Modern Information Retrieval

Chapter 13
Structured Text Retrieval
with Mounia Lalmas

Introduction

- Text documents often contain structural information
- Structural information can be exploited at several stages of the information retrieval process
    - Indexing stage
    - Retrieval stage
    - Result presentation stage
    - Querying stage

Structuring Power

Structuring Power

- Until the 1990s, various structured text retrieval models appeared in the literature
- They comprise three main parts:
  - a model of the text, which specifies the character set, synonyms, stop words, stemming;
  - a model of the structure, which specifies the markup language, the index structure, the type of structuring; and
  - a query language, which specifies what can be asked, and what the answers are

Structuring Power

- We contrast the structuring power of structured text retrieval models according to three main aspects:
    - explicit vs. implicit structure
    - static vs. dynamic structure
    - single vs. multiple hierarchical structure

Explicit vs. Implicit Structure

- Most structured text retrieval models are based on the explicit structure of the documents
    - They work on the basis that the documents are composed by sections, chapters, titles, and so on
- The structure is usually provided through the use of a markup language
- For example,

```
section CONTAINS "red wine"
```

- Will return all sections that contain the sentence “red wine”

Explicit vs. Implicit Structure

- **Implicit structure**: the structure of the document is not explicitly distinguished from its text content
  - Documents are modeled as sequences of tokens without distinguishing a word token from a markup token
- A structural element is therefore constructed at querying time
- An example of a query that refers to an implicit structure is as follows:

```
("<section>" FOLLOWING "</section>")
CONTAINS "red wine"
```

- The `section` element only exists at querying time

Static vs. Dynamic Structure

- Some text retrieval models allow the specification of dynamic structures in the query
- This allows the systems to return elements that have not been explicitly marked-up in the the documents
  - In XQuery and XQuery Full-Text, this is done by element construction
  - In other models, dynamic structure is a natural part of the model

Static vs. Dynamic Structure

- Consider the following structured text document:

```
SPIRE, “Patagonia, Chile”, 2001, The conference
....
```

- Let us assume that the above document is explicitly structured by the following grammar that acts as a document schema:

```
entry := conference ’,’ area ’,’ year ’,’ content’.’
conference := text ;
area := ’“’ text ’”’ ;
year := digit digit digit digit ;
content := text ;
text := ( letter | ’ ’ )+ ;
```

Static vs. Dynamic Structure

- Every document instance must conform to the grammar
  - The instance takes the form of a parsed string called “p-string”
- With this schema, the area symbol does not distinguish the country “Chile” and the region “Patagonia"
- This distinction can be done at query time by introducing a small grammar fragment:

```
AreaG  :=  { area      :=  ( region ’ , ’ )+ country ;
             country   :=  letter + ;
             region    :=  letter + ; }
```

Static vs. Dynamic Structure

- The p-strings model provides a simple query language for adding additional grammar fragments
- Given the document d_j, the following query returns a p-string containing the area element with a given country and region explicitly identified:

  `(area in` d_j`) reparsed by AreaG`

Single vs. Multiple Hierarchies

- The type of structure most used with structured text retrieval models is a hierarchical organization
- Text retrieval models using implicit structure assume, for simplicity, a single hierarchy
- Approaches based on explicit structure assume that multiple structural hierarchies are present on the same document
- Querying with these models has only been possible with respect to a single hierarchy
  - It is not possible to query mixing hierarchies because many undefined cases appear
- That is, queries must be resolved in one hierarchy and then projected to another hierarchy

Single vs. Multiple Hierarchies

- This has changed with the work of Alink *et al*
  - In this work, additional XPath based navigational steps have been introduced to allow moving from one hierarchy to another
- Example of query:

```
$doc//paragraph[
    ./select-narrow::Verb CONTAINS
"hiking" and
    ./select-narrow::Region CONTAINS
"Patagonia"
]
```

Early Text Retrieval Models

Early Text Retrieval Models

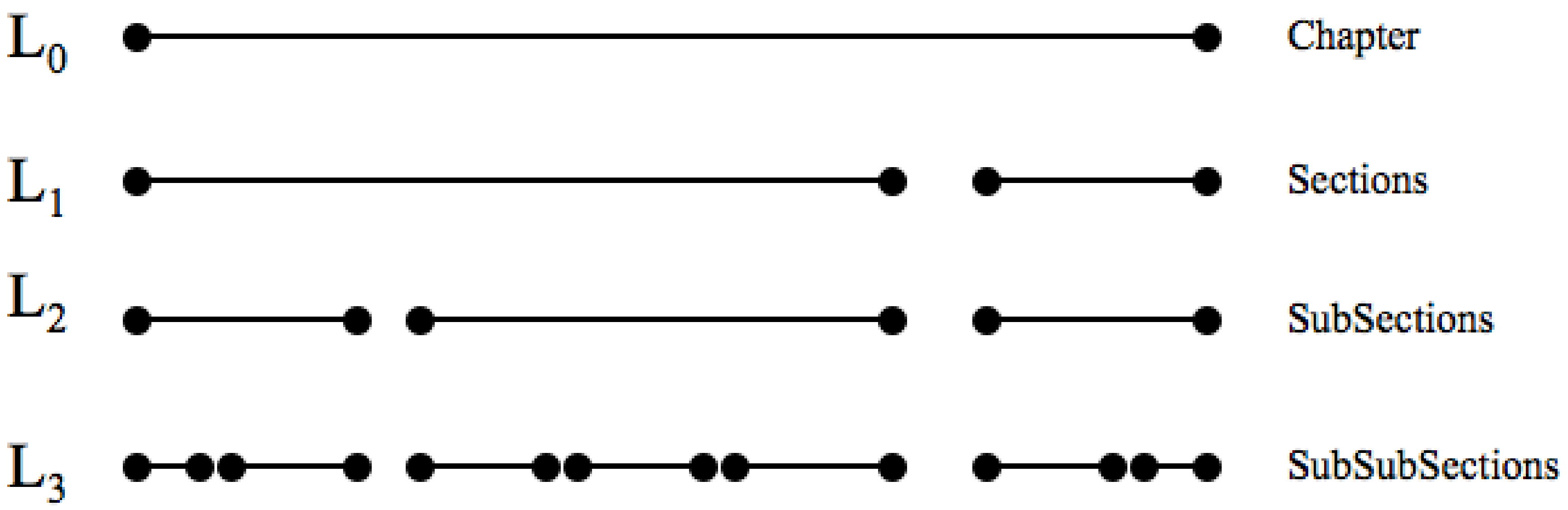
- We discuss now two early structured text retrieval models
- We use:
  - The term **match point** to refer to the position in the text of a word which matches the user query
  - The term **region** to refer to a contiguous portion of the text
  - The term **node** to refer to a structural component of the document

Non-Overlapping Lists

- Burkowski divided the whole text of each document into non-overlapping text regions collected in a list
- Since there are multiple ways to divide a text in non-overlapping regions, multiple lists are generated
- Text regions from distinct lists might overlap

Non-Overlapping Lists

- Example of the representation of the structure in the text of a document through four separate (flat) indexing lists



Non-Overlapping Lists

- A single inverted index is built in which each structural component stands as an entry in the index
- Associated with each entry, there is a list of text regions as a list of occurrences
- Moreover, such list could be easily merged with the traditional inverted index for the words in the text
- Since the text regions are non-overlapping, the types of queries which can be asked are simple:
    - select a region that contains a given word (and does not contain other regions)
    - select a region A that does not contain any other region B (where B belongs to a list distinct from the list for A)
    - select a region not contained within any other region

Proximal Nodes Model

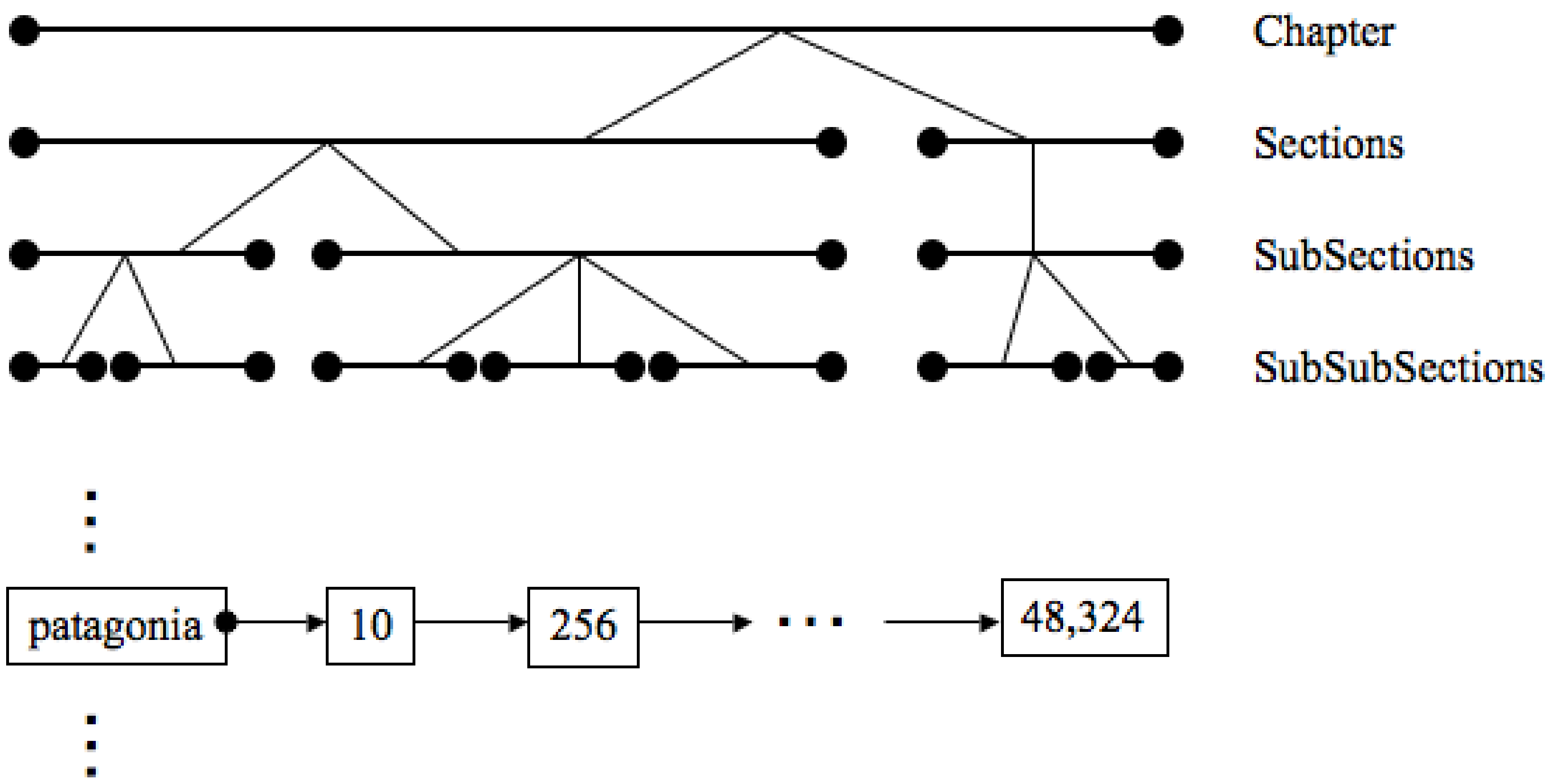
- Baeza-Yates and Navarro proposed a model based on proximal nodes
- This model allows defining independent hierarchical indexing structures over the same document text
- These indexing structures are strict hierarchies composed of chapters, sections, paragraphs, pages, and lines, which are called nodes
- Two distinct hierarchies might refer to overlapping text regions

Proximal Nodes Model

- Given a user query, the compiled answer is formed by nodes that all come from only one of the hierarchies
    - This is to allow for faster query processing at the expense of less expressiveness
- Nested text regions are allowed in the answer set

Proximal Nodes Model

- They use a hierarchical index for structural components and a flat index for words:



Proximal Nodes Model

- The query language allows:
  - the specification of regular expressions
  - the reference to structural components by name
  - the combination of these
- In this sense, the model can be viewed as a compromise between expressiveness and efficiency

Proximal Nodes Model

- Consider, for instance, the query

```
    (*section)  with ("Patagonia")
```

- It searches for sections, subsections, or subsubsections which contain the word “Patagonia”
- A simple query processing strategy is:
    - to traverse the inverted list for the term “Patagonia”
    - for each entry in the list, search the hierarchical index looking for sections, subsections, and subsubsections containing that occurrence of the term

Proximal Nodes Model

- A more sophisticated query processing strategy is as follows:
  - For the first entry in the list for “Patagonia”, search the hierarchical index as before
  - Verify whether the innermost matching component also matches the second entry in the list
  - If it does, we immediately conclude that the larger structural components above it (in the hierarchy) also do
  - Proceed then to the third entry in the list, and so on

Proximal Nodes Model

- The query processing is accelerated because only the nearby nodes need to be searched at each time
  - This is the reason for the term “proximal nodes”
- This model allows formulating queries that are more complex than those based on non-overlapping lists
- To speed up query processing, however, only nearby nodes are looked at, which imposes restrictions on the answer set retrieved

XML Retrieval

XML Retrieval

- Nowadays, XML retrieval is almost a synonym for structured text retrieval
- INEX provided test sets and a forum for the evaluation and comparison of XML retrieval approaches

Challenges in XML Retrieval

- XML Retrieval Task: to exploit the structure of XML documents to select the best document components
- These answers should be ranked according to their likelihood of relevance to the queries
- Classic information retrieval models make use of term statistics, such as TF and IDF, to rank documents
- Indexing algorithms for XML retrieval require similar terms statistics, but at the element level
- One could simply replace document by element and define:
    - within-element term frequency, ETF
    - inverse element frequency, IEF

Challenges in XML Retrieval

- Not all element types will satisfy the users when returned as answers to queries
  - Some elements may be too small, or be of a type that does not contain informative text
- The challenges are:
  - to determine what are the best categories of elements to return as answers to a query, and
  - how to use this information to rank elements

Challenges in XML Retrieval

- XML documents are not just documents composed of elements of various types and sizes
- There are also relationships among the elements, as provided by the logical structure of the XML markup
- These relationships among elements can be used to improve XML retrieval
- For instance, consider a collection of scientific articles
- It is reasonable to assume that the “abstract" is a better indicator of what the article is about than a section on “future work"

Challenges in XML Retrieval

- Two distinct tasks:
  - To provide a score expressing how relevant an element is to a query
  - To decide, from several overlapping relevant elements, which one to return as the best answer
- This is because of the nested nature of XML documents
  - If an element has been estimated relevant to a query, it is likely that its parent element will also be estimated relevant to the query
- However, returning a paragraph and its enclosing section should be avoided

Challenges in XML Retrieval

- The final challenge is how to interpret structural constraints
- Early work in XML retrieval required query constraints to be strictly matched by the results returned
- However, specifying structural constraints in the query is not an easy task
- Users may not have a clear idea of the structural nature of the searched collection

Indexing Strategies

- In XML retrieval, in contrast to retrieval of “flat” documents, there are no *a priori* fixed retrieval units
- The simplest approach to allow the retrieval of elements at any level of granularity is to index all elements
  - Each element thus corresponds to a document, and conventional information retrieval indexing techniques can be used
- Term statistics are calculated based on the concatenation of the text of the element and that of its descendants

Indexing Strategies

- With respect to the calculation of IEF, the previous approach ignores the issue of nested elements
- Alternatively, IEF can be estimated across elements of the same type or across documents
  - The former greatly reduces the impact of nested elements on the IEF value of a term, but does not eliminate it as elements of the same type can be nested within each other
  - The latter is the same as using inverse document frequency, which completely eliminates nested elements

Indexing Strategies

- We can aggregate the term statistics of an element with the statistics of each of its children elements
  - This overcomes the issue of calculating IEF across nested elements
- Aggregated-based ranking uses the aggregated representation of elements to rank elements
- An alternative approach is to only index leaf elements
  - This implies that term statistics will only be calculated for leaf elements, which can then be used to rank the leaf elements themselves
- This also overcomes the issue of calculating IEF across nested elements

Indexing Strategies

- It has also been common to discard elements smaller than a given threshold
  - They are often considered not meaningful retrieval units
- However, they should still be indexed, in particular when a propagation mechanism for scoring is used

Indexing Strategies

- In Mass and Mandelbrod, a separate index is built for each selected element type
  - For a collection of scientific articles, for instance, these types may include article, abstract, section, subsection, paragraph
- The statistics for each index are then calculated separately
- In this case, the term statistics are likely to be more uniform and consistent
- Further, this approach greatly reduces the term statistics issue arising from nested elements

Indexing Strategies

- It is not yet clear which indexing strategy is the best, as which approach to follow would depend on the collection
- In addition, the choice of the indexing strategy has an effect on the ranking strategy

Ranking Strategies

- Many of the retrieval models for plain text documents have been adapted to XML retrieval
- These models have been used to estimate the relevance of an element based on its content
- However, to use evidence coming from the context of the element increases retrieval performance
- Depending on the indexing strategy, specific strategies are needed to rank elements at all levels of granularity
- Finally, structural constraints must be processed to provide results that satisfy the structural criteria of a query

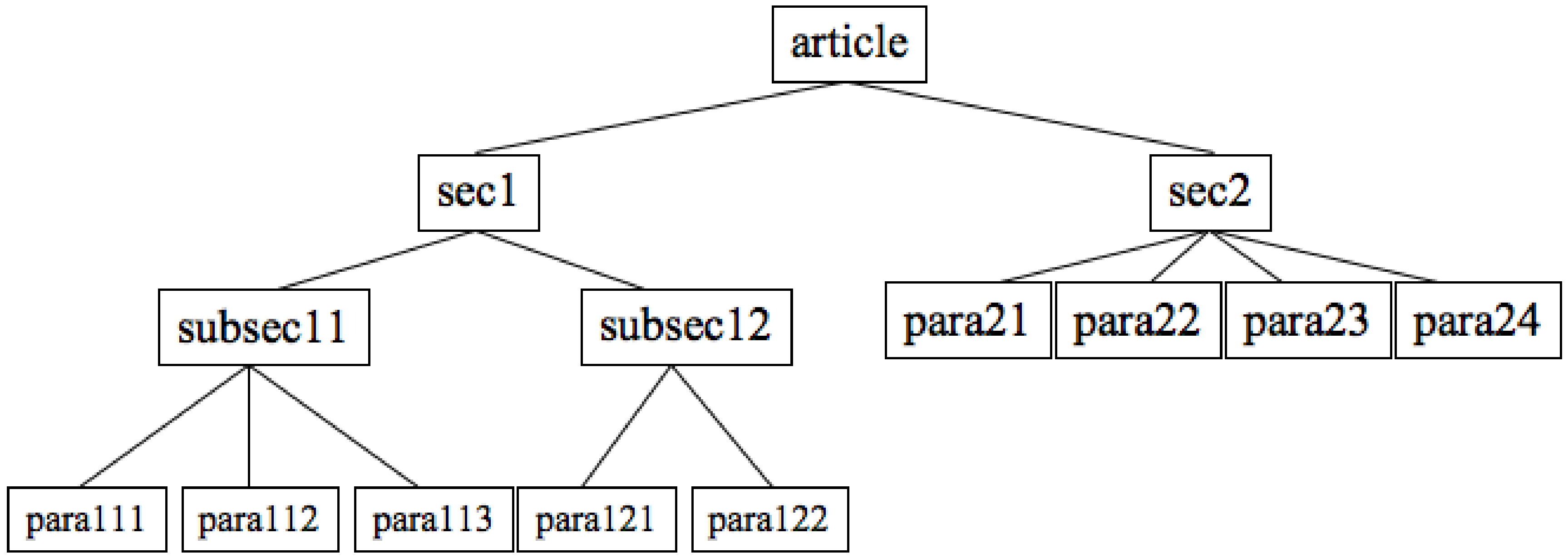
Ranking Strategies

- For illustration purpose, we will use the sample XML document shown below

```
<article>
<sec>
<subsec>
<p> ...  wine ...  patagonia ...  </p>
<p> ...  wine ...  </p>
<p> ...  patagonia ...  </p>
</subsec>
<subsec>
<p> ...  </p>
<p> ...  </p>
</subsec>
</sec>
<sec>
<p> ...  </p>
<p> ...  wine ...  </p>
<p> ...  </p>
<p> ...  </p>
</sec>
</article>
```

Ranking Strategies

- The tree structure of our sample XML document



Element Scoring

- All ranking strategies require a scoring function that estimates the relevance of an element for a given query
- With the propagation strategy, the scoring function is applied to the leaf elements only
- In other cases, it is applied to all the retrievable elements
- The scoring function is usually based on standard IR models

Element Scoring

- To illustrate, we describe how XML-specific features can be incorporated in a language modeling framework
- Given
  - a query $q = (k_1, k_2, ..., k_n)$
  - an element e and its corresponding element language model M_e
- The elements are ranked in decreasing order of $P(e|q)$ as follows:

$$P(e|q) \propto P(e)P(q|M_e)$$

  where

  - $P(e)$ is the prior probability of relevance for element e and
  - $P(q|M_e)$ is the probability of a query being generated by the element language model M_e

Element Scoring

- Using a multinomial language model, for instance, $P(q|M_e)$ can be calculated as:

$$P(q|M_e) = \prod_{k_i \in q} P(k_i|M_e, \lambda)$$

- If the term probabilities are computed based on Jelinek-Mercer smoothing, we obtain

$$P(k_i|M_e, \lambda) = \lambda P(k_i|e) + (1 - \lambda) P(k_i|C)$$

where

  - $P(k_i|e)$ is the probability of query term k_i in element e
  - $P(k_i|C)$ is the probability of query term k_i in the collection
  - λ is the smoothing parameter

Element Scoring

- $P(k_i|e)$ is the element model based on element term frequency, ETF
- $P(k_i|C)$ is the collection model based for instance on inverse element frequency, IEF

Element Scoring

- Statistics for our sample XML document ($\lambda = 0.8$)

Elements	$P(\text{wine}\|e)$	$P(\text{wine}\|M_e)$	$P(\text{patag....}\|e)$	$P(\text{patag....}\|M_e)$	$length(e)$
para111	0.2	0.180	0.333	0.327	15
para112	0.6	0.500	0	0.060	10
para113	0	0.020	0.25	0.260	12
para121	0	0	0	0	8
para122	0	0	0	0	10
para21	0	0	0	0	20
para22	0.5	0.420	0	0.060	14
para23	0	0	0	0	10
para24	0	0	0	0	18
subsec11	0.243	0.215	0.216	0.233	37
subsec12	0	0	0	0	18
sec1	0.155	0.144	0.138	0.170	58
sec2	0.113	0.110	0	0.060	62
article	0.127	0.122	0.063	0.111	126
$P(k_i\|C)$	0.1	$\lambda = 0.8$	0.3	$\lambda = 0.8$	

Element Scoring

- The term statistics are based on the content of the elements and of their descendants
- $length(e)$ is the number of terms in e
- The length of an inner element is the sum of the length of each of its children elements
- Elements with no query terms are ignored

Element Scoring

- The ranking of the elements forming our sample document is given below

```
0.0588 para111
0.0500 subsec11
0.0300 para112
0.0252 para22
0.0246 sec1
0.0135 article
0.0066 sec2
0.0052 para113
```

Element Scoring

- Here, we have assumed a constant prior relevance probability, so $P(e)$ was ignored
- However, with the above language modeling framework, XML-specific features can be incorporated through $P(e)$
- For instance, a bias towards long elements can be incorporated by setting:

$$P(e) = \frac{length(e)}{\sum_{e'} length(e')}$$

Element Scoring

- Other XML-specific features that can be incorporated into a language modeling framework include:
    - the number of topic shifts in an element
    - the path length of an element
- Overall, accounting for element length is crucial in XML retrieval
- Other XML-specific features can help, depending on the retrieval task

Contextualization

- The context of an element can provide valuable evidence on what an element is or is not about
- This is because all the terms in the document can be used to score the element for a given query
  - For instance, the fact that an element does not contain all query terms, but is contained in a document that contains all query terms, is likely to be of importance when assessing relevance
- This strategy can be implemented by combining the element score with the document score
- The process of combining the score of the element and that of its context is referred to as contextualization

Contextualization

- A contextualization technique is to use the document containing the root element as context
- This means combining the score of the element to that of the XML document containing that element
- The combination can be as simple as the average of the two scores
- A scaling factor can be used to emphasize the importance of one score compared to the other

Contextualization

- Some or all ancestors of an element can also be used as context
  - For instance, the parent element alone can be used as context
- Using the context is nothing else than capturing relationships between elements in XML retrieval
- Properly accounting for these relationships consistently improves retrieval performance

Propagation

- A propagation strategy is required when the indexing strategy indexes only leaf elements
- The retrieval score of an inner element is calculated on the basis of the scores of its descendant elements
- Let e be a non-leaf (inner) element, e_ℓ a leaf element contained in e and q a query
- Let $score(.)$ be the scoring function used to rank elements, then
  - $score(e_\ell, q)$ is calculated directly from the index of e_ℓ
  - $score(e, q)$ is calculated by a propagation mechanism

Propagation

- The most common propagation mechanism consists of a weighted sum of the retrieval scores
- The number of children elements of an element can be used as a weight
- For instance, in the GPX approach, $score(e, q)$ is calculated as follows:

$$score(e, q) = D(m) \times \sum_{e_c} score(e_c, q)$$

  where

  - e_c is a child element of e
  - m is the number of retrieved children elements of e

Propagation

- $D(m) = 0.49$ if $m = 1$ (e has only one retrieved child element) and 0.99, otherwise
- The value of $D(m)$, called the decay factor, depends on the number of retrieved children elements
- If e has one retrieved child:
  - the decay factor of 0.49 means that an element with only one retrieved child will be ranked lower than its child
- If e has several retrieved children:
  - the decay factor of 0.99 means that an element with many retrieved children will be ranked higher than its children

Propagation

- We use the GPX technique to illustrate the propagation strategy applied to our sample XML document
- We take the scores of the leaf elements produced by the element-only scoring strategy
- As `subsec11` has three children elements, then $D(3) = 0.99$ since $m = 3$
- Thus $score($ `subsec11` $, \{$wine, patagonia$\}) =$
$$0.99 \times (0.0588 + 0.0300 + 0.0052) = 0.0931$$

Propagation

- Ranking using the GPX propagation strategy, showing the damping factors $D(m)$ between ( ), we get:

```
0.0931 subsec11 (0.99)
0.0588 para111
0.0574 article (0.99)
0.0456 sec1 (0.49)
0.0300 para112
0.0252 para22
0.0123 sec2 (0.49)
0.0052 para113
```

- The length of the path between an inner element and its leaf elements can be used as weight
- In the XFIRM system, this distance is used in the propagation mechanism

Propagation

- The (simplified) score of an inner element e for a query q is given as follows:

$$\begin{aligned} score(e,q) &= \rho \times m \times \sum_{e_\ell} \alpha^{d(e,e_\ell)-1} \times score(e_\ell,q) \\ &\quad +(1-\rho) \times score(root,q) \end{aligned}$$

  where

  - m is the total number of retrieved leaf elements contained in e
  - $d(e,e_\ell)$ is the distance between elements e and e_ℓ in the document tree
  - $score(root,q)$ is the retrieval score of the $root$ element
  - ρ emphasizes the importance of the element score versus that of the document score in the contextualization strategy

Propagation

- Propagation mechanisms led to good retrieval performance
- Particularly, the GPX propagation mechanism produced top performance for the retrieval tasks of the INEX campaigns, showing its versatility for XML retrieval

Aggregation

- Aggregation is based on the work of Chiaramella *et al* on structured document retrieval
- The representation of an XML element can be viewed as the aggregation of:
  - its own content representation, and
  - the content representations of structurally related elements
- Retrieval can be based on these aggregated representations

Aggregation

- An element’s content representation is generated using standard indexing techniques
- An aggregation function is used to generate the representation of the non-leaf elements
- The aggregation function can include parameters specifying how the representation of an element is influenced by that of its children elements

Aggregation

- To illustrate, we describe an approach based on the language modeling framework
- For an element e, the probability of a query term term k_i given a language model based on the **own** element content $M_{e_{own}}$ is given by:

$$P(k_i|M_{e_{own}}) = (1-\lambda)P(k_i|e_{own}) + \lambda P(k_i|C)$$

  where

  - λ is the smoothing parameter controlling the influence of the background collection model C

Aggregation

- In our working example, only leaf elements own content
- All inner elements are made of the content of their children elements
- If we assume $\lambda = 0.8$, the $P(k_i|M_{e_{own}})$ are given below

	elements	$P(\text{wine}|M_{own})$	$P(\text{patagonia}|M_{own})$
leaf elements	para111	0.180	0.327
	para112	0.500	0.060
	para 113	0.020	0.260
	para22	0.420	0.060

Aggregation

- Now assume that e has several children, e_j, each with their own language model M_{e_j}
- Then, the aggregation function can be implemented as a linear interpolation of language models:

$$P(k_i|M_e) = \sum_{e_j} \omega_j P(k_i|M_{e_j})$$

where $\sum_{e_j} \omega_j = 1$

- The ω parameters model the contribution of each language model to the aggregation

Aggregation

- For our working example, we assume equal contribution from each of the children
  - It means dividing $P(k_i|M_{e_j})$ by the number of children
- We obtain the aggregated term statistics shown below

	elements	$P(\text{wine}|M_e)$	$P(\text{patagonia}|M_e)$
inner elements	subsec11	0.233	0.216
	sec1	0.117	0.108
	sec2	0.105	0
	article	0.111	0.054

Aggregation

- For instance, since `subsec11` has three children, the aggregated score for the term wine is then

$$P(wine|M_{subsec11}) = 1/3 \times (0.180 + 0.500 + 0.020) = 0.233$$

- $w = 1/3$ since we have equal contribution by all involved element language models
- The ranking is produced by estimating the probability that each element generates the query
- Other approaches for dealing with aggregation use fielded BM25 and probabilistic models
- An important issue with the aggregation method is the estimation of parameters

Merging

- Mass and Mandelbrod used an indexing strategy where a separate index is created for each element type
- Let us assume that a retrieval model is used to rank the elements in each index
- This results in separate ranked lists, one for each index, that will be merged to return to the user a single list
- For this, normalization is necessary to take into account the variation in size of the elements of the indexes
- For each index, the $score(q, q)$ is calculated, which is the score of the query as if it were an element in the collection
- For each index, the element score is normalized with $score(q, q)$

Processing Structural Constraints

- At INEX, structural constraints are viewed as hints as to where to look to find valuable information
- The reason for this view is two-fold
    - First, it is well known that users of IR systems do not always properly express their information need
        - For instance, a user asking for a paragraph on a given topic may not realize that valuable content is scattered across several paragraphs
    - Second, the belief that satisfying the content criterion is, in general, more important that satisfying the structural criterion
        - For example, the content in a title is probably the most important content of a section of a document

Processing Structural Constraints

- A first approach is to build a dictionary of tag synonyms
- For example, consider the elements:
    - `<p>` that corresponds to paragraph type
    - `<p1>` that corresponds to the first paragraph in a sequence of paragraphs
- It would be quite logical to consider `<p>` and `<p1>` as equivalent tags
- The dictionary can also be built from processing past relevance data
    - If query asks for `<section>` elements, then all types of elements assessed as relevant for that query are considered equivalent to the `<section>` tag

Processing Structural Constraints

- A second technique is that of structure boosting
- In this technique, the element score is first generated ignoring the structural constraint of the query
- Then, the element score is boosted according to how the structural constraint is satisfied by the element
- Consider the following content-and-structure query q expressed in the NEXI query language:

```
//article[about(., wine)]//sec[about(., patagonia)]
```

- This query will be divided into two subqueries:

```
q1 = //article[about(., wine)]
q2 = //sec[about(., patagonia)]
```

Processing Structural Constraints

- The element scoring strategy applied to subquery q2 yields these scores for the content criteria (c_score):

elements	c_score	s_score
para111	0.327	0.4
para113	0.261	0.4
subsec11	0.233	0.6
sec1	0.170	1
article	0.111	0.7

- The table also show the structure scores with respect to matching section elements (s_score) for the each element type (tag)

Processing Structural Constraints

- Let us assume that the content-based scores are boosted as follows for q2:

$$b\_score(e, q2) = 0.8 \times c\_score(e, q2) + 0.2 \times s\_score(e, sec)$$

- The boosted scores, b_score, are shown below

elements	c_score	s_score	b_score
para111	0.327	0.4	0.341
para113	0.261	0.4	0.288
subsec11	0.233	0.6	0.293
sec1	0.170	1	0.336
article	0.111	0.7	0.378

Processing Structural Constraints

- For simplicity, for query q1 we only consider articles as results
- Now for the full query q, we need to combine:
    - the boosted score of each element retrieved for query q2, and
    - the boosted score of the article containing that element and retrieved for query q1
- This can be defined, for instance, as follows:

$$s\_score(e,q) = b\_score(e,q2) \times b\_score(article,q1)$$

Processing Structural Constraints

- The resulting ranked list of elements is given below

```
0.0459 article
0.0415 para111
0.0409 sec1
0.0373 subsec11
0.0350 para113
```

Processing Structural Constraints

- An important issue here is to determine the actual level of imprecise matching
- For our example, this would translate into how should we set the values of the structure scores s_score
- The techniques described here were evaluated in the context of INEX, where the relevance of an element was assessed based on its content only

Removing Overlaps

- An XML retrieval system aims at returning the most relevant elements for a given user’s query
- When an element has been estimated relevant to a given query, it is likely that its ancestors will also be estimated as relevant
- Thus, several elements may be contained in the result list, eventually leading to a considerable amount of redundant information being returned to users

Removing Overlaps

- Returning redundant information has been shown to be distracting for the users
- Thus, it is necessary to decide which of these relevant but overlapping elements should be returned
- A first approach is to remove overlapping elements directly from the ranked list of elements
- Another approach is to select the highest ranked element from the result list, removing any ancestor and descendant elements with lower ranks
- The process is then applied recursively

Removing Overlaps

- A number of approaches have been developed where the actual structure of the document is considered
    - Mass and Mandelbrod look at the distribution of retrieved elements in the XML document structure in addition to their score
- Overall, techniques that consider the document tree structure tend to outperform those that do not
- There is, however, the issue of answer time, as the removal of overlaps is done at query time
- An interesting question to investigate is the effect of the original result list on the overlap removal strategy
- There are indications that a good initial result list might lead to a better overlap-free results list

XML Retrieval Evaluation

XML Retrieval Evaluation

- Research on structured text retrieval grew with:
  - the adoption of XML as the markup language for structured documents
  - the setup of the INitiative for the Evaluation of XML retrieval (INEX), the equivalent of TREC for evaluating XML retrieval effectiveness
- INEX established an infrastructure in the form of large test collections and appropriate measures

Document Collections

- The document collection used in INEX up to 2004 consisted of 12,107 articles marked-upin XML
- The articles were selected from 12 magazines and 6 transactions of the IEEE Computer Society
- On average, an article contains 1,532 XML nodes, where the average depth of the node is 6.9
- In 2005, the collection was extended with further publications from the IEEE Computer Society
- A total of 4,712 new articles were added, giving a total of 16,819 articles and 11 million elements

Document Collections

- Since 2006, INEX uses a different document collection, made from English documents from Wikipedia
- The collection consists of the full-texts, of 659,388 articles in XML from the Wikipedia project
- On average, an article contains 161.35 XML nodes, where the average depth of an element is 6.72
- This collection has a richer set of tags (1,241 unique tags compared to 176 in the IEEE collection)
- It also includes a large number of cross-references (represented with XLink)

Topics

- INEX identified two types of topics:
  - **Content-only (CO) topics**, which are information need statements that ignore the document structure
  - **Content-and-structure (CAS) topics**, which are information need statements that refer to both the content and the structure of the elements
- CO and CAS topics reflect users with varying levels of knowledge about the structure of the collection

Topics

- CAS topics fit the needs of users who want to take advantages of knowledge on the document structure to improve the quality of the results
- CAS topics are more likely to fit the needs of expert users, such as librarians or patent experts
- As in TREC, an INEX topic consists of the standard title, description, and narrative fields
- For CO topics, the title is a sequence of terms
- For CAS topics, the title is expressed using NEXI, a path-based query language for XML

Topics

- In 2005, the CO topics were extended into Content-Only + Structure (CO+S) topics
- The CO+S topics included a CAS title (`<castitle>`) field
- This field includes knowledge in the form of structural constraints
- CAS titles were expressed in the NEXI query language
- An example of a CO+S topic is given in the next slide

Topics: CO+S from INEX 2005

```
<inex_topic topic_id="231" query_type="CO+S">
<title>Markov chains in graph related algorithms</title>
<castitle>
   //article//sec[about(., +"markov chains" +algorithm +graphs)]
</castitle>
<description>Retrieve information about the use of markov
chains in graph theory and in graphs-related algorithms.
</description>
<narrative>I have just finished my Msc. in mathematics, in
   the field of stochastic processes. My research was in a subject
   related to Markov chains. My aim is to find possible
   implementations of my knowledge in current research. I’m mainly
   interested in applications in graph theory, that is, algorithms
   related to graphs (...)
</narrative>
</inex_topic>
```

Retrieval Tasks

- XML retrieval systems need to determine the appropriate level of element granularity to return to the users
- In INEX, a relevant element is defined to be at the right level of granularity if it:
  - discusses all the topics requested in the user query – it is exhaustive to the query
  - does not discuss other topics – it is specific to that query
- Up to 2004, the task of an XML retrieval system in INEX was to return those elements that are most relevant

Retrieval Tasks

- Within this generic task, two main sub-tasks were defined:
  - **CO sub-task**, which makes use of the CO topics, where an effective XML retrieval system is one that:
    - retrieves the most specific elements
    - retrieves only those which are relevant to the requested topic
  - **CAS sub-task**, which makes use of CAS topics, where an effective system is one that:
    - retrieves the most specific document components
    - match the structural constraints specified in the query

Retrieval Tasks

- This led to two CAS sub-tasks:
  - **SCAS sub-task** (strict content-and-structure)
  - **VCAS sub-task** (vague content-and-structure)
- The following two sub-tasks were defined in 2005:
  - **Focused sub-task**
  - **Thorough sub-task**

Relevance

- In XML retrieval, the relevance evaluation is complicated by the need to consider the structure in the documents
- An element and one of its children elements can both be relevant to a given query
- However, the child element may be more focused on the topic of the query than its parent element, which may contain additional irrelevant content
- In this case, the child element is a better element to retrieve than its parent element

Relevance

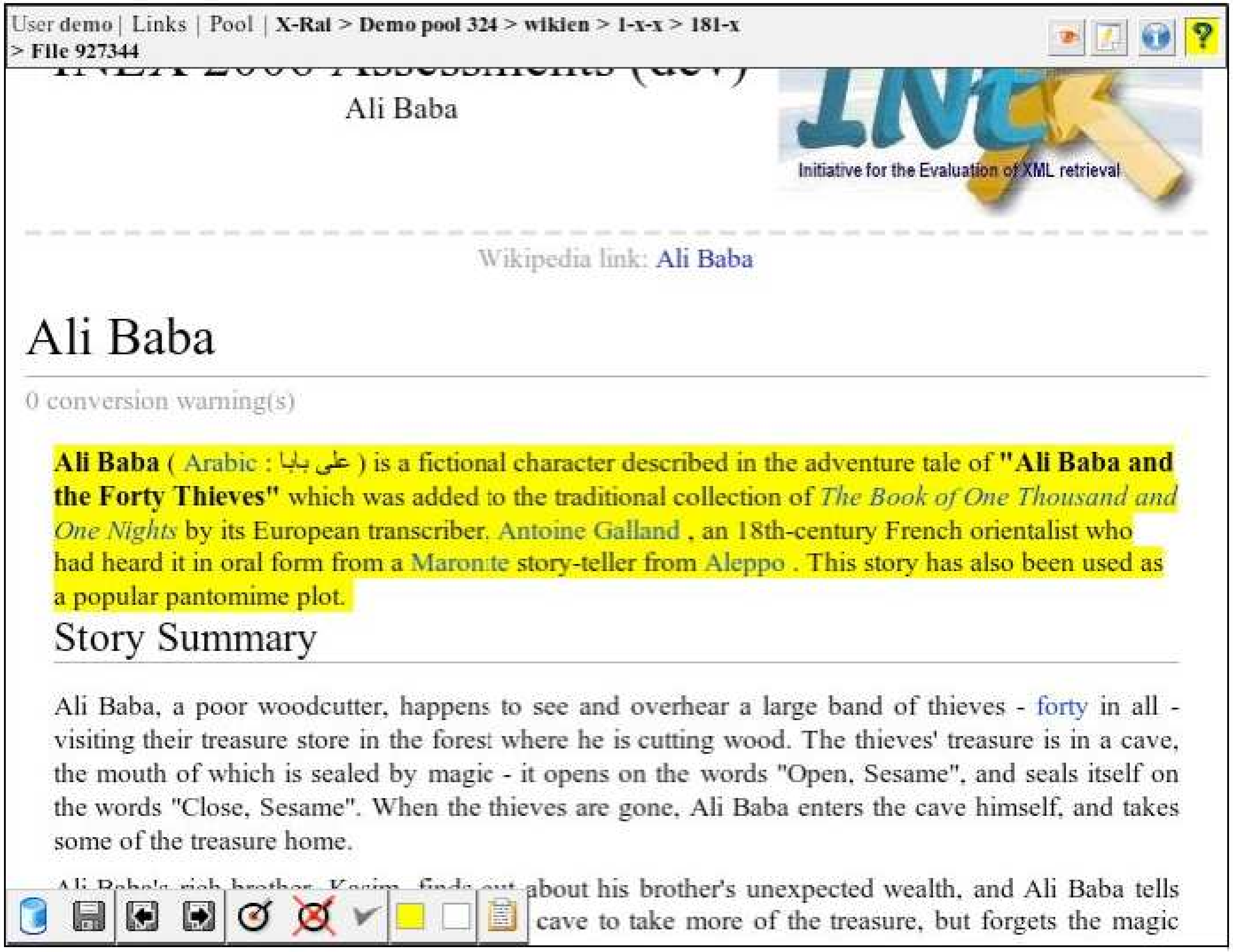
- To accommodate the specificity aspect, INEX defined relevance along two dimensions:
  - **Exhaustivity**, which measures how exhaustively an element discusses the topic of the user’s request
  - **Specificity**, which measures the extent to which an element focuses on the topic of request
- In addition, a scale is necessary to allow the representation of how exhaustive or how specific is an element
  - Binary values of relevance cannot reflect this difference
  - For example, a system that retrieve the only relevant section in a book is more effective than one that returns a whole chapter

Relevance

- INEX therefore adopted a four-point relevance scale for the exhaustivity and specificity dimensions:
  - Not – Marginally – Fairly – Highly
- Each year, assessors provided the relevance assessments through a relevance assessment tool
- The assessment process is composed of two phases
  - In the first phase, assessors highlighted text fragments containing only relevant information
  - In the second phase, for all elements within highlighted passages, assessors were asked to assess their exhaustivity

Relevance

- INEX 2006 assessment interface:


User demo | Links | Pool | X-Rai > Demo pool 324 > wikien > 1-x-x > 181-x > File 927344

Ali Baba

Wikipedia link: Ali Baba

Ali Baba

0 conversion warning(s)

Ali Baba (Arabic : علی بابا) is a fictional character described in the adventure tale of **"Ali Baba and the Forty Thieves"** which was added to the traditional collection of *The Book of One Thousand and One Nights* by its European transcriber, Antoine Galland , an 18th-century French orientalist who had heard it in oral form from a Maronite story-teller from Aleppo . This story has also been used as a popular pantomime plot.

Story Summary

Ali Baba, a poor woodcutter, happens to see and overhear a large band of thieves - forty in all - visiting their treasure store in the forest where he is cutting wood. The thieves' treasure is in a cave, the mouth of which is sealed by magic - it opens on the words "Open, Sesame", and seals itself on the words "Close, Sesame". When the thieves are gone, Ali Baba enters the cave himself, and takes some of the treasure home.

about his brother's unexpected wealth, and Ali Baba tells cave to take more of the treasure, but forgets the magic

Relevance

- Statistical analysis of the INEX 2005 results showed that, in terms of comparing retrieval performance, ignoring the exhaustivity dimension led to similar results
- As a result, in INEX 2006 relevance has been defined along the specificity dimension only
- In future INEX campaigns, researchers will examine a greater range of focused retrieval tasks

Measures

- Measuring XML retrieval effectiveness requires considering the dependencies among the elements
- Users may locate additional relevant information by browsing or scrolling down the elements
- This motivates the need to consider elements from where users can access relevant content
- The alternative would lead to a much too strict evaluation scenario

Measures

- From 2002 to 2004, INEX used **inex_eval**, which applies the measure of precall to XML elements
- Inex_eval is based on the number of retrieved and relevant elements
- Systems that return relevant but overlapping elements will be evaluated as more effective than those that do not return overlapping elements

Measures

- The classic definition of precision and recall can be modified to reflect this view:

$$Precision = \frac{amount\ of\ relevant\ information\ retrieved}{total\ amount\ of\ information\ retrieved}$$

$$Recall = \frac{amount\ of\ relevant\ information\ retrieved}{total\ amount\ of\ relevant\ information}$$

- Instead of counting the number of relevant items retrieved, we are measuring the amount of relevant text retrieved

Measures

- More formally, let:
  - $hlength(e)$ be the length in characters of highlighted content in element e for a given topic
  - $length(e)$ be the total number of characters contained in e
  - $Trel$ be the total amount of (highlighted) relevant information in the collection for the topic
  - $erank(i)$ be a function that returns the element at rank i
- Precision at rank r, indicated by $P@r$, is the fraction of retrieved relevant information up to rank r:

$$P@r = \frac{\sum_{i=1}^{r} hlength(erank(i))}{\sum_{i=1}^{r} length(erank(i))}$$

Measures

- Recall at rank r, indicated by $R@r$, is the fraction of relevant information retrieved up to rank r:

$$R@r = \frac{1}{Trel} \times \sum_{i=1}^{r} hlength(erank(i))$$

- The definition of $Trel$ depends on whether returning overlapping elements is allowed or not
  - For the thorough sub-task, $Trel$ is the total number of highlighted characters across all elements
  - For the focused sub-task, $Trel$ is the total number of highlighted characters across all documents
- Other measures include precision values at fixed recall levels, and mean average precision have been defined

Query Languages

Query Languages

- When searching unstructured text, users are naturally limited in the expressive power of their queries
- With structured text and a query language that supports its use, users can write more precise queries
    - For example, “I want a paragraph discussing penguin near to a picture labeled South Pole”
- Query languages are an integral part of XML and structured text retrieval

Characteristics

- The requirements for query languages for structured text retrieval can be divided into three main classes:
  - content constraints
  - pattern matching constraints
  - structural constraints

Content Constraints

- These are concerned with the specification of the content aspect of the information need
- Various types of content constraints exist:
  - **Word:** the document fragments to be returned should contain or approximate the query words
  - **Context:** conditions imposed on the positions of the words in the text
  - **Weight:** conditions on the importance of words and/or context constraints in the document fragments
  - **Boolean:** where all the above can be combined using Boolean operators

Content Constraints

- In (traditional) databases, processing a query yields a non-ranked list of document fragments
  - In IR, the list would be ranked
- Also, in databases, it is commonly the case that the words must be contained in the fragment to be returned
  - In IR, containment is replaced by **aboutness**

Pattern Matching Constraints

- These allow the retrieval of text fragments that match some specified pattern such as strings, prefixes, suffixes, substrings, or regular expressions in general

Structural Constraints

- These allow the specification of the structural aspect of the information need
- There are three main types of structural constraints:
  - **Target result:** users have the opportunity to specify which particular structural results they are targeting
  - **Support condition:** structure can be used to specify structural constraints other than that of the desired results
  - **Result construction:** results can be built from several fragments within or across documents
- It should however be noted that an increase in expressiveness entails an increase in time complexity

Classification of Languages

- XML query languages can be classified as
  - content-only
  - content-and-structure query languages

Content-only Queries

- Content-only queries make use of content constraints to express user information needs
- They are suitable for XML search scenarios in which users do not know the document structure
- XML retrieval systems must still determine what are the best fragments

Content-and-structure Queries

- This type of query provides a means for users to specify their content and structural information needs
- It is towards the development of this type of queries that most research on XML query languages lies
- There are three main categories of content-and-structure query languages, namely:
  - tag-based languages
  - path-based languages
  - clause-based languages
- The complexity and the expressiveness of these languages increase from tag to clause-based queries

Tag-based queries

- These queries allow users to annotate words that specifies a structural constraint to target as the result
- For example, the information need “retrieve sections about red wine” would be expressed as follows:

```
section: red wine
```

- They do not cater for support conditions and result constructions
- An example of a tag-based query language is XSEarch

Path-based Queries

- These queries are based upon the syntax of XPath to encapsulate the document structure in the query
- Examples of path-based query languages: XPath 1.0, XIRQL, and NEXI
- The information need “retrieve sections about red wine in documents about Chile” expressed in NEXI:

```
//document[about(.,Chile)]//section[about(.,red)]
```

- Path-based queries allow for expressing target results (“section” element above) and support conditions (“document ” about “Chile”)

Path-based Queries

- XIRQL allows for weights to be assigned to structural constraints
- For instance, the following XIRQL query:

```
//section[0.6 .//* $cw$ "Chile" + 0.4 .//section $cw$ "wine"]
```

- Any tag-based query can be rewritten using a path-based query language
- For example, in NEXI as follows:

```
//section[., about(red wine)]
```

Path-based Queries

- Moreover, any content query can be expressed as a path-based query
- For example as follows in the NEXI query language:

```
//*[., about(red wine)]
```

- This query asks for any element and at any level of granularity about “red wine”

Clause-based queries

- These queries use nested clauses to express information needs, very similarly to SQL
- The most prominent clause-based language for XML retrieval is XQuery, the W3C proposed standard
- A typical clause-based query is made of three clauses:
  - a **for clause** to specify support conditions
  - a **where clause** to specify content constraints
  - a **return clause** to specify target fragments and the construction of new fragments as result

Clause-based queries

- The following information need “retrieve document sections with the title penguins” would be expressed as follows in XQuery:

```
for $x in /document/section
    where $x/title=penguins
    return $x/section
```

- XQuery Full-Text extends XQuery with powerful text search operations, including:
  - context constraints
  - ranking functionality

XML Query Languages

- XML content-only query languages are specified in the same way as in flat text retrieval
- We present here:
  - two path-based query languages: XPath and NEXI, and
  - two clause-based query languages, namely XQuery and XQuery Full-Text
- These query languages provide a good overview of the most recent developments on XML query languages

XPath

- XPath (XML Path language) is a query language defined by the W3C, which primary purpose is to access or navigate to components of an XML document
- In addition, XPath provides facilities for the manipulation of strings, numbers and Boolean operations
- The most important type of expressions in XPath is the location path. For example:

  ```
  book/publisher/@isbn
  ```

  is a location path, where:
  - `book` and `publisher` are steps that navigate to children elements with names “book” and “publisher”
  - `@isbn` is a step that navigates to attributes with name “isbn”

XPath

- All the steps are separated by “/”
  - This means, for example, that the location path selects the `isbn` attributes that are directly below `publisher` elements
- Publisher elements are referred to as children of book elements
- The navigation steps can be separated by “//”
  - This means that the location path navigates to the current element and all its descendant elements before it applies the next step

XPath

- For example:
  - `book//title` navigates to all title elements that are directly or indirectly below a book element,
  - `//title` will select all title elements in the document
- Special steps include the self step denoted "." and the parent step denoted ".."
- For example:
  - `.//book` returns any book elements contained in the current node
  - `../publisher` returns the publisher elements of the parent of the current node

XPath

- Also, XPath uses wildcards such as “*” and “@*” to navigate to elements and attributes with any name
  - E.g. `book/*` and `book/publisher/@*`
- At each step, predicates can be specified between “[ ]”, which must be satisfied for the nodes to be selected
  - For example, `//book[@year=2002]/title`
- The standard comparison operators $=$, $!=$, $<$ and $<=$ can also be used in the predicates
- Existential predicates are used to check whether a certain path expression returns a non-empty result
  - For example, `//publisher[city]` selects publishers for which the city information is given

XPath

- Positional predicates are used to navigate according to the position of an element in the document tree
  - For example, `//publisher/country[1]/city`
- The comparisons and existential conditions can be combined with **and**, **or** and **not()**
  - For example, `not(@year = 2002)`
- An important function in XPath is the Boolean function `contains()`
  - It takes two string arguments and returns true if the first string contains the second string, and false otherwise
  - This function can be used to check whether an element contains in its text a specified string

XPath

- As such, XPath is not a query language that can be directly used for content-oriented XML retrieval
- XPath is, however, used by other XML query languages or has inspired other XML query languages

NEXI

- The Narrowed Extended XPath I (NEXI) was developed at INEX
- It has been used by INEX participants to express realistic content-and-structure queries to form the test collection
- The enhancement comes from the introduction of a new function, named `about()`
- This function requires an element to be about the content
- This is to reflect that an element can be relevant to a given query without actually containing any of the words used in the query

NEXI

- The reasons for choosing a small subset of XPath were two-fold:
    - First, the queries to INEX showed high syntactic and semantic error rates in the use of XPath path location
    - The second reason was that NEXI was defined for the purpose of performing retrieval evaluation
- Positional predicates are not allowed, as they do not bring anything in terms of effectiveness evaluation
- Also, all target elements must have at least one content condition, i.e., one `about()` function
- It is indeed a mechanical process to return, for instance, the title of sections on a given topic

NEXI

- An example of a NEXI query is:

```
//article[about(.//body, "artificial intelligence")]//
            body[about(., chess) and about(., algorithm)]
```

- NEXI was developed to construct topics for the purpose of evaluating XML retrieval effectiveness
- It is therefore the task of the XML retrieval system to interpret a NEXI query
- The interpretation is with respect to the `about()` condition and the structural constraint

XQuery

- XQuery includes XPath as a sub-language
- Additionally, it allows to query multiple documents and combine the results into new XML fragments
- The core expressions of XQuery are the FLWOR expressions
- The following XQuery expression lists the publishers whose average price of books is less than 50 euros:

```
for $pub in distinct-values (doc("pub.xml")//publisher)
let $a := avg(doc("bib.xml")/book[publisher = $pub]/price)
    where $a < 50
    order by $pub/name
    return <publisher> { $pub/name , $a } </publisher>
```

XQuery

- A FLWOR expression starts with one or more `for` and `let` clauses, each binding a number of variables
  - The variables bound within the `for` clause are used to iterate over the elements of the result sequence of an expression
  - The variables bound within the `let` clause are used to iterate over the entire sequence
- An optional `where` clause specifies selection conditions
- Further, an optional `order by` clause specifies sorting criteria
- Finally, a `return` clause specifies the results to be returned

XQuery

- Consider again our XQuery expression example:

```
for $pub in distinct-values (doc("pub.xml")//publisher)
let $a := avg(doc("bib.xml")/book[publisher = $pub]/price)
    where $a < 50
    order by $pub/name
    return <publisher> { $pub/name , $a } </publisher>
```

- The `for` clause binds the variable $pub such that it iterates over the publisher elements in the document entitled “pub.xml” in the order that they appear
- The `distinct-values` function eliminates duplicates in “pub.xml”

XQuery

- The `let` clause binds the variable $a to the average price of books from publisher $pub
- Then, those publisher elements for which the condition in the `where` clause is true are selected
- The resulting bindings are sorted by the `order by` clause on the publisher name in $pub ($pub/name)
- The `return` clause creates a new publisher element that contains the name element of the publisher $pub
- The results are new fragments, as they were not in the XML original documents

XQuery

- XQuery is a powerful query language for XML retrieval, and can be viewed as the SQL for XML
- It is a language that is mostly appropriate for data-centric XML retrieval

XQuery Full-Text

- XQuery Full-Text is an XML query language that extends XQuery with powerful text search capabilities
- XQuery Full-Text allows to specify that the results should be ranked according to how relevant they are
- The added text search capabilities are the result of the introduction of a new XQuery expression, *FTContainsExpr*
- For instance, the following *FTContainsExpr* expression:

```
//book[./title ftcontains {"red" "wine"} all]//author
```

  returns the authors of books whose title contains all the specified words, here “red” and “wine”

XQuery Full-Text

- XQuery Full-Text defines primitives for searching text, such as phrase, word order, and word proximity
- It also allows the specification of:
  - Letter cases in matched words, the use of stemming, thesauri, stop words, content pattern matching, and many more
- For instance, the following *FTContainsExpr* expression restricts the proximity of the matched words to appear within a window of six words:

```
//book[./title ftcontains {"red"  "wine"}
       all window at least 6 words]//author
```

XQuery Full-Text

- The expression below looks for matches to the word “growing” in its various forms, **e.g.** “grow”, “grows”:

```
//book[./title ftcontains "growing" with stems]//author
```

- The ranking of results is provided with the introduction of *FTScoreClause* expressions
- We illustrate with an IR search-like example:

```
for $b score $s in //book[./title ftcontains {"red" "wine"} all]
    order by $s descending
    return <book isbn="{$b/@isbn}" score="{$s}"/>
```

XQuery Full-Text

- XQuery Full-Text was not designed to implement a specific scoring method, but to allow an implementation to proceed as it wishes
- XQuery Full-Text has all the characteristics required by both data and document-centric XML retrieval applications
- From a content-oriented XML retrieval perspective, XQuery Full-Text may be viewed as far too complex for many end-users to master