at-front places them at the end of the respective chain. This is the cause of the method's dismal lookup performance. Incidentally, STL's `hash_map` implementation also inserts new hash table entries at the beginning of the respective chain, which is one reason why it does not perform so well in this benchmark.

A hash-based dictionary implementation employing the move-to-front heuristic is largely insensitive to the size of the hash table. In fact, even when indexing text collections containing many millions of different terms, a relatively small hash table, containing perhaps 2^{16} slots, will be sufficient to realize efficient dictionary lookups.

Extensible in-memory postings lists

The second interesting aspect of the simple in-memory index construction method, besides the dictionary implementation, is the implementation of the extensible in-memory postings lists. As suggested earlier, realizing each postings list as a singly linked list allows incoming postings to be appended to the existing lists very efficiently. The disadvantage of this approach is its rather high memory consumption: For every 32-bit (or 64-bit) posting, the indexing process needs to store an additional 32-bit (or 64-bit) pointer, thus increasing the total space requirements by 50–200%.

Various methods to decrease the storage overhead of the extensible postings lists have been proposed. For example, one could use a fixed-size array instead of a linked list. This avoids the

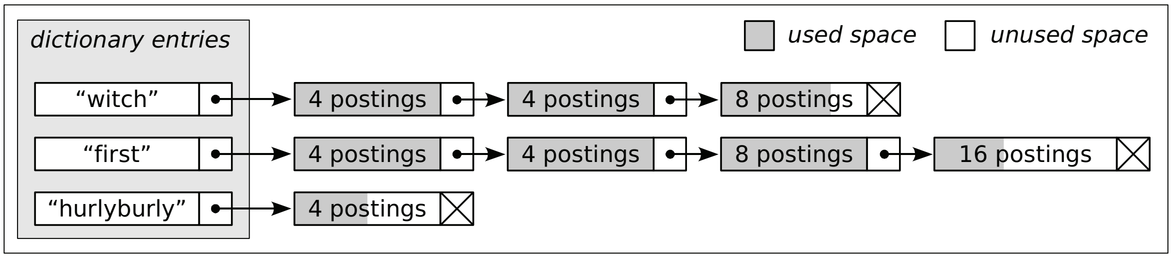

Figure 4.9 Realizing per-term extensible postings list by linking between groups of postings (*unrolled linked list*). Proportional pre-allocation (here with pre-allocation factor $k = 2$) provides an attractive trade-off between space overhead due to `next` pointers and space overhead due to internal fragmentation.

storage overhead associated with the `next` pointers in the linked list. Unfortunately, the length of each term's postings list is not known ahead of time, so this method requires an additional pass over the input data: In the first pass, term statistics are collected and space is allocated for each term's postings list; in the second pass, the pre-allocated arrays are filled with postings data. Of course, reading the input data twice harms indexing performance.

Another solution is to pre-allocate a postings array for every term in the dictionary, but to allow the indexing process to change the size of the array, for example, by performing a call to `realloc` (assuming that the programming language and run-time environment support this kind of operation). When a novel term is encountered, *init* bytes are allocated for the term's postings (e.g., $init = 16$). Whenever the capacity of the existing array is exhausted, a `realloc` operation is performed in order to increase the size of the array. By employing a proportional pre-allocation strategy, that is, by allocating a total of

$$s_{\text{new}} = \max\{s_{\text{old}} + init, k \times s_{\text{old}}\} \tag{4.2}$$

bytes in a reallocation operation, where s_{old} is the old size of the array and k is a constant, called the *pre-allocation factor*, the number of calls to `realloc` can be kept small (logarithmic in the size of each postings list). Nonetheless, there are some problems with this approach. If the pre-allocation factor is too small, then a large number of list relocations is triggered during index construction, possibly affecting the search engine's indexing performance. If it is too big, then a large amount of space is wasted due to allocated but unused memory (internal fragmentation). For instance, if $k = 2$, then 25% of the allocated memory will be unused on average.

A third way of realizing extensible in-memory postings lists with low storage overhead is to employ a linked list data structure, but to have multiple postings share a `next` pointer by linking to *groups* of postings instead of individual postings. When a new term is inserted into the dictionary, a small amount of space, say 16 bytes, is allocated for its postings. Whenever the space reserved for a term's postings list is exhausted, space for a new group of postings is

Table 4.6 Indexing the TREC45 collection. Index construction performance for various memory allocation strategies used to manage the extensible in-memory postings lists (32-bit postings; 32-bit pointers). Arranging postings for the same term in small groups and linking between these groups is faster than any other strategy. Pre-allocation factor used for both `realloc` and *grouping*: $k = 1.2$.

Allocation Strategy	Memory Consumption	Time (Total)	Time (CPU)
Linked list (simple)	2,312 MB	88 sec	77 sec
Two-pass indexing	1,168 MB	123 sec	104 sec
`realloc`	1,282 MB	82 sec	71 sec
Linked list (grouping)	1,208 MB	71 sec	61 sec

allocated. The amount of space allotted to the new group is

$$s_{\text{new}} = \min\{limit, \max\{16, (k-1) \times s_{\text{total}}\}\}, \quad (4.3)$$

where *limit* is an upper bound on the size of a postings group, say, 256 postings, s_{total} is the amount of memory allocated for postings so far, and k is the pre-allocation factor, just as in the case of `realloc`.

This way of combining postings into groups and having several postings share a single `next` pointer is shown in Figure 4.9. A linked list data structure that keeps more than one value in each list node is sometimes called an *unrolled linked list*, in analogy to loop unrolling techniques used in compiler optimization. In the context of index construction, we refer to this method as *grouping*.

Imposing an upper limit on the amount of space pre-allocated by the grouping technique allows to control internal fragmentation. At the same time, it does not constitute a performance problem; unlike in the case of `realloc`, allocating more space for an existing list is a very lightweight operation and does not require the indexing process to relocate any postings data.

A comparative performance evaluation of all four list allocation strategies — simple linked list, two-pass, `realloc`, and linked list with grouping — is given by Table 4.6. The two-pass method exhibits the lowest memory consumption but takes almost twice as long as the grouping approach. Using a pre-allocation factor $k = 1.2$, the `realloc` method has a memory consumption that is about 10% above that of the space-optimal two-pass strategy. This is consistent with the assumption that about half of the pre-allocated memory is never used. The grouping method, on the other hand, exhibits a memory consumption that is only 3% above the optimum. In addition, grouping is about 16% faster than `realloc` (61 sec vs. 71 sec CPU time).

Perhaps surprisingly, linking between groups of postings is also faster than linking between individual postings (61 sec vs. 77 sec CPU time). The reason for this is that the unrolled linked list data structure employed by the grouping technique not only decreases internal fragmentation, but also improves CPU cache efficiency, by keeping postings for the same term close

```
buildIndex_sortBased (inputTokenizer) ≡
    position ← 0
    while inputTokenizer.hasNext() do
        T ← inputTokenizer.getNext()
        obtain dictionary entry for T; create new entry if necessary
        termID ← unique term ID of T
        write record R_position ≡ (termID, position) to disk
        position ← position + 1
    tokenCount ← position
    sort R_0 ... R_{tokenCount−1} by first component; break ties by second component
    perform a sequential scan of R_0 ... R_{tokenCount−1}, creating the final index
    return
```

Figure 4.10 Sort-based index construction algorithm, creating a schema-independent index for a text collection. The main difficulty lies in efficiently sorting the on-disk records $R_0 \ldots R_{tokenCount-1}$.

together. In the simple linked-list implementation, postings for a given term are randomly scattered across the used portion of the computer's main memory, resulting in a large number of CPU cache misses when collecting each term's postings before writing them to disk.

4.5.2 Sort-Based Index Construction

Hash-based in-memory index construction algorithms can be extremely efficient, as demonstrated in the previous section. However, if we want to index text collections that are substantially larger than the available amount of RAM, we need to move away from the idea that we can keep the entire index in main memory, and need to look toward disk-based approaches instead. The sort-based indexing method presented in this section is the first of two disk-based index construction methods covered in this chapter. It can be used to index collections that are much larger than the available amount of main memory.

Recall from the beginning of this section that building an inverted index can be thought of as the process of reordering the sequence of term-position tuples that represents the text collection — transforming it from collection order into index order. This is a sorting process, and sort-based index construction realizes the transformation in the simplest way possible. Consuming tokens from the input files, a sort-based indexing process emits records of the form (*termID*, *position*) and writes them to disk immediately. The result is a sequence of records, sorted by their second component (*position*). When it is done processing the input data, the indexer sorts all tuples written to disk by their first component, using the second component to break ties. The result is a new sequence of records, sorted by their first component (*termID*). Transforming this new sequence into a proper inverted file, using the information found in the in-memory dictionary, is straightforward.

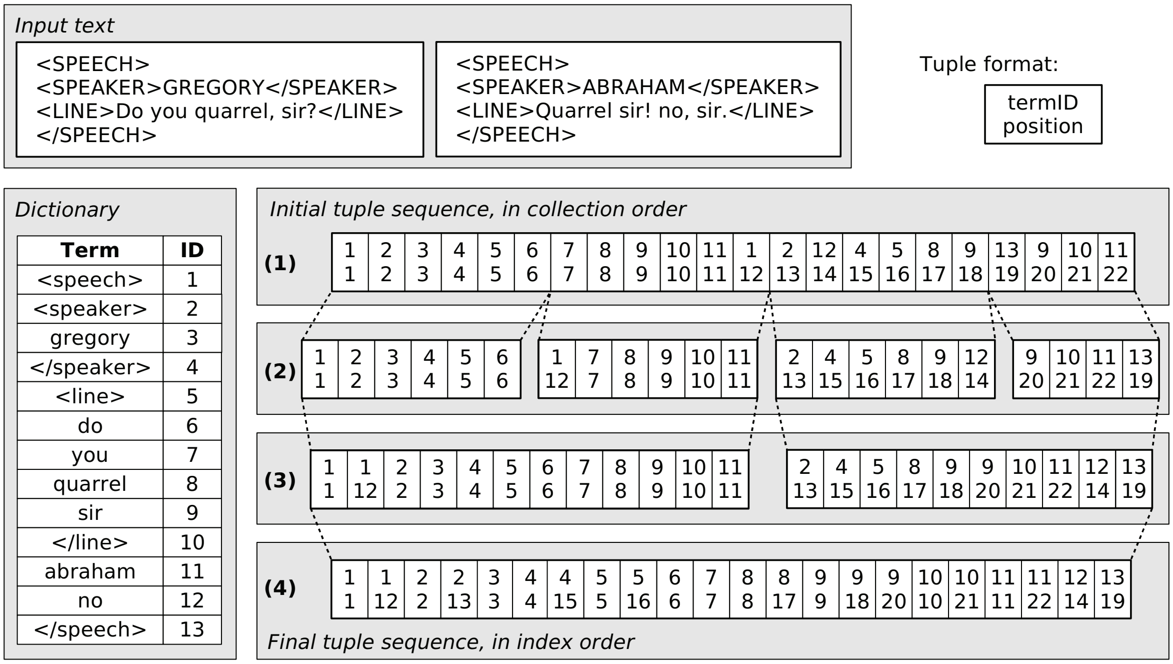

Figure 4.11 Sort-based index construction with global term IDs. Main memory is large enough to hold 6 (*termID*, *position*) tuples at a time. (1)→(2): sorting blocks of size ≤ 6 in memory, one at a time. (2)→(3) and (3)→(4): merging sorted blocks into bigger blocks.

The method, shown as pseudo-code in Figure 4.10, is very easy to implement and can be used to create an index that is substantially larger than the available amount of main memory. Its main limitation is the available amount of disk space.

Sorting the on-disk records can be a bit tricky. In many implementations, it is performed by loading a certain number of records, say n, into main memory at a time, where n is defined by the size of a record and the amount of available main memory. These n records are then sorted in memory and written back to disk. The process is repeated until we have $\lceil \frac{tokenCount}{n} \rceil$ blocks of sorted records. These blocks can then be combined, in a *multiway merge operation* (processing records from all blocks at the same time) or in a *cascaded merge operation* (merging, for example, two blocks at a time), resulting in a sorted sequence of *tokenCount* records. Figure 4.11 shows a sort-based indexing process that creates the final tuple sequence by means of a cascaded merge operation, merging two tuple sequences at a time. With some additional data structures, and with the *termID* component removed from each posting, this final sequence can be thought of as the index for the collection.

Despite its ability to index text collections much larger than the available amount of main memory, sort-based indexing has two essential limitations:

- It requires a substantial amount of disk space, usually at least 8 bytes per input token (4 bytes for the term ID + 4 bytes for the position), or even 12 bytes (4 + 8) for larger text collections. When indexing GOV2, for instance, the disk space required to store the temporary files is at least $12 \times 44 \times 10^9$ bytes (= 492 GB) — more than the uncompressed size of the collection itself (426 GB).
- To be able to properly sort the (*termID*, *docID*) pairs emitted in the first phase, the indexing process needs to maintain globally unique term IDs, which can be realized only through a complete in-memory dictionary. As discussed earlier, a complete dictionary for GOV2 might consume more than 1 GB of RAM, making it difficult to index the collection on a low-end machine.

There are various ways to address these issues. However, in the end they all have in common that they transform the sort-based indexing method into something more similar to what is known as *merge-based index construction*.

4.5.3 Merge-Based Index Construction

In contrast to sort-based index construction methods, the merge-base method presented in this section does not need to maintain any global data structures. In particular, there is no need for globally unique term IDs. The size of the text collection to be indexed is therefore limited only by the amount of disk space available to store the temporary data and the final index, but not by the amount of main memory available to the indexing process.

Merge-based indexing is a generalization of the in-memory index construction method discussed in Section 4.5.1, building inverted lists by means of hash table lookup. In fact, if the collection for which an index is being created is small enough for the index to fit completely into main memory, then merge-based indexing behaves exactly like in-memory indexing. If the text collection is too large to be indexed completely in main memory, then the indexing process performs a *dynamic partitioning* of the collection. That is, it starts building an in-memory index. As soon as it runs out of memory (or when a predefined memory utilization threshold is reached), it builds an on-disk inverted file by transferring the in-memory index data to disk, deletes the in-memory index, and continues indexing. This procedure is repeated until the whole collection has been indexed. The algorithm is shown in Figure 4.12.

The result of the process outlined above is a set of inverted files, each representing a certain part of the whole collection. Each such subindex is referred to as an *index partition*. In a final step, the index partitions are merged into the final index, representing the entire text collection. The postings lists in the index partitions (and in the final index) are usually stored in compressed form (see Chapter 6), in order to keep the disk I/O overhead low.

The index partitions written to disk as intermediate output of the indexing process are completely independent of each other. For example, there is no need for globally unique term IDs; there is not even a need for numerical term IDs. Each term is its own ID; the postings lists in each partition are stored in lexicographical order of their terms, and access to a term's list can

```
buildIndex_mergeBased (inputTokenizer, memoryLimit) ≡
    n ← 0   // initialize the number of index partitions
    position ← 0
    memoryConsumption ← 0
    while inputTokenizer.hasNext() do
        T ← inputTokenizer.getNext()
        obtain dictionary entry for T; create new entry if necessary
        append new posting position to T's postings list
        position ← position + 1
        memoryConsumption ← memoryConsumption + 1
        if memoryConsumption ≥ memoryLimit then
            createIndexPartition()
    if memoryConsumption > 0 then
        createIndexPartition()
    merge index partitions I_0 ... I_{n−1}, resulting in the final on-disk index I_final
    return

createIndexPartition () ≡
    create empty on-disk inverted file I_n
    sort in-memory dictionary entries in lexicographical order
    for each term T in the dictionary do
        add T's postings list to I_n
    delete all in-memory postings lists
    reset the in-memory dictionary
    memoryConsumption ← 0
    n ← n + 1
    return
```

Figure 4.12 Merge-based indexing algorithm, creating a set of independent sub-indices (*index partitions*). The final index is generated by combining the sub-indices via a multi-way merge operation.

be realized by using the data structures described in Sections 4.3 and 4.4. Because of the lexicographical ordering and the absence of term IDs, merging the individual partitions into the final index is straightforward. Pseudo-code for a very simple implementation, performing repeated linear probing of all subindices, is given in Figure 4.13. If the number of index partitions is large (more than 10), then the algorithm can be improved by arranging the index partitions in a priority queue (e.g., a heap), ordered according to the next term in the respective partition. This eliminates the need for the linear scan in lines 7–10.

Overall performance numbers for merge-based index construction, including all components, are shown in Table 4.7. The total time necessary to build a schema-independent index for GOV2 is around 4 hours. The time required to perform the final merge operation, combining the n index partitions into one final index, is about 30% of the time it takes to generate the partitions.

```
mergeIndexPartitions (⟨I_0, ..., I_{n−1}⟩) ≡
    create empty inverted file I_final
    for k ← 0 to n − 1 do
        open index partition I_k for sequential processing
    currentIndex ← 0
    while currentIndex ≠ nil do
        currentIndex ← nil
        for k ← 0 to n − 1 do
            if I_k still has terms left then
                if (currentIndex = nil) ∨ (I_k.currentTerm < currentTerm) then
                    currentIndex ← I_k
                    currentTerm ← I_k.currentTerm
        if currentIndex ≠ nil then
            I_final.addPostings(currentTerm, currentIndex.getPostings(currentTerm))
            currentIndex.advanceToNextTerm()
    delete I_0 ... I_{n−1}
    return
```

Figure 4.13 Merging a set of n index partitions $I_0 \ldots I_{n-1}$ into an index I_{final}. This is the final step in merge-based index construction.

The algorithm is very scalable: On our computer, indexing the whole GOV2 collection (426 GB of text) takes only 11 times as long as indexing a 10% subcollection (43 GB of text).

There are, however, some limits to the scalability of the method. When merging the index partitions at the end of the indexing process, it is important to have at least a moderately sized read-ahead buffer, a few hundred kilobytes, for each partition. This helps keep the number of disk seeks (jumping back and forth between the different partitions) small. Naturally, the size of the read-ahead buffer for each partition is bounded from above by M/n, where M is the amount of available memory and n is the number of partitions. Thus, if n becomes too large, merging becomes slow.

Reducing the amount of memory available to the indexing process therefore has two effects. First, it decreases the total amount of memory available for the read-ahead buffers. Second, it increases the number of index partitions. Thus, reducing main memory by 50% decreases the size of each index partition's read-ahead buffer by 75%. Setting the memory limit to $M = 128$ MB, for example, results in 3,032 partitions that need to be merged, leaving each partition with a read-ahead buffer of only 43 KB. The general trend of this effect is depicted in Figure 4.14. The figure shows that the performance of the final merge operation is highly dependent on the amount of main memory available to the indexing process. With 128 MB of available main memory, the final merge takes 6 times longer than with 1,024 MB.

There are two possible countermeasures that could be taken to overcome this limitation. The first is to replace the simple multiway merge by a cascaded merge operation. For instance, if 1,024 index partitions need to be merged, then we could first perform 32 merge operations

Table 4.7 Building a schema-independent index for various text collections, using merge-based index construction with 512 MB of RAM for the in-memory index. The indexing-time dictionary is realized by a hash table with 2^{16} entries and move-to-front heuristic. The extensible in-memory postings lists are unrolled linked lists, linking between groups of postings, with a pre-allocation factor $k = 1.2$.

	Reading, Parsing & Indexing	Merging	Total Time
Shakespeare	1 sec	0 sec	1 sec
TREC45	71 sec	11 sec	82 sec
GOV2 (10%)	20 min	4 min	24 min
GOV2 (25%)	51 min	11 min	62 min
GOV2 (50%)	102 min	25 min	127 min
GOV2 (100%)	205 min	58 min	263 min

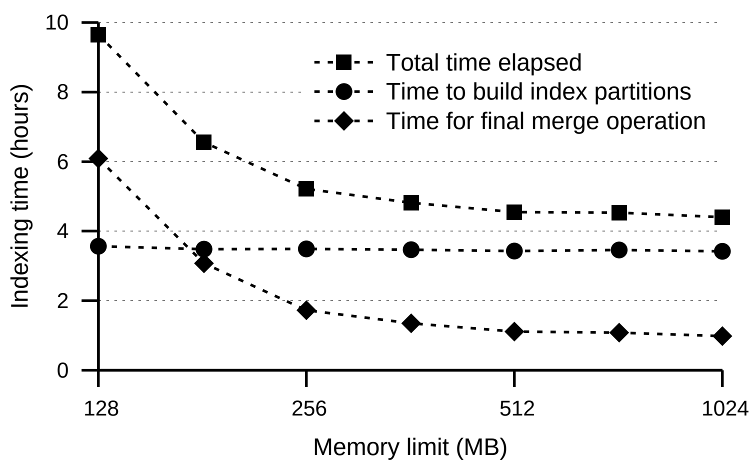

Figure 4.14 The impact of available RAM on the performance of merge-based indexing (data set: GOV2). The performance of phase 1 (building index partitions) is largely independent of the amount of available main memory. The performance of phase 2 (merging partitions) suffers severely if little main memory is available.

involving 32 partitions each, and then merge the resulting 32 new partitions into the final index. This is a generalization of the process shown in Figure 4.11, which depicts a cascaded merge operation that works on 2 partitions at a time. The second countermeasure is to decrease the space consumed by the postings lists, by means of compressed in-memory inversion. Compressing postings on-the-fly, as they enter the in-memory index, allows the indexing process to accumulate more postings before it runs out of memory, thus decreasing the number of on-disk index partitions created.

In conclusion, despite some problems with the final merge operation, merge-based index construction lets us build an inverted file for very large text collections, even on a single PC. Its advantage over the sort-based method is that it does not require globally unique term IDs. It is therefore especially attractive if the number of dictionary terms is very large. Another great advantage of this index construction algorithm over the sort-based method is that it produces an in-memory index that is immediately queriable. This feature is essential when the search engine needs to deal with dynamic text collections (see Chapter 7).

4.6 Other Types of Indices

In our discussion of index data structures, we have limited ourselves to the case of inverted indices. However, an inverted index is just one possible type of index that can be used as part of a search engine.

A *forward index* (or *direct index*) is a mapping from document IDs to the list of terms appearing in each document. Forward indices complement inverted indices. They are usually not used for the actual search process, but to obtain information about per-document term distributions at query time, which is required by query expansion techniques such as pseudo-relevance feedback (see Chapter 8), and to produce result *snippets*. Compared to extracting this information from the raw text files, a forward index has the advantage that the text has already been parsed, and the relevant data can be extracted more efficiently.

Signature files (Faloutsos and Christodoulakis, 1984) are an alternative to docid indices. Similar to a Bloom filter (Bloom, 1970), a signature file can be used to obtain a list of documents that *may* contain the given term. In order to find out whether the term actually appears in a document, the document itself (or a forward index) needs to be consulted. By changing the parameters of the signature file, it is possible to trade time for speed: A smaller index results in a greater probability of false positives, and vice versa.

Suffix trees (Weiner, 1973) and *suffix arrays* (Manber and Myers, 1990) can be used to efficiently find all occurrences of a given n-gram sequence in a given text collection. They can be used either to index character n-grams (without tokenizing the input text) or to index word n-grams (after tokenization). Suffix trees are attractive data structures for phrase search or regular expression search, but are usually larger than inverted indices and provide less efficient search operations when stored on disk instead of in RAM.

4.7 Summary

In this chapter we have covered the essential algorithms and data structures necessary to build and access inverted indices. The main points of the chapter are:

- Inverted indices are usually too large to be loaded completely into memory. It is therefore common to keep only the dictionary, which is relatively small compared to the size of the entire index, in memory, while storing the postings lists on disk (Section 4.2).
- For a large text collection even the dictionary might be too large to fit into RAM. The memory requirements of the dictionary can be substantially decreased by keeping an incomplete dictionary in memory, interleaving dictionary entries with on-disk postings lists, and exploiting the fact that all lists in the index are sorted in lexicographical order (Section 4.4).
- A sort-based dictionary implementation should be used, and postings lists should be stored on disk in lexicographical order, if prefix queries are to be supported by the search engine (Sections 4.2 and 4.3).
- For each on-disk postings list, a per-term index, containing a subset of the term's postings, can be used to realize efficient quasi-random access into the list (Section 4.3).
- Highly efficient in-memory index construction can be realized by employing a hash-based in-memory dictionary with move-to-front heuristic and by using grouped linked lists to implement extensible in-memory postings lists (Section 4.5.1).
- If the amount of main memory available to the indexing process is too small to allow the index to be built completely in RAM, then the in-memory index construction method can be extended to a merge-based method, in which the text collection is divided into subcollections, dynamically and based on the available amount of main memory. An index for each subcollection is built using the in-memory indexing method. After the entire collection has been indexed, the indices for the individual subcollections are merged in a single multiway merge operation or a cascaded merge process (Section 4.5.3).
- The performance of merge-based index construction is essentially linear in the size of the collection. However, the final merge operation may suffer severely if too little main memory is available for the subindices' read buffers (Section 4.5.3).

4.8 Further Reading

Good entry points for an in-depth understanding of the architecture and performance of inverted indices are provided by Witten et al. (1999, chapters 3 and 5) and Zobel and Moffat (2006). A high-level overview of the index data structures employed by Google around 1998 is given by Brin and Page (1998).

Moffat and Zobel (1996) discuss query-time efficiency issues of inverted files, including the per-term index data structure ("self-indexing") used for random list access that is outlined in Section 4.3. Rao and Ross (1999, 2000) demonstrate that random access issues arise not only in the context of on-disk indices but also for in-memory inverted files. They show that binary search is *not* the best way to realize random access for in-memory postings lists.

Heinz and Zobel (2003) discuss single-pass merge-based index construction and its advantages over the sort-based method. They also examine the efficiency of various in-memory dictionary implementations, including the move-to-front heuristic described in Section 4.5.1 (Zobel et al., 2001), and propose a new dictionary data structure, the *burst trie* (Heinz et al., 2002), which achieves a single-term lookup performance close to a hash table but — unlike a hash table — can also be used to resolve prefix queries.

Memory management strategies for extensible in-memory postings lists (e.g., unrolled linked lists) are examined by Büttcher and Clarke (2005) and, more recently, by Luk and Lam (2007).

With a naïve implementation of the final merge procedure in merge-based index construction (and variants of sort-based indexing), the total storage requirement is twice as large as the size of the final index. Moffat and Bell (1995) describe a clever technique that can be used to realize the merge in situ, re-using the disk space occupied by already processed parts of the input partitions to store the final index.

Faloutsos and Christodoulakis (1984) give a nice overview of signature files, including some theoretical properties. Zobel et al. (1998) discuss the relative performance of inverted files and signature files in text search. They come to the conclusion that, for many applications, inverted files are the better choice. Carterette and Can (2005) argue that, under certain circumstances, signature files can be almost as fast as inverted files.

Suffix trees made their first appearance under the name *position trees* in a paper by Weiner (1973). Ukkonen (1995) presents a linear-time construction method for suffix trees. Clark and Munro (1996) discuss a variant of suffix trees that allows efficient search operations when stored on disk instead of main memory.

4.9 Exercises

Exercise 4.1 In Section 4.3 we introduced the concept of the *per-term index* as a means to improve the index's random access performance. Suppose the postings list for some term consists of 64 million postings, each of which consumes 4 bytes. In order to carry out a single random access into the term's postings list, the search engine needs to perform two disk read operations:

1. Loading the per-term index (list of synchronization points) into RAM.
2. Loading a block B of postings into RAM, where B is identified by means of binary search on the list of synchronization points.

Let us call the number of postings per synchronization point the *granularity* of the per-term index. For the above access pattern, what is the optimal granularity (i.e., the one that minimizes disk I/O)? What is the total number of bytes read from disk?

Exercise 4.2 The introduction of the per-term index in Section 4.3 was motivated by the performance characteristics of typical hard disk drives (in particular, the high cost of disk seeks). However, the same method can also be used to improve the random access performance of in-memory indices. To confirm this claim, you have to implement two different data structures for in-memory postings lists and equip them with a **next** access method (see Chapter 2). The first data structure stores postings in a simple array of 32-bit integers. Its **next** method operates by performing a binary search on that array. In the second data structure, an auxiliary array is used to store a copy of every 64th posting in the postings list. The **next** method first performs a binary search on the auxiliary array, followed by a *sequential scan* of the 64 candidate postings in the postings list. Measure the average single-element lookup latency of both implementations, working with lists of n postings, for $n = 2^{12}, 2^{16}, 2^{20}, 2^{24}$. Describe and analyze your findings.

Exercise 4.3 Building an inverted index is essentially a sorting process. The lower bound for every general-purpose sorting algorithm is $\Omega(n \log(n))$. However, the merge-based index construction method from Section 4.5.3 has a running time that is linear in the size of the collection (see Table 4.7, page 130). Find at least two places where there is a hidden logarithmic factor.

Exercise 4.4 In the algorithm shown in Figure 4.12, the memory limit is expressed as the number of postings that can be stored in RAM. What is the assumption that justifies this definition of the memory limit? Give an example of a text collection or an application in which the assumption does not hold.

Exercise 4.5 In Section 4.5.1 we discussed the performance characteristics of various dictionary data structures. We pointed out that hash-based implementations offer better performance for single-term lookups (performed during index construction), while sort-based solutions are more appropriate for multi-term lookups (needed for prefix queries). Design and implement a data structure that offers better single-term lookup performance than a sort-based dictionary and better prefix query performance than a hash-based implementation.

Exercise 4.6 (project exercise) Design and implement an index construction method that creates a schema-independent index for a given text collection. The result of the index construction process is an on-disk index. The index does not need to use any of the optimizations discussed in Sections 4.3 and 4.4.

- Implement the in-memory index construction method described in Section 4.5.1. When run on typical English text, your implementation should be able to build an index for a collection containing approximately $\frac{M}{8}$ tokens, where M is the amount of available main memory, in bytes.

- Extend your implementation so that the size of the text collection is no longer limited by the amount of main memory available to the indexing process. This will probably require you to write a module that can merge two or more on-disk indices into a single index.

4.10 Bibliography

Bender, M., Michel, S., Triantafillou, P., and Weikum, G. (2007). Design alternatives for large-scale Web search: Alexander was great, Aeneas a pioneer, and Anakin has the force. In *Proceedings of the 1st Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR)*, pages 16–22. Amsterdam, The Netherlands.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426.

Brin, S., and Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117.

Büttcher, S., and Clarke, C. L. A. (2005). *Memory Management Strategies for Single-Pass Index Construction in Text Retrieval Systems.* Technical Report CS-2005-32. University of Waterloo, Waterloo, Canada.

Carterette, B., and Can, F. (2005). Comparing inverted files and signature files for searching a large lexicon. *Information Processing & Management*, 41(3):613–633.

Clark, D. R., and Munro, J. I. (1996). Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391. Atlanta, Georgia.

Faloutsos, C., and Christodoulakis, S. (1984). Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems*, 2(4):267–288.

Heinz, S., and Zobel, J. (2003). Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology*, 54(8):713–729.

Heinz, S., Zobel, J., and Williams, H. E. (2002). Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223.

Luk, R. W. P., and Lam, W. (2007). Efficient in-memory extensible inverted file. *Information Systems*, 32(5):733–754.

Manber, U., and Myers, G. (1990). Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327. San Francisco, California.

Moffat, A., and Bell, T. A. H. (1995). In-situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550.

Moffat, A., and Zobel, J. (1996). Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379.

Rao, J., and Ross, K. A. (1999). Cache conscious indexing for decision-support in main memory. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 78–89. Edinburgh, Scotland.

Rao, J., and Ross, K. A. (2000). Making B^+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486. Dallas, Texas.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.

Weiner, P. (1973). Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11. Iowa City, Iowa.

Witten, I. H., Moffat, A., and Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images* (2nd ed.). San Francisco, California: Morgan Kaufmann.

Zobel, J., Heinz, S., and Williams, H. E. (2001). In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271–277.

Zobel, J., and Moffat, A. (2006). Inverted files for text search engines. *ACM Computing Surveys*, 38(2):1–56.

Zobel, J., Moffat, A., and Ramamohanarao, K. (1998). Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490.