The Amberfish Text Retrieval System

Nassib Nassar *Etymon Systems, Inc.*

Table of Contents

Preface

This document serves as an introduction to the Amberfish text retrieval system. It is intended for both beginning UNIX users and professional system administrators. Amberfish is text-based software and does not currently have a graphical user interface, although it is hoped that the tools provided will be easy to integrate into existing systems. Except where otherwise noted, a UNIX platform is assumed.

1 Tutorial

1.1 Introduction

The easiest way to understand how to use Amberfish is to index and search a small group of text files.

The process of *indexing* creates a set of files that collectively are referred to in this document as a *database*. Once the database has been created, the original text can be searched via the database. The `af` tool performs both of these tasks as well as others.

1.2 Indexing

Here is a simple indexing command ('`$`' is the UNIX prompt):

```
$ af -i -d mydb -C -v *.txt
```

The '`-i`' option indicates that we want to index files, and the list of files is expected to be at the end of the line ('`*.txt`'). The '`-d`' option specifies the database name, in this case, '`mydb`'. The '`-C`' option indicates that we want to create a new database (overwriting any existing database called '`mydb`'). Finally, the '`-v`' option tells `af` to print information about the indexing process as it goes along. (Another option, '`-m`', may be added to increase the amount of memory used for indexing, which will generally reduce indexing time.)

The indexing process creates a group of files beginning with '`mydb`' (i.e. '`mydb.db`', '`mydb.dt`', and several others). These are the index files that make up the database. We can use the `af` tool with this database to search the files we have indexed.

> *Note:* As an optional step after indexing, the database can be *linearized*. In the example above, this would be done with the command, `af -L -d mydb`. Linearizing improves search time but has the disadvantage of being a very slow process. Another disadvantage is that it fixes the database so that no more documents can be added.

1.3 Searching

Here is a simple searching command:

```
$ af -s -d mydb -Q 'cat & (dog | mouse)'
```

The '`-s`' option tells `af` that this is a search operation. The '`-q`' option specifies a Boolean search *query*, which represents what you are searching for. The entire query is enclosed between two apostrophes so that the UNIX shell will not modify its contents. The special characters, '`&`' and '`|`', are Boolean operators meaning '*and*' and '*or*' respectively. In other words, the above query means, "Find all documents that contain the word, '`cat`', and that

also contain either '`dog`' or '`mouse`'." Note that these words are '*case-insensitive*', meaning that '`cat`', '`CAT`', and '`caT`' are interchangeable.

The command returns a list of matching document references, each one taking the form:

```
score dbname docid parent filename begin end
```

where '`score`' is a relevance score, '`dbname`' is the database name, '`docid`' is an unique number identifying the document within the database, '`parent`' is the *docid* of the document that "contains" this document (or 0 if no such relationship exists), '`filename`' is the name of the file containing the document, and '`begin`' and '`end`' are byte offsets indicating the beginning and ending of the document within the file. Often there is a one-to-one correspondence between files and documents, in which case the '`begin`' and '`end`' values may be ignored. The '`dbname`' value is only useful if one is searching multiple databases at once (for example, see Section 1.10 [Multiple databases], page 8).

In the above example the words, '`cat`', '`dog`', and '`mouse`', are each an individual query *term*. When the '`-Q`' option is used, Amberfish searches for each query term separately and then combines the results according to the Boolean operators. Amberfish also supports a non-Boolean *free-text* query type, which is invoked by using '`-q`' instead of '`-Q`'. It consists of a list of terms separated by spaces and no Boolean operators, for example:

```
$ af -s -d mydb -q 'cat dog mouse'
```

This is roughly similar to a Boolean query with all the terms joined by '`or`', such as:

```
$ af -s -d mydb -Q 'cat | dog | mouse'
```

> *Note:* The free-text search option ('`-q`') is not yet fully implemented but is planned for an upcoming release. For now please use '`-Q`' when trying the examples in this tutorial.

In the next several examples we concentrate on the structure of a single query term, with the understanding that terms can be combined arbitrarily into Boolean expressions or free-text queries.

1.4 Right truncation

Amberfish supports *right truncated* queries such as the following:

```
$ af -s -d mydb -q 'car*'
```

The query term, '`car*`', finds all documents containing the word, '`car`', '`cars`', or '`carpet`', or any other word that starts with the prefix, '`car`'. A term such as '`s*`' will match any word that starts with the letter, '`s`', and it may take a while for Amberfish to process; not to mention the fact that it is very likely to match every document in the database.

1.5 Phrases

To enable phrase searching, the '`--phrase`' option must be given to `af`, together with '`-i`' and '`-C`', at the time of indexing.

Here is an example of a phrase search:

```
$ af -s -d mydb -q '"Dinu Lipatti"'
```

This means, "Find all documents that contain the phrase, '`Dinu Lipatti`'." (Note that the entire phrase must be enclosed in double quotes and that spaces are not significant.) Amberfish defines a phrase in this context as the word, '`Dinu`', followed by the word, '`Lipatti`', with no other words in between the two. Phrases may include more than two words, as in '`John Quincy Adams`'; and phrase words may be right truncated, for example, '`Emil* Durkheim`'.

1.6 Multiple documents in a file

A document can consist of an entire file or a portion of a file. Amberfish records '`begin`' and '`end`' byte offsets for each document as demarcation of the document within the file that contains it. By default the whole file is treated as a single document. For example, a file called '`sample.txt`' that is 12000 bytes in size will be indexed as a single document with '`begin`' and '`end`' byte offsets, 0 and 12000, respectively.

The `af` tool includes the '`--split`' option as a method of instructing Amberfish that the files to be indexed contain multiple documents. The '`--split`' option is used to specify a string delimiter that indicates the boundaries between documents in a file. For example:

```
$ af -i -d mydb -C --split '#####' -v *.txt
```

As the files, '`*.txt`', are indexed, they are scanned for the string, '`#####`'. Each instance of '`#####`' is interpreted as the beginning of a new document, and each new document is indexed individually. Note that each instance of '`#####`' is considered to be part of the document that follows it, as opposed to the document that precedes it. If the string delimiter happens to include text, rather than merely '`#####`', it will (normally) be indexed as text.

The division of files into multiple documents can be verified with `af -l` after the files have been added to the database (see Section 1.11 [Listing database information], page 8).

The `af --fetch` command prints a portion of a file to standard output:

```
$ af --fetch filename begin end
```

where '`filename`', '`begin`', and '`end`' are taken from the output of `af -s` (see Section 1.3 [Searching], page 3) or `af -l` (see Section 1.11 [Listing database information], page 8).

The '`--split`' option does not work with the `xml` document type, which uses a different method of dividing files into documents (see Section 1.12 [More about XML], page 8).

1.7 Fields

Amberfish allows searching on specific fields within documents. Support for various file formats is provided by *document type* modules. The document type must be specified to `af` at the time of indexing, using the '`-t`' option. Each document is individually associated

with a document type as it gets added to the database. The default document type (if none is specified) is `text`, which does not recognize any fields and therefore does not support field searching.

Here is an example of searching on a field:

```
$ af -s -d mydb -q 'Title/cat'
```

This means, "Find all documents that contain the word, '`cat`', in the '`Title`' field."

Unlike search words, field names may be case-sensitive, so that '`Title`' and '`title`' might be two different fields. Whether or not fields are case-sensitive is determined by the document type. For example, XML elements and attributes are case-sensitive; therefore an XML document type would most likely have case-sensitive fields. As we shall discuss later, it is best not to mix document types within a single database if those document types have incompatible views about case-sensitivity.

1.8 Fields and XML

XML is a good context for exploring field searching. The following examples make use of the `xml` document type, which supports nested fields (i.e. fields within fields).

Suppose we index the following XML data, contained in a file called, '`jones.xml`':

```
<Document>
   <Author>
      <Name>
         <FirstName> Tom </FirstName>
         <LastName> Jones </LastName>
      </Name>
   </Author>
</Document>
```

with the following command:

```
$ af -i -d mydb -C -t xml -v jones.xml
```

The `xml` document type views this document as containing two words, '`Tom`' and '`Jones`', each located at a certain *field path* within the document:

```
/Document/_c/Author/_c/Name/_c/FirstName/_c/Tom
/Document/_c/Author/_c/Name/_c/LastName/_c/Jones
```

The character, '`/`', separates the field names, and in this case each field except for '`_c`' corresponds to an XML element. (Below we shall see an example in which a field corresponds to an XML attribute.) The '`_c`' is a special field defined by `xml` that means, "element content." Thus the following search:

```
$ af -s -d mydb -q '/Document/_c/Author/_c/Name/_c/LastName/_c/Jones'
```

will return '`jones.xml`' as a matching result. These queries also will return a positive match:

```
$ af -s -d mydb -q '/.../Document/_c/Author/_c/Name/_c/LastName/_c/Jones'
$ af -s -d mydb -q '/.../_c/Author/_c/Name/_c/LastName/_c/Jones'
```

```
$ af -s -d mydb -q '/.../Author/_c/Name/_c/LastName/_c/Jones'
$ af -s -d mydb -q '/.../_c/Name/_c/LastName/_c/Jones'
$ af -s -d mydb -q '/.../Name/_c/LastName/_c/Jones'
$ af -s -d mydb -q '/.../_c/LastName/_c/Jones'
$ af -s -d mydb -q '/.../LastName/_c/Jones'
$ af -s -d mydb -q '/.../_c/Jones'
$ af -s -d mydb -q '/.../Jones'
$ af -s -d mydb -q 'Jones'
```

The '`...`' is defined by Amberfish as, "a sequence of any zero or more fields." A '`/.../`' that begins a field path can be left out completely. For example, these two queries yield the same results:

```
$ af -s -d mydb -q '/.../LastName/_c/Jones'
$ af -s -d mydb -q 'LastName/_c/Jones'
```

The '`...`' can be used anywhere within a field path. For example, the following queries match '`jones.xml`':

```
$ af -s -d mydb -q '/Document/_c/Author/_c/Name/.../Jones'
$ af -s -d mydb -q 'Name/.../LastName/.../Jones'
```

The first of the two examples above will match '`Jones`' anywhere within the author's name, not necessarily only his last name. The second matches only a last name of Jones, but it need not be the author; for example, it would match a document containing the following fragment:

```
<Bibliography>
   <Reference Type="book">
      <Title> Text searching the old fashioned way. </Title>
      <Name>
         <FirstName> Indiana </FirstName>
         <LastName> Jones </LastName>
      </Name>
   </Reference>
</Bibliography>
```

Other queries that would match the above fragment:

```
$ af -s -d mydb -q 'Reference/_a/Type/book'
$ af -s -d mydb -q 'Reference/_a/.../book'
$ af -s -d mydb -q 'Reference/.../book'
```

The '`_a`' is another special field defined by `xml` that means, "attribute content." Thus '`_c`' and '`_a`' allow one to distinguish between attribute and element searching if desired. In constructing queries for this document type, it is always necessary to specify '`_c`', '`_a`', or '`...`' after an element field name and before the next field name or the search word.

Phrase searching with fields is done this way:

```
$ af -s -d mydb -q 'Title/.../"text searching"'
```

or in a multiple term expression:

```
$ af -s -d mydb -Q 'Title/.../"text searching" &
                    Name/.../Indiana & Name/.../Jones'
```

1.9 Relevance ranking

By default, Amberfish attempts to rank search results based on relevance to the query. It computes a score for each document in the result set and sorts the results by score.

1.10 Multiple databases

The `af` tool can search multiple databases, for example:

```
$ af -s -d patents1978 -d patents1979 -d patents1980 -q 'mousetrap'
```

The query is applied to each of the three databases, '`patents1978`', '`patents1979`', '`patents1980`', and the results are merged into a single result set.

1.11 Listing database information

The `af` tool can be used to list information about an existing database. The '`-l`' option lists the documents contained in a database:

```
$ af -l -d mydb
```

Each line of output takes the form:

```
docid parent filename begin end doctype
```

where '`docid`' is an unique number identifying the document within the database, '`parent`' is the *docid* of the document that "contains" this document (or 0 if no such relationship exists), '`filename`' is the name of the file containing the document, '`begin`' and '`end`' are byte offsets indicating the beginning and ending of the document within the file, and '`doctype`' is the name of the document type associated with the document.

1.12 More about XML

The `xml` document type contains several features related to the structure of XML documents.

Amberfish treats an XML document as an hierarchy of nested documents, which can be useful because it increases the resolution of search results. This is regulated with the `af` option '`--dlevel`', which limits how many levels of resolution will be processed during indexing. For example:

```
$ af -i -d mydb -C -t xml --dlevel 2 -v medline.xml
```

Specifying '`--dlevel 1`' will define each XML file as a single document, bounded by the outermost XML element. (This is the default.) Specifying '`--dlevel 2`' will descend another level to the children of the outermost element and consider these to be documents

nested within the outer documents, etc. This feature should be used very cautiously, because of the dramatic increase in disk space and processing time required for progressively higher '`--dlevel`' values.

The result of using the '`--dlevel`' option is that search results can be very specific. The results returned by `af -s` consist of the innermost documents (as allowed by '`--dlevel`') that match the query. The '`--style`' option can be used with `af -s` to print the lineage of each result document, for example:

```
$ af -s -d mydb -q 'ArticleTitle/.../"adipose tissue"' --style=lineage
```

This produces an indented hierarchy of "ancestor" documents above each result document.