Corpus Querying

Instructions on how to write a sketch grammar have been moved to a separate page.

Page Contents

1. Corpus Querying
   1. Corpus Query Language (CQL)
   2. Basic Queries
      1. Simple attribute-value queries
      2. Regular expressions
      3. Searching for position numbers
      4. Searching for structures
   3. Advanced operators
      1. Using within and containing operators
      2. Using the within operator in parallel corpora
      3. Using meet and union operators
      4. Using swap and ccoll operators
   4. Queries exploiting word sketches
      1. Queries exploiting thesaurus
      2. Queries exploiting word sketch triples
      3. Queries exploiting word sketch seeks
   5. Global conditions
   6. Query examples

Corpus Query Language (CQL)

The language was developed at the Corpora and Lexicons group, IMS, University of Stuttgart in the early 1990s, see ?IMS Corpus Workbench. The CQL as used in Sketch Engine is an extension to the original language and varies in several ways. This documentation describes the CQL as implemented in manatee 2.114 (released January 2015).

- A query consists of a regular expression over attribute expressions and/or structures.
- The attributes used in the examples provided below are word and tag. These examples assume that in our corpus every word has an associated part-of-speech tag referred to as tag.

Basic Queries

Simple attribute-value queries

- The general form to query a positional attribute value is

  [attr="value"]

- For example, very often, you only want to look for a given word (e.g. teapot), so attr would be word and the value would be teapot

  [word="teapot"]]
You might want to broaden the search, for example you want to find all occurrences of words beginning with `confus`. The full form is

```
[word="confus.*"]
```

but you can make use of the so called default attribute (in this example selected as `word`) so we can simply use

```
"confus.*"
```

the default attribute can be changed using the drop-down list under the CQL box.

- Case is significant to the query processor. If you want case-insensitive search, include `(?i)` in a string

```
"(?i)on"
```

- We often want a wild-card word: any single word, it doesn't matter which. We use the "match any token" operator `[` (similar to the dot for "match any character" in regular expressions over strings): "confus.*`[` "by" This query finds all sequences of a word beginning with `confus`, followed by any word followed by `by`.

- We search for exactly two words between `confus.*` and `by` with

```
"confus.*`[]{2}` "by"
```

- We search for between 0 and 3 words between `confus.*` and `by` with

```
"confus.*`[]{0,3}` "by"
```

- The following comparison operators are also possible: `=`, `!=`, `<=`, `>=`, `!<=`, `!>=`, `==`, `!==`. For `<=`, `>=`, `!<=`, `!>=` operators, the attribute value is compared in such a way that alphabetical parts of the value are compared lexicographically and numerical parts numerically. The intended usage of this feature focuses on structure attributes, so that one can search for `<doc id>="AB2010CD">` and that will include documents with id such as "BB0000CD", "AB2011CD" or "AB2010CE". The `==`, `!==` operators differ from their single-character-counterparts in how they treat regular expression meta-characters. Normally such characters have to be escaped by backslash to gain their standard value, i.e. to find all dots, one needs to query for `word="\."]. Since this might be sometimes cumbersome, one can use `==`, `!==` which evaluate the value as a fixed string and not a regular expression. Note that even in case of `==`, `!==` for obvious reasons two characters need to be escaped anyway: the quote (`"`) and the backslash (`\`).

### Regular expressions

Regular expressions can be used for attribute values in almost all cases.

The regular expression operators available are:

- disjunction (`|`),
- Kleene star (`*`, as in our "confus.*" example above, this matches any number of repetitions, including 0),
- plus operator (`+`, matching 1 or more repetitions),
- optionality operator (`?`, optional, i.e. matches 0 or 1 occurrence)
- the interval operator

```
{n, k}
```
matches between \( n \) and \( k \) repetitions. If \( k \) is omitted, at least \( n \) repetitions are matched. If the interval has the form

\[
\{n\}
\]

exactly \( n \) repetitions are matched. The examples below will clarify this.

- Each attribute expression is -- roughly speaking -- evaluated against the word (and/or other, additional attributes) at a given corpus position. It has the form

\[
[\text{Boolean expression}]
\]

that is, an attribute expression is a \textit{boolean expression} surrounded by brackets.

- A boolean expression is a set of \textit{attribute value tests}, combined with the usual boolean expression operators conjunction (\&), disjunction (\|) and negation (\!). Parentheses may be used in the usual way.

- You may use back references within a regular expression by putting a part of the string into parenthesis and referencing it with \NUMBER\ starting with 2. E.g.: "(abra)kad\2" would match "abrackadabra" and "(a) (b) (c) \4\3\2" would match "abccab".

**Searching for position numbers**

- You can query particular corpus positions (token numbers) by [\#POSITION], e.g. to get the positions 100 and 210, you could do:

\[
[#100 | #210]
\]

- For a range of positions, use:

\[
[#100-210]
\]

- A negation can be achieved using the ! operator:

\[
[!#100-210]
\]

**Searching for structures**

- It is possible to use structures in your search. If \textit{s} is a valid structure in your corpus, then \textit{<s>} matches the beginning of the structure, \textit{</s>} matches its end and \textit{<s/>} matches the whole structure including all tokens inside it.

- In the same way positional attributes are included in the query, one can limit the search on particular structures by their structure attribute values. The following will find the beginnings of all documents with an id of 2011, where a proper noun (singular) must occur at the beginning of a sentence, followed by an arbitrary number of unspecified words, and finally followed by a verb.

\[
<\text{doc id="2011"> [tag="N.*"] []* [tag="VB.*"]</text{doc}>
\]

- The \textit{N-th} structure (in the order as appearing in the corpus) might be selected using the \textit{<doc \#N>} syntax, e.g. to retrieve the fifth document, one would use:

\[
<\text{doc \#5}>
\]

- The negation of the previous query ("all documents except for the fifth") is possible as well:
Advanced operators

Using *within* and *containing* operators

- If the corpus has sentence, paragraph or document markup, rather than constraining the match by specifying a number of tokens, we can specify it as within a unit (Here s for sentence.) We search for *confus* followed by *by* within a sentence with:

  "confus.*" []* "by" within <s/>

- A generalization of the previous example is "QUERY within QUERY" so that you can match e.g. all noun phrases within a sequence starting and ending with a verb:

  [tag="N.*"]+ within [tag="VB.*"] []* [tag="VB.*"]

Note that while the entire expression is matched, only the first query before *within* is highlighted in the concordance as the node or KWIC.

- As the counterpart to the *within* query, there is also a *containing* query with obvious semantics, you can e.g. match all sentences containing more than one noun:

  <s/> containing []* [tag="N.*"] []* [tag="N.*"] []*

- Similarly, you can generalize to "QUERY containing QUERY" and construct a query matching a sequence starting and ending with a verb and containing at least one noun:

  [tag="VB.*"] []* [tag="VB.*"] containing [tag="N.*"]

- Both of the *within/containing* queries support a shortcut of *within/containing NUMBER* which expands to *within/containing []{NUMBER}.*

- The *within* and *containing* operators might be mutually nested in an arbitrary way, making it possible to formulate complex queries like the following one which tries to look up particles:

  [tag="PR.*"] within [tag="V.*"] [tag="AT0"]? [tag="AJ0"]* [tag="(PR.?|N.*)"] [tag="PR.*"] within <s/>

- *within!* and *containing!* are negations of *within* and *containing* ("not within", "not containing").

Using the *within* operator in parallel corpora

- The *within* operator can also be used for querying aligned parts of parallel corpora (such as europarl5_de_en). Simply write a query of the following structure:

  <CQL query> within <aligned part identifier>: <CQL query for aligned part>

that will give you only those results of the first "CQL query", whose aligned part matches the second "CQL query".

For example:

  [word="car"] within europarl5_de: [word="Auto"]
will give you (more-or-less) occurrences of word “car” translated as “Auto” in German. (To be more precise, it returns those occurrences of word “car”, whose aligned part contains word “Auto”. This is not necessarily the translation of the first word, but mere random co-occurrence in the aligned part - unless the corpus is aligned word-to-word).

**Using meet and union operators**

- **meet** queries represent a specific type of contextual queries: let's say you want to match every noun which is surrounded by a verb in a -3/+3 context. You can achieve this using the following query:

  (meet [tag="N.*"] [tag="VB.*"] -3 3)

Only the first part ([tag="N.*"]) is highlighted as KWIC in the concordance, the [tag="VB.*"] is used as a contextual filter in the search

- **union** queries can be used to collect the results of meet queries. E.g. if you'd like to extend the previous example by all adjectives surrounded by a verb in -2/+2 context, you can do that in the following way:

  (union (meet [tag="N.*"] [tag="VB.*"] -3 3) (meet [tag="A.*"] [tag="VB.*"] -2 2))

- Both meet and union may occur wherever a positional attribute might be placed and can be combined with within/containing queries as demonstrated in the example below:

  containing (meet [lemma="have"] [tag="P.*"] -5 5) containing (meet [tag="N.*"] [lemma="blue"])

**Using swap and ccoll operators**

- The "swap" function swaps the KWIC with the selected collocations. The syntax is swap (<COLLNUM>, <ONEPOSITION>), e.g.:

  [swap (1, ws ("car", "modifier", "new"))]

  This will basically reverse the word sketch relation: the former KWIC becomes the first collocation and vice versa..

- A "ccoll" function relabels the given collocation. The syntax is: ccoll (<OLDCOLLNUM>, <NEWCOLLNUM>, <ONEPOSITION>), e.g.:

  [ccoll (1, 2, ws ("car", "modifier", "new"))]

  This relabels the first collocation as second.

  [ccoll (3, 1, ccoll (1, 3, ws(2, 6543)))]

  This relabels 1 to 3 and back: a no-op.

**Queries exploiting word sketches**

All the query types in this section assume that you are working with a corpus where word sketches are available.
Queries exploiting thesaurus

On corpora where the distributional thesaurus built on top of the word sketches is compiled, the ~ similarity operator can be used that searches for top N items similar to its argument. The syntax is as follows:

[WSATTR~NUMBER"word"]

where WSATTR is the positional attribute used in word sketches (mostly lempos or lemma). The result of such a query is "word" and top NUMBER items from thesaurus similar to "word".

A number of shorthands is available:

- [WSATTR~"word"] which has the same semantics as above with NUMBER defaulting to 10-base logarithm of the frequency of "word" in the corpus.

- just ~"word" or ~NUMBER"word" where the WSATTR is automatically selected.

Note that for obvious reasons regular expressions are not supported.

Example:

~"car-n" will match car (as noun) and top N items from thesaurus as described above provided that WSATTR is lempos.

Queries exploiting word sketch triples

You can lookup a particular word sketch concordance (i.e. concordance of headword-collocate occurrences for a particular grammatical relation) by the following:

[ws(headword,relation,collocation)]

Regular expressions can be used for all three parameters. Example:

[ws("test-n","object_of","conduct-v")]

This will retrieve a concordance of all test-n that are object_of conduct-v.

Queries exploiting word sketch seeks

Knowing a particular seek offset in the word sketch data files, the related concordance can be retrieved using:

[ws(level,seek)]

The level can be 0, 1 or 2 for the level of headwords, grammatical relations or collocations, respectively. The seek depends on particular corpus compilation, hence this kind of queries is mainly suitable for technical manipulation and combination of word sketch concordances.
Global conditions

- A global conditions part might be appended which postulates additional global constraints on positional attribute values. To make use of it, relevant positions must be prefixed by a numeric label, such as 1:[word="car"].
- Global conditions are introduced using the & operator and may occur only at the very end of the query.
- The example below would retrieve all neighbouring pairs of words with the same tag:

\[
1:[] \ 2:[] \ & \ 1.tag = 2.tag
\]

- A frequency function might be used to further limit the search:

\[
1:[] \ 2:[] \ & \ 1.tag = 2.tag & f(1.tag) > 1000
\]

Query examples

Look for...

- **thank** starting with either upper or lower case:

  "[tT]hank"

- A word beginning with **confuse**, followed by a preposition or a personal pronoun:

  "confuse.*" [tag="IN" | tag="PP"]
  "confuse.*" ([tag="IN"] | [tag="PP"])
  "confuse.*" [tag="IN|PP"]

  The three alternatives have the same effect, but are handled at a different level of evaluation: the first at the level of boolean expressions, the second at the level of attribute expressions, and the third at the level of regular expressions over the character alphabet.

- the same, but with at most 10 words in between:

  "confuse.*" [{0,10} [tag="IN" | tag="PP"]

- the same, but without full stops in between:

  "confuse.*" [word!="\."{0,10} [tag="IN" | tag="PP"]

  The backslash is needed to escape the dot, otherwise it will be treated as the matchall symbol of the regular expressions at the level of strings. If the backslash is omitted, all one-character tokens are excluded.

- a sequence of an adjective, a noun, a conjunction and another noun:

  [tag="JJ.*"] [tag="N.*"] "and|or" [tag="N.*"]

- a noun, followed by either **is** or **was**, followed by a verb ending in **ed**:

  [tag="N.*"] "is|was" [tag="V.*" & word=".*ed"]

- similar, but **is** or **was** followed by a past participle (which is described by a particular POS tag, VBD):

  [tag="N.*"] "is|was" [tag="VBD"]

- **catch** or **caught**, followed by a determiner, any number of adjectives and a noun, or a noun, followed by **was** or **were**, followed by **caught**:

  "catch|caught" [tag="DT"] [tag="JJ"]* [tag="N.*"] | [tag="N.*"] "was|were" "caught"
• **look** or **bring**, followed by either **up** or **down** with at most 10 non-verbs in between:

"look|bring" [tag !="VB.*"]{0,10} "up|down"