Alexey Sokirko, 2005
sokirko@yandex.ru

User Manual for DWDS/Dialing Concordance

version 1.7.2

Introduction

The main purpose of DWDS/Dialing Concordance ("DDC") is to search for words or sequences of words together with morphological patterns. DDC is created to help linguists to find a particular collocation or word in the given context. Roughly speaking, the functionality of DDC is similar to search engine “SARA“, that is accessible at the site of British National Corpus (see [1]).

Generally, the majority of web search engines were designed to look for information. It means that they must be able to process all text formats and aggregate all text variance into one representation. After all they are to give the most relevant hits to the end-user. A web search engine usually uses stop word lists, various dictionaries of synonyms, dictionaries of proper names, and it indexes the corpora by files, not by sentences. All this is useless for a linguistic search engine. For example, using a stop word list is senseless, since for a linguist stop words are the most relevant words of the language (prepositions, conjunctions and so on). Therefore we cannot simply use a web search engine for linguistic purposes, but are to design a special version of a search engine.

There are two possible strategies to search something in a corpus. On the one hand, we can create a program like Unix "grep", which doesn't use any index at all. It is a good solution because we really can create a very sophisticated query language. Once a sentence (a hit) is found, and its morphological interpretation is built, it is possible to go through the sentence many times and to check all conditions, which the query contains. Of course, the speed of the process will be equal to the speed of indexing, i.e. to the speed of linguistic processors (morphology, syntax). Apart from the speed there is one more disadvantage of this solution. When the search goal lies at the end of the corpus, the end-user is to wait till the whole corpus is processed. And for our opinion, the last consideration is the most important cause to give up the grep-like searching and choose another alternative, i.e. an index-based search engine.

Definitions

1. Source File

Generally a source file is a file, from which DDC reads information to index. A list of source files is the first input parameter for the indexing process. The second parameter is a file of options. Source files are used only during the indexing, and they can be removed afterwards. The current version supports the following types of source files:

1. Plain text (Russian Win-1251, German ISO-8859-1), extension *.txt;
2. Html text (Russian Win-1251, German ISO-8859-1), extension *.html, *.htm;
3. Xml text (TEI encoding scheme or ddc_document encoding scheme), extension *.xml (see below);

If source file is an xml file then DDC extracts some bibliographical information from it and stores it as an additional index, otherwise the bibliographical index is empty. For XML parsing DDC uses TinyXml project (see [3]). If a source file is a tar archive, then DDC should first extract all files from it.

2. Corpus File

A list of corpus files is built upon a list of source files extracting everything from all source archives. So if a source list doesn't contain archives then the lists of corpus files and source files are identical, otherwise it contains also files from archives, whose names are prefixed by the name of the archive. For each corpus file DDC maintains a file break and bibliographical information.

3. Occurrence

An occurrence is an integer number (4 bytes), which refers a position of a token. DDC stores lists of occurrences for each token in special files. During the querying it is the most time-consuming task for DDC to retrieve these lists as fast as possible.

4. Index Set

An index set consists of the list of strings (which are also called "index items") and corresponding lists of their occurrences in the corpus, for example:

mother -> 1, 100, 457
mothered -> 5006
mothering -> 2, 120, 147
...

A string to index can contain any char except \0. All strings of one index set are stored in a special file. An index set has two names: the short one and the long one. These names can be used interchangeably in queries. Optionally one index set can have a storage.

5. Index Storage

An index storage is a sequence of integers $X_1...X_N$, where N is the number of tokens in the corpus. Each X_i points to an indexed string, for example for the token index it points to a token. The order of $X_1...X_N$ is just the same as it was in the input corpus. For example using "Token" index storage DDC can reproduce the whole corpus word by word. By default the first index("Token" index) of the corpus has an index storage, for the other indices this option is switched off .

6. Corpus break

A "break" is a border between two adjacent sentences, paragraphs, files or other text parts. Generally, a break of a type **t** is an integer end offset of a corpus part. Type **t** can be sentence, a clause, a file etc. The ordered concatenation of all parts of type **t** is the corpus itself, so it means that there is no intersection between these chunks and no uncovered parts. One break collection of type **t** has a short and a long name.

7. Hit

A hit is a text chunk between two adjacent corpus breaks of one break collection that satisfies the input query.

8. Corpus period

A "corpus period", "internal subcorpus" or a "search period" is a break, which is introduced to restrict the memory usage. Corpus period always coincides with a file break. The size of one corpus period is 5000000 by default and can be determined manually using field "UserMaxTokenCountInOnePeriod" in the options file. While evaluating a query DDC deals only with one corpus period at a time, so DDC applies the input query to each corpus period, and then concatenates the results.

9. Highlighted words

Let Q be a query, and let Q_1, Q_1,…. Q_n, be atomic queries, which constitute Q. Let H be a hit, which satisfies query Q. Then we would call each occurrence of an atomic query Q_i (= a word or a punctuation mark) in H a highlighted word. For example, let Q be

mother && father

let H be

The mother and the father like their mothers and fathers

then highlighted words would be the following:

The mother and the father like their mothers and fathers

The way of highlighting depends upon the general output format and can be defined in the option file (see HtmlHighlighting and TextHighlighting fields).

10. Relevant document

A relevant document is a corpus file that contains at least one hit

11. Text area

Text area is a part of corpus xml file, for example a title or an appendix. A border between two text areas is a sort of corpus breaks, that's why a token of a corpus file should be part of exactly one text area. All corpus files should contain the same number of text areas. A text area could be empty. All text areas are declared in the options file, and if the input corpus consists of XML files, then the options file should have at least one text area declaration (usually body declaration). For non-XML files text areas are not relevant.

Options file

File of options is a text file, which contains corpus options, one option per line. Some options are applicable only during the indexing, other – only during the querying, and there are some options that are applicable on both phases. Among the latter there are two most important options: the index type and the corpus language.

An index type determines a general structure of DDC indices and break collections. In principle, the following types are defined:

- **DWDS_Index**. A type for corpora, where words have not assigned annotations, which are written in the corpus files. For example, the input text can be a plain text. DDC always builds a token index and a file break collection for this index type. Optionally DDC can build "Thes" index (an index for thesaurus descriptions of words), "MorphPattern" index (as index for morphological patterns of words) and a sentence break collection.

- **MorphXML_Index.** A type for xml-texts, if words of these texts have annotations, which are written in the xml-structure. DDC always builds a token index and a "MorphPattern" index. It also creates a file and a sentence break collection See here the description of xml-format of this index type.

- **Free_Index.** This index type is called free and therefore it should be defined in the options file (fields "Indices" and "HitBorders"). The corpus should consist of xml-files. Besides a TEI bibliographical header each file should have a body element, where text is written in CWB format (see [2]). The original CWB format has been a little bit changed:

  1. Instead of line breaks that are used to delimit records in the input file, DDC uses a special tag ("l")

  2. The first column (=the token column) can be prefixed with "-#" string, which leads to excluding this token from the index (but not its properties).
  (see here for an example).

  A corpus language is a switcher which chooses tokenizer and morphology system during indexing (under **DWDS_Index**) and querying (for all index types). DDC supports now Russian, German and English languages.
  Consequently the first two lines of an options file should look like as follows:

  IndexType <type>
  <lang>,
  where <type> can be DWDS_Index, MorphXML_Index, Free_Index
  <lang> can be Russian, German, or English.
  For example:
  IndexType DWDS_Index
  Russian

  Other options can be grouped as follows:
  1) Common options

Option	Description
CaseInsensitive	Make the search case-insensitive by default

OutputBibliographyOfHits	If it is on, then DDC outputs bibliographical information of hits, instead of their file names.
LocalPathPrefix <string>	The common prefix of each corpus file name that should be replaced by "InternetPathPrefix", when DDC outputs file names.
InternetPathPrefix <string>	See LocalPathPrefix
UserMaxTokenCountInOnePeriod <n>	Set the size of internal subcorpora. The greater the value of this parameter is, the faster querying procedures work, and the more memory the program needs.
DisableDefaultQueryLexicalExpansion	If on, then DDC does not expand each word in the input query with its possible word forms. It has the same effect as though each word in the query has an exact operator.
HtmlHighlighting <string>	This field describes how DDC should highlight the words in a hit if DDC outputs HTML. The <string> is a semicolon-separated tuple of 4 items. The first item contains a string that should be inserted before the first highlighted word (in order to visualize the highlighting). The second item contains a string that should be inserted after the first highlighted word. The third item is inserted before the following highlighted words. The fourth item is inserted after the following highlighted words. The default value is "`<STRONG><FONT COLOR=red>; </FONT></STRONG>; <STRONG><FONT COLOR=red>; </FONT></STRONG>`"
TextHighlighting <string>	This field describes how DDC should highlight the words in a hit if DDC outputs plain text. See format of <string> in field "HtmlHighlighting". The default value is "`&&; &&; _&; &_`"
ResumeOnIndexErrors	If on, then DDC ignores source xml-files with a wrong structure while indexing the input corpus, otherwise DDC stops at the first errors.
MaxRegExpExpansionSize <n>	The maximal number of index items that can be included in an expansion set of one regular expression, default value is 1000000.
NoContextOperator	Prohibits context operator in the query language ("#cntxt")
ShowNumberOfRelevantDocuments	If on, DDC calculates the number of relevant documents, otherwise the member that holds this number is set to 0.
LeftKwicContextSize <n>	<n> determines the size of the left context for each kwic-line while generating file summaries (snippets). The default value is 4.
RightKwicContextSize <n>	<n> determines the size of the right context for each kwic-line while generating file summaries (snippets). The default value is 4.
NumberOfKwicLinesInSnippets <n>	<n> determines the number of kwic lines in snippets. The default value is 10.
DwdsCorpusInterface	Enables DWDS-like formatting for output hits.

Bibl	This field can occur many times in the options file. One line with this field describes one bibliographical field (such as date of publication, author and so on). The bibliographical field can be predefined(“orig”, ”scan”, ”date”, ”page”, “pagerank”) or free (user-defined). Predefined bibliographical fields have special processing in DDC, for example, field “scan” is used to build a hit header. Free bibliographical fields can be integer or string. For both types one can use the general filter operator or general order operators. Field “Bibl” has the following format: Bibl <type> <visibility> <name> <xpath>, where • <type> can be “string” or “integer”; • <visibility> can be “1” or “0”, if it is “1” then DDC shows the value of the field for each hit header; • <name> is name of this field; • <xpath> is an Xpath (see [6] for Xpath specification).
TextArea	This field can occur many times in the options file. One line with this field describes one text area. This field has the following format: TextArea <name> <xpath>, where • <name> is name of this field; • <xpath> is an Xpath (see [6] for Xpath specification).
`TfIdfRank`	float parameter (0<=0<1) for tf.idf weighting
`NearRank`	float parameter (0<=0<1) for near weighting
`PositionRank`	float parameter (0<=0<1) for position weighting

2) Options that can be used under DWDS_Index or MorphXML_Index:

ArchiveIndex	If on, DDC archives all indices (see this for more detail)
QueryOnlyFiles	If on, DDC doesn’t create a sentence break collection

3) Options that can be used only under DWDS_Index

Option	Description
UseDwdsThesaurus	Create and use “Thes” index under DWDS_Index
IndexPunctuation	Index punctuation marks
UseParagraphTagToDivide	If on, the tokenizer finds "</p>" in the input texts in order to divide the text into paragraphs
EmptyLineIsNotSentenceDelim	If on, an empty line in the input texts is not an end of sentence
DontUseIndention	If on, DDC doesn’t use indention to find paragraph breaks
IndexMorphPatterns	If on, DDC creates “MorphPattern” index

4) Options that can be used only under Free_Index

Option	Description
IndexChunks	Enables indexing and querying using chunks(see [2]), otherwise chunks are ignored.
Indices <s>	<s> is a string of index declarations, delimited by “;”. One index declaration is a tuple [**x a b s**], where **x** is a long name of the index, **a** is a short name, and **b** can “archive” or “normal”. If **b** is “archive”, then this index is archived, otherwise it is not. The last member **s** can be ”storage” or “storage_omit”, if it is ”storage” then the index is supplied with a storage during indexing, otherwise storage is not built. By default the first index is build with a storage (if it is not manually prohibited), other indices are built without storages.
HitBorders <s>	<s> is a string of break collection declarations, delimited by “;”. One break collection declaration is a triplet [**x:a:default**] or a pair [**x:a**] , where **x** is a long name of the collection, **a** is a short name. If a break collection is called default, then it is used for queries, which do not contain break collection specification („within“ operator).
IndicesToShow <s>	<s> is a list of integers, where each integer **i** is an index of some declaration, declared in option ”Indices”, so for each **i** it is true that 0<**i** <**n**, where **n** is the number index declarations. For each output word this list determines the order of the indices, whose values are used to represent this word in the output hit. The integers are separated by blanks. By default IndicesToShow is equal to “1”, that means that words are represented only by the first index (normally it is the token itself). If some index is mentioned in IndicesToShow, then it must have an index storage, that is built during indexing.
InterpDelimiter <s>	<s> is a string that should be used to separate different interpretations of the one word. By default it is ‘#’.

Here are some examples of options files:

1. Russian plain texts:

```
IndexType DWDS_Index
Russian
IndexMorphPatterns
UserMaxTokenCountInOnePeriod 5000000
HtmlHighlighting .<first>;</first>;<rest>;</rest>
```

2. German XML texts with bibliography:

```
IndexType DWDS_Index
German
UseParagraphTagToDivide
EmptyLineIsNotSentenceDelim
DontUseIndention
UserMaxTokenCountInOnePeriod 10000000
IndexMorphPatterns
OutputBibliographyOfHits
```

```
IndexPunctuation
Bibl string 1 textClass /TEI.2/teiHeader/profileDesc/textClass/keywords/term
Bibl string 0 author /TEI.2/teiHeader/fileDesc/sourceDesc[@id="orig"]/biblFull/titleStmt/author
Bibl string 0 date /TEI.2/teiHeader/fileDesc/sourceDesc[@id="orig"]/biblFull/publicationStmt/date[@id="first"]
Bibl string 0 orig /TEI.2/teiHeader/fileDesc/sourceDesc[@id="orig"]/bibl
Bibl string 0 scan /TEI.2/teiHeader/fileDesc/sourceDesc[@id="scan"]/bibl
Bibl string 0 page /TEI.2/teiHeader/fileDesc/sourceDesc[@id="orig"]/seriesStmt/idno[@type="page"]
TextArea body /TEI.2/text/body
TextArea title /TEI.2/text/title
```

3. An options file for a web-site search engine:

```
IndexType DWDS_Index
Russian
LocalPathPrefix /home/sokirko
InternetPathPrefix www.aot.ru/
QueryOnlyFiles
CaseInsensitive
```

4. An example for free index:

```
IndexType Free_Index
English
Indices  [Token w normal];[Lemma l normal]
HitBorders [s:sentence:default];[c:clause]
OutputBibliographyOfHits
IndexChunks
Bibl string 0 orig /TEI.2/teiHeader/fileDesc/sourceDesc[@id="orig"]/bibl
Bibl string 0 date /TEI.2/teiHeader/fileDesc/sourceDesc[@id="orig"]/biblFull/publicationStmt/date[@id="first"]
Bibl integer 0 relevance /TEI.2/teiHeader/fileDesc/relevance
TextArea body /TEI.2/text/body
```

Index item formats

Unless a corpus is indexed under Free_Index type, each index item has its own predefined format, according to the name of index:

- One string of "Token" index can be either a sequence of chars in Russian (Windows-1251), English, or German (ISO-8859) alphabets plus digits and hyphens or it can be a sequence of punctuation marks. These strings cannot contain a mixture of alphabets and punctuation marks. A punctuation marks is one of the following chars:
  ^!"§$%&/()=?*'_:;+#-.,\{}[].
  It is possible to index texts that contain chars other than punctuation marks and alphabets, but those chars wouldn't be found by DDC.

- One string of "MorphPattern" index must have the following format:
  $P\ f_1,f_2,..,f_n$
  where P is a part of speech and $f_1,f_2,..,f_n$ are morphological features, for example
  SUB plu,nom,mas
  These strings are eventually converted to the inner "MorphPattern" format which looks like this:
  $@P@@f_1@@f_2@\dots@f_n$
  If the input token has more than one morphological annotations than these annotations are delimited by "#" (Chars "#" and "@" are supposed not to occur inside morphological patterns), for example, if word has two patterns
  "SUB plu," and "VER prt", then the inner formatted string would be:

@SUB@@plu$_1$@#@VER@@prt@

Obviously if the index type is **Free_Index**, one can introduce an index called "MorphPattern" and fill it with the strings in the inner format. If this is fulfilled, then one can use a special query construction, which refers this index.

- One string of "Thes" index should have the following format:

  T_1:T_2: … :T_n

  where T_i is a thesaurus node. For example:

  Human:Programmer:Sokirko

  DDC supports a special query construction, which refers this index.

Archived indices

DDC can archive indices for tokens and their properties. An archived index is normally 30% smaller than the same normal index, but it takes up to 5% more time to browse an archived index, than a normal one. The principal idea of archiving is adopted from Jakarta Lucene Project [6]. Under Free_Index the last attribute of an index description is responsible, whether DCC should archive this index or not. Under MorphXML_Index or DWDS_Index DDC archives all indices, if option ArchiveIndex is included in the options file.

Working under MorphXML_Index

Under MorphXML_Index DDC indexes xml-files, where each word has a morphological annotation. On the one hand, this type is better than simple DWDS_Index type**,** since under DWDS_Index each token has always one and the same morphological annotation regardless of the context. On the other hand, the indices, which are built under this index type, are compatible to DWDS_Index, and that's why one can use the compact atomic query construction, instead of the bulky general construction, which one has to use under Free_Index.

One MorphXML_Index document has a header with bibliographical information and a sequence of sentences. Each sentence is a sequence of words, to which a sequence of morphological annotations is attached. One morphological annotation contains three parts:

1. lemma,
2. part of speech,
3. a set of morphological features.

A part of speech or a morphological feature as string constants must consists of English chars, digits or hyphens. Lemma should be in the alphabet of the corpus.

The complete document type definition for in Appendix 1. An example for this document type definition is in Appendix 2.

*Working under **Free_Index***

In order to illustrate, how DDC works under Free_Index, let us consider an example. Suppose we have a text of two German sentences:

Mutter kam nach Hause, deswegen war ich froh. Mutter war auch froh.

Suppose we can annotate this text with sentence breaks, clause breaks, NP, VP, and PP chunks. Also we could attach to each word a lemma. This can be written in CWB format[2] . Suppose also that we can add to this text a small header, which contains its bibliographical information. This would result a file, which we would call “text1.table” (see Appendix 3).

This “text1.table” should actually be a well-formed XML-file. Note, that the second occurrence of “Mutter” is prefixed with “-#”, which leads to excluding this occurrence from the token index.

After source file creation we can create a source file list (let it be “corpus.con“), which would contain only one line:

```
text1.table
```

Then we can create a file of options:

corpus.opt:

```
German
IndexType Free_Index
Indices  [Token w normal];[Lemma l normal]
HitBorders [s:sentence:default];[c:clause]
OutputBibliographyOfHits
IndexChunks
Bibl string 0 orig /TEI.2/teiHeader/fileDesc/sourceDesc[@id="orig"]/bibl
Bibl string 0 body /TEI.2/text/body
```

This options file register two indices: one for tokens and one for lemmas, and two break collections: one for sentences and one for clauses. Note, that it is obligatory to use the short names of break collections as the tag names in the source file.

This index structure enables us, for example, to query the corpus in the following way:

1. Mutter
   // produce the first sentence but not the second, since the second occurrence of
   // „Mutter” is deleted.
2. froh #within clause
   // produce the second clause of the first sentence and the second sentence
3. $l=MUTTER
   // produce all words whose lemma is “MUTTER” (two occurrences, not one!)
4. $l=MUTTER && !Mutter
   // produce only the second (deleted) occurrence of “Mutter”.
5. ^pp
   // produce the first sentence and highlight “nach Hause”

Now we want also to supply each output word with a lemma. To achieve it we are to add IndicesToShow and InterpDelimiter options and supply lemma index with a storage:

corpus.opt:

```
German
IndexType Free_Index
Indices  [Token w normal];[Lemma l normal storage]
HitBorders [s:sentence:default];[c:clause]
OutputBibliographyOfHits
IndexChunks
Bibl string 0 orig /TEI.2/teiHeader/fileDesc/sourceDesc[@id="orig"]/bibl
Bibl string 0 body /TEI.2/text/body
IndicesToShow 1 2
InterpDelimiter ^
```

After reindexing the corpus all output words would be supplied with lemmas, for example:

query: Mutter

answer:
Die^die   Mutter^MUTTER   kam^kommen nach^NACH   Hause^HAUS .^.

Reading bibliographical information from XML

If some corpus file is an xml-file, then DDC tries to get from it bibliographical descriptions(bibliographical fields). These bibliographical fields would be used to describe the document in which the hit is found.   They also can be used to filter and to sort hits. Some bibliographical fields are predefined and have special processing in DDC, other fields are user-defined. However all bibliographical fields should be listed in the options file in Bibl field, since it is the only way to set an xpath of bibliographical field.

Here is the list of all predefined bibliographical fields:

Name	Description
body	The bibliographical field is the only mandatory for indexing xml files. It contains an xpath to document body, from which DDC retrieves tokens to index.
Scan	This is the bibliography of the document, which was built during its scanning. In html format or text format the piece of information always starts a hit header. This field cannot be used for filtering or sorting.
Orig	This is the original bibliography of the document, which normally contains the title of the document. If there is no “scan” bibliography for the document, then a hit header is started with the original bibliography. This field cannot be used for filtering or sorting.
Date	Document issue date in format YYYY-MM-DD. This field is indexed as dates. The index is used in the date filter operator and the date sort operators.
PageRank	This XML node contains the outward page rank of the document. The value of the page rank should be integer and 0<=PageRank<=1000
page	This field contains the number of the first page in the xml-file. This field is indexed as integer.

A user-defined bibliographical field (some field that is not called “orig”,”date”, “page”, “body” or “scan”) is always indexed as a set of strings. For such set of strings it is possible to use the general bibliographical filter.

If some xpath, which is defined for a bibliographical field, does not point to a leaf in the XML structure, then DDC concatenates all descendent text elements using delimiter “:”.

Page breaks

The source texts can be divided into pages. In all formats (Free_Index, DWDS_Index, MorphXML_Index) the following construction denotes a page break:

<pb n=”**cc**”/>, where **cc** can be a decimal number, for example:

<pb n=”123”/>

The text below <pb n=”123”/> is considered to be on page number 123 until the next <pb/> tag or the end of the text. The order of page numbers is not relevant. It is possible to specify the starting page number in the bibliographical header (see predefined field page). If the staring page is specified, then it would refer to the beginning of the text till the first <pb/> in the body. For example:

```
Bibl string 0 body /example/ body
Bibl integer 0 page /example/header/startpage/@index

<example>
    <header>
        <startpage index="4>
    </header>
    <body>
        text on page 4
        <pb n="5"/>
        text on page 5
        <pb n="6"/>
        text on page 6
    <body>
<example>
```

Creating and indexing a corpus

To establish a corpus one should create two files that have the same name but different extensions:

- File of options (*.opt)
- File of source files (*.con).

These files as well as the index files, which DDC creates, should be always in the same directory. To start indexing one should type[1]:

```
ConcordIndex <file.con>
```

where <file.con> is file of source files.

As it was said ConcordIndex would use the options file, which is in the same folder as the file of source files. ConcordIndex return "0" on a successful execution otherwise one should read error messages, which ConcordIndex prints.

During corpus creation, one should keep in mind the following restrictions:

1. Number of running tokens for one corpus couldn't be more than 2^{29}(536 mln.). This restriction is followed from the fact that DDC uses "fseek" function to move the file pointer to a specified location for storage files and files of occurrences. This function works with signed double word file offsets ("long"). Notice, that one occurrence is also a double word.

2. The length of the input token cannot be more than 4096 for Free_Index and MorphXmlIndex. For DWDS_Index DDC uses its own tokenizer, which cannot produce tokens that exceed 4096 chars, so for DWDS_Index this restriction is irrelevant

If the input corpus does not satisfy these restrictions, then ConcordIndex could not index it.

The process of indexing starts with loading of the list of source files. Each file should

[1] Here we presuppose that DDC was already installed on the computer.

be stored on the local hard disk. After indexing the source files can be removed. The options file should always stay, since DDC uses it during loading and querying.

Document relevance (Ranking)

The rank of a document (or more generally, a hit) is a sum of outward page rank (for example, the number of links to this document) and the measure of correlation between the input query and the document. The outward page rank is an integer, stored in the XML files.

The measure of correlation between the input query Q and the document D is calculated using the following formula:

$R(D,Q) = \lambda_1 * TFIDF(D,Q) + \lambda_2 / Near(D,Q) + \lambda_3 / Position(D,Q)$,

where $\lambda_1+\lambda_2+\lambda_3 = 1$

Parameters λ_1, λ_2, λ_3 could be set in the options file: `TfIdfRank` (λ_1), `NearRank` (λ_2), `PositionRank` (λ_3).

TFIDF (D,Q) is an augmented version of the standard tf.idf weighting, that can be declared as follows. Let q_1, q_2,…,q_n be different query items that are found in document D.

$$TFIDF(D,Q) = \frac{\sum_{i=1}^{n} TFIDF(D,q_i)}{n}$$

$TFIDF(D,q) = \beta + (1-\beta)\cdot tf(D,q)\cdot idf(D,q)$, where β=0.4

$$tf(D,q) = \frac{freq(D,q)}{freq(D,q) + 0.5 + 1.5\cdot\frac{len(D)}{avg\_dl}},$$

where freq(D, q) is the frequence of q in D, len(D) is the length of D in words, avg_dl is the average document length in the corpus

$$idf(q) = \frac{\log(\frac{|c|+0.5}{cf(q)})}{\log(|c|+1)},$$

where |c| is the number of documents in the corpus, cf(q) is the number occurrence of q in the coprus.

Near(D,Q) is the length of the minimal representative document segment that contains the maximal number of different query items. For example,

Q = (a && b),
D = “a c b c c b c b a”,
Near(D,Q) = 2 (the length of the minimal representative substring is 2 (“b a”))

Position(D,Q) is the position of the first minimal representative document segment, for example

Q = (a && b),
D = “a c b c c b c a”,
Near(D,Q) = 3
Position(D,Q) = 1

In other words, ranking using TFIDF (D,Q) is based on the following principles:

1. The more frequent the input query items occur in D, the higher is the rank (due tf(q));
2. The larger D is, the higher is the rank (due to tf(q));

3. The more frequent the input query items occur in the whole corpus, the lower is the rank (due to idf(q));
4. The shorter the representative document segment is, the higher the rank is (due to NEAR(D,Q));
5. The closer to the beginning the representative document segment is, the higher the rank is (due to Position(D,Q)).

Notice that to some extent Position(D,Q) is used to express different priorities of text areas of document. The order of text areas declarations in the options file determines the order of indexing of these text areas, for example, if there are two text areas in the options file

```
TextArea title /xml/title
TextArea body /xml/body
```

Since text area "title" is declared first, it is indexed earlier than "body", that's why "title" is closer to the beginning of document

It is also important to mention that TFIDF(D,Q) can be applied not only to documents (i.e. corpus files) but also to other hits (sentences, paragraphs and so on)

DDC query language

DDC query language is defined inductive. First one should define "atomic" queries, which refer one word, then those, that refer sequence of words and at last queries, which use logical operations.

Atomic queries:

Type	**Examples**	**Result**
Word[2]	Hause	All hits that contain some morphological variant of "Hause".
	\,	All hits that contain a comma.
	\?!	All hits that contain "?!" (a sequence of punctuation marks, which was indexed as one token).
Word*	Ha*	All hits that contain a word that starts with "Ha".
*Word	*ken	All hits that contain a word that ends with "ken".
/regexp/	/^Ha.*ken$/	All hits that contain a word that ends with "ken" and starts with "Ha".
@Word	@Hause	All hits that contain word form "Hause".
[P F][3]	[SUB sin] [VER aux,inf,]	"P" is a lexical category; "F" is a comma-delimited list of morphological features. Under DWDS_Index both lists depend upon the language of the corpus.
{H}	{AMTBERUFG}	All hits that contain a subtype of the given thesaurus

[2] Word should satisfy the token definition, if this token consists of punctuation marks, then it should be prefixed with a backslash.

[3] This construction refer "MorphPattern" index, which should be created under IndexType=DWDS_Index or under IndexType=MorphXml_Index. This index can be manually created under IndexType=Free_Index, but the input index items should be properly formatted.

	{EIG}	node. To use this construction one should enable UseDwdsThesaurus in the options file. If this option is enabled, then DDC create "Thes" index, which would be used for this type of query.
$I=W	$Token=W	This atomic query is useful under Free_Index. "$I" is a registered index name (specified in field "IndicesStr") and "W" can be a complete indexed string or a regular expression (with backslashes). If "I" is "Token" or "Thes" then one can use above-mentioned atomic queries, since, for example, query "$Token=Mutter" or is equal to query "Mutter" and query "$Thes=/:EIG:/" is equal to {EIG}. But for MorphPattern index a query like $MorphPattern=/[SUB plu]/ doesn't yield a valid result, since morph patterns are stored in a special format.
	$w=/^Ha.*ken$/	
%L	%mutter	Finds all hits, which contain lemma "L". This construction refers "Lemma" index and can be used under MorphXml_Index . One can use it also under Free_Index, but then "Lemma" index should be in the index definition.

Let X_1,X_2, … X_N be some atomic queries and let d_1,d_2,…,d_n be integers, $0<d_i<32$, then the following constructions are queries for sequence of tokens:

"X_1 X_2 … X_N"	"mein Haus"	All hits that contain "mein Haus".
	"Haus [VER]"	All hits that contain "Haus" which is followed by a verb.
"X_1 #d_1 X_2 #d_2 … #d_{n-1} X_N"	"mein #1 Haus"	All hits that contain "mein", which is followed by "Haus", and may be there is one word between them
	"Haus #10 [VER]"	All hits that contain "Haus", which is followed by a verb, and may be there are up to ten tokens between them.

Let Q1, Q2, Q3 be some queries, then the following constructions are also queries:

(Q1 && Q2)	Hause && [SUB sin]	All hits that contain Q1 and Q2
(Q1 && !Q2)	[SUB sin] && !Haus	All hits that contain Q1 and not Q2
(Q1 || Q2)	[VER aux,inf,] || [SUB sin]	All hits that contain Q1 or Q2
near(Q1;Q2;n)	NEAR (Hause ; [SUB]; 2)	All hits that contain "Hause" and some noun, and the distance

		between them is less or equal 2. The order of the occurrences is not important.
near(Q1;Q2;Q3;n)	NEAR (Haus;Mutter;Vater; 10)	All hits that contain "Haus" and "Vater", and there are up to 10 words between them and one of those words is "Mutter".

Here are some examples of complicated DDC queries:
"@Es [VER sin] [SUB nom] und [SUB nom]" && !geben
near(das; [SUB] with *chen; 0)

The meaning of a query can be also modified by so called query operators. Query operators are special strings, which are attached to the end of a query. If Q is a query, then the following constructions are also queries:

Name	Prototype	Example	Description
Context operator	Q #cntxt **N**	Vater #cntxt 2	DDC prints the actual hit together with its right and left contexts (**N** is the size of context (0<**N**<6)). The context doesn't exceed the file borders. This operator is ignored if "NoContextOperator" option is included in the options file or if the file break collection is used.
Within operator	Q #within C	Vater #within file	Instructs DDC to use break collection "C" instead of the default one. "C" can be a name of any registered break collection.
Corpus chooser	Q :$c_1,c_2,\ldots,c_n$	Mutter :dw1,dw2	This query operator is processed only by a DDC server (in the distributed model). This operator instructs DDC to redirect the input queries only to the selected corpora ($c_1,c_2,\ldots,c_n$,) ignoring other output sockets.
Date operator	Q #is_date[**d**]	Mutter #is_date[1990-01-01]	Finds documents whose issue date is **d** (in the example it is "1990-01-01"). The format of date should be "yyyy-mm-dd".
Less by rank operator	Q #less_by_rank	Mutter #less_by_rank	DDC puts the output hits in ascending order using the document rank.

Greater by rank operator	Q #greater_by_rank	Das #greater_by_rank	DDC puts the output hits in descending order using the document rank.
Size operator	Q #is_size[**d**]	Mutter #is_size[10]	DDC outputs only hits of length **d** (measured in tokens).
Less by date size	Q #less_by_size Q #less_by_size[d_1] Q #less_by_size[d_1,d_2]	Mutter #less_by_size	DDC puts the hits in ascending order using their size in tokens Parameter d_1 is the lower bound, d_2 is the upper bound.
Greater by size operator	Q #greater_by_size Q #greater_by_size[d_1] Q #greater_by_size[d_1,d_2]	Das #greater_by_date	DDC puts the hits in descending order using their size in tokens. Parameter d_1 is the upper bound, d_2 is the lower bound.
Common filter operator	Q #has_field[n, v]	Mutter #has_field[author,Grass] Mutter #has_field[author,Gr*] Mutter #has_field[textClass,*Berlin] Mutter #has_field[textClass,/Berlin/]	DDC outputs only hits that are found in the documents whose user-defined bibliographical field "n" is restricted by "v"[4].
General greater operator	Q #greater_by[n] Q #greater_by_size[n.d_1] Q #greater_by_size[n,d_1,d_2]	Das #greater_by[relevance,2,1]	DDC puts the hits in descending order using free bibliographical field, called "n". Parameter d_1 is the upper bound, d_2 is the lower bound.
General less operator	Q #less_by[n] Q #less_by_size[n.d_1] Q #less_by_size[n,d_1,d_2]	Das #less_by[relevance,1,2]	DDC puts the hits in ascending order using free bibliographical field, called "n". Parameter d_1 is the lower bound, d_2 is the upper bound.
File names operator	Q #file_names	Mutter #file_names	If specified, then DDC does not output bibliography for each hit, instead of it DDC prints file names. This operator makes sense only if option "OutputBibliographyOfHits" is specified in the options file, otherwise DDC outputs file names by default.

[4] Regular expression (in slashes) can be used for string bibliographical fields, but it is not necessary for a regular expression to cover the whole string, but it is possible only to be found in it.

It is worth to tell that date operators and common filter operator are senseless, if the input corpus doesn't contain the corresponding bibliographical information. If there is some bibliographical information but not each corpus file has it, then ascending sort operators can produce undesirable results, since the files with missing bibliography would be always the first ones.

Output formats

As the result of queries the user always gets a sequence of hits. This sequence can be in different formats, but the content is almost always the same:

1. In the beginning DDC outputs information about the number of found hits, which is followed by a sequence of hits.
2. For each hit there should be a reference to a corpus file, which this hit contains.
3. Together with the hit string DDC outputs information about words that causes DDC to include this hit into the resulted set. These words are called "highlighted words".

These three pieces of information can be in the following formats:

- Text format. Each hit is written on a separate line. Each hit contains at least 3(may be more) fields, delimited by "###":

  1. a file reference or a scan/orig bibliographical field depending on whether option OutputBibliographyOfHits is in the options file;
  2. the contents of "date" bibliographical field;
  3. possibly many values of the visible user defined bibliographical field , delimited by "###";
  4. the hit string;

  For the distributed version (see ddc_xml with –text option) the first line contains numbers of hits for all corpora and the second line contains the number of all relevant documents, for example, the first two lines could be the following:

  ```
  Corpora Distribution:9=5+4;
  Relevant Documents Distribution:3=2+1;
  ```

  Here "9" is the number of all found hits in all corpora. "5" is the number of hits in the first corpus, "4" is the number of hits in the second corpus. "3" is the number of relevant documents in all corpora; "2" is the number of relevant documents for the first corpus; "1" is the number of relevant documents for the second corpus.
- Html format. If OutputBibliographyOfHits is included in the options file, then DDC outputs bibliographical reference for each hit, otherwise if LocalPathPrefix and InternetPathPrefix options are initialized, then DDC creates an HTML reference to a corpus file, replacing LocalPathPrefix with InternetPathPrefix, otherwise DDC prints for each hit the corpus file name, which this hit contains.

- Table format. DDC creates for each hit the following line:

$$n_1=v_1 \text{ \textbackslash x2 } n_2=v_2 \text{ \textbackslash x2 } \ldots\ n_i=v_i \text{ \textbackslash x2 keyword=}v_{key} \text{ \textbackslash x2 page=}v_{page} \text{ \textbackslash x2 HitStr}$$

where $n_1,n_2,\ldots,n_i$ are the names of all bibliographical fields, $v_1,v_2,\ldots,v_i$ are values of the fields for the document. v_{key} is a concatenation of all unique highlighted words, v_{page} contains the page number, where this hit was found. "HitStr" is the hit string itself. "\x2" is the character with number 2.

If somebody searches a pattern using a file break collection, then DDC outputs hit strings in a compressed format that are usually called "snippets". Each DDC snippet contains not more than ten kwic lines for each found file (the particular number of lines can be set using NumberOfKwicLinesInSnippets parameter). The lines should represent different query parts if the input query is not atomic. For example, for query "mother && father" the snippet should contain both words ("mother" and "father").

Querying a corpus

In order to query a corpus one should run the program called "ConcordConsole". It has a command-line interface with the following commands:

help – print help information
load <Concordance Project> - load project
query <query> - run a query
set limit <limit> - set the maximal number of hits to find (default 10)
set out <file> - print hits to <file>
set result_format html|txt|table - set the format of hits
exit - exit
showtime - toggle show time switcher

For example if one has created and indexed a corpus called "tst.con", then the following instructions would print the results for query Mutter||Vater to file "test.html":

```
load tst.con
set out test.html
set result_format html
query Mutter||Vater
exit
```

Distributed model

DDC can run as a distributed application. Under this paradigm DDC functions as a number of network agent with different roles. These agents communicate between themselves over the network using TCP/IP and the contents of their messages is determined by their roles, which can be the following:

1. A corpus thread is a thread that holds one corpus; this corpus can be queried via a TCP/IP socket, which was opened by this corpus thread.
2. A corpus server is a server that can receive queries, redirect it to all corpus threads that are known to this server, concatenate the results and outputs them to a corpus client.
3. A corpus client is a program that can work via TCP/IP with a corpus server.

All these programs can be run on different computers or even under different operating systems.

Corpus threads and corpus server

DDC reads information about the corpus threads from so called local corpora file. This file contains descriptions of corpora, for which ConcordDaemon should create corpus threads. Each line of the file contains four pieces of information:

1. Corpus name ;
2. IP Address of the socket to open for this corpus;

3. Port of the socket;
4. Local path to the project corpus file[5].

For example:

```
GermanCorpus 192.168.1.77   50001   /home/sokirko/TestCorpora/german.con
RussianCorpus 192.168.1.77   50002   /home/sokirko/TestCorpora/russian.con
```

The configuration information for the corpus server is stored in a server sockets file. This file contains descriptions of output sockets, which should be queried by the server, and one description of the input server socket, which is used to query the server itself. Each line of the file contains three pieces of information:

1. Corpus name[6] ;
2. IP Address of the socket to open for this corpus;
3. Port of the socket;

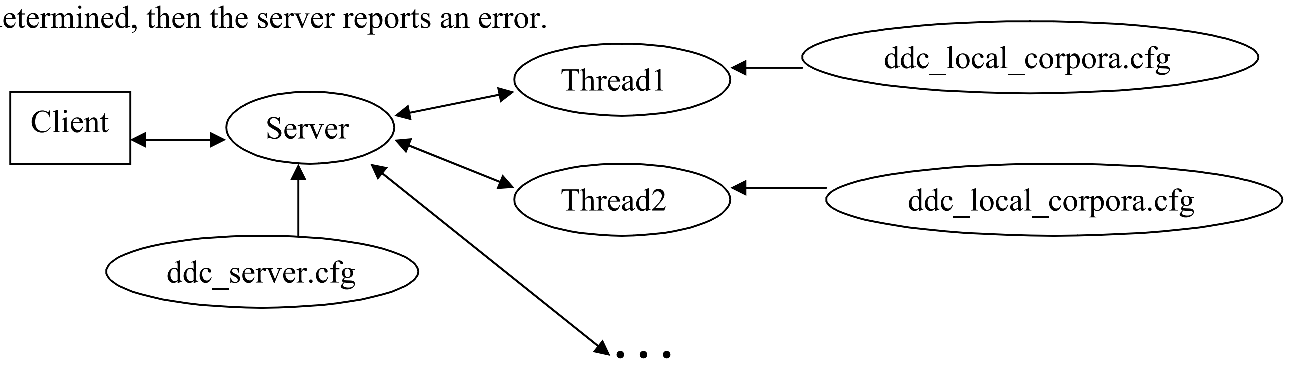
For example:

```
server    192.168.1.77   50000
Corpus1 192.168.1.77   50001
Corpus2 192.168.1.78   50002
```

The first line contains input server socket description (the name “server” is predefined). The input socket would be opened in the listening mode, other sockets would be initialized only upon arriving of a query and closed after it’s processing. In the example if the input server socket(192.168.1.77:50000) receives a query, then the server redirects it to sockets “Corpus1” and “Corpus2”. If the input query contains a corpus chooser operator, then the server uses only the selected corpora, for the last example a query can be as follows:

Mutter && Vater : Corpus1

The querying is accomplished in a parallel mode, so the server redirects the query and then waits for results. If some socket doesn’t answer in the timeout, which the client has determined, then the server reports an error.



DDC socket messages

The only communication method between clients, servers and threads is based upon socket packets. For DDC socket packets are strings, which are sent over a network. Depending upon the recipient the packets have their predefined formats and resp. their meanings. An input server socket can recognize two commands:

1. “run_query” with six parameters:
    a. name (is always “Distributed”);
    b. query - query string itself;
    c. format - output hit format (“html”, “text” or “table”);
    d. start - the starting index, from which DDC retrieves hits;

[5] The path should be full, i.e. under Unix it should start with char ‘/’.
[6] This corpus name can be used in a „Corpus chooser operator“, see „DDC Query Language”

e. limit - the maximal number of hits to retrieve;
f. timeout – timeout for querying in seconds.

2. "get_paradigm" with three parameters:
   a. a word for which DDC should print a paradigm;
   b. flag that let DDC print only lemma information without a full paradigm;
   c. language identifier.

Strictly speaking, the second command is not part of DDC machinery, since it deals only with morphological dictionaries and browses no DDC index at all. But this command is used to show a full paradigm of a word to a user, which is useful during the search process.

A <u>corpus thread socket</u> can recognize two commands:

1. "get_first_hits" with three parameters:
   a. query - query string itself;
   b. limit - the maximal number of hits to retrieve;
   c. timeout – timeout for querying in seconds;
2. "get_hit_strings" with three parameters:
   a. format - output hit format ("html", "text" or "table");
   b. start - the index, starting from which DDC retrieves hits;
   c. limit - the maximal number of hits to retrieve.

Normally a client sends to a server a "run_query" message, the server sends a "get_first_hits" message to all its output sockets, then it concatenates results, sorts them by the order identifiers, which is attached to each hit, then the server sends a "get_hit_strings" message to the sockets that have parts of the requested subsequence of hits.

Each packet syntactically looks like this:

command $l_1\ l_2 \ldots l_3$

where command can be "run_query", "get_paradigm", "get_hit_strings" or "get_first_hits", and $l_1\ l_2 \ldots l_3$ is a list of command parameters, delimited by char \001. For example

run_query Distributed∇Test&&West∇html∇1∇10∇199 [7]

Decoding from this string one can get the following message parameters:

command is "run_query",
name is "Distributed",
query is "Test&&West",
format is "html",
start is 1,
limit is 10,
timeout is 199.

ConcordDaemon

The functionalities of corpus threads and corpus servers are implemented by one and the same program, which is called ConcordDaemon. Upon starting ConcordDaemon looks for the local corpora file and for the server sockets file. Then it creates corpus threads and corpus servers and starts waiting for incoming messages. Here is a summary of the usage of ConcordDaemon:

ConcordDaemon <action> [--local <l>] [--server <s>] [--protocol <p>],

[7] We replace in this example char \001 with ∇, since char \001 in not printable.

where <action> can be “start” (to start ConcordDaemon) or “stop” (to stop ConcordDaemon), option “--local” is used to redefine the name of the local corpora file (by default $RML/Bin/ddc_local_corpora.cfg), option “--server“ is used to redefine the name of the server socket file (by default $RML/Bin/ddc_server.cfg), option “--protocol“ is used to change the verbosity level, <p> can be “weak”, “medium” or “heavy”.

ConcordDaemon prints information about the querying process to log files. The log files are situated in $RML/Logs/concord. This catalog should be created before running ConcordDaemon. Logs are stored for each date separately, for example “01November2004.log“ contains all log information for 1 November 2004. There are two types of log messages: loading log messages (which show the loading status of a corpus thread or the corpus server) and working log messages (which are issued upon a query arrival) .

The basic loading messages are the following:

- “Loading corpora” , it is issued by a corpus thread, when it starts to load the corpus from the disk;
- “Create Listener” , it is issued by a corpus thread, when it starts to create a listening socket;
- “LoadServer” , it is issued by the corpus server, when it starts to create itself
- “waiting for accept“ , it is issued by the corpus server or a corpus thread, when it successfully finishes creating a listening socket.

While starting, a corpus thread or a corpus server should issue at least two messages: “Loading corpora”, resp. “LoadServer”, and then “waiting for accept”, if the second message has not arrived, then some error message must be printed with an explanation.

During the querying each socket can log the incoming packets, but by default only a corpus server sockets prints them. The full logging can be switched on using option “ --protocol heavy” . The incoming packets are printed in the following format:

C recv “**packet**”,

where “C” is a corpus name or “server” if it is the server socket, and “packet” is the incoming packet itself.

Client “Search” (a CGI Script)

Program “Search” is a simple CGI-program[8] that can interact with a corpus server. The program receives a query from an HTML form, which should look like as follows:

File s.htm

```
<html>
<body>
<form method="POST" action="../cgi-bin/search">
<input type="text" name=SearchText size="136">
<input type="submit" value="Submit" name="New">
<input type="hidden" name="CorporaName" value="DialingDocsCorpora">
<input type="hidden" name="HeaderFormat"
value="<br>Hits: %i-%i of %i in %i relevant documents<br>">
<input type="hidden" name="TemplateFile" value = "../htdocs/s.htm">
<input type="hidden" name="MoreButton" value = "more">
<input type="hidden" name="Timeout" value = "100">
<input type="hidden" name="ResultLimit" value = "10">
<!--ResultText-->
</form>
</body>
</html>
```

[8] The sources of the program uses “Cgic” library (see www.boutell.com/**cgic**)

Line 03 contains a starting tag of the form and a reference to the CGI-script itself.
Line 04 contains a text control for the input query.
Line 05 contains the button, which submits this form.
Line 06 contains a name of a corpus server, to which the query would be sent. The IP and port of the corpus server is in $RML/Bin/concord_cgi.cfg, whose syntax is equal to the syntax of a server sockets file.
Line 07 contains a format, which would be used to print the header of a query result. Here the first "%i" stands for the starting index, from which DDC retrieves hits, the second %i stands for the end index of the retrieved hits, and the third %i stands for the whole number of hits and the last %i stands for the number of relevant documents.
Line 08 contains the self-reference to this HTML-page.
Line 09 contains a title for the button, which is used to get more hits after the initial query has been executed.
Line 10 contains the value for timeout, after which the corpus server should stop processing the input query.
Line 11 contains the maximal number of hits, which can be output at once on one HTML-page.
Line 12 contains the special commentary block (<!--ResultText-->), which would be replaced by the query result.
Upon pressing "submit" button this script builds a run_query message, sends it to "CorporaName" server, gets the results, takes the input form as a prototype, replaces <!--ResultText--> with the query results, and if there are more hits, than the script has now, then it adds "MoreButton" button to the end of the form, .
This CGI-script logs the input queries to $RML/search.log.

Client "ddc_xml" (a command-line tool)

Program "ddc_xml" is a simple console tool that can interact with a corpus server. Its parameters are the following:

ddc_xml <query> <options>,
where <query> is a query,
<options> can be
-Limit N is the maximal number of hits to find
-text if specified, let the program create a text file(by default is xml)
-timeout N set the timeout in seconds (by default it is 200 seconds)

Examples:
```
ddc_xml  "test&&west" -Limit 100
ddc_xml  "test&&west" -Limit 100 -text
ddc_xml  test  >out.xml
```

Upon starting this program loads file "$RML/Bin/ddc_xml_server.cfg". The syntax of "ddc_xml_server.cfg" should be the same as in a server sockets file. In this file ddc_xml finds the first description of "server" socket and tries to establish a connection to it. If it was successful, ddc_xml sends a run_query message. This run_query message builds from the query the input parameters of ddc_xml. If "-text" option is specified, then the output format is set to "text", otherwise it should be "table". When ddc_xml receives the results and if "-text" is specified then it simply prints the results to the standard output and exits, otherwise it loads file "$RML/Bin/ddc_template.xml", which is a prototype xml file to build xml using DDC results in "table" format. This file contains a set of place holders, which are marked with "##". For example:

<entry>

<date> ##date## <date>
<date> ##context## <date>
</entry>

Here "##date##" and "##context##" are place holders. Each place holder should be replaced by the value of bibliographical field with the same name (for example, ##date## should be replaced with "date" bibliographical field), with two exceptions:

1. If place holder is called "query", then it should be replaced with the input query.
2. If place holder is called "context", then it should be replaced with the hit string.

References

[1] British National Corpus (http://sara.natcorp.ox.ac.uk/lookup.html).
[2] Stuttgart Corpus Encoding Tutorial
http://www.ims.uni-stuttgart.de/projekte/CorpusWorkbench/CWBTutorial/cwb-tutorial.pdf
[3] TinyXml project: http://sourceforge.net/projects/tinyxml
[4] Text Encoding Initiative http://www.tei-c.org/
[5] Jakarta Lucene Project http://jakarta.apache.org/lucene/docs/index.html
[6] TinyXPath project: http://sourceforge.net/projects/tinyxpath

Appendix 1. DTD for documents under MorphXml_Index

```
<?xml version="1.0" encoding="UTF-8"?>
<!--DDC exchange format -->
<!--last changes: 2004-08-13 by AG -->
<!ELEMENT ddc_document (header, text)>
<!ATTLIST ddc_document
        id CDATA #REQUIRED
>
<!ELEMENT header (bibl , author , title , date , textClass+ , idno*)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT bibl (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT idno (#PCDATA)>
<!ATTLIST idno
        type (ext_archive | ext_id) #REQUIRED
>
<!ELEMENT textClass (#PCDATA)>
<!ATTLIST textClass
        term CDATA #REQUIRED
>
<!ELEMENT title (#PCDATA)>

<!ELEMENT text (s)>
<!ELEMENT s (w | pb)+>
<!ELEMENT w (#PCDATA | ana)*>
<!ELEMENT ana EMPTY>
<!ATTLIST ana
        lemma CDATA #REQUIRED
        pos CDATA #REQUIRED
        gram CDATA #IMPLIED
>
<!ELEMENT pb EMPTY>
<!ATTLIST pb
        n CDATA #REQUIRED
>
```

Appendix 2. An example of a document under MorphXml_Index

File corpus.con:

```
text1.xml
```

File corpus.opt:

```
English
IndexType MorphXML_Index
OutputBibliographyOfHits
Bibl string 0 date /ddc_document/header/date
Bibl string 0 orig /ddc_document/header/bibl
Bibl string 0 textClass /ddc_document/header/textClass
Bibl string 1 author /ddc_document/header/author
TextArea body /ddc_document/text
```

File text1.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ddc_document id="1">
        <header>
                <bibl>Anzengruber, Ludwig, Der Gwissenswurm</bibl>
                <author>Anzengruber, Ludwig</author>
                <title>Der Gwissenswurm</title>
                <date>1874</date>
                <textClass term="1">Belletristik</textClass>
                <textClass term="2">Komцdie</textClass>
                <idno type="ext_archive">Gutenberg-DE</idno>
                <idno type="ext_id">1946/04/Zt19460314_005_0021_f</idno>
        </header>
<text>
<s>
      <pb n="1"/>
      <w> I
            <ana lemma="I" pos="PN" gram="1"/>
      </w>
      <w> was
            <ana lemma="be" pos="VERB" gram="sg,past-simp"/>
      </w>
      <w> in
            <ana lemma="in" pos="PREP" gram=""/>
            <ana lemma="in" pos="ADJ" gram=""/>
            <ana lemma="in" pos="NOUN" gram="narr,sg"/>
      </w>
      <w> Berlin
            <ana lemma="Berlin" pos="NOUN" gram="sg,loc"/>
      </w>
      <w> .
       <ana lemma="punct" pos="PUNCT"/>
      </w>
</s>
</text></ddc_document>
```

Appendix 3. An example for a document under Free_Index

File corpus.con:

```
text1.table
```

File corpus.opt:

```
German
IndexType Free_Index
Indices  [Token w normal];[Lemma l normal]
HitBorders [s:sentence:default];[c:clause]
OutputBibliographyOfHits
IndexChunks
Bibl string 0 orig /TEI.2/teiHeader/fileDesc/sourceDesc[@id="orig"]/bibl
Bibl string 0 date /TEI.2/teiHeader/fileDesc/sourceDesc[@id="orig"]/biblFull/publicationStmt/date[@id="first"]
Bibl string 1 page /TEI.2/teiHeader/fileDesc/sourceDesc[@id="orig"]/pb/@n
```

TextArea body /TEI.2/text/body

In "*corpus.opt*" we have defined two token indices ("Token" and "Lemma"). It means, that a token description should always contain two strings, delimited by a tabulation ("\t"), this first string should be a token and the second should be a lemma. As it was already said one token description is always written in <l> </l> tags.

File text1.table:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<TEI.2>
      <teiHeader>
            <fileDesc>
                  <sourceDesc id="orig">
                        <bibl>A test for free indexing</bibl>
                        <biblFull>
                              <publicationStmt>
                                    <date id="first">2004-11-03</date>
                              </publicationStmt>
                        </biblFull>
                        <pb n="4"/>
                  </sourceDesc>
            </fileDesc>
      </teiHeader>
      <text>
            <body>
                  <s>
                        <c>
                              <np>
                                    <l>Mutter   MUTTER</l>
                              </np>
                              <vp>
                                    <l>kam      KOMMEN</l>
                                    <pp>
                                          <l>nach     NACH</l>
                                          <np>
                                                <l>hause    HAUS</l>
                                          </np>
                                    </pp>
                              </vp>
                        </c>
                        <c>
                              <l>,  ,</l>
                              <l>deswegen DESWEGEN</l>
                              <vp>
                                    <l>war      SEIN</l>
                              </vp>
                              <np>
                                    <l>ich      ICH</l>
                              </np>
                              <l>froh     FROH</l>
                              <l>.  .</l>
                        </c>
                  </s>
                  <pb n="6"/>
                  <s>
                        <c>
                              <np>
                                    <l>-#Mutter MUTTER</l>
                              </np>
                              <vp>
                                    <l>war      SEIN</l>
                                    <l>auch     AUCH</l>
                                    <l>froh     FROH</l>
                              </vp>
                              <l>.  .</l>
                        </c>
                  </s>
            </body>
      </text>
</TEI.2>
```