XML Technologies

XQuery Full Text

Christian Grün
Database & Information Systems Group
Universität Konstanz
Information Retrieval

- IR is about finding *information*, not *data*\(^1\)
- IR is focused on *full texts*, not the *document structure*

History

- **19th century**: creation of first printed *indices* for *library catalogs*
- **1960–1970**: Boolean queries, Vector Space Model
- **1970–1990**: digitalization of science and library data (OPACs)
- **1990–2000**: emergence of the web, first search engines
- **since 2000**: Web 2.0, semantic search, social networks
- **Today**: online IR has become *the standard approach* to find information

---

\(^1\) Ricardo Baeza-Yates and Berthier Ribeiro-Neto. Modern Information Retrieval, 1999
Information Retrieval

Back then...

Today...

...name five different search engines!

- However, the number of online catalogs has never been bigger
XQuery: String Search

- XQuery provides string functions that can be used for full-text requests
- Examples:
  
  declare variable $text := "Find a or B";
  contains($text, "a") or contains($text, "B"),
  matches($text, "a|B"),
  matches($text, "a|b", "i"),
  matches($text, "\Wa\W|\Wb\W", "i"),
  matches($text, "(^|\W)\a(\$|\W)|(^|\W)\b(\$|\W)", "i"),
  analyze-string($text, "(^|\W)\a(\$|\W)|(^|\W)\b(\$|\W)", "i"")

What is happening here? Which problems can you spot?

Solution in XQuery Full Text: $text contains text { "a","b" }
XQuery Full Text

- brings IR and XML worlds together
- enables query writers to search *full texts in structured* text data
- support for *arbitrary languages*
- basic syntax: `...text... contains text ...query...`

**Processing steps**

1. *evaluate* text and query expressions
2. *tokenize* resulting strings
3. *match* query strings against input text
4. compute *score value* (double), return boolean value
Tokenization

- a string is split into *normalized units*, which can be better compared (words, phrases, symbols, ...)
- example: "Was wünschen Sie?" ⇒ "was" "wunsch" "sie"
- whitespaces serve as delimiters; special characters are removed, words are *stemmed*, case is changed to *lower case*

**Challenge: what are good tokens?**

- how to handle fixed terms, hyphens, etc? U.S.A., aujourd'hui, 3.14159265358979...
- what about compounds? Donaudampfschifffahrtsgesellschaftskapitän²
- what about East Asian texts? それが困難である！

² [de.wikipedia.org/wiki/Donaudampfschifffahrtsgesellschaftskapitän](de.wikipedia.org/wiki/Donaudampfschifffahrtsgesellschaftskapitän)
Tokenization Techniques

Case folding

- reduce all letters to (usually) lower case
- in most cases, upper and lower case can be ignored
- exceptions prove the rule: AIDS vs. aids; SAP vs. sap
- language-specific issues, e.g. Turkish dotless: KAPISI ⇔ Kapısi
- many languages don’t have upper and lower case (...makes it easier)

Case folding in XQuery

- by default, case will always be normalized
- normalization can be suppressed:
  "US govt" contains text "us" using case sensitive
Diacritics

- A diacritical mark is a glyph added to a letter; e.g. ʻ ـ " ́ ̀ ʻ .
- English has only a few diacritical words (mostly loanwords), e.g. naïve
- Found more often in other languages, e.g. ačiū!, façade, mañana
- Also used in non-Western texts, e.g. край, القرآن, です
- Normalization may increase number of characters: München ⇒ Muenchen

**Diacritics in XQuery**

- By default, diacritical characters will always be normalized
- Once again, normalization can be suppressed:
  "città" contains text "citta" using diacritics sensitive
Stop words

- mostly used to reduce size of index structures
- usual stop words: articles, prepositions, conjunctions: to, and, the, of, ...
- essential in the past, but more and more obsolete  
  (most current search engines index as much as possible)
- counterproductive: some search phrases mainly consists of stop words
- prominent examples: “To be, or not to be”, “Let it be”

Stop words in XQuery

- by default, no stop words are removed
- stop words can be requested in the query or via a URI:
  "one and two" contains text "one or two"  
  using stop words at "stopwords.txt"
Stemming

- inflected/derived words are reduced to their stem, base or root form
- a stem needs not be the same as the morphological root of a word
  example: fish, fishing ⇒ fish, but: argue, arguing ⇒ argu
- stemming is highly dependent on text language
- composed terms are challenging, e.g. "ich kaufe ein" ⇒ "ich" "einkaufen"?

Approaches

- lookup tables: based on dictionaries
- stochastic model: trained from an initial set of root forms
- lemmatization: determine part of speech, apply normalization rules
- suffix-stripping: progressively removes usual word suffixes
Porter Stemmer (1980)

- prominent suffix-stripping algorithm for English texts
- five rule sets are applied consecutively to an input term
- the first rule is demonstrated in more detail:

1A: "sses$" $\Rightarrow$ "ss"  
    "ies$" $\Rightarrow$ "i"  
    "[^s]s$" $\Rightarrow$ ""  
    "caress" $\Rightarrow$ "caress", "cats" $\Rightarrow$ "cat"

1B: only applied if prefix contains at least one vowel and consonant:
    "eed$" $\Rightarrow$ "ee"  
    "ed$" $\Rightarrow$ ""  
    "ing$" $\Rightarrow$ ""  
    "plastered" $\Rightarrow$ "plaster", "bled" $\Rightarrow$ "bled"

3 tartarus.org/~martin/PorterStemmer
Porter Stemmer (cont.)

1B if the second or third rule of 1B applies, the following is done:
   "at$" ⇔ "ate"  conflat(ed) ⇔ conflate
   "bl$" ⇔ "ble"  troubl(ed) ⇔ trouble
   "iz$" ⇔ "ize"  siz(ed) ⇔ size

if string before suffix contains at least one vowel and consonant:
   "eed$" ⇔ "ee"  feed ⇔ feed, agreed ⇔ agree

if string ends with double consonant other than LL, SS, ZZ:
   "(.)\1$" ⇔ ""  hopp(ing) ⇔ hop, fall(ing) ⇔ fall

if string before suffix contains one vowel enclosed by consonants,
and does not ends with W, X, or Y:
   "ing$" ⇔ "e"  fail(ing) ⇔ fail, fil(ing) ⇔ file

1C if string contains a vowel:
   "y$" ⇔ "i"  happy ⇔ happi, sky ⇔ ski
Stemming in XQuery

- stemming can be activated as follows:
  "AIDS" contains text "aid" using stemming
- an additional option exists to specify the used language:
  "Häuser" contains text "Haus" using language 'de' using stemming using diacritics sensitive
- all full text options can be globally specified in the query prolog
- BaseX provides a function `ft:tokenize(...)` for tokenizing strings\(^4\):
  declare ft-option using stemming;
  ft:tokenize("...here we see how a text is stemmed!")

\(^4\) docs.basex.org/wiki/Full-Text_Module
Stemming in XQuery, more examples

What does the following query do?

```xquery
declare ft-option using stemming;
let $words := ("computer", "computational", "computation", "compute")
let $stem := "comput"
return every $word in $words
  satisfies ft:tokenize($word) = $stem
```

- shorter version:
  ```xquery
  not(ft:tokenize("computer computational computation compute",
                  map { 'stemming': true() }) != "comput")
  ```

- stem input in different languages:
  ```xquery
  for $ln in ('German', 'English', 'Italian')
  return ft:tokenize("computers",
                     map { 'stemming': true(), 'language': $ln })
  ```
Thesaurus

- groups words together according to their similarity of meaning
- different fields of interests imply different semantics (e.g. pool, board)
- many special-purpose thesauri exist

Thesaurus in XQuery

- attach specified thesaurus and find synonymy (UF = used for):
  "word" contains text "term" using thesaurus at "thesaurus.xml" relationship "UF"
- use default thesaurus (if available) and find narrow term (NT):
  "users" contains text "people" using thesaurus default relationship 'NT' at most 2 levels
- most thesaurus specifics are implementation-defined
Matching Algorithms

Wildcard search

- Example: "jabadoo" contains text "jaba.*" using wildcards
- Allowed modifiers:
  . arbitrary character
  .? zero or one character
  .* zero or more characters
  .+ one or more characters
  .{\d+, \d+} single period, matching a single arbitrary character

❓ What could be a (technical) reason for the limited set of modifiers?
❓ Which index structures are (not) suitable for wildcard searches?
Approximate search

- added to BaseX (not part of the specification)
- based on the Damerau-Levenshtein distance (using Ukkonen cut-off)
- can be used instead of wildcard search (combination is not allowed)

Examples

- find texts similar to "Stanley Kubrick"
  //*[text() contains text "stanly kubrik" using fuzzy]
- only accept strings with a maximum of 2 errors
  (# db:lserror 2 #) {
    "Rhythmus" contains text "ritmus" using fuzzy
  }
Levenshtein Distance

- string metric to measure the difference between two character sequences
- edit distance \((k)\): integer value, representing the number of single-character edits required to transform one string into another
- supported edit operations:
  - deletion: hand \(\Rightarrow\) han
  - insertion: hand \(\Rightarrow\) handy
  - substitution: sand \(\Rightarrow\) hand
- additional edit operation introduced by Damerau distance:
  - transposition: hadn \(\Rightarrow\) hand

Compute the edit distance of the following three strings:
wunderbar \(\Rightarrow\) sonderbar, hundert, allerhand
Damerau-Levenshtein Distance

- costs for computing distance: $O(m \cdot n)$
- optimized (using Ukkonen cut-off): $O(k \cdot \min(m,n))$
- formal definition:

$$D_{ij} = \min \left( X_i + 1, X_d + 1, X_s + C_{\text{substitution}}, X_t + C_{\text{transposition}} \right)$$

$$C_{\text{substitution}} = \begin{cases} 
1, & \text{if } S_1[i] \neq S_2[j] \\
0, & \text{else}
\end{cases}$$

$$C_{\text{transposition}} = \begin{cases} 
1, & \text{if } S_1[i] = S_2[j-1] \land S_1[i-1] = S_2[j] \\
\infty, & \text{else}
\end{cases}$$

Definition and example have been adopted from ntz-develop.blogspot.de/2011/03/fuzzy-string-search.html
Scoring

• in contrast to exact search, we often need to rate the relevance of single query results
• factors that may influence the relevance of a hit:
  • number of occurrences of a search string
  • its position in the text
  • quality of a text, based on statistical or subjective ratings
  • number of other texts referencing a text
• texts can even be relevant without containing the search terms:
  • think of: typing errors, the usage of synonyms, or related terms
  • a text may be relevant if it’s referenced by many other texts that contain the search string
Precision & Recall

- **Precision**: percentage of returned texts that are *relevant to the user*
- **Recall**: percentage of *relevant texts* that are returned
- the image shows the following:
  - left region: *relevant texts*
  - oval: *retrieved texts*
  - left, red region: *relevant, but not retrieved texts*
  - red region in the oval: *returned, but not relevant texts*

⚠️ What are the most desirable values for Precision and Recall?

The image has been taken from: en.wikipedia.org/wiki/Precision_and_recall
**TF-IDF measure**

- most popular ranking mechanism for document collections
- indicates how relevant a single term is to a document in a collection
- the Term Frequency (TF) reports how often a term occurs in a document
  - \( f(t, d) = n \) means: term \( t \) occurs \( n \) times in document \( d \)
  - normalized frequency, including max. number of any term \( w \) in a document \( d \):
    \[
    \text{tf}(t, d) = \frac{f(t,d)}{\max \{ f(w, d) : w \in d \}}
    \]
- the Inverse Document Frequency (IDF) indicates how common or rare a single term \( t \) is across all documents \( D \):
  - \( \text{idf}(t, D) = \log \left( \frac{|D|}{1 + \{ d \in D : t \in d \}} \right) \)
  - \( \{ d \in D : t \in d \} \) means: number of documents \( d \) that contain term \( t \)
- problem: IDF needs to be updated whenever collection changes
Scoring in XML documents

- traditional IR was designed for *plain, unstructured full texts*
- semistructured full texts in XML are *structured by elements*
- the structure of a document can be helpful to *improve scoring*
- if we have no meta information on an XML document, we can take advantage of *statistical information* on text nodes, such as

- their **string length**: short texts may be more relevant
- their **depth**: text nodes on higher documents levels could be ranked higher
- their **position**: assign higher ranking to nodes with few preceding siblings
- advantage: works with all XML documents
Scoring in XQuery

- XQuery scores are *double values*, ranging from 0-1
- they will be *implicitly bound* to items resulting from full-text operations
- score variables can be added to `let` and `for` clauses of FLWOR expressions

**Examples**

- Returns a single score value resulting from a full-text operation:
  ```xml
  let score $s := 'A B C' contains text 'A'
  return 'Score: ' || $s
  ```
- Sorts books by a score value and returns their titles:
  ```xml
  for $t score $s in //book[text contains text 'Prozess']
  order by $s descending
  return $t/title
  ```
Advanced Features

Cardinality

- number of matches of a term may be controlled via cardinality selections
- syntax:
  occurs [exactly|at least|at most]... times
  occurs from ... to ... times

Logical operators

- results may be combined via ftand, ftor, ftnot and not in
- ... not in ... (mild not) returns all hits that match A unless it is a part of B
- Example:
  'New Orleans' contains text 'new' not in 'new york'
Positional filters

Filters matches based on their position in a string:

- **ordered**: matches if terms are aligned in ascending order: 'a b c' contains text 'c' ft and 'b' ordered
- **at start, at end, entire content**: match terms at specified position
- **[same|different][sentence|paragraph]**:
  - specified terms must all be located in the same or a different sentence/paragraph
  - the notion of *sentences* and *paragraphs* is implementation-defined
- **distance [exactly|at least|at most]... [words|sentences|paragraphs]**
  - find words in a specified distance or within a specific window
Containment

The specified query inputs may be handled in different ways:

- **any**: at least one of the specified inputs must be found (*default*):
  \[ 'a \ b' \text{ contains text } \{ 'c', 'b' \} \text{ any} \]

- **all**: all specified inputs must be found:
  \[ 'a \ b' \text{ contains text } \{ 'b', 'a' \} \text{ all} \]

- **any word**: all inputs are tokenized, and at least one token must be found:
  \[ 'a \ b' \text{ contains text } \{ 'c \ b' \} \text{ any word} \]

- **all words**: all inputs are tokenized, and all resulting tokens must be found:
  \[ 'a \ b' \text{ contains text } \{ 'b \ a' \} \text{ all words} \]

- **phrase**: the specified inputs are joined and handled as one string:
  \[ 'a \ b' \text{ contains text } \{ 'a', 'b' \} \text{ phrase} \]

What Boolean values are returned by the queries?
Semistructured Data

- full-text querying poses additional challenges when *mixed content* is to be queried
- this is particularly true if we have no *meta information* on the input:
  - which elements contain *structural contents*?
  - which XML fragments are *semistructured*?
- if text extends over *multiple elements*, an implementation must decide to either respect or ignore *element boundaries*

Do you see the challenge?

- `<div><b>No</b> Way!</div>` contains text "No"
- `<xml><line>he comes</line><line>soon.</line></xml>` contains text "comes soon"
XQuery Full Text in BaseX

Scoring

- support for TF/IDF was discarded in favor of local scores
- currently, only the number of tokens in a text is considered when computing score values
- the function `ft:score()` returns the score value of an item:

  ```xml
  for $t in ('A', 'A A', 'A B')
  return ft:score($t contains text 'A')
  ```

Counting results

- the function `ft:count()` indicates how often a term has been found:

  ```xml
  ft:count(db:open('DB')//*[text() contains text 'abc']])
  ```
Highlighting results

- `ft:mark()` puts new marker elements around the found terms in the text nodes of a full-text index request.
  - example query, using `<b/>` as marker element:
    ```xml
    ft:mark(
      db:open('DB')/*[text() contains text { 'one','two' }], 'b'
    )
    ```
  - possible result:
    ```xml
    <text>There may be <b>one</b> or <b>two</b> solutions</text>
    ```
- `ft:extract()` works as `ft:mark()`, but it additionally extracts and returns relevant parts of text nodes:
  - possible result: 
    ```xml
    <text>... be <b>one</b> or <b>two</b> solutions ...</text>
    ```

Find more details at: docs.basex.org/wiki/Full-Text_Module
Preview: Implementation

- efficient full-text search is only possible if texts are \textit{indexed}
- an \textit{inverted index} is used to find matching text nodes:

\texttt{<A>
  <B>xw xy</B>
  <C>
    <D>x</D>
    <D>xy xw</D>
    <D>x y</D>
  </C>
  <E>y x</E>
</A>}

\texttt{\begin{tabular}{|r|c|c|}
\hline
pre & dist & size & data \\
\hline
1 & 1 & 11 & A \\
2 & 1 & 1 & B \\
3 & 1 & 0 & xw xy \\
4 & 3 & 6 & C \\
5 & 1 & 1 & D \\
6 & 1 & 0 & x \\
7 & 3 & 1 & D \\
8 & 1 & 0 & xy xw \\
9 & 5 & 1 & D \\
10 & 1 & 0 & x y \\
11 & 10 & 1 & E \\
12 & 1 & 0 & y x \\
\hline
\end{tabular}}

Depicted above: XML document, tree representation, table mapping, trie index

(Compressed) \textbf{Tries} are popular full-text index structures
Index-supported XQuery expressions

The optimizer will do its best to rewrite a full-text query for index access:

Original query plan

XPath, shown on the left:

/A/C[D/text() contains text "x"]

Optimized query plan

Equivalent XQuery expression:

for $c in /A/C
where $c/D/text() contains text "x"
return $c