

Chapter 4: Building the Inverted Index

Objectives

- Learn how search engines index text
- Learn the basic inverted indexing algorithm

1 Introduction

Having discussed parsing, the next step is to describe how text is indexed. The point of using an index is to increase the speed and efficiency of searches of the document collection. Without some sort of index, a user's query must sequentially scan the document collection, finding those documents containing the search terms. Consider the "Find" operation in Windows; a user search is initiated and a search starts through each file on the hard disk. When a directory is encountered, the search continues through each directory. With only a few thousand files on a typical laptop, a typical "find" operation takes a minute or longer. Assume we have a fast processor executing an efficient sequential search algorithm. Further, assume we have a fast processor and high speed disk drives that enable a scan of one million documents per second (figure a document to be 4KB). This implies a scan rate of 400 GB per second, a generous processing assumption. Currently, a web search covers at least one billion documents. This back of the envelope calculation puts us at around 1000 seconds or a tad longer than sixteen minutes (and we were generous in estimating the speed of the scan). Users are not going to wait sixteen minutes for a response. Hence, a sequential scan is simply not feasible.

B-trees (balanced trees) were developed for efficient database access in the 1970's [B-tree ref]. A survey of B-tree algorithms is found in [xxx]. The B-tree is designed to remain balanced even after numerous updates. This means that even in the presence of updates, retrieval remains at a speed of $O(\log n)$ where n is the number of keys. Hence, update is well balanced with retrieval, both require $O(\log n)$.

Within the search engine domain, data are searched far more frequently than they are updated. Certainly, users make changes to documents, but it is not clear that a searcher requires the latest change to satisfy most search requests. Given this situation a data structure called an *inverted index* is commonly used by search engines.

An inverted index is able to do many accesses in $O(1)$ time at a price of significantly longer time to do an update, in the worst case $O(n)$. Index construction time is longer as well, but query time is generally faster than with a b-tree. Since index construction is an off-line process, shorter query processing times at the expense of lengthier index construction times is an appropriate tradeoff.

Finally, inverted index storage structures can exceed the storage demands of the document collection itself. However, for many systems, the inverted index can be compressed to around ten percent of the original document collection. Given the alternative (of twenty minute searches), search engine developers are happy to trade index construction time and storage for query efficiency.

An inverted index is an optimized structure that is built primarily for retrieval, with update being only a secondary consideration. The basic structure *inverts* the text so that instead of the view obtained from scanning documents where a document is found and then its terms are seen (think of a list of documents each pointing to a list of terms it

contains), an index is built that maps terms to documents (pretty much like the index found in the back of this book that maps terms to page numbers). Instead of listing each *document* once (and each term repeated for each document that contains the term), an inverted index lists each *term* in the collection only once and then shows a list of all the documents that contain the given term. Each document identifier is repeated for each term that is found in the document.

An inverted index contains two parts: an *index of terms*, (generally called simply the *term index*, *lexicon*, or *term dictionary*) which stores a distinct list of terms found in the collection and, for each term, a *posting list*, a list of documents that contain the term.

Consider the following two documents:

D1: The GDP increased 2 percent this quarter.

D2: The spring economic slowdown continued to spring downwards this quarter.

An inverted index for these two documents is given below:

2 → [D1]
continued → [D2]
downwards → [D2]
economic → [D2]
GDP → [D1]
increased → [D1]
percent → [D1]
quarter → [D1] → [D2]
slowdown → [D2]
spring → [D2]
the → [D1] → [D2]
this → [D1] → [D2]
to → [D2]

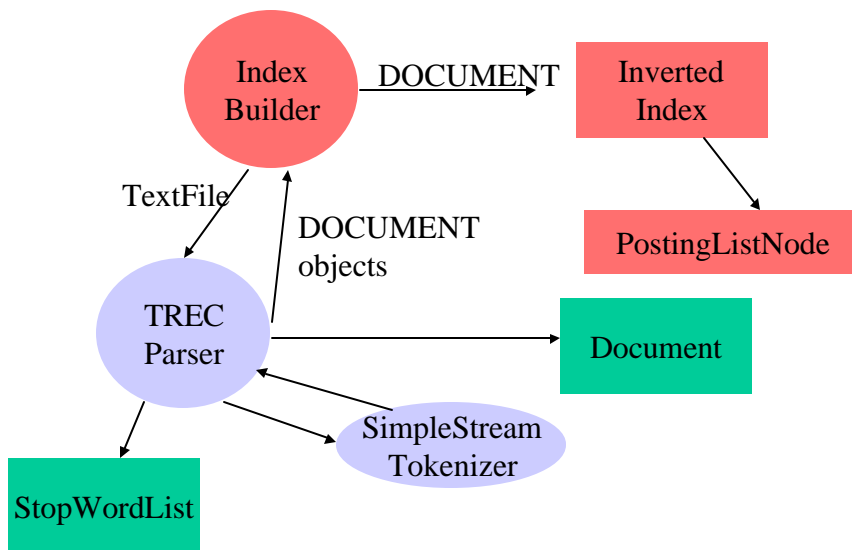
As shown, the terms *continued*, *economic*, *slowdown*, *spring*, and *to* appear in only the second document, the terms *GDP*, *increased*, and *percent*, and the numeral 2 appear in only the first document, and the terms *quarter*, *the*, and *this* appear in both documents.

We now briefly describe the data structures often used to build the index and the posting list.

2 Index Construction

As we discussed in Chapter 2, the IndexBuilder object is at the heart of building the index. Figure 10 is similar to that given in Figure 5, but now the components that actually do the index construction are drawn in red to illustrate that these are the ones we will be focusing on during this chapter.

Figure 10: Indexing Architecture



The IndexBuilder drives the indexing process. When instantiated, the *build* method in this object actually parses documents using the Parser. As mentioned in Chapter 2, the parser returns a list of Document objects that contain parsed terms. The parsed documents objects are then sent to the *add* method in the inverted index. Once the index

is built, users are able to efficiently submit queries against the text. When completed, the index is written to disk with the *write* method.

2.1 ***Index***

The index is simply a list of terms. Since we rarely traverse this list, a hash table is often used as the data structure for the index. A hash table permits quick access to a term by applying a hash function on the term. If two terms have the same hash entry, a collision resolution algorithm (such as simply using a linked list of collisions) can be employed. The Java JDK hides the details of collisions from the user with the `HashMap` class. Once we find the correct entry for a term, a pointer will exist to the relevant posting list.

The index also may contain other useful information about a term. The size of the posting list indicates the number of documents that contain the term. Typically this is a value stored with the term entry in the index. Other values might include the term type, namely, is the term a phrase, an acronym, a number, etc.

A crucial assumption with many information retrieval systems is that the index fits in memory. If only single term entries are stored in the index then this is easily the case. A 10GB web collection often used for testing information retrieval systems (see Chapter xxx—`trec`) has fewer than one million distinct terms. With an average term length of around six characters, only 6 MB are needed to store the actual text of the index. If we add four bytes for a document frequency, a byte for a type tag, and a four-byte pointer to the inverted index, we now have 15 bytes for each term requiring a total of 15 MB to store the index. Such storage requirements are easily supported by main memory configurations.

We note that ultimately, in some large diverse collections, it is not possible to guarantee storing the index in memory, and in such cases, if the index exceeds its allotted memory, a b-tree is often used to represent just the term entries in the index. This has the advantage of efficiently finding the entry in $O(\log_b N)$ where N is the number of terms time. The b-tree eliminates the need for the index to fit entirely into main memory at a small cost of no longer being able to simply hash on the term and immediately go to the memory address of the corresponding posting list.

2.2 **Posting List**

A posting list indicates, for a given term, which documents contain the term. Typically, a linked list is used to store the entries in a posting list. This is because in most retrieval operations, a user enters a query and all documents that contain the query are obtained. This is done by hashing on the term in the index and finding the associated posting list. Once the posting list is obtained, a simple scan of the linked list yields all the documents that contain the query. A `PostingListNode` object is used in `SimpleIR` to store the document identifier and the term frequency for each occurrence in a `PostingList`.

2.3 **Document List**

The inverted index only needs a term dictionary, namely the index, and the corresponding posting lists. However, we found it convenient to also include the *documentList* variable in our *InvertedIndex* object. The *documentList* is simply an `ArrayList` of `Document` objects (see Chapter 2 for more details on `Document` objects). Its purpose is to map a document identifier to a `Document` object. The `Document` object contains information unique to each document, for example, the author, title, date published, text file that

```

/* this class actually can build the inverted index */
public class IndexBuilder implements java.io.Serializable {

    private Properties configSettings;      /* config settings object          */
    private int      numberOfDocuments;    /* num docs in the index              */
    private String   inputFileName;       /* name of the input file being read  */
    private ArrayList documentList = new ArrayList();

    IndexBuilder (Properties p) {
        configSettings = p;
    }

    /* build the inverted index, reads all documents */
    public InvertedIndex build () throws IOException {

        boolean endOfFile = false;
        int offset = 0;
        Document d;

        InvertedIndex index = new InvertedIndex();
        index.clear();

        inputFileName = configSettings.getProperty("TEXT_FILE");
        TRECParser parser = new TRECParser(inputFileName, stopWordFileName);
        documentList = parser.readDocuments();

        Iterator i = documentList.iterator();
        while (i.hasNext()) {
            d = (Document) i.next();
            index.add(d);
        }

        index.write(configSettings); // write the index to a file
        return index;
    }
}

```

contains the document, etc.). Storing the list of document objects with the inverted index makes sense because the only time it is updated is when we update the inverted index.

3 Index Builder

The index builder drives the indexing process. The constructor reads the configuration file with the Properties object to identify what stop word list to use and what text files are used as input. The *build* method calls the parser, and the parser returns a set of document objects. Next, the index builder loops through all the document objects and calls the *add* method associated with the *InvertedIndex* object to add each document to the inverted

index. The IndexBuilder is designed so that different inverted indexes can be constructed for different document collections. Also different parsers are easily incorporated for different types of documents. Once all documents have been processed, the *write* method is used to store the *InvertedIndex* object to disk.

4 The Inverted Index Object

Consider an object-oriented design for a text search application; the inverted index is clearly a needed object. The figure below shows the inverted index objects instance

```
package ir;
import java.util.*;
import java.io.*;
import java.lang.*;
import java.text.*;
import ir.SimpleStreamTokenizer;

/* actual inverted index object -- built by the index builder -- loaded from disk */
/* by the index loader */

public class InvertedIndex implements java.io.Serializable {

    private HashMap    index;          /* actual inverted index          */
    private ArrayList  documentList;   /* stores structured info about each doc */

    /* constructor initializes by setting up the document object */
    InvertedIndex () {
        index = new HashMap();
        documentList = new ArrayList();
    }

    public void add(Document d) {
    }

    /* returns a posting list for a given term */
    public LinkedList getPostingList(String token) {
        LinkedList result = new LinkedList();

        if (index.containsKey(token)) {
            result = (LinkedList) index.get(token);
        } else {
            result = null;
        }
        return result;
    }
}
```

variables and method declarations for SimpleIR. The variable *index* defines the list of terms in the inverted index. The java data structure *HashMap* is used as it provides a layer of abstraction over a hashing table. The hashing algorithm is concealed to us (we can change the hashing algorithm if that is so desired) and collision resolution is handled by the *HashMap* implementation. Hence, the *get* method associated with a *HashMap* provides a term as a key to the map, and the *HashMap* returns the corresponding object. The *put* method stores entries in the *HashMap*. When the inverted index is initialized, the constructor instantiates a null *HashMap*, called *index*.

4.1 Adding a Document

To add a document to the index, SimpleIR has an *add* method associated with the *InvertedIndex*. We saw in Chapter 2 that the *Document* object was created and populated by the parser. The *add* method simply accepts a *Document* object and adds it to the inverted index. The *add* method is given below.

```

public void add(Document d) {
    String token;          /* string to hold current token          */
    TermFrequency tf;     /* holds current term frequency */
    Set termSet;          /* set of terms to add to index */
    LinkedList postingList; /* list of docID, tf entries    */
    HashMap terms = new HashMap();
    Integer documentFrequency;
    int df;

    /* add all the terms in the document to the index */
    long docID = d.getDocumentID(); // current document ID
    terms = d.getTerms();           // get current term map
    termSet = terms.keySet();
    Iterator i = termSet.iterator();

    /* loop through the terms and add them to the index */
    while (i.hasNext()) {
        token = (String) i.next();

        /* if we have the term, just get the existing posting list */
        if (index.containsKey(token)) {
            postingList = (LinkedList) index.get(token);
        } else {
            /* otherwise, add the term and then create new posting list */
            postingList = new LinkedList();
            index.put(token, postingList);
        }

        /* now add this node to the posting list */
        tf = (TermFrequency) terms.get(token);
        PostingListNode currentNode = new PostingListNode(docID, tf);
        postingList.add(currentNode);
    }
}

```

The method begins by getting the document identifier for the document to be added. Next, the list of distinct terms is obtained from the Document. An iterator is obtained for this list, and now, a loop begins which adds each term to the posting list. For a given term, we first check to see if a posting list already exists for this term. If it does, we simply retrieve the posting list (using the same *containsKey* method used in the *getPostingList* method). If no posting list exists, we instantiate a null *LinkedList* and associate this null list with the *index* *HashMap*. At this point, the term we are adding has a *postingList* associated with it. All that remains is to retrieve the *postingList*, instantiate

a new PostingListNode and add it to the postingList. The PostingListNode contains just the information that we wish to store for an occurrence of the term within a document.

The PostingListNode is a relatively straightforward document and is given below.

```
/* This will store and retrieve an entry in a posting list          */
/* typically an entry in a posting list contains the document identifier */
/* and the term frequency                                         */
public class PostingListNode implements Serializable {

    long documentID;
    TermFrequency tf;

    /* simple constructor with no args -- should not be called */
    public PostingListNode(long id, TermFrequency tf) {
        documentID = id;
        this.tf = tf;
    }

    public long getDocumentID() {
        return documentID;
    }

    public short getTF() {
        return tf.getTF();
    }
}
```

It simply contains a documentID and a TermFrequency object. The TermFrequency is a separate object because we wish to handle a case that does not come up too often. A two-byte term frequency can store 32767 occurrences of a term in a given document. Should the term have a higher frequency, we do not want an overflow condition to occur.

Instead of checking for this overflow condition throughout all of SimpleIR, we simply build a TermFrequency object that has a special *increment* method. The special increment method stops incrementing when the terms frequency exceeds 32767 and avoids an overflow condition. While capping the term frequency at 32767 may at first glance seem to skew the calculations, in fact, if a document is long enough to contain a single word 32,767 times, it is probably not the document you are looking for! (Besides,

it turns out, as we will see in discussing similarity measures, that very large term frequencies do not proportionately impact the relevance measure).

```
public class TermFrequency implements Serializable {  
  
    private static final short MAX_VALUE = 32767;  
    short tf;  
  
    public TermFrequency() {  
        tf = 1;  
    }  
  
    /* increment the tf as long as we have room for an increment */  
    public void Increment () {  
        if (tf <= MAX_VALUE) {  
            tf = (short) (tf + 1);  
        }  
    }  
  
    public void Set (short value) {  
        tf = value;  
    }  
  
    public short getTF () {  
        return tf;  
    }  
}
```

4.1.1 Adding our Sample Documents

Consider once again our two-sample document collection:

D1: The GDP increased 2 percent this quarter.

D2: The Spring economic slowdown continued to spring downwards this quarter.

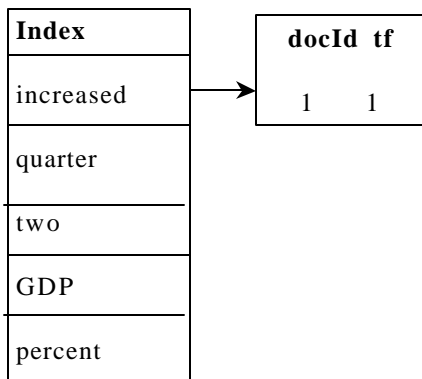
After the parser is called, we have two document objects. The Document objects are shown below. As discussed in chapter 2, punctuation is removed, all terms are converted to lowercase, and stop words are removed. The `distinctTerms` instance variable in the Document object contains a `HashMap` of all of the terms in a document. Hashing on the term yields the term frequency (*tf*) of the term. Since *spring* occurs twice in document two, it is our only example of a term with a *tf* higher than one.

D1	
documentID	1
distinctTerms	
term	tf
increased	1
quarter	1
2	1
gdp	1
percent	1

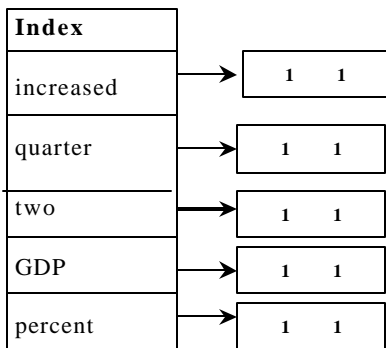
D2	
documentID	2
distinctTerms	
term	tf
spring	2
quarter	1
economic	1
slowdown	1
continued	1

Let us trace the *add* method, and assume it was sent the Document object that contains Document 1. The variable *termset* contains the set of distinct keys in the hashmap for document 1. This will be *{increased, quarter, 2, gdp, percent}*. Now we iterate through this list. Upon encountering *increased* we check if the term dictionary, *index* has it. It

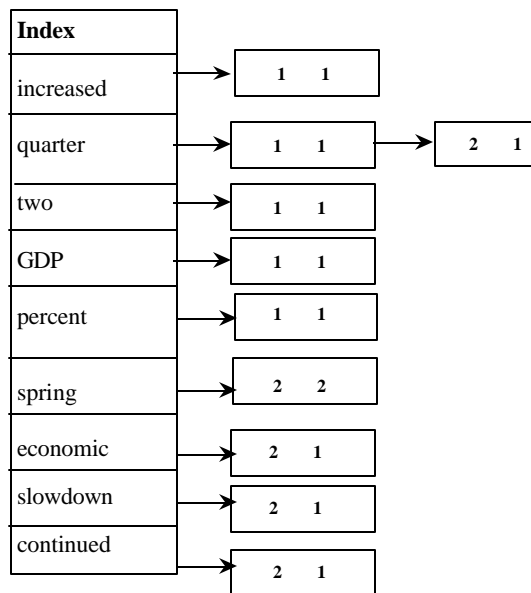
does not; so, we generate a new posting list and attach it to the *index*. Next we look up using the hashmap in document one the term frequency for *increased*, it is set to one so we now can generate a TermFrequency object that stores this value. We then create a PostingListNode that contains the documentID and the term frequency, both of which are one. We then add this PostingListNode to the posting list for Frequency. At the end of this iteration of the loop, we now have an inverted index that looks like:



All remaining terms in document 1 are processed similarly. At the end of processing document 1, our inverted index looks like:



After processing the second document, the inverted index will appear as:



Retrieving a Posting List

The *getPostingList* method takes a given term as an argument and returns a *LinkedList* that is the posting list associated with the term. If no linked list is available, this means the term does not currently exist in the inverted index and a value of *null* is returned. When a *token* is passed to *getPostingList*, the *HashMap* is checked with the *containsKey* method. This implements a hashing function on the term and returns true if the term exists in the index. When the term exists, we return the corresponding posting list, otherwise, a null is returned.

5 Processing a set of Documents

Now that we have a parser and an `InvertedIndex` method we can show how documents are added to the index.

The *build* method in the `IndexBuilder` object as given above illustrates a simple index build routine. It starts by instantiating a new `InvertedIndex` object. The index is cleared and a property `TEXT_FILE` is read from disk. `TEXT_FILE` indicates the file that should be indexed. A `STOPWORD_FILE` enables us to dynamically change the list of stop words used by the search engine. Next, a `TRECParser` is instantiated for this file with a given list of stop words.

It may seem overly trivial to only index a single file, but in truth, this *build* routine is designed for scalability. Most web search engines use many different processors to parse documents. By separating the parser from the index building, it is possible to launch numerous instances of the parser on numerous machines. Each parser accepts as a parameter the file to parse and the stopwords to use. After the parser is instantiated, the *readDocuments* method is called to actually read and parse the documents. A list of document objects is returned.

```

/* build the inverted index, reads all documents */
public InvertedIndex build () throws IOException {

    boolean endOfFile = false;
    int offset = 0;
    Document d;

    InvertedIndex index = new InvertedIndex();
    index.clear();

    /* get the name of the input file to index */
    inputFile = configSettings.getProperty("TEXT_FILE");

    /* now read in the stop word list */
    String stopWordFileName = configSettings.getProperty("STOPWORD_FILE");

    /* now lets parse the input file */
    TRECParser parser = new TRECParser(inputFile, stopWordFileName);
    documentList = parser.readDocuments();

    /* put the document list into the inverted index */
    index.setDocumentList(documentList);

    Iterator i = documentList.iterator();
    System.out.println("Starting to build the index");
    while (i.hasNext()) {
        d = (Document) i.next();
        index.add(d);
    }
    return index;
}

```

An interesting side effect is that we can easily build separate stop words for separate domains. All that is needed is a controlling routine that invokes multiple parsing threads and dispatches them to different machines. As discussed in Chapter 2, each parser returns a list of documents so when a parser returns, its results are sent to the index builder. We describe the controller for the more scalable, distributed, multi-processing solution later in Chapter xx. For now, we focus on simply parsing a single file.

The *setDocumentList* simply copies the list of documents into the InvertedIndex Object. Once this is done, we loop through the documents that were parsed and call the *add* method to add them to the InvertedIndex. This simple loop is responsible for adding

all of the documents to the `InvertedIndex`. Notice that the `IndexBuilder` requires no knowledge of how the `InvertedIndex` is actually structured, a simple `add` method hides all the details of how a document is actually added. This will come in handy when we talk about compressing the `InvertedIndex` in chapter xxx.

6 Updating the inverse document frequency

We For query processing, many search engines use an automatically defined weight for each keyword. Many weights exist but we only describe a simple one for now. For a given term t , the *inverse document frequency* (*idf*) can be computed as:

$Idf(t) = \log_{10}(N / df)$ where *IDF* is the inverse document frequency, N is the number of documents and df is the number of documents that contain the term. For a 1,000 document collection a term that occurs in all 1,000 documents has an *idf* of $\log(1,000 / 1,000) = 0$. Hence, a term that occurs with such a frequently probably does not help any query find relevant documents. A term that occurs in only one document should be given a high weight. The *idf* of a term that occurs in only one document has a weight of $\log(1,000 / 1) = 3.0$.

Many search engines have a stage where the *idf* is computed during index time, right after the inverted index is populated. We do not do that here, because of the data structures that are being used. The fact that we are using the Java JDK means that the `LinkedList` object used to store the posting list includes a `size` method that obtains the size of the posting list. Looking at the JDK source, it can be seen that the object has a private instance variable called `size` that is used to compute this. Hence, any update to a posting list results in an update to the `size` variable. The size of the posting list is really just the

document frequency. The bottom line is that the *idf* can easily be computed at query time instead of at index time because we are naturally tracking the document frequency simply because we are using the JDK LinkedList object to store our posting list.

7 Saving the Inverted Index

The inverted index we described in this chapter must entirely fit into memory prior to being saved. In large search engines, we cannot assume that we have sufficient memory (even though reasonably sized machines are now being built with up to 8 GB of RAM per processor). We have started with this simplifying assumption so that we could describe the basic operations of the inverted index. With the code we provided here, storing the inverted index for subsequent use is trivial. We simply make use of the JDK's serialization operators such as *readObject* and *writeObject* that takes an existing object and stores it on disk. The figure below shows the `InvertedIndex.write` method used to save the `InvertedIndex` object described in this chapter.

```

/* this will the index to disk */
public void write(Properties configSettings) {
    FileOutputStream ostream = null;
    ObjectOutputStream p = null;

    String outputFile = configSettings.getProperty("INDEX_FILE");
    try {
        ostream = new FileOutputStream(outputFile);
        p = new ObjectOutputStream(ostream);
    }
    catch (Exception e) {
        System.out.println("Can't open output file.");
        e.printStackTrace();
    }

    try {
        p.writeObject(this);
        p.flush();
        p.close();
        System.out.println("Inverted index written to file ==> "+outputFile);
    }
    catch (Exception e) {
        System.out.println("Can't write output file.");
        e.printStackTrace();
    }
}

```

The output file location is identified from the configuration file. Once this is done an `ObjectOutputStream` is opened and the `writeObject` method is called to write the object to disk. We will see in Chapter xx how much more work is required to handle a situation where we cannot fit the entire index into memory.

8 Summary

We described the basics of creating an inverted index. Essentially, after parsing, documents are fed to an `add` method that adds an inverted index. The inverted index consists of both a term dictionary and, for each term, a *posting list* that is a list of documents that contain the term. We demonstrated how an `IndexBuilder` simply passes documents to the `add` method to create the inverted index. A key simplifying assumption made in this chapter is that the entire inverted index fits in primary memory. When we will relax this assumption in Chap xx, we will end up with a far more complicated `IndexBuilder`.

Exercises

1. Run the index builder on some documents. Do all posting lists have about the same number of terms? Compute the min, max, and average sized posting list.
2. Run the index builder on several different test document collections; compute the index overhead required for each document.
3. Profile the index builder. Identify which routines are called the most and identify the percentage of time taken on each method. Suggest techniques that can be used to improve performance of the index builder.