Element Relationship: Exploiting Inline Markup for Better XML Retrieval

Philipp Dopichaj

University of Kaiserslautern
Dept. of Computer Science
P. O. Box 3049
D-67653 Kaiserslautern, Germany
`dopichaj@informatik.uni-kl.de`

Abstract: With the increasing popularity of semi-structured documents (particularly in the form of XML) for knowledge management, it is important to create tools that use the additional information contained in the markup. Although research on textual XML retrieval is still in its early stages, many retrieval approaches and engines exist. The use of inline markup in these engines so far is very limited. We introduce the concept of element relationship and describe how it can improve similarity calculation. We illustrate our ideas with examples based on an existing document collection.

1 Textual XML Retrieval

In traditional Information Retrieval (IR), a user has an information need and wants to obtain documents fulfilling that need from a document base [BYRN99]. The situation is essentially the same in retrieval on document-centric XML; one important difference is that documents are not assumed to be atomic units, that is, the retrieval engine should return the most specific fragments satisfying the query.

The traditional IR techniques can be used for semi-structured data as well, but as they do not make use of the additional information contained in the markup, they are likely not to yield the best results possible. Because of this, new, XML-specific query languages and retrieval engines were developed. Structure-based query languages like XPath and XQuery assume that the searcher has an intimate knowledge of the documents to be queried, as they expect him to formulate queries based on the element names and nesting. A draft version of XQuery Full-Text adds a contains operator that supports comparison using standard IR techniques; the user still has to specify exact paths, however. Other query languages are closer to the ones used in traditional IR [FG01, TW02].

All these approaches use the XML markup to some extent. Markup can be used at several levels in an XML document schema: *Block-level markup* can be used to embed metadata (like authors' names) and to represent the structure of the document; examples include ⟨body⟩ in (X)HTML and ⟨section⟩ in DocBook [WM99]. *Inline markup* is used on single words or (short) phrases to convey the meaning or intended representation of the marked-up contents.

Block-level markup is very important for finding the document fragments to return for a query, but that is not the topic of our paper. Inline markup can be very useful for indexing and comparison purposes, because it may hint at the correct way to interpret an element's contents; making good use of inline markup is the main focus of this paper.

In Section 2, we will provide example scenarios where existing retrieval approaches offer no satisfactory solution. In Section 3, we describe our concept of element relationship that addresses these problems, followed by conclusions and a description of further research options in Section 4.

2 Motivation

In this section, we shall motivate why there is a need to make better use of inline markup in XML retrieval. We do this by providing several example scenarios that are inadequately supported by the existing query languages and retrieval engines. The context of all scenarios is the collection of Linux Howto documents collected by the Linux Documentation Project (TDLP)[1]. The documents are marked up in DocBook [WM99], an XML- (and SGML-) based markup language for the creation of computer-related texts.

Example 1 *Adam does not know the details of DocBook markup, but he can distinguish various basic types of search terms. When he searches for information about the command shell `bash`, he should be able to specify that the word `bash` is only relevant if it is used as a technical term; in particular, it is of no interest if "to bash" occurs in normal text.*

Example 2 *Betty knows DocBook well, but she is interested in a higher level of abstraction, because she knows that any of several element types might contain the relevant text. For example, when she searches for information about `save`, she is interested in ⟨command⟩s and ⟨menuitem⟩s (among others), but not on hints about saving paper.*

The users in these examples would benefit from a level of detail between simple keyword-based search and complex XML path queries. Typical query languages do not support this intermediate level: Either they are purely keyword-based or they require detailed knowledge of the relevant tag names, like XIRQL [FG01] or XQuery.

Example 3 *Charlie has performed a search and found a document fragment that almost, but not quite, satisfies his information need. He proceeds to search for similar documents.*

No major XML retrieval engine directly supports documents as queries. It is possible to transform the document to a query, but this will lead to one of two problems:

- The converted query is too specific and matches only the original document (if the markup is converted to XPath constraints); this can easily happen if the input document is short and contains detailed markup.

- The converted query is too general so that the semantic information contained in the markup is lost.

[1] http://www.tldp.org

What is needed for good results in this example is a retrieval engine that supports some form of fuzzy element matching.

Example 4 *Howto author Dorothy wants to mention the shell* `bash` *in her text, but she is not sure whether* ⟨productname⟩ *or* ⟨application⟩ *is the appropriate markup.*

Given the wealth of elements provided by DocBook, it is not surprising that the semantics of some elements are very similar, so it is frequently hard to choose. Another problem lies in the authors' laziness or less than perfect knowledge of DocBook: An examination of the Linux Howtos revealed that technically incorrect markup is fairly common. Even the DocBook reference [WM99] concedes that this problem exists: "`Emphasis` is often used wherever its typographic presentation is desired, even when other markup might theoretically be more appropriate." Because of this, retrieval engines should support approximate matching of elements.

Example 5 *Eric considers the amount of markup necessary for something as simple as command-line input to be excessive and omits all but the top-level tags.*

This is a real problem at least in the Linux Howtos; the reason for this 'lazy' markup is probably the high number of semantic markup options available to the author that cause work without much apparent benefit (the rendered presentation might not change anyway). It is unrealistic to expect the retrieval engine to reconstruct the missing elements, but it can make sure that equivalent fragments with complete and incomplete markup compare almost equal.

3 Element Relationship

In order to address the problems mentioned in the previous sections, we introduce the concept of *element relationship* which allows us to partially substitute elements with other, similar elements in the retrieval process.

The first two examples from the previous section illustrate that we need a level of abstraction above that of element names: In both cases, the searchers were willing to supply detail on the markup structure, but not at that level of detail. It seems reasonable to form groups of related tags and offer an input field for each of them. The number of groups should be small, because otherwise it is still too complicated. For DocBook, the following list might be a reasonable starting point:

- Computer-related text (e. g. user input, program output and listings)
- Emphasized text (e. g. text marked as emphasized, keywords and index terms)
- Metadata (e. g. author and revision information)
- "Normal" text (everything else)

(These categories are not necessarily free of overlaps, but as we shall see later, this poses no problem and can indeed be used to our advantage.) The result is an interface that

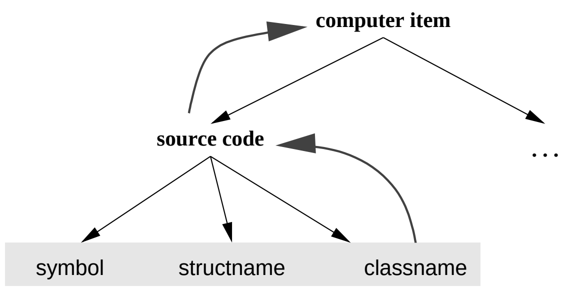

Figure 1: Gradual generalization when searching specific element types.

is still usable without having to learn a complex query language, but offers more power than simple keyword-based languages.

If we use an existing document as a query ("more like this"), the query obviously contains elements instead of categories. Due to the problems with ambiguous or misused markup, searching for element contents *only* in elements of the same type may lead to omitting many good matches. On the other hand, simply searching for the contents in *all* text, no matter what markup is used, sacrifices precision.

In this case, we want the ability to gradually generalize the element, that is, matches contained in the same type of element receive the highest score and going up in a hierarchy of categories reduces the score. Figure 1 illustrates this principle: A search for text marked up with ⟨classname⟩ would first search all ⟨classname⟩ elements, then (at reduced score) all **source code** (⟨symbol⟩, ⟨structname⟩, . . .), then all **computer items** etc.

3.1 Facets of Element Similarity

The question that arises at this point is: What can we base our element grouping on? There is no single aspect that can be used in isolation to calculate the similarity of two element types. Instead, there are several, somewhat related options:

Tag names. Ideally, element names should convey their meaning without any further information; XXL [TW02] makes this assumption and uses a separate ontology to relate these names. In our experience, meaningful names (i. e., names that correspond to unabbreviated words) for XML tags are the exception rather than the rule. Very often, cryptic abbreviations like ⟨qandadiv⟩ from DocBook or ⟨br⟩ from (X)HTML are used, and considering that even humans have problems interpreting these names without further information, it seems unrealistic to expect computers to manage that task.

Syntactic restrictions. XML schema languages like DTD, XML Schema or Relax NG provide provisions for defining the syntactic structure of the documents in that schema, in particular the permitted nesting of elements. In DocBook, for example, the element ⟨copyright⟩ may only contain the elements ⟨holder⟩ and ⟨year⟩. This information is easily parseable, but its use for our purposes is limited: In most cases, either all inline elements are allowed as sub-elements or none.

Semantics. Considering the previous remarks, it appears to be necessary to use further information to establish semantic relations between element types, for example, grouping related element types (see Figure 1). This information is typically available in the form of documentation aimed at authors of documents, but actually making use of that information can be very time-consuming.

Contents. If a significant number of documents is available, we can use statistical methods based on the contents of the XML elements. One simple approach would be to use statistics of character classes like upper/lower case letters, digits, etc. to differentiate the element categories; for example, UNIX paths typically contain a disproportionate number of slashes ("/"). More sophisticated approaches could use the words both in the element and in its context in order to obtain classifiers.

Visual appearance. Normally, document-centric XML is meant to be rendered for presentation to the user. The number of semantic inline tags typically exceeds the number of available formatting options of the output format, so many tags are represented in the same way. While much of the semantics contained in the markup is lost, the mapping is not arbitrary: Even though several unrelated tags might be represented in the same way, *related* tags usually have the *same* formatting. In DocBook, for example, the computer-related entities like filenames, computer I/O and environment variable names are all likely to be rendered in a fixed-width font. The transformation from XML to the rendered representation can be specified in XSLT style sheets.

Each of these aspects can be used as the basis for a similarity measure comparing two elements. Instead of creating a similarity matrix containing the similarities of all pairs of elements, we want to have a more compact representation that it is comprehensible to a human reader. Reconsidering our examples, we can see that some form of categorization (with overlapping categories) would be most useful. The number of elements in DocBook (and most other XML-based languages) is too high for a single level of categories to be sufficient – we would end up with either too many or with too broad categories.

The solution is to use an almost hierarchical representation, where categories can contain sub-categories (*almost* because of overlaps). This keeps the number of members in each category low but enables us to take the query categories from higher-level categories.

The *element relationship graph* (ERG) is a directed, acyclic graph. The nodes are labeled with either an element name (*element nodes*) or a category label (*category nodes*). Element nodes may have several incoming edges (because categories may overlap). The category nodes are partitioned into *aspect sets* corresponding to the aspects mentioned above; no two nodes from different aspect sets have a direct connection. In essence, this means that we have sub-graphs that are disjoint except for the element nodes.

As we hinted at when describing the aspects, the construction of an ERG can only be automated in some cases. In the case of a graph based on element semantics, there is no option but to create the graph manually, based on the documentation. Considering that there are typically hundreds of elements in a given schema – about 300 in DocBook –, this may seem like a daunting task.

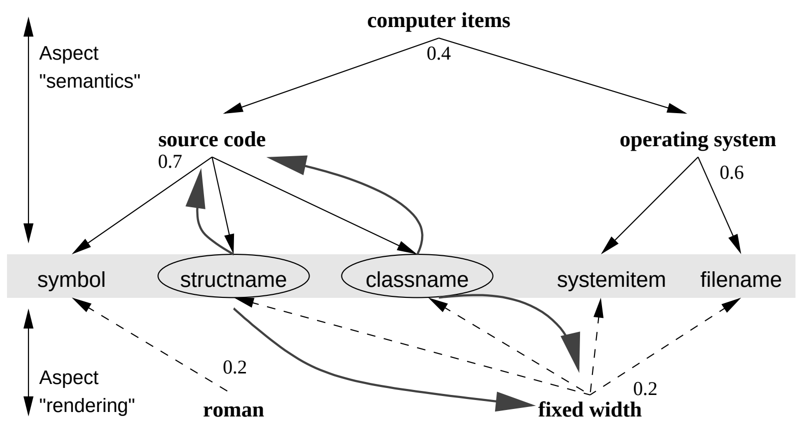

Figure 2: Calculating similarity in the ERG. The names in the gray bar are tags, the labels in the upper part are (semantic) concepts, the labels in the lower part are presentation styles.

It is rarely necessary to start from scratch, given nothing more than a list of element names and descriptions: For didactic reasons, tutorials and reference material for a schema normally describe the elements in related groups. For DocBook, for example, we have a section about "Logical Divisions: The Categories of Elements in DocBook" in the reference manual [WM99, Section 2.5], and a quick reference card where the easily parseable XML source code is available. Thus, while the task is still far from trivial, it turns out to be manageable, as we shall see in Section 3.3.

3.2 Element Similarity in the Element Relationship Graph

The graph we have described so far provides information about (almost hierarchical) relationships of elements and newly-introduced categories, but it does not quantify element similarity. A first approach could be to define the distance of two elements – which can be seen as the inverse of similarity – to be the shortest path between them, ignoring the direction of the edges. This approach is not entirely satisfactory, however, because not all possible paths are equal: The information that the elements ⟨structname⟩ and ⟨classname⟩ are both rendered in the same font is not as meaningful as the information that they are both in the semantic group **source code** (see Figure 2).

Bergmann [Ber98] examined a similar problem in the context of similarity measures for taxonomies in structural Case-Based Reasoning, which only needs minor modifications to be used in our context. We start by labeling each inner node n with a number $c_n \in [0,1]$ denoting the *coherence* of the group formed by the direct descendants. Furthermore, the values must satisfy the following condition: If the inner node a is an ancestor of d, $c_a < c_d$ must hold. As we shall see presently, this condition ensures that similarity can never increase if we increase the level of generality.

Given two different nodes n_1 and n_2, we can then easily calculate their similarity: We need to find the set of their closest common ancestors A; the similarity is $\max_{a \in A}(c_a)$.

In Figure 2, the nearest set of common ancestors for ⟨structname⟩ and ⟨classname⟩ is $\{\textbf{source code}, \textbf{fixed width}\}$, and the resulting similarity is $\max(c_{\textbf{source code}}, c_{\textbf{fixed width}}) = \max(0.7, 0.2) = 0.7$.

The difference between this approach and an approach based purely on distance is obvious if we look at ⟨systemitem⟩ and ⟨classname⟩: The closest common ancestor based on path length is **fixed width**, but **computer items** has a higher coherence value, so the resulting similarity is 0.4 instead of 0.2.

3.3 Constructing an Example Element Relationship Graph

To show that it is feasible to construct an ERG, we created one for DocBook and the aspects of visual appearance and contents.

Visual appearance of the output is determined by XSL style sheets, so we took the official ones[2] and wrote a script to derive a graph from them, using the style sheets for transforming to HTML as a basis. We are concerned about the markup of inline elements, so we only used the file inline.xsl to avoid unnecessary clutter of the resulting graph.

The style sheets also contain templates not directly tied to HTML tags for modularization. For example, the template inline.italicmonoseq indicates that the text is both italic and monospace; having this intermediate node in the graph has the advantage of expressing that *both* features are present. If the ERG contained two separate links instead, only one of them would be used for similarity calculation, so some information would be lost. Of course, there are also links from the corresponding HTML elements to this intermediate node, so that elements having only one of these are still similar to elements having both.

The effort needed was low: It took one person less than two hours total, and a significant fraction of that time was spent removing templates that are not relevant in our context. Removing these nodes is only necessary for making the graph easier to comprehend; leaving the additional nodes in the graph would not result in worse similarity calculation.

Next, we looked at a possible semantic grouping of the elements based on the quick reference we mentioned previously[3]. It contains 46 overlapping groups of elements, but not all groups are relevant to us because we are only interested in the 32 groups containing inline elements. The authors categorized the elements with a focus on quick look-up, not on semantic similarity, so we needed to modify them slightly.

Then we successively merged the low-level categories until we reached the high level of abstraction mentioned above (computer-related text, emphasized text, metadata, normal text). DocBook's main application area is computer texts, so it is not surprising that the **computer items** category is the most complex one, with 11 sub-categories in three levels.

The last step is to assign the coherence values. We found it easiest to create an internally consistent labeling (on a scale from 0 to 1) for the sub-graph of each aspect. When merging the sub-graphs, we then assigned a weight to each of them denoting the relative importance. For example, the aspect of semantics is much more important than the aspect of presentation, so the weights were 1.0 and 0.4. The final coherence value of a category is then determined by multiplying the preliminary coherence value and the importance of the

[2]http://docbook.sourceforge.net/projects/xsl/ (version 1.66.1)
[3]http://www.dpawson.co.uk/docbook/qrefplain.xml

```
⟨programlisting⟩
    ⟨prompt⟩# ⟨/prompt⟩
    ⟨userinput⟩⟨command⟩ln -s⟨/command⟩ /dev/hdc /dev/dvd⟨/userinput⟩
⟨/programlisting⟩
```

(a) Text with inline markup (from the DVD Playback Howto; slightly reformatted)

Term	DocID	Count	Enclosing markup
`ln`	1	1	{ ⟨command⟩, ⟨programlisting⟩,⟨userinput⟩ }
`s`	1	1	{ ⟨command⟩, ⟨programlisting⟩,⟨userinput⟩ }
`dev`	1	2	{ ⟨programlisting⟩, ⟨userinput⟩ }
`hdc`	1	1	{ ⟨programlisting⟩, ⟨userinput⟩ }
`dvd`	1	1	{ ⟨programlisting⟩, ⟨userinput⟩ }

(b) Corresponding index entries. Note that the two occurrences of ⟨dev⟩ could only be merged because they have the same enclosing markup.

Figure 3: Storing markup information in the index

corresponding sub-graph. This approach makes it easy to adjust the relative importance later without needing to revisit all nodes individually.

Overall, the construction of an initial version of the ERG took less than one day. Of course this original version may well need to be refined based on feedback from users in everyday use.

3.4 Building the Index Structures

Before the document base can be searched, we need to construct an index for better performance. We use the vector model (see [BYRN99]) for basic similarity calculation, so we need an *inverted index* mapping each word to the list of documents containing the word. In addition to that, we need to record what markup enclosed each occurrence of the word. Figure 3 shows an example of how markup is stored in the index.

There are several important points concerning the index:

- Only *inline* element names are recorded in the index, as only they are needed for element relationship considerations. We could also store information about the corresponding categories in the index, but that would prevent us from changing the ERG without rebuilding the index, with little benefit.
- Only the presence of a given element name is recorded, that is, we only store a flag whether a given element name occurred in the enclosing markup; we omit the path.
- The terms in the table can occur multiple times, with different enclosing markup. For example, if the text in Figure 3(a) included another reference to `dev`, enclosed *only* in ⟨userinput⟩, a new row would be added to the index table.

As mentioned above, we must store the information about the markup in the index, so its size will increase compared to the basic vector model. The most straightforward way of storing this information is a bit vector, each bit of which signifies whether the corresponding markup element is among the enclosing elements of the word. These vectors are extremely sparse: Less than 1 % of the index entries in our example contained any markup information, none contained more than three different elements.

Another aspect that contributes to an increased index size is that we need to store more entries, because the same word can occur several times with different enclosing markup. In the worst case, this could be a factor of 2^e where e is the number of inline markup elements. Realistically, the overhead is much lower, because most words are not marked up at all, and the few that are are marked up in few combinations; experiments with the Linux Howtos indicate that the number of index entries increases by approximately 10 %.

We need another index for the ERG that supports efficiently finding both the parents of a given node and finding all *element* descendants of a given node. The number of nodes is considerably lower than the number of documents, so we can afford some redundancy in exchange for better performance.

3.5 Search Process

One important issue that we have not addressed yet is the actual retrieval process from query to results. As mentioned in the previous section, we use the vector model for basic similarity calculation at the category level and use weighted sums to aggregate the category similarities to a global similarity. In the indexing phase, each document is parsed, and the occurrence of each word along with its containing element names is recorded. (Recording the containment information is crucial for calculating the similarity at the category level.)

The query is formulated on the basis of high-level categories, and although the documents contain detailed markup, we use the element relationship graph for grouping the element contents into categories at the same level as the query. For each category, we calculate a similarity as follows: $\text{sim}_{\text{global}} = \sum \text{sim}_i w_i$.

The weights w_i with $\sum w_i = 1$ represent the relative importance of category c_i. For example, in our example scenario, **computer items** are very important compared to free text, as they have stricter, that is, less ambiguous, semantics. The retrieval process (given a query composed of word sets in the query categories) is as follows:

1. For each category in the query:
   (a) Obtain the descendant elements from the ERG.
   (b) Search the main index for items that are embedded in one of the elements obtained in the previous step and match the query words for this category.
   (c) Calculate the resulting similarities, taking into account the coherence value.
2. Merge the similarity lists.
3. If documents with higher similarities might be found by broadening the categories, do so and repeat the main step.

The last step in particular warrants some explanation: It would be very time-consuming to always traverse the whole relationship graph for broadening categories. Fortunately this is not really necessary, as typical users are only interested in the top few documents, so we need continue broadening when we can be sure (because of the coherence value) that any further documents will have a lower similarity than those we already know. This is similar in spirit to the concept of query relaxation described in [SB00].

4 Conclusions and Future Work

In this paper, we outlined how we can improve textual XML retrieval both as far as the query interface and as far as the similarity calculation is concerned. We achieve this by introducing element relationship, which can be used to determine how similar two elements from a given schema are.

We have shown that the construction of the ERG for a reasonably well-documented XML-based language can be accomplished in very short time, and that the increase in index size is tolerable. One important component that is still missing is an experimental verification of the retrieval quality of our approach. That retrieval system should be put to a test in the next INEX workshop[4], but we have to consider that the current test documents use only visual markup akin to HTML. In this context, we also need to investigate how our approach can be integrated into existing retrieval engines and approaches.

It is conceivable that we can use the element relation graph for specifying local tokenizers and similarity measures, but at this point it is unclear whether this can be accomplished without an unacceptable price in performance.

References

[Ber98] Ralph Bergmann. On the Use of Taxonomies for Representing Case Features and Local Similarity Measures. In Lothar Gierl and Mario Lenz, editors, *Proceedings of the 6th German Workshop on Case-Based Reasoning (GWCBR'98)*. Universität Rostock, 1998.

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, Harlow, Essex, England, 1999.

[FG01] N. Fuhr and K. Großjohann. XIRQL: A Query Language for Information Retrieval in XML Documents. In W.B. Croft, D. Harper, D.H. Kraft, and J. Zobel, editors, *Proceedings of the 24th Annual International Conference on Research and Development in Information Retrieval*, pages 172–180. ACM Press, New York, 2001.

[SB00] Jürgen Schumacher and Ralph Bergmann. An Efficient Approach to Similarity-Based Retrieval on Top of Relational Databases. In Enrico Blanzieri and Luigi Portinale, editors, *EWCBR*, volume 1898 of *Lecture Notes in Computer Science*. Springer, 2000.

[TW02] Anja Theobald and Gerhard Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. In *Extending Database Technology*, 2002.

[WM99] Norman Walsh and Leonard Muellner. *DocBook: The Definitive Guide*. O'Reilly & Associates, Sebastopol, 1999.

[4]for 2004's workshop, see http://inex.is.informatik.uni-duisburg.de:2004/