

MG4J (big): The Manual

MG4J (big): The Manual

Table of Contents

1. A Quick Tour of MG4J	1
Building your first index	1
Building a compressed collection	2
More options	3
Querying MG4J	4
More sophisticated queries	5
A semantic index	5
A TREC index	7
2. Behind the scenes: The indexing process	9
Introduction	9
Preamble: terms, dictionaries and term-related maps	9
Scan: Building batches	10
Time/space requirements	15
Combining batches	15
Virtual fields in MG4J	16
Virtual fields and virtual fragments	17
Document resolvers	17
What is a document resolver actually doing: virtual texts and gaps	19
Payload-based indices	19
3. Performance	21
Indexing Time	21
Setting up the index structure	21
Setup Time	21
Query Time	22
4. Clusters & Partitioning	23
Documental vs. Lexical	23
Partitioning vs. Clustering	23
Creating a Cluster	23
5. Accessing MG4J indices programmatically	24

Chapter 1. A Quick Tour of MG4J

Building your first index

Indexing in MG4J is centered around documents, either exposed by means of sequences or of collections. For the time being, let us concentrate on collections, which are randomly addressable lists of documents.

Each document in a collection is associated with a *title* and a *URI*. Typical titles are filenames, or titles from HTML documents. URIs can be the actual URL of a page. To build our first document collection, we use the main method of the class `FileSetDocumentCollection`, which allows to build and serialize a set of documents specified by their filenames. As a typical case, we will build a collection out of your Javadoc documentation directory. Supposing your Javadocs are located in `/usr/share/javadoc`, you may try the following:

```
find /usr/share/javadoc/ -iname \*.html -type f | \
  egrep -v "(package-|-tree|class-use|index-.*.html|allclasses)" | \
  java it.unimi.dsi.big.mg4j.document.FileSetDocumentCollection \
    -f HtmlDocumentFactory -p encoding=UTF-8 javadoc.collection
```

Let us try to understand what's happening. We are providing as input to the main method of the class a list of files, one per line. Moreover, we are specifying (using the `-f` option) a *factory*, that is, something that will turn a pure stream of bytes (provided, in this case, by a file) into a document made by several *fields* (for instance, title and main text). The factory needs to know the encoding of the files, and we are specifying UTF-8 as a *property*. All this information is serialised and stored in a file named `javadoc.collection`. Note that since we are using a standard MG4J factory, we can avoid to write the full factory class name (`it.unimi.dsi.big.mg4j.document.HtmlDocumentFactory`).

If you try and look into the file `javadoc.collection`, you will discover that this is indeed a typical, serialized version of a Java object; note that the file *is not going to contain* the files that are part of the collection, but only their name. This means, in particular, that the very existence of the collection will depend on the existence of the files spanned by the collection; in other words, deleting or modifying any of the indexed file may cause inconsistency in the collection (and, more importantly, in the index produced in the following steps). This is true of almost every collection: document collections may base their existence on some external data (files, web pages, mailbox files etc.), and they usually become inconsistent as soon as such data are modified, changed or deleted.

It is now time to index our collection. To do so, we simply pass the collection to the main method of the class `IndexBuilder`, which scans all documents in the collection and produces a number of indices, one for each field of the collection. The number of fields depends on the factory used to produce documents: in our case, we will get indices for the title (the content of the HTML `title` element, if present; the filename is used, instead, if the title element is absent) and the body (the textual content of the entire HTML page). Additionally, `FileSetDocumentCollection` sets the URI of each document to a URI pointing to the absolute location of the file in the file system; the document title is, once more, going to be the title appearing in the HTML content.

```
java it.unimi.dsi.big.mg4j.tool.IndexBuilder \
  --keep-batches --downcase -S javadoc.collection javadoc
```

The class `IndexBuilder` has a large number of options, as it runs in sequence the two phases of the indexing process. These phases are also available separately, mainly in the case of very large collection (hundreds of millions of documents) for which the memory limits are rather tight. Note that we did not specify a memory option, for instance, `--Xmx256M`, as it is not necessary (and might be even pernicious)

on newer Java virtual machines, which allocate memory dynamically; if you run into memory problem, please allow for more memory.

In this example, we have used the `--downcase` option that forces all the terms to be downcased: this means that the index will collapse words that differ only for the presence of upper/lowercase letters. For example, terms `String` and `string` will not be distinguished. More generally, you could specify a different *term processor* for custom term modification (in this case, the `DowncaseTermProcessor` class has been implicitly chosen). The `-S` option specifies that we are producing an index for the specified collection (`javadoc.collection`): if the option was omitted, `Index` would expect to index a document sequence read from standard input (more about this below). The `--keep-batches` option is not used normally, but we specify it here so to have a look at the temporary files generated during the indexing process. The last, unflagged option, `javadoc`, is the only mandatory option for `Index`, and it is the *index basename*, the basename after which all index files are stemmed.

Since our collection has documents containing two fields, named `title` and `text`, there will be two sets of index files: each will be named, by convention, with the index basename followed by the field name (separated with a dash). Hence, there will be index files named `javadoc-title.something` and files named `javadoc-text.something`.

We have now built indices, and we are ready to query them using a web server. This is very easy in MG4J: we just run the main method of the `Query` class specifying the `-h` option and passing as argument the indices and (for showing snippets) the collection:

```
java it.unimi.dsi.big.mg4j.query.Query -h -i FileSystemItem \  
-c javadoc.collection javadoc-text javadoc-title
```

We can now either use the command line (if you have **rlwrap** installed, you can put it to good use), or open the search page by pointing our browser to `http://localhost:4242/Query` and start querying the collection. Note that `-i` option, which specifies what to link to result items: the specified class links a file in the file system using a local HTTP server (the observation about class names made for factories applies here, too).

Note that the names we specified for the indices (e.g., `javadoc-text`) are actually URIs, so you can add options much like in a web query. For instance, `javadoc-text?inMemory=1` would load the index into main memory, whereas `javadoc-text?mapped=1` would try to use low-level memory-mapping features of the operating system to cache the most frequently used part of the index in main memory.

Building a compressed collection

During the indexing process, it is possible to build a compressed version of the collection used to build the index itself. There are several ways to do that (and you can program your own). The easy way is to use the `-B` option, which accepts a basename from which various files will be generated. By default, MG4J will generate a `SimpleCompressedDocumentCollection` (but you can write your kind of collection, provide a `DocumentCollectionBuilder` for it, and just pass it to `Scan`). For instance,

```
java it.unimi.dsi.big.mg4j.tool.IndexBuilder \  
-B javacomp --downcase -S javadoc.collection javadoc
```

would generate during the indexing process a collection, which would be named `javacomp.collection`, that you can pass to `Query`. The collection is actually a `ConcatenatedDocumentCollection` that exhibits a set of component instances of `SimpleCompressedDocumentCollection`, one per batch (this arrangement makes collection construction more scalable). This is an important fact to know, because if you move

`javacomp.collection` somewhere else you will also need to move all files stemmed from `javacomp@`, which contain the component collections.

Note that in this particular case there is no need to build another collection—the `FileSetDocumentCollection` used to build the index can be happily passed to `Query`. This is, however, not always the case, as MG4J builds indices out of *sequences*—objects that expose the data to be indexed in a sequential fashion. A typical example is the default, built-in `InputStreamDocumentSequence`. Assume you have a file `documents.txt` that contains one document per line. You can index it as follows:

```
java it.unimi.dsi.big.mg4j.tool.IndexBuilder \  
  --downcase -p encoding=UTF-8 javadoc <documents.txt
```

Note the `-p encoding=UTF-8` option, which sets the encoding of the text file. This command will create a single index with field name `text` (you can change the field name with another property—see the `InputStreamDocumentSequence Javadoc`). When you query the index, results will be displayed as numbers (positions in the original text), as `Query` has no access to a document collection. But if you specify the `-B` option, you can build on the fly a collection that can be used by `Query` to display snippets.

The kind of collection that is create is customisable. The interface `DocumentCollectionBuilder` specifies what a *collection builder* should provide to be used at indexing time, and a builder can be specified with the `--builder-class` option. For instance, by specifying `--builder-class ZipDocumentCollectionBuilder` you will get back the behaviour of the obsolete `-z` option—building a `ZipDocumentCollection`.

There are many other collections you can play with—they are contained in the package `it.unimi.dsi.big.mg4j.document`. There are collections for reading from JDBC databases, comma-separated files, and so on (and, of course, you can write your own). Some collections let you play with other collections: `ConcatenatedDocumentCollection` exhibits a set of collection as a single collection that concatenates their content. `SubDocumentCollection` exhibits a contiguous subset of documents of a given collection as a new collection. Some of these classes have constructor that follow `dsutils`'s `ObjectParser` conventions, and thus can be constructed directly for the command line. One such class is `SubDocumentCollection`; the following command line uses the `-o` option to build such a collection on the fly:

```
java it.unimi.dsi.big.mg4j.tool.IndexBuilder \  
  --downcase -oSubDocumentCollection\ (javadoc.collection,0,10\ ) mini
```

The above command would just index the first ten documents of `javadoc.collection` (see the `Javadoc` of `SubDocumentCollection` for more details). You can then use the option `-o` to pass the same collection to `Query`, or build a compressed collection during the indexing phase.

More options

All tools and classes used so far have a large number of options that make them highly configurable. For instance, there are other properties of a factory that can be specified—please have a look at the `Javadoc` of the document factory you are using. For instance, a common property is `wordreader`, which makes it possible to specify a different instance of `WordReader`—the class that it used to segment text into words and non-words. The standard `WordReader (FastBufferedReader)` considers just letters and digits as part of a word, but you can choose your variant, and even specify it directly on the command line: for instance, `-pwordreader=FastBufferedReader\ (_)` specifies that underscores should be considered as part of a word. More generally, you can specify an expression that follows `dsutils`'s `ObjectParser` conventions and that will be used to instantiate a `WordReader`.

All MG4J tools implement the standard `--help` option, which will display a detailed help text.

Querying MG4J

Querying MG4J is easy if you already used a text-indexing system. The simplest possible query is a single term, e.g., `class`: the answer that you will obtain by such a query is the set of all documents (in our case: all files among those that have been indexed) that contain the word `class` (or any other uppercase/lowercase variant thereof).

There are several additional operators you might want to try:

- **AND**: writing more than one term (separated by whitespace) means that you want to look for documents that contain *all the specified words* (not necessarily in the same order or consecutively); for example, the query `InputStream Reader encoding` means that you want to look for documents that contain *all the given words*; you can convey the same meaning by using the operator `&` (a.k.a. AND), thus writing `InputStream & Reader & encoding` instead;

- **OR**: if you want to write a disjunctive query you can use the operator `|` (a.k.a. OR); thus, for example, the query `InputStream | Reader | encoding` means that you are looking for documents that contain *any of the given words*;

- **NOT**: you can use the operator `!` (a.k.a. NOT) to mean negation; thus, for example, the query `InputStream & !Reader` means that you are looking for documents that contain the first term *but not the second*;

- **phrase**: you can force consecutivity by using quotation marks; thus `"InputStream Reader"` means that you want to look for documents that contain these two words *consecutively*;

- **proximity restriction**: you can limit your search to documents where the words you are searching appear within a limited portion of the document; this is done with the tilda operator; for example, `(InputStream Reader)~5` means that you are looking for documents where the two given words appear (in any order) within 5 words from each other;

- **ordered AND**: writing more than one term separated by `<` will find documents containing the given terms in the specified order.

- **wildcard search**: you can perform wildcard searches by appending `*` at the end of a term; for example, `term*` will look for documents containing "term", "terms", "termed" and so on.

- **parentheses**: you can use parentheses to enforce priority when building complex queries; parentheses are not needed in many cases, but they are necessary, for example, when a boolean query is written within a phrase; for example, if you want to look for the word `InputStream` followed by `Reader` or `Writer`, you will enter the query `"InputStream (Reader | Writer)"`.

- **index specifiers**: prefixing a query with the name of an index followed by a colon you can restrict the search to that index. The name of an index is by default the name of the field that it has indexed, so `title:Reader` will search for `Reader` just in titles.

- **range queries**: if you created an index containing *payloads* (dates, integers, etc.) you can perform range queries using square brackets and two dots: for instance, assuming the existence of a field `date` the query `[20/2/2007 .. 23/2/2007]` will search for documents whose date is between 20 February and 23 February 2007, inclusive.

MG4J will emphasise intervals satisfying the query. By clicking on the link of a document, the document will be opened in the browser.

The description we have just given just scratches the surfaces of the queries you can write with MG4J: *all* the operators can be freely combined, obtaining very sophisticated constraints on the documents returned. More information on this topic can be found in the documentation of the package `it.unimi.dsi.big.mg4j.search`.

More sophisticated queries

MG4J actually provide very sophisticated query tuning. In particular, it provides *scorers*, which let you reorder the documents satisfying a query depending on some criterion. To use this features, you must use the command line interface, albeit all settings will be used for the subsequent web queries.

Type `$` to get some help on the available options. A basic command is `$mode`, which lets you choose the kind of result: just the document number and title, the intervals, snippets and so on. Some options require a full index and a collection (for instance, snippets). The most interesting command, however, is `$scorer`, that lets you choose a scorer for your documents. For instance,

```
$score BM25Scorer VignaScorer
```

reproduces the standard settings, using a BM25 scorer and a scorer that shows firsts documents satisfying your queries more frequently and in smaller intervals, linearly combined with equal weight. Scorers are described in the documentation of the package `it.unimi.dsi.big.mg4j.search.score`.

When you use a scorer, it is a good idea to use *multiplexing*: when multiplexing is on, each query is multiplexed to all indices (by default, a query is directed to the first index specified on the command line). Just type

```
$mplex on
```

Of course, you can always choose a specific index with the colon notation. You can also change the weight of your indices (which is particularly useful when multiplexing):

```
$weight text:1 title:3
```

In this way, weight-based scorers will usually consider the `title` field three times more important than the `text` field.

You can also change the way snippets (or intervals) on display are chosen: MG4J provides an *interval selector*, a class that will try to choose the best intervals to be shown. You can set the maximum length of an interval, and the maximum number of intervals:

```
$selector 3 40
```

will show at most three intervals, and intervals longer than 40 characters will be broken. All these changes are reflected in the web interface.

If you want to learn more about query resolution, you should have a look at the documentation of the class `it.unimi.dsi.big.mg4j.query.QueryEngine`, which embodies all the logic used to answer queries in MG4J.

A semantic index

For our next example, we will put to good use the semantically annotated snapshot of the English Wikipedia [http://barcelona.research.yahoo.net/dokuwiki/doku.php?id=semantically_annotated_snapshot_of_wikipedia]

created at Yahoo!. The collection exhibits Wikipedia articles as a number of *parallel texts*, one of which is the sequence of tokens, whereas others provide information like "this token is a person's name". MG4J provides an *alignment* operator that can be used with parallel texts to align results of two queries—in practise, you can ask which results of an arbitrary query match certain semantic conditions. The support has a few rough edges, but it's an interesting example nonetheless.

First of all, you must get the collection, for instance through Yahoo!. The collection is made by a number of text files (stored, say, in `/your/wiki/dir/`), which must be recorded in a `WikipediaDocumentCollection` as follows:

```
find /your/wiki/dir/ -type f \  
    java it.unimi.dsi.big.mg4j.document.WikipediaDocumentCollection wiki.collection
```

Similarly to a `FileSetDocumentCollection`, the serialized collection will contain references to the files and also a compacted representation of pointers to the start of each record in each file. You can now index as before, and invoke `Query`. In this particular case, we use tokens and some semantic tagging.

```
java it.unimi.dsi.big.mg4j.query.Query -h \  
    -c wiki.collection wiki-token wiki-WSJ
```

We're now ready. For instance, the query

```
Washington ^ WSJ: (B\E\PERSON | B-I\PERSON)
```

will search for "Washington", but only in those positions that have been marked as person names. This happens because the alignment operator `^` solves the left query, and then keeps just those results whose positions are the same as those of the second query, which can be on a completely different index. Note that the left and right query of an alignment operator are completely arbitrary, and the overall query is a standard query on the first index. Thus,

```
"(Washington ^ WSJ: (B\E\PERSON | B-I\PERSON)) was"
```

would search for "Washington" as a person name, but only if immediately followed by "was".

If you click on the title of a result, you will be brought to the corresponding Wikipedia page, as the factory embodied in the `WikipediaDocumentCollection` sets the document URI to the Wikipedia page. If you want a more technical view of what's happening, you can use the `GenericItem` class, which will display in a very simple manner the content of all fields.

Another interesting property of the Wikipedia examples is the end-of-sentence markers (§) are indexed. You can use another fairly exotic operator, *Brouwerian difference*, to restrict your results to queries that are true *inside a sentence*. The semantics of query in MG4J is a set of *minimal intervals* that represent region of text that satisfy the query. For instance, for the query `was killed` the intervals describe the smallest regions of text in which `was` and `killed` do appear. But the difference operator (a minus) will eliminate the intervals generated by the left query that contain one or more interval from the right query. Thus,

```
was killed - §
```

will perform the same search, but we will see *only those results for which there are regions of text not containing §*. In other words, results will be restricted to be within a sentence, as matches (i.e., again, regions of text) that cross sentence borders will be killed by the difference operator.

Finally, the *index remapping* operator comes handy in two situations: display the results of a field using another, parallel field, or applying positional operators to results from different fields. If you search for `WSJ: (B\E\ :PERSON | B-I\ :PERSON)`, the resulting snippets will be rather ugly:

```
Document #205 [2.000000] Protected_areas_of_Tasmania
WSJ: ...0 0 0 B-N:CARDINAL I-N:CARDINAL I-N:CARDINAL I-N:CARDINAL
0 B-E:PERSON I-E:PERSON 0 0 0 0 0 B-N:CARDINAL 0 ...
```

```
Document #258 [1.999152] List_of_people_by_name:_Kea-Kel
WSJ: ? 0 0 B-E:PER_DESC 0 0 0 B-E:PERSON ? 0 0 0 0 ? 0 ...
```

This is correct, as MG4J is displaying results from the `WJS` field. It is however easy to *remap* those results to another index: if we try `(WSJ: (B\E\ :PERSON | B-I\ :PERSON)) {{WSJ->token}}`, the result will be like

```
Document #205 [2.000000] Protected_areas_of_Tasmania
token: ...but it contains no fewer than 495 separate Protected Areas
with a total area of 22 , ...
```

```
Document #258 [1.999152] List_of_people_by_name:_Kea-Kel
token: ? List of people by name : Kea-Kel ? Access to rest of list ? Access ...
```

Snippets are now represented using the parallel content of the `token` field.

Assume now that we want to find a person's name *immediately followed* by the term "was". A direct attempt would be trying the query `"WSJ: (B\E\ :PERSON | B-I\ :PERSON) was"`: the result would be an error message ("The phrase operator requires subqueries on the same index"). This is correct, because intervals returned by the two subqueries of the phrasal operators are on different indices—mixing them makes no sense. However, if you're sure that you are handling indices on parallel texts, the idea *does* make sense, and we can convince MG4J about this as follows:

```
{token, WSJ}>"(WSJ: (B\E\ :PERSON | B-I\ :PERSON) {{WSJ->token}}) (token:was) "
```

```
Document #225 [1.996363] Days_of_our_Lives
token: ...family tree by way of SORAS . ? Abby was rapidly aged to a teenager . ? Abby .
```

```
Document #467 [1.995153] Airey_Neave
token: ...in Northern Ireland . ? In 1975 , Neave was the campaign manager for Margaret
```

In this example, we qualified also "was" with an index selector to avoid problems in case multiplexing is on.

A TREC index

In this section we discuss thoroughly the construction of an index based on the TREC GOV2 collection (Text REtrieval Conferences are series of events organized by the National Institute of Standards and Technology to evaluate scientifically and reproducibly systems for information retrieval). TREC collections must be bought to be used, but they are very commonly used for scientific work. In this example

we use a `TRECDocumentCollection` to index GOV2 (25 million web pages). Be warned that different collections have slightly different formats, and `TRECDocumentCollection` might need some tweaking to work with them (we are making it more and more flexible on a per-request basis).

GOV2 data comes as a list of files in directories named `GX000`, `GX001`, and so on. The file themselves are zipped (you can create the collection with unzipped files, however, if you want faster access).

```
find GX??? -iname \*.gz | \  
  java it.unimi.dsi.big.mg4j.document.TRECDocumentCollection \  
  -f HtmlDocumentFactory -p encoding=ISO-8859-1 -z trec.collection
```

After some grinding, you'll get the collection. Note that this process is mainly useful for accessing later the collection in a random fashion—for instance, to generate snippets.

Since we want to index anchor text, we must now generate URIs that will represent each document.

```
java it.unimi.dsi.big.mg4j.tool.ScanMetadata -S trec.collection -u trec.uris
```

Note the `-U` option. GOV2 contains many duplicate (and even triplicate) URLs, modulo trivial normalizations such as adding a bar after the host name. The `-U` option is a very crude way of making them unique. (A more principled mechanism would involve merging all documents with identical URLs, but that should have been addressed when GOV2 was built.)

We are now ready to build our index:

```
java it.unimi.dsi.big.mg4j.tool.IndexBuilder -S trec.collection \  
  -t snowball.PorterStemmer -a -v anchor:trec.vdr trec
```

We set a rather large batch size, assuming that a lot of memory is available. As we said, `Scan` will try to detect low-memory conditions and dump batches automatically, but you can lower the batch size, in case you run into out-of-memory errors. We also require a downcasing Porter stemmer (all Snowball [<http://snowball.tartarus.org/>]-based stemmers downcase terms). Beware again: you will be generating hundreds of batches, so you must be able to open a few thousand files in the combination phase. When the indexing process is completed, you can query the index as usual.

Chapter 2. Behind the scenes: The indexing process

Introduction

The main point of MG4J is the construction of *inverted indices*: an inverted index is just like the index you can find at the end of a book is a list of the occurrences in the text of every term. Building an inverted index is a complex process that MG4J perform essentially in two phases. Furthermore, there is another step that is called *term map construction* that is optional, depending on the kind of functionalities you require of your index.

Besides traditional indices, MG4J provides *payload-based* indices, which are used to store metadata associated to documents such as dates, integers, and so on.

In this chapter we will try to dissect the whole process to give you an idea of what happens when you run the `Index` class.

Preamble: terms, dictionaries and term-related maps

Before starting our description of the indexing process, a brief introduction is necessary to present some basic concepts. MG4J has to do with documents (e.g., HTML files, mail messages etc.), and every document is composed by fields (e.g., the fields of a mail message will probably be its subject, sender, recipient, body etc.). Although, as we shall see, MG4J will provides support for non-textual fields, its "bread and butter" is with textual fields, and for the time being we shall assume that we are dealing with documents composed of just one textual field.

A textual field (in our simplified view: a document) is a sequence of words: it is up to the factory producing the document to decide how to choose words (e.g., one may want to discard digits or not), and how to *segment* the document. For instance, the typical letter-to-nonletter transition used to split Western languages does not work very well with, say, Chinese. However, once segmentation has produced suitable words, they must be turned into indexable *terms*: for instance, you may want to downcase your words, but at the same time you may want to keep "ph" (as in "this soap's ph") separated from "Ph" (as in "Ph.D. degree"). You may also want to make more dramatic transformations, such as *stemming*, or avoid indexing a term altogether. All these operation are performed by a *term processor*, which can be specified on the command line. The option `--downcase`, for instance, selects for you the class `it.unimi.dsi.big.mg4j.index.DowncaseTermProcessor`. The chosen processor is recorded into the index structure: this is essential for interpreting queries correctly.

Note that in the design of other search engines segmentation and processing are somehow mixed into a generic tokenisation phase. We prefer to split clearly between *linguistic* and *algorithmic* term processing. Linguistic processing depends only on the writing customs of a language, whereas algorithmic processing might be language neutral (we do not exclude, however, that it might be language dependent, too).

If you scan the whole document collection, you can collect all terms that appear in it; the set of all such terms is called the *term dictionary*. Note that every term in the dictionary appears in some (at least one) document, and probably it will appear in many documents, possibly even many times in some documents. (By the way: terms that appear in just one document are called *hapax legomena*, and they are far more frequent than one might expect in many collections, especially due to typos).

MG4J, like any other indexing tool, does not treat internally terms as character sequences, but it uses numbers. This means that terms in the dictionary are assigned an index (a number between 0 and the dictionary size minus 1), and that this index is used whenever the application needs to refer to a term. Usually, indices are assigned in lexicographical order: this means that index 0 is assigned to the first term in lexicographic order, index 1 to the next one and so on). The assignment between terms and indices is stored in a suitable data structure, that compactly represents both the dictionary and the map.

There are many possible different representations of this assignment, each with certain memory requirements and each allowing different kind of access to the data.

- The simplest kind of representation of a dictionary is the *term list*: a text file containing the whole dictionary, one term per line, in index order (the first line contains term with index 0, the second line contains term with index 1 etc.). This representation is not especially efficient, and access-time is prohibitive for most applications. Usually, a file containing the term list is stemmed with `.terms`; if the terms are *not* sorted lexicographically, the file is stemmed with `.terms.unsorted`.

- A much more efficient representation is by means of a *monotone minimal perfect hash function*: it is a very compact data structure that is able to answer correctly to the question “What is the index of this term?” (more precisely, “What is the lexicographical rank of this term in the term list?”). You can build such a function from a *sorted* term list using the (main method of) implementations available in Sux4J [<http://sux4j.dsi.unimi.it/>].

- Monotone minimal perfect functions are very efficient and compact, but they have a serious limit. As we said before, they can answer correctly to the question “What is the index this term?”, but *only for terms that appear in the dictionary*. In other words, if the above question is posed for a term that does not appear anywhere, the answer you get is completely useless. This is not going to cause any harm, if you are sure that you will never try to access the function with a term that does not belong to the dictionary, but it will become a nuisance in all other cases. To solve this problem, you can *sign* the function. A signed function will answer with a special value (-1) that means “the word is not in the dictionary”. You can sign any function using the signing classes in `dsiutils` [<http://dsiutils.dsi.unimi.it/>] (e.g., `ShiftAddXorSignedFunction`).

- Signed and unsigned monotone minimal perfect hash functions are ok, as long as you don’t need to access the index with wildcards. Wildcard searches require the use of a *prefix map*. A prefix map is able to answer correctly to questions like “What are the indices of terms starting with these characters?”. This is meaningful only if the terms are lexicographically sorted: in this case, the indices of terms starting with a given prefix are consecutive, so the above question can be answered by giving just two integers (the first and the last index of terms satisfying the property). You can build a prefix map by using the main method of one of the implementation of the `PrefixMap` interface, e.g., `ImmutableExternalPrefixMap` from `dsiutils`—actually, this is exactly what happens when you use `IndexBuilder`, albeit you can specify a different class for the term map using an option.

Scan: Building batches

In this step, MG4J scans the whole document collection producing the so-called batches. Batches are subindices limited to a subset of documents, and they are created each time the number of indexed documents reaches a user-provided threshold, or when the available memory is too little.

An occurrence is a group of three numbers, say (t,d,p) , meaning that *term with index t appears in document d at position p* . Here, both the term and document are represented by a long integer, called, in the second case, the *document pointer*, which is in most cases the position of the document in the document collection (0 for the first document, 1 for the second document and so on). Position is an integer that represents where the term occurs in the document.

To understand what the scanning phase really does, suppose you have three documents:

Document pointer	Document
0	I love you
1	God is love
2	Love is blind
3	Blind justice

Here is the dictionary produced initially by the scanning phase:

Term index	Term
0	blind
1	god
2	i
3	is
4	justice
5	love
6	you

Now, at least conceptually, this is the list of occurrences:

**Occurrences
(in the same
order as they
are found when
scanning the
documents)**

(2,0,0)	(5,0,1)
(6,0,2)	(1,1,0)
(3,1,1)	(5,1,2)
(5,2,0)	(3,2,1)
(0,2,2)	(0,3,0)
(4,3,1)	

This simply means that:

- term 2 (I) appears in document 0 at position 0;
- term 5 (love) appears in document 0 at position 1;
- term 6 (you) appears in document 0 at position 2;

and so on. Inverted lists can now be obtained by re-sorting the occurrences in increasing term order, so that occurrences relative to the same term appear consecutively:

Term	Occurrences
0 (blind)	(0,2,2) (0,3,0)
1 (god)	(1,1,0)
2 (i)	(2,0,0)
3 (is)	(3,1,1) (3,2,1)
4 (justice)	(4,3,1)
5 (love)	(5,0,1) (5,1,2) (5,2,0)
6 (you)	(6,0,2)

Now, the indexer must:

- scan all documents and extract occurrences;
- if the list of terms have not yet been obtained, gather new terms as they are found;
- sort the terms in alphabetical order, renumbering all occurrences correspondingly;
- (if required) renumber the documents and sort them in increasing order,
- sort, at least partially, the occurrences found in increasing term order;
- when the number of accumulated documents reaches a given threshold, create a subindex containing the current batch of occurrences.

The last point needs further explanation. Since occurrences are *a lot* it is not reasonable to think that they can be all kept in memory. What the indexing pass does is keeping an internal batch where occurrences are stored as they are found; when the batch is full, it is ordered by term, and flushed out on disk under the form of a subindex. Every batch will be in term order, but different batches may (and usually, will) contain occurrences of the same termG.

Getting back to the example given in Chapter 1, where we indexed the collection `javadoc.collection`, the basename of the resulting index is going to be `javadoc` (as usual, completed with the field name). After running `IndexBuilder`, we get the following files:

```
-rw-r--r--  1 vigna  vigna      430 May 13 12:02 javadoc-text.cluster.properties
-rw-r--r--  1 vigna  vigna     144 May 13 12:02 javadoc-text.cluster.strategy
-rw-r--r--  1 vigna  vigna    20k May 13 12:02 javadoc-text.frequencies
-rw-r--r--  1 vigna  vigna    29k May 13 12:02 javadoc-text.globcounts
-rw-r--r--  1 vigna  vigna   1.1M May 13 12:02 javadoc-text.index
-rw-r--r--  1 vigna  vigna    52k May 13 12:02 javadoc-text.offsets
-rw-r--r--  1 vigna  vigna   5.8M May 13 12:02 javadoc-text.positions
-rw-r--r--  1 vigna  vigna    54k May 13 12:02 javadoc-text.posnumbits
-rw-r--r--  1 vigna  vigna     387 May 13 12:02 javadoc-text.properties
-rw-r--r--  1 vigna  vigna     9.4k May 13 12:02 javadoc-text.sizes
-rw-r--r--  1 vigna  vigna     1.2k May 13 12:02 javadoc-text.stats
-rw-r--r--  1 vigna  vigna   175k May 13 12:02 javadoc-text.termmap
-rw-r--r--  1 vigna  vigna   412k May 13 12:02 javadoc-text.terms
-rw-r--r--  1 vigna  vigna    14k May 13 12:01 javadoc-text@0.frequencies
-rw-r--r--  1 vigna  vigna    19k May 13 12:01 javadoc-text@0.globcounts
-rw-r--r--  1 vigna  vigna   3.8M May 13 12:01 javadoc-text@0.index
-rw-r--r--  1 vigna  vigna    40k May 13 12:01 javadoc-text@0.offsets
-rw-r--r--  1 vigna  vigna    37k May 13 12:01 javadoc-text@0.posnumbits
-rw-r--r--  1 vigna  vigna    342 May 13 12:01 javadoc-text@0.properties
-rw-r--r--  1 vigna  vigna    4.7k May 13 12:01 javadoc-text@0.sizes
```

```

-rw-r--r--  1 vigna  vigna  264k May 13 12:01 javadoc-text@0.terms
-rw-r--r--  1 vigna  vigna  11k May 13 12:02 javadoc-text@1.frequencies

-rw-r--r--  1 vigna  vigna  15k May 13 12:02 javadoc-text@1.globcounts

-rw-r--r--  1 vigna  vigna  2.7M May 13 12:02 javadoc-text@1.index
-rw-r--r--  1 vigna  vigna  31k May 13 12:02 javadoc-text@1.offsets

-rw-r--r--  1 vigna  vigna  29k May 13 12:02 javadoc-text@1.posnumbits

-rw-r--r--  1 vigna  vigna  342 May 13 12:02 javadoc-text@1.properties

-rw-r--r--  1 vigna  vigna  4.5k May 13 12:02 javadoc-text@1.sizes
-rw-r--r--  1 vigna  vigna  213k May 13 12:02 javadoc-text@1.terms
-rw-r--r--  1 vigna  vigna  3.1k May 13 12:02 javadoc-text@2.frequencies

-rw-r--r--  1 vigna  vigna  4.2k May 13 12:02 javadoc-text@2.globcounts

-rw-r--r--  1 vigna  vigna  234k May 13 12:02 javadoc-text@2.index
-rw-r--r--  1 vigna  vigna  10k May 13 12:02 javadoc-text@2.offsets

-rw-r--r--  1 vigna  vigna  9.3k May 13 12:02 javadoc-text@2.posnumbits

-rw-r--r--  1 vigna  vigna  336 May 13 12:02 javadoc-text@2.properties

-rw-r--r--  1 vigna  vigna  221 May 13 12:02 javadoc-text@2.sizes
-rw-r--r--  1 vigna  vigna  80k May 13 12:02 javadoc-text@2.terms
-rw-r--r--  1 vigna  vigna  429 May 13 12:02 javadoc-title.cluster.properties

-rw-r--r--  1 vigna  vigna  144 May 13 12:02 javadoc-title.cluster.strategy

-rw-r--r--  1 vigna  vigna  1.5k May 13 12:02 javadoc-title.frequencies

-rw-r--r--  1 vigna  vigna  1.5k May 13 12:02 javadoc-title.globcounts

-rw-r--r--  1 vigna  vigna  25k May 13 12:02 javadoc-title.index
-rw-r--r--  1 vigna  vigna  5.2k May 13 12:02 javadoc-title.offsets

-rw-r--r--  1 vigna  vigna  9.9k May 13 12:02 javadoc-title.positions

-rw-r--r--  1 vigna  vigna  1.6k May 13 12:02 javadoc-title.posnumbits

-rw-r--r--  1 vigna  vigna  376 May 13 12:02 javadoc-title.properties

-rw-r--r--  1 vigna  vigna  2.5k May 13 12:02 javadoc-title.sizes
-rw-r--r--  1 vigna  vigna  1.1k May 13 12:02 javadoc-title.stats
-rw-r--r--  1 vigna  vigna  31k May 13 12:02 javadoc-title.termmap

-rw-r--r--  1 vigna  vigna  61k May 13 12:02 javadoc-title.terms
-rw-r--r--  1 vigna  vigna  752 May 13 12:01 javadoc-title@0.frequencies

-rw-r--r--  1 vigna  vigna  752 May 13 12:01 javadoc-title@0.globcounts

-rw-r--r--  1 vigna  vigna  11k May 13 12:01 javadoc-title@0.index

```

```
-rw-r--r--  1 vigna  vigna      2.1k May 13 12:01 javadoc-title@0.offsets
-rw-r--r--  1 vigna  vigna      791 May 13 12:01 javadoc-title@0.posnumbits
-rw-r--r--  1 vigna  vigna      329 May 13 12:01 javadoc-title@0.properties
-rw-r--r--  1 vigna  vigna     1.2k May 13 12:01 javadoc-title@0.sizes
-rw-r--r--  1 vigna  vigna     32k May 13 12:01 javadoc-title@0.terms
-rw-r--r--  1 vigna  vigna     761 May 13 12:02 javadoc-title@1.frequencies
-rw-r--r--  1 vigna  vigna     761 May 13 12:02 javadoc-title@1.globcounts
-rw-r--r--  1 vigna  vigna     11k May 13 12:02 javadoc-title@1.index
-rw-r--r--  1 vigna  vigna     2.1k May 13 12:02 javadoc-title@1.offsets
-rw-r--r--  1 vigna  vigna     883 May 13 12:02 javadoc-title@1.posnumbits
-rw-r--r--  1 vigna  vigna     330 May 13 12:02 javadoc-title@1.properties
-rw-r--r--  1 vigna  vigna     1.2k May 13 12:02 javadoc-title@1.sizes
-rw-r--r--  1 vigna  vigna     29k May 13 12:02 javadoc-title@1.terms
-rw-r--r--  1 vigna  vigna      41 May 13 12:02 javadoc-title@2.frequencies
-rw-r--r--  1 vigna  vigna      41 May 13 12:02 javadoc-title@2.globcounts
-rw-r--r--  1 vigna  vigna     400 May 13 12:02 javadoc-title@2.index
-rw-r--r--  1 vigna  vigna      91 May 13 12:02 javadoc-title@2.offsets
-rw-r--r--  1 vigna  vigna      44 May 13 12:02 javadoc-title@2.posnumbits
-rw-r--r--  1 vigna  vigna     320 May 13 12:02 javadoc-title@2.properties
-rw-r--r--  1 vigna  vigna      57 May 13 12:02 javadoc-title@2.sizes
-rw-r--r--  1 vigna  vigna     1.2k May 13 12:02 javadoc-title@2.terms
```

As you can see, there are several new files (they could be more or less, depending on the number of documents stored on your system): each file whose names starts with `javadoc-text@` belongs to a certain subindex, that was generated using a batch of occurrences. The `javadoc-text.properties` file contains global information pertaining all subindices. Other files, such as the `.sizes` files, contain the list of the document sizes (the number of words contained in each document). The latter is useful for statistical purposes, but it might also be used by the indices, to establish better compression methods for the inverted lists. The `.terms` files, instead, contains the terms indexed in each batch. Note that each subindex can be queried separately, albeit you will need to generate manually a term map if you want to write query by term and not by term number (`IndexBuilder` creates such a map just for the whole index). The class `it.unimi.dsi.util.ImmutableExternalPrefixMap`, for instance, can be used to this purpose.

Now, if you look into the `javadoc-text.properties` file, you will find some information:

```
documents = 4090
terms = 32634
postings = 976736
maxcount = 3425
indexclass = it.unimi.dsi.big.mg4j.index.FileHPIndex
skipquantum = 0
skipheight = 8
coding = FREQUENCIES:GAMMA
coding = POINTERS:DELTA
coding = COUNTS:GAMMA
coding = POSITIONS:DELTA
termprocessor = it.unimi.dsi.big.mg4j.index.DowncaseTermProcessor
batches = 3
field = text
size = 57884884
maxdocsize = 53057
occurrences = 4599358
```

You can see some the overall number of occurrences (4599358), the number of batches (3) and the maximum size (number of words) of a document (53057). Similar information is available on a per-batch basis looking at the remaining `.properties` files.

The files starting with `javadoc-text.cluster` present a *cluster view* of the set of batches just built. Essentially, they provide dynamic access to the entire set of batches as a single index. More information can be found in the documentation of the package `it.unimi.dsi.big.mg4j.index.cluster`.

Time/space requirements

The scanning phase is, by far, the most time/space consuming. MG4J will work with little memory, but more memory will make it possible to build larger batches, which can then be merged more quickly and without opening too many files. You should set the JVM memory as high as you can go, and a number of documents per batch that does not cause too many compactions (or most of the time will be spent in the garbage collector), always keeping in mind that larger batches are better. If you experience out-of-memory errors (but it shouldn't happen!), just lower the number of documents per batch. Note that the memory compaction performed by MG4J seems to make the JVM erroneously think that there is too much garbage collection, sometimes resulting in an `OutOfMemoryError` due to excessive garbage-collector overhead. Please use the option `-XX:-UseGCOverheadLimit` to overcome the problem.

The kind of document sequence is going to influence heavily the indexing time. The best way of providing data to MG4J is to stream documents to the standard input, separating them with a character (usually, newline or NUL). This is the default choice if you do not specify explicitly a collection. Other kind of collections (e.g., database-based collections) might be reasonably efficient, but, for instance, do not expect great results from document sequences retrieving documents directly from the file system one at a time.

Remember that the indexer will produce a number of subindices, and this number will depend on the overall number of occurrences (which is, essentially, proportional to the total document size). Combining these subindices (or accessing them using an on-the-fly index combiner) has a cost in time that increases logarithmically with the number of subindices. Moreover, for each subindex an on-the-fly combiner needs to allocate buffers, so the memory cost for batch or on-the-fly combination increases linearly with the number of subindices. The rule of thumb is that you should try to make batches as large as possible, but you should also check the logs because working with an almost full heap can slow down Java significantly, and exaggeratedly large batches can cause slowdown because of the large number of cache misses.

Combining batches

Once you have the batches, you must *combine* them in a single index (in the `IndexBuilder` example, combination has been handled for you). Note that MG4J allows you to combine *any* set of indices, which means, for instance, that if your collection is split in several piece you can index the pieces separately and combine them later. MG4J distinguish three type of index combination:

1. *Concatenation* takes a list of indices and builds a new index as follows: the first document of the second index is renumbered to the number of documents of the first index, and the others follow; the first document of the third index is renumbered to the sum of number of documents of the first and second index, and so on. The resulting index is identical to the index that would be produced by indexing the *concatenation* of document sequences producing each index. This is the kind of combination that is applied to batches, unless documents were renumbered.

2. *Merging* assumes that each index contains a separate subset of documents, with non-overlapping number, and merges the lists accordingly. In case a document appears in two indices, the merge operation is stopped. Note that no renumbering is performed. This is the kind of combination that is applied to batches when documents have been renumbered, and each batch contains potentially non-consecutive document numbers.

3. *Pasting* relaxes further the assumptions of merging: each index is assumed to index a (possibly empty) part of a document. For each term and document, the positions of the term in the document are gathered (and possibly suitably renumbered). If the inputs that have been indexed are text files with newline as separator, the resulting index is identical to the one that would be obtained by applying the UN*X command **paste** to the text files. This is the kind of combination that is applied to *virtual documents*, described in the next section.

Please consult the Javadoc of the package `it.unimi.dsi.big.mg4j.document` and of the above classes for more information.

Virtual fields in MG4J

As we explained, documents usually originate from some stream in the form of byte sequences; every such sequence representing a document is then interpreted by some *document factory* that actually maps the byte sequence into a set of fields. For example, the `it.unimi.dsi.big.mg4j.document.HtmlDocumentFactory` translates a sequence of bytes into a set of fields, such as the title of the HTML document and its body. The factory deals with all the problems of translating bytes into characters, of establishing which parts of the document should be retained (e.g., in the case of HTML, discarding tags), of determining word borders etc.

There are cases, though, when the content of a document actually refers to another document in the collection: for example, it is well known that a HTML document may contain *anchors*, that are pieces of text that link to (and, at least conceptually, refer to) another document, specified via a URI.

As an example, consider the following document, with URI `http://foo.bar/one.html`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <title>This is document one</title>
  <body>
    <p>Here you can find a <a href="http://foo.bar/two.html">document \
      containing a lot of information about Mongolia</a>.
  </body>
</html>
```

The piece of text that reads:

document containing a lot of information about Mongolia

is actually an anchor that refers to another document (with URI `http://foo.bar/two.html`) and this fact should somehow be made explicit when indexing the collection. For example, in some sense, the word `Mongolia` should be taken as appearing in the document `http://foo.bar/two.html`, even if it may not even be mentioned in the text at that page.

This situation is dealt with by MG4J with the special notion of *virtual field*. Understanding how virtual fields actually work requires some patience, and some knowledge of the internal organization of document collections and factories; the reader may want to skip this section, reserving it for later.

Virtual fields and virtual fragments

As we briefly said, every document factory is responsible for turning raw byte sequences into documents. In particular, every factory transforms a sequence of bytes into a number of *fields*. Every field has a *name* and a *type*: for example, a document factory for mail documents might contain fields such as `subject`, `from`, `to`, `body`, `date` etc. The type of a field determines which values the field might contain. The two most important types of fields (and currently the only ones that MG4J is able to index) are *textual fields* and *virtual fields*.

A textual field, as the reader may guess, is just a piece of text, that is recognized as composed by words: words of a textual fields are the atoms of MG4J indexing system for textual fields. How words are really singled out from the stream of characters is a subtle problem that is dealt with by something that is called a *word reader* in the MG4J jargon, but we reserve a more comprehensive explanation of how this actually works for later.

Let us consider, instead, virtual fields. To make our explanations more concrete, let us consider the `HTMLDocumentFactory`: as we said above, this factory produces fields out of a HTML document. Actually, the factory has three fields: two of them (`text` and `title`) are textual, and one (`anchor`) is virtual.

A virtual field produces pieces of text that are to be referred to other documents, possibly belonging to the collection. To establish a precise terminology, let us call *referrer* the document that we are considering, and *referee* the document to which a certain piece of referrer is referring to. Now, the referrer produces in a virtual field a number of fragments of text, each referring to a certain referee. Hence, the content of a virtual field is conceptually a list of pairs made by a piece of text (called *virtual fragment*) and by some string that is aimed at representing the referee (called the *document spec* because it should somehow specify which document we are referring to).

In the case of the `HTMLDocumentFactory`, the `anchor` field is the list of all anchors contained in the document; the `document spec` is a URL (as specified in the `href` attribute) whereas the virtual fragment is the content of the anchor element. To be more precise, the actual implementation of the factory in MG4J considers not only the content of the anchor, but also some surrounding text, called the anchor context. This is only incidental, though: the important point is that a certain piece of text is associated with the `document spec`.

Note that as far as document factories are concerned, there is no fixed way to map `document spec` into actual references to documents in the collection. This is resolved, in MG4J, by the notion of *document resolver*.

Document resolvers

A document resolver is an object that is able to map the document spec produced by some document factory into actual references to documents in the collection: more precisely, given a document spec, the resolver will decide whether the spec really refers to a document in the collection or not, and in the first case it will find out to which document the spec refers to.

You don't need to deal with document resolvers until you try to index virtual fields. This is something that actually MG4J does only on demand: this is why in the example of the previous section we ignored the problem. Indeed, when we issued the command:

```
java -Xmx256M it.unimi.dsi.big.mg4j.tool.IndexBuilder \  
  --downcase -S javadoc.collection javadoc
```

we asked MG4J to index *only the textual fields of the collection* (whose documents were, as you remember, HTML documents). This means that only titles and texts were indexed, but no anchors (some of you may have noticed that MG4J emitted a brief warning about this fact, logging that `Virtual field anchor is not being indexed`; use `-a` or explicitly add field among the indexed ones).

Now, if you want to index also anchors you might explicitly ask for it, or you may use the `-a` option:

```
java -Xmx256M it.unimi.dsi.big.mg4j.tool.IndexBuilder \  
  -a --downcase -S javadoc.collection javadoc
```

If you try to do so, you will get an exception, saying that `No resolver was associated with virtual field anchor`: to understand the meaning of this exception we need to build a document resolver that is able to translate the document spec produced for the field anchor by the `HTMLDocumentFactory` into references to documents of the collection. Note that every document spec needs a different kind of document resolver, and you need to know which document resolver fits the needs of a certain virtual field.

In the case of anchors, the job is done by the `URLMPHVirtualDocumentResolver` class, that turns URLs into document pointers (i.e., references to documents). To build a URL document resolver, you first need to find the URLs of the document within your collection; you can list them as follows

```
java -Xmx256M it.unimi.dsi.big.mg4j.tool.ScanMetadata \  
  -S javadoc.collection -u javadoc.urls
```

This command scans the whole collection and produces a (text) file called `javadoc.urls` that contains the URLs of the collection in their order (of course, the collection URIs must actually be URLs). Note that in the case of our collections, URLs will actually be just file names.

By the way, you can use `ScanMetadata` also to extract other information (e.g., the document titles) from your collection.

Now that you have a list of URLs, one per document, you can build the document resolver you need by calling:

```
java -Xmx256M it.unimi.dsi.big.mg4j.tool.URLMPHVirtualDocumentResolver \  
  -o javadoc.urls javadoc-anchor.resolver
```

This command produces the resolver you need to index your anchor fields. Now, you can try again to index the whole collection, running:

```
java -Xmx512M it.unimi.dsi.big.mg4j.tool.IndexBuilder \  
-a -v anchor:javadoc-anchor.resolver --downcase \  
-S javadoc.collection javadoc
```

What is a document resolver actually doing: virtual texts and gaps

To understand what we just did, it is useful to think that conceptually all the virtual fragments that refer to a given document of the collection should be thought of as producing a single text, called the *virtual text*. So, for example, all the text of anchors referring to `file:/usr/share/javadoc/java/java/lang/String.html` should be concatenated and thought of as a single virtual text that will be indexed as a part of `file:/usr/share/javadoc/java/java/lang/String.html`.

Indeed, if you start the query engine again

```
java it.unimi.dsi.big.mg4j.query.Query -h -i FileSystemItem \  
-c javadoc.collection javadoc-text \  
javadoc-title javadoc-anchor
```

you will be able to input queries such as `text:implementation AND anchor:buffer` that are matched by all documents that contain the word `implementation` in their text and the word `buffer` in (some of their) anchor(s).

Some caution should be exercised here. When indexing, the virtual text is actually (somehow) built by concatenating the anchor text. This means that virtual fragments coming from different anchors are actually concatenated. This fact might produce false positive results. For example, queries like `anchor:(buffer AND long)` are matched by documents that contain both the word `buffer` and the word `long` in their anchors, but not necessarily in the *same anchor*.

To avoid such kinds of false positives, you can play with *virtual gaps*: the virtual gap is a positive integer, and it is the virtual space left between different virtual fragments. For example, if the virtual gap is 64 (the default), anchors are concatenated by leaving 64 "empty words" between subsequent fragments.

Hence, for example, if you input a query like `anchor:(buffer AND long)~64` you will be sure that only documents that contain both words in the *same anchor* will be found. Of course, this time you might have false negatives, if some anchor is longer than 64 words. If you want, while indexing you can specify a different virtual gap; for example:

```
java -Xmx512M it.unimi.dsi.big.mg4j.tool.IndexBuilder \  
-a -g anchor:100 -v anchor:javadoc-anchor.resolver \  
--downcase -S javadoc.collection javadoc
```

runs exactly as before, but leaving a virtual gap of 100 words between successive fragments.

Payload-based indices

MG4J provides a special kind of index, called *payload-based index*, that is used to store not text but rather metadata (dates, integers, etc.) related to a document. It is the default way of storing non-textual fields. Essentially, a payload-based index leverages the structure of a text-based index: it has no counts or

positions, but each posting has a *payload*—a piece of data related to the document referred by the posting. In this way, by creating an index with a single posting list (related to the term #) we are effectively storing metadata related to each document. The main advantage of this approach is that we get almost for free the sophisticated skipping structure of MG4J's indices, and support for splitting, combination, and so on.

From the user viewpoint there is no particular difference between standard and payload-based indices, except that the latter do not provide some files that would be nonsensical, such as the file of sizes or the global occurrence count, and that searching a payload-based index is rather different from searching an index (instead of term-based operators and Boolean combinators you just get range queries).

Chapter 3. Performance

Indexing Time

MG4J provides a great flexibility in index construction. For instance, you can choose several different codes for the components of the index, and moreover you can decide to drop parts you are not going to use (e.g., positions). All these choices have a significant impact on performance. Building a collection during the indexing phase will of course slow down the whole process.

In general building large batches is a good idea if you have a lot memory; you can set the tentative batch size using the `-s` option. However, if your collection contains a large number of terms (e.g., if it contains many *hapax legomena*—terms that occur just once in the collection) a very large number of objects will be generated. This can cause a massive amount of garbage collection if you're relatively tight on memory. For this reason, there is a limit on the number of terms indexed at once (see the `-M` option of `IndexBuilder` and `Scan`).

Setting up the index structure

In MG4J, you can choose the codes used for compression. As a general rule, *nonparametric codes are quicker than parametric codes*. Thus, Golomb codes for document pointers have an excellent compression rate, but δ codes have very good compression, too, and can be decoded more quickly. The point here is that for unary, γ , shifted γ and δ codes MG4J uses *precomputed decoding tables* that speed up decompression by an order of magnitude. The default choice (γ for frequencies, δ for pointers, γ for counts, and δ for positions) is very reasonable. For maximum speed you could even try to use γ everywhere (as it is quicker to decode if the precomputed decoding tables fail).

Another important trick is that of discarding what you don't need. The default MG4J index type is called *high-performance*: it contains all information (pointers, counts, positions) but it is only partially interleaved—positions are kept in a separate file. This satisfies most needs, but if you are just using BM25 or TF/IDF scoring, there is no need to store positions in your index: you can force a standard, interleaved index, and store just what you need (e.g., `-cPOSITIONS:NONE` will eliminate positions from the index).

By default, indices contain a *skipping structure* that makes skipping index entries faster. Skipping structures introduce a slight overhead when scanning sequentially a list (so you should disable them using the `--no-skips` option if you don't need them), but in general they make query processing significantly faster. Skipping structures are based on two parameters: the *quantum* q and the *height* h . The quantum dictates how often the skipping structure should index positions in the inverted list. The height dictates how far the skipping structure is able to jump in one shot (an index is able to skip in one shot as far as q^{2^h}). However, as h grows the memory required to build the skip structures grows exponentially: the rule of thumb is setting h as large as possible without incurring in out-of-memory errors.

Sizing the quantum is a more complex issue, as it depends on the structure of the inverted list. Dense inverted lists require smaller quanta. Since version 3.0, MG4J makes it possible to just specify the *percentage* of the index size occupied by the skipping structure, and let some machinery compute the correct quantum. You can also specify a quantum explicitly, but it will be the same for all lists, which is usually not a good thing.

Setup Time

Once the index has been created, there are many ways in which you can improve query resolution time. First of all, an index can be read from disk, memory-mapped, or directly loaded into main memory. These three solutions work with increasing speed and increased main memory usage. The default is

to read an index from disk, but you can add suitable options to the index URI (e.g., `mapped=1` or `inmemory=1`—see the `Index.UriKeys` documentation) to force your preferences. Analogously, *offsets* are necessary to locate, inside the index file, the posting list of a certain term. By default they are read from disk using a `SemiExternalOffsetList`, but you can load them in memory if you prefer so. If you load sizes (e.g., because you want to run a scorer that needs them) there is a suitable URI option (e.g., `succinctsizes=1`) that will load sizes in a highly compact format. This is particularly useful when pasting large indices.

To get more options, you can partition your index. Once you have a cluster formed by several sub-indices, you can decide which sub-indices go to memory, which will be mapped, and so on.

Query Time

Once you are convinced that your setup is reasonable you should generate a wired `BitStreamIndexReader/BitStreamHPIndexReader`. The latter are the generic classes used by MG4J to read an index: thus, they incorporate all the logic required to handle literally hundreds of types of indices. However, you can use the Ruby script `genbitstreamreaders.rb` provided with MG4J to generate additional instances that are wired to a specific index type. When loading an index, MG4J will fetch dynamically (by reflection) the wired class and will log (at `INFO` level) that it is using a wired class instead of the generic class. (The standard MG4J distribution contains wired classes for the default index-construction options.)

The Ruby script above prints a list of commands involving a C compiler (by default, `gcc`). Actually, the commands use the C preprocessor to filter a driver file contained in the source tree (`BitStream[HP]IndexReader.c`). Executing the output of the Ruby script will generate all possible wired classes (hundreds), but you can also select manually the classes you prefer to generate.

The simplest way to understand the wiring process is having a look at the output of the Ruby script: essentially, defining the symbol `GENERIC` you obtain the generic driver. Otherwise, you can define symbols `SKIPS` and `PAYLOADS` if you want these features, and then you must specify assertions with name `frequencies`, `pointers`, `counts`, and `positions` that either select a code or disable a feature (as in `Combine`'s command line options). The symbol `CLASSNAME` defines the wired class name, and *must* be generated following the algorithm contained in the Ruby script, or MG4J will mistakenly load wired classes that are not adapt for your index.

Chapter 4. Clusters & Partitioning

Documental vs. Lexical

MG4J provides a completely generic way of combining indices into clusters. This feature can be used, for instance, to support incremental indexing, but it goes way beyond that. An index is just a *composite* in the design-pattern sense, and can be built by combining different indices. For instance, you can index separately two sets of documents and then use the two resulting indices as a single index using a concatenation-based cluster index. Alternatively, you can actually combine the indices, getting a new index.

More generally, a cluster exhibits a set of *local* indices as a single *global* index. Clusters, moreover, can be *documental* or *lexical*. In a documental cluster, each document of the global index appears exactly once in each local index. In a lexical cluster, each term of the global index appears exactly once in each local index. These two types of clusters satisfy different needs: documental clusters, for instance, can be used to keep a set of documents with high static rank in a separate index living on faster storage, whereas lexical cluster can be used to load in memory the inverted lists of terms that appear more frequently in user queries.

Partitioning vs. Clustering

The opposite of clustering is *partitioning*. Partitioning an index means dividing its inverted lists using some criterion, and, not surprisingly, partitioning can be documental or lexical. MG4J provides tool that make it possible partitioning using a custom strategy specified by a Java class, so it is very easy to process indices (even large indices) and partition them in several ways (obvious splitting strategies, such as uniform strategies, are actually built-in). You should try the `PartitionDocumentally` and `PartitionLexically` tools to get an idea of what can be done and have a look at the documentation of the `it.unimi.dsi.m4j.cluster` package.

Of course, the suite of combination tools used to combine batches can be used for the opposite process—taking the set of local indices making up a cluster and turning them into a single combined index, which will contain the same data of the original cluster, but in a different format. Clusters, partitioning and combining are thus several facets of the same idea—that is, that an index is actually a composite object.

Creating a Cluster

A cluster is simply defined by a property file having property `indexclass` equal to the suitable cluster class (e.g., `it.unimi.dsi.big.mg4j.index.cluster.DocumentalConcatenatedCluster`), a clustering strategy specified by the property `strategy` and one `localindex` property for each local index (see also the documentation of `it.unimi.dsi.big.mg4j.index.cluster.IndexCluster`). You can run `IndexBuilder` with the `--keep-batches` option to have a look at the generated cluster files, which expose the batches as a single index.

Chapter 5. Accessing MG4J indices programmatically

Constructing an index and querying it using the `Query` class is fine, but usually MG4J must be integrated in some kind of environment. In this chapter we describe how to access programmatically an index using MG4J. A (small but growing) list of heavily commented examples is available in the `it.unimi.dsi.big.mg4j.example` package.

In general, the first thing you need is to load an index. To do that, you must use the `Index.getInstance()` method, which will arrange for you a number of things, like finding the right index class, possibly loading a term map, and so on. Usually you will have more than an index (e.g., title and main text).

The second piece of information that is necessary for the following phases is an *index map*—a data structure mapping a set of symbolic *field names*, which will be used to denote the various indices, to the actual indices (e.g., to the actual instances of the `Index` class). There are simple ways to build such maps on the fly using `fastutil` [<http://fastutil.dsi.unimi.it/>] classes (see the `RunQuery` example). Another important map is the map of term processors, which maps each field name to the respective term processor. Usually the term processor is the one used to build the index, which can be recovered from the `Index` instance, but different choices are possible

There are now several ways to access MG4J. Given a textual query, the query is parsed and turned into an internal *composite representation* (essentially, a tree). Then, a *builder visitor* visits the tree and builds a corresponding *document iterator*, which will return the documents that satisfy the query.

At the basis of the query resolution, *index iterators* provide results from the index (i.e., documents in which a term appears, and other information): in other words, they are used as iterators corresponding to the leaves of the query tree. These can be combined in various ways (conjunction, disjunction, etc.) to form document iterators. Document iterators return documents satisfying the query and, for each document, a list of *minimal intervals* representing the regions of text satisfying the query. At that point, *scorers* are used to rank the documents returned by the document iterator.

You can handle this chain of events at many different levels. You can, for instance, build your own document iterators using the various implementations of `DocumentIterator`. Or you can create queries (i.e., composite built using the implementations of `it.unimi.dsi.big.mg4j.query.node.Query`), and turn them into document iterators. You can even start from a textual query, parse it to obtain a composite internal representation, and then go on.

Nonetheless, the simplest way is to use a façade class called `QueryEngine` that tries to do all the dirty work for you. A query engine just wants to know which parser you want to use (`SimpleParser` is the default parser provided with MG4J), which builder visitor you want to use, and which index map. The builder visitor is a visitor class that is used to traverse the internal representation of a query and compute the corresponding document iterator. The default visitor, `DocumentIteratorBuilderVisitor`, is very simple but fits its purpose. You might want to change it, for instance, to reduce object creation.

A query engine has many tweakable parameters, that you can find in the Javadoc documentation. However, its main advantage is that its method `process()` takes a textual query, a range of ranked results, and a list in which to deposit them, and does everything for you. You can easily get results from MG4J in this way.

A different route is that of customizing the `QueryServlet` class that MG4J uses for its HTTP/HTML display. This might simply involve changing the Velocity script that displays the results (and which is set by a system variable—see the class `HttpQueryServer`) or actually modifying the class code.