The Index-based XXL Search Engine for Querying XML Data with Relevance Ranking

Anja Theobald, Gerhard Weikum

University of the Saarland, Germany
WWW: http://www-dbs.cs.uni-sb.de
E-mail: {theobald, weikum}@cs.uni-sb.de

Abstract. Query languages for XML such as XPath or XQuery support Boolean retrieval: a query result is a (possibly restructured) subset of XML elements or entire documents that satisfy the search conditions of the query. This search paradigm works for highly schematic XML data collections such as electronic catalogs. However, for searching information in open environments such as the Web or intranets of large corporations, ranked retrieval is more appropriate: a query result is a rank list of XML elements in descending order of (estimated) relevance. Web search engines, which are based on the ranked retrieval paradigm, do, however, not consider the additional information and rich annotations provided by the structure of XML documents and their element names. This paper presents the XXL search engine that supports relevance ranking on XML data. XXL is particularly geared for path queries with wildcards that can span multiple XML collections and contain both exact-match as well as semantic-similarity search conditions. In addition, ontological information and suitable index structures are used to improve the search efficiency and effectiveness. XXL is fully implemented as a suite of Java servlets. Experiments with a variety of structurally diverse XML data demonstrate the efficiency of the XXL search engine and underline its effectiveness for ranked retrieval.

1. Introduction

1.1 Motivation

XML is becoming the standard for integrating and exchanging data over the Internet and within intranets, covering the complete spectrum from largely unstructured, ad hoc documents to highly structured, schematic data [Kos99]. XML data collections can be viewed as a directed, labeled data graph with XML elements as nodes (and their names as node labels) and edges for subelement relationships as well as links both within and across documents [ABS00]. A number of XML query languages have been proposed, such as XPath, XML-QL, or the recently announced W3C standard XQuery. These languages combine SQL-style logical conditions over element names, content, and attributes with regular-expression pattern matching along entire paths of elements. The result of a query is a set of paths or subgraphs from a given data graph that represents an XML document collection; in information retrieval (IR) terminology this is called Boolean Retrieval.

This search paradigm makes sense for queries on largely schematic XML data such as electronic product catalogs or bibliographies. It is of very limited value, however, for searching highly heterogeneous XML document collections where either data comes from many different information sources with no global schema or most documents have an ad hoc schema or DTD with element names and substructures that occur only in a single or a few documents. The latter kind of environment is typical for document management in large intranets, scientific data repositories such as gene expression data collections and catalogs of protein structures, and, of course, also for the Web. For example, a bank has a huge number of truly semistructured documents, probably much larger in total size than the production data held in (object-) relational databases; these include briefing material and the minutes of meetings, customer-related memos, reports from analysts, financial and business news articles, and so on. Here, the variance and resulting inaccuracies in the document structures, vocabulary, and document content dictate ranked retrieval as the only meaningful search paradigm.

So the result of a query should be a list of potentially relevant XML documents, elements, or subgraphs from the XML data graph, in descending order of estimated relevance. This is exactly the rationale of today's Web search engines, which are also widely used for intranet search, but this technology does not at all consider the rich structure and semantic annotations provided by XML data. Rather state-of-the-art IR systems restrict themselves to term-frequency-based relevance estimation [BR99, Ra97] and/or link-based authority ranking [BP98,Kl99,KRR+00]; note that the latter has been fairly successful for improving the precision of very popular mass-user queries but does not help with advanced expert queries where recall is the main problem.

This paper presents a query language, coined XXL (for Fle<u>x</u>ible <u>X</u>ML Search <u>L</u>anguage), and the prototype implementation of the XXL search engine, as steps towards more powerful XML querying that reconciles the more schematic style of logical search conditions and pattern matching with IR-style relevance ranking.

1.2 Related Work

Related approaches, which combine search over structural and textual information, have already been proposed in the context of hypertext data (see, e.g., [CSM97, BAN+97, FR98, MJK+98]). However, this work predates the introduction of XML and does not have the same expressiveness for querying semistructured data as more recent languages such as XML-QL [XMLQL] or XQuery [XQuery]. Extending such XML query languages with text search methods has been suggested by [CK01, FKM00, FG00, HTK00, NDM+00, TW00]. However, [NDM+00] and [FKM00] are limited to pure keyword search for Boolean retrieval and do not support relevance ranking. In contrast, the simultaneously developed languages XIRQL [FG00] and XXL [TW00] (the latter is our own approach) have been designed to support ranked retrieval. A restricted approach along these lines is [HTK00], which assumes advance knowledge of the document structure and provides similarity search only on element contents, not on element names. Extensions of conventional database query languages a la SQL in order to support similarity search have been addressed also in the WHIRL project [Coh98, Coh99], but with focus on structured data with uniform, fixed schema and not in the context of XML. Another text retrieval extension of XML-QL has recently been proposed by [CK01]: XML data is mapped onto relational storage and similarity conditions are translated into subqueries that are evaluated using WHIRL. To our knowledge, none of the above approaches uses ontological information in their similarity metrics.

1.3 Contribution and Outline of the Paper

The contributions of this paper are twofold:

1) At the conceptual level, we show how to reconcile pattern matching along paths of XML elements with similarity conditions and IR-style relevance ranking as well as

simple ontological reasoning. In contrast to the prior work on probabilistic query evaluation on structured data, most notably [Coh98, Coh99], our language XXL supports path expressions in the spirit of modern XML query languages. In contrast to IR research, XXL can exploit the structural information and the rich semantic annotations that are immanent in XML documents. In comparison to our own prior work reported in [TW00] the current paper goes a significant step further and integrates ontological relationships as a basis for effective similarity search.

2) At the implementation level, we present techniques for efficiently evaluating a simple but widely useful class of XXL queries using several index structures and a heuristic strategy for decomposing compound search conditions into subqueries. The index structures include an element path index similar to the data guides approach of [MAG+97, MWA+98], a term-occurrence index as commonly used in IR engines for element contents, and an ontological index for the occurrences of element names. None of these structures is fundamentally new, but their combination proves to be a powerful and, to our knowledge, novel backbone for XML query evaluation with relevance ranking. Our prototype implementation of XXL is fully operational as a suite of Java servlets and includes a Java-based GUI for graphically composing XXL queries. We report measurements that demonstrate the effectiveness and efficiency of the XXL search engine.

2. XXL: A Flexible XML Search Language for Ranked Retrieval

2.1 Example Scenario

As an example, consider the following fragments of three XML documents about zoos of the world. The first document is a web portal to XML documents about zoos. The second and the third document contain descriptions of animals in particular zoos.

URL 1: http://www.myzoos.edu/zoos.xml

```
<zoos> Zoos of the World
  <name href=”http://www.allzoos.edu/american_zoos.xml”> San Diego Wild Animal Park,CA,USA </name>
  <name href=”http://www.animals.edu/european_zoos.xml”> Tierpark Berlin, Germany </name>
   ...
</zoos>
```

URL 2: http://www.allzoos.edu/american_zoos.xml

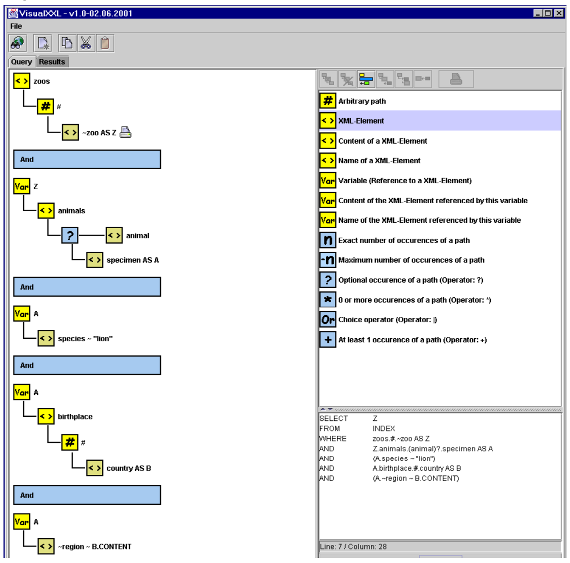
```
<animal_park name=”San Diego Wild Animal Park”
                country=”USA”  city=”Escondido”>
  <animals>
    <animal name=”Teddy”>
      <specimen>
        <species>brown bear</species>
        <range>Europe</range>
        <location no=”411”/>
        <birthplace href=”
           http://www.animals.edu/european_zoos.xml#
           XPointer(id(‘Tierpark Berlin’))”/> ...
      …
</animal_park>
```

URL 3: http://www.animals.edu/european_zoos.xml

```
<zoo name=“Tierpark Berlin“ country=“Germany“
      city=“Berlin“>
  <animals>
    <specimen name=”Bobby”>
      <species>snow leopard</species>
      <region>Central Asia</region>
      <location no=”327”/>
      <birthplace href=”
                  http://www.parks.edu/asian_zoos.xml#
                  XPointer(id(‘Zoo Tokyo’))”/>…
    <enclosure no=“327“>
    <size>16m2</size> …
</zoo>
```

As an example query consider the search for “zoos that have big cats such as lions which are born in a zoo of their natural geographic area”. This query is easily expressible in our language XXL, shown in Fig. 1a, with keywords in boldface. The XXL search engine also has a graphical user interface based on a Java that allows clients to compose queries in an interactive manner. Fig. 1b shows a screenshot with the graphical representation of query Q1.

Q1:
Select Z
From http://www.myzoos.edu/zoos.xml
Where zoos.#.~zoo **As** Z
And Z.animals.(animal)?.specimen **As** A
And A.species ~ "lion"
And A.birthplace.#.country **As** B
And A.~region ~ B.CONTENT


VisualXXL - v1.0-02.06.2001
File
Query
Results
zoos
#
~zoo AS Z
And
Z
animals
?
animal
specimen AS A
And
A
species ~ "lion"
And
A
birthplace
#
country AS B
And
A
~region ~ B.CONTENT
Arbitrary path
XML-Element
Content of a XML-Element
Name of a XML-Element
Variable (Reference to a XML-Element)
Content of the XML-Element referenced by this variable
Name of the XML-Element referenced by this variable
Exact number of occurences of a path
Maximum number of occurences of a path
Optional occurence of a path (Operator: ?)
0 or more occurences of a path (Operator: *)
Choice operator (Operator: |)
At least 1 occurence of a path (Operator: +)
Line: 7 / Column: 28

```
SELECT      Z
FROM        INDEX
WHERE       zoos.#.~zoo AS Z
AND         Z.animals.(animal)?.specimen AS A
AND         (A.species ~ "lion")
AND         A.birthplace.#.country AS B
AND         (A.~region ~ B.CONTENT)
```

Fig. 1: Query Q1 in **a)** textual form (left) and **b)** represented in the Visual XXL GUI (right)

In this query uppercase characters denote element variables that are bound to a node (i.e., element) and its attributes of a qualifying path (i.e., A, B, Z in our example), # is a wildcard placeholder for arbitrary paths, ? indicates an optional element on a path, and dots stand for path concatenation. ~ is a similarity operator for semantic similarity, which is used as a unary operator when applied to element names and as a binary operator when applied to element content.

XML data can be viewed as a directed, labeled data graph where the vertices are XML elements marked with the element names. Each node consists of a unique object identifier (oid) and a label. A label can be an element name, an attribute name, the content of an element, or the value of an attribute. The edges represent element-attribute, element-subelement, or element-element links within XML documents, as well as element-element relations between elements of different XML documents based on XLink and XPointer. To simplify matters, we will focus on simple XLinks which can be recognized by href attributes as well as simple XPointers which consists of absolute references (root(), id()) and a sequence of relative references such as child() and descendant(). Fig. 2 shows (the relevant part of) the XML data graph for the sample data introduced above. In this graph we distinguish *n-nodes* for element names and attribute names and *c-nodes* for contents/values.

Queries such as Q1 are evaluated by traversing the XML data graph, possibly using "shortcuts" obtained from index lookups, and comparing the various search conditions against the nodes of the data graph. Exact-match conditions such as "Z.animals.(animal)?.specimen **As** A" return true or false for a given subgraph, and the end node of a matching path is bound to an element variable, A in this case. Similarity conditions such as "A.~region ~ B.CONTENT" assign a similarity score to a candidate element or path.

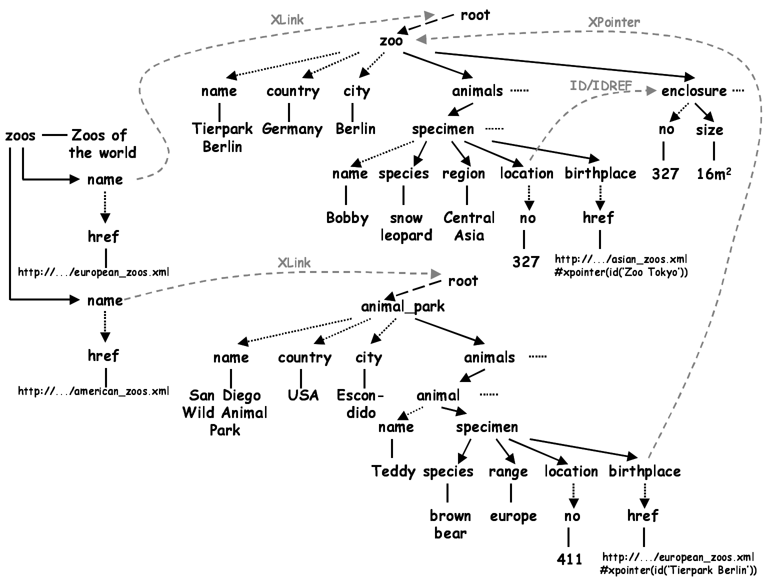


Fig. 2: XML data graph

The similarity scores of different conditions are composed using simple probabilistic reasoning. Complete results, as determined by the element variables that appear in the Select clause, are returned in descending order of similarity to the query (i.e., relevance). With the given sample data, both Bobby, the snow leopard, and Teddy, the brown bear, will be returned as search results, but Bobby will be ranked higher because snow leopards have a closer relationship to lions. Note that Teddy qualifies as an approximate match although he lives in an "animal park" rather than a "zoo" and the geographic distribution of his species is indicated by "range" rather than "region"; that is, element names with sufficient semantic similarity to the ones in the query are considered as relevant, too.

2.2 XXL Syntax and Semantics

The Select clause of an XXL query specifies the output, for example a list of URLs or all bindings of certain element variables. The From clause defines the search space, which can be a set of (seed) URLs or the index structure that is maintained by the XXL search engine. The Where clause of an XXL query specifies the search conditions. We define the Where clause of a query as the logical conjunction of *path expressions*, where a path expression is a regular expression over *elementary conditions* and an elementary condition refers to the name or content of a single element or attribute.

In addition to the standard set of operators on strings and other simple data types ("=", "≤", etc.), the elementary conditions supported by XML include a semantic similarity operator "~", which is used as a unary operator on element names and as a binary operator on element content.

Each path expression can be followed by the keyword "As" and a variable name that binds the end node of a qualifying path (i.e., the last element on the path and its attributes) to the variable. A variable can be used within path expressions, with the meaning that its bound value is substituted in the expression.

The relevance-enabled semantics of a query then is to return a ranked list of approximately matching subgraphs called *result graphs* each with a measure of its relevance (i.e., semantic similarity) to the query. Our relevance measure is defined inductively as follows. We interpret the similarity score for an elementary condition as a relevance probability. Then we need to combine the relevance probabilities for elements with regard to elementary conditions into a relevance measure for a path or subgraph with regard to a composite query condition. In the absence of any better information, we simply postulate probabilistic independence between all elementary conditions, and derive the combined probabilities in the straightforward standard way (i.e., by simply multiplying probabilities for conjunctions and along the elements of a path, etc.)

The result of an XXL query is a subgraph of the XML data graph, where the nodes are annotated with local relevance probabilities for the elementary search conditions given by the query. These relevance values are combined into a global *relevance score* for the entire result graph. Full details of the semantics of XXL and especially the probabilistic computation of similarity scores can be found in [TW00].

2.3 Ontology based similarity

The use of ~ as a binary operator, i.e., in the form "element ~ term", requires a two-step computation. The first step of the computation determines similar terms (with relevance score π_1) to the given term based on the ontology. The second step computes the tf*idf-based relevance (π_2) of each term for a given element content (where tf*idf refers to the standard formula of IR-style relevance based on term frequencies (tf) and inverse document frequencies (idf), see, e.g., [BR99]). The element content under consideration then satisfies the search condition with relevance $\pi_1 \cdot \pi_2$.

To define the basic probabilities π_1 and π_2 we now give details on the similarity metric for terms or element names within an ontology. Consider the example shown in Fig. 3 as a part of an ontology about animals. In XXL an ontology is a directed acyclic labeled graph G=(V,E) where V is the set of nodes, the terms (in our case element names, attribute names, and terms in element/attribute contents), and E is the set of edges (see, e.g., [MWK00] for more general variants of ontological graphs). The outgoing edges of a node have integer weights, which are unique among siblings, as shown in Fig. 3, thus leading to a total order among siblings.

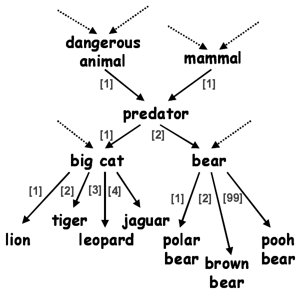


Fig. 3: Part of an ontology

Edges describe relationships between hypernyms (broader terms) and hyponyms (narrower terms), the weights and order of edges express the similarity among sibling nodes. For example, the local similarity of big cats is given as follows: 1 – lion, 2 – tiger, 3 – leopard, 4 – jaguar, and so on. This means that lions are more related to tigers than to leopards.

In order to use an ontology for similarity search we have to define the similarity of two graph nodes (terms) of the ontology. The general rationale for this similarity metric is to use the number of edges on the shortest path between two graph nodes as their "semantic distance", and then derive a normalized similarity metric from this consideration. The notion of distance is actually a bit more sophisticated because we consider also the "semantic distance" between all siblings of a node rather than treating all siblings as equally related to each other. Generally, when we refer to paths in G, we interpret the edges of the DAG as undirected edges.

The similarity between two nodes of the ontology is defined as follows:

a) Let v_1 and v_2 be nodes with v_1 being an ancestor of v_2 or vice versa; then $\text{dist}(v_1,v_2) = \text{length}(v_1,v_2)$.
   Examples: dist(dangerous animal, brown bear)=3; dist(predator, brown bear)=2

b) A node $p \in V$ on the shortest path between two nodes v_1 and v_2 is called the lowest common ancestor (lca) of v_1 and v_2 if there are two paths $p \dots v_1$ and $p \dots v_2$ such that $\text{length}(p, v_1)$ and $\text{length}(p,v_2)$ are both minimal. (If there are multiple shortest paths, we choose one arbitrarily.)
   Example: predator is the lca of lion and brown bear.

c) Let v_1 and v_2 sibling nodes with parent node p; then the sibling distance of v_1 and v_2 is defined as $$\text{siblingdist}(v_1,v_2) = \frac{|\text{weight}(p,v_1) - \text{weight}(p,v_2)|}{\text{maxweight}}$$
   where maxweight is the maximal weight of all outgoing edges of node p.
   Example: siblingdist(lion, leopard) = (|1 – 4|) / 4 = 3/4

d) Let v_1 and v_2 be arbitrary nodes of T, let p be the lca of v_1 and v_2, and let p_1 and p_2 be children of p such that $p\ p_1 \dots v_1$ and $p\ p_2 \dots v_2$ are paths of T; then the distance of v_1 and v_2 is defined as
   $$\text{dist}(v_1,v_2) = \text{length}(p_1,v_1) + \text{length}(p_2,v_2) + \text{siblingdist}(p_1,p_2)$$
   Example: dist(lion, brown bear) = 1 + 1 + (|1 – 2|)/2 = 2.5

e) Now we can define the normalized similarity of two nodes v_1 and v_2 of an ontology tree T as $\text{sim}(v_1,v_2) = 1 / (1 + \text{dist}(v_1,v_2))$
   Example: sim(lion, tiger) = 1 / (1+1/4) = 0.8

With each node of the ontology we can associate a set of synonyms. These are treated as a single semantic concept, with the same predecessors and successors in the graph. For example, the node "big cat" could have a fifth child with the three synonyms "cougar", "puma", "mountain lion".

3. Index Support for XXL Queries

Similarly to Web search engines, the XXL engine builds on precomputed index structures for evaluating the various search conditions in a query. These indexes are constructed and maintained based on a crawler that periodically or incrementally traverses documents that are reachable from specified roots within the intranet or Web fragment of interest. We use three index structures, element path index (EPI), element content index (ECI), and ontology index (OI), which are described in the subsequent subsections.

3.1 Element Path Index (EPI)

The EPI contains the relevant information for evaluating simple path expressions that consist of the concatenation of one ore more element names and path wildcards #. Each element name that occurs at least once in the underlying data is stored exactly once in the index. Associated with each element name is a list of its occurrences in the data: the URL of the document and the oid of the element. Furthermore, short index-internal pointers to the parent and the children of an element are stored with each element occurrence. In addition to this information on parent-child edges of the data graph, outgoing (XLink and XPointer) links such as href attributes can optionally be stored with each element occurrence in the index. Attributes are treated as if they were children of their corresponding elements, along with a flag to indicate the attribute status.

The parent-child information of the EPI is illustrated in Fig. 4a, which refers to the example data graph shown in Fig. 2. For notation we treat the entire index as if it were a nested relation. The subentries of the form "↑..." are short, index-internal pointers. The optionally maintained information on links is depicted in Fig. 4b, again referring to the example data of Fig. 2.

```
(element name,                                         | (element name,
 (element instance, pointer to the father element),    |   (element instance, pointer to linked element)
                         (pointer to a child element)) |
zoo                                                    | birthplace
   [URL1, 0]      ---                                  |     [URL2, 12]        ↑zoo (…)
                       ↑name (URL1, 1)                 |     [URL3, 13]        ↑zoo (URL2, 1)
                       ↑name (URL1, 2)                 |     …
                       …                               | location
   [URL2, 1]      ↑… (URL2, …)                         |     [URL2, 10]        ↑enclosure (URL2, 30)
                       ↑name (URL2, 2)                 |     [URL3, 11]        ↑enclosure (URL3, …)
                       ↑country (URL2, 3)              | …
                       …                               |
   …                                                   |
```

Fig. 4: a) Schema for EPI on parent-child relationships (left side), and **b)** links (right side)

The EPI very much resembles the notion of "data guides" introduced [GW97] and similar structures in other XML query engines (e.g., [FM00]). However, the EPI only stores paths of length 2 explicitly and reconstructs longer paths by combining multiple index entries based on the index-internal pointers. This way the EPI can answer the following kinds of subqueries in a very efficient manner:

- retrieve all URLs and element IDs for a given element name,
- retrieve all children (and link successors) of a given element instance or a given element name,
- retrieve all descendants or ancestors of a given element, up to a specified depth,
- test whether there is a path from element x to element y.

Elements retrieved from the EPI for a given subquery always have a relevance score of 1.

The EPI is implemented as an in-memory red-black tree on element names; index-internal pointers are virtual memory addresses. The entire index is loaded from disk when the XXL search engine starts.

3.2 Element Content Index (ECI)

The element content index (ECI) contains all terms, that is word stems, that occur in the contents of elements and attributes. For stemming we use the Porter algorithm [BR99]. Each term has associated with it its inverse document frequency (the idf value, which is the quotient of the total number of elements that are known to the index and the number of elements that contain the term), its occurrences, and for each occurrence the term frequency (the tf value). So the ECI largely corresponds to a standard text index as used by virtually all Web search engines [BP98, BR99]. The main difference is that our units of indexing and tf*idf computations are elements rather than entire documents. For query evaluation the ECI is used to answer subqueries of the form: find all element instances that contain a given term, along with the corresponding tf*idf values.

Our implementation of the ECI uses the text retrieval engine of Oracle8i, known as interMedia [Ora8i]. We store element contents and attribute values in a database table of the form *data (URL, oid, content)*, similarly to [BR01, FK99], and create an interMedia index for the attribute „content“. Thus we are able to exploit Oracle’s special Contains predicate for ranked text retrieval using the tf*idf formula for relevance scores.

3.3 Ontology Index (OI)

The quality of similarity search and result ranking on semistructured data can be improved by exploiting ontological information. Motivated by the visionary but still somewhat vague idea of a “Semantic Web” [SemWeb], we consider element (and attribute) names as the semantically most expressive components of XML data. This assumes that many XML documents will be constructed according to standardized, domain-specific ontologies (see, e.g., www.ebxml.org for e-business data, www.fpml.org for financial instruments, or www.geneontology.org for bioinformatics data) with meaningful, carefully chosen element names. Note that this does by no means imply that all data is schematic: ontologies may include a large number of terminological variants, and there is large diversity of ontologies for the same or overlapping application domains anyway.

The ontology index (OI) contains all element names that occur in the indexed XML data, and organizes these names into an ontological graph as explained in Section 2.3. The XXL query processor exploits the OI for query expansion: a path expression of the form “~e” for an element name e is expanded into a disjunctive path expression of the form “e | $term_1$ | ... | $term_k$”, with k terms returned by the OI (with relevance scores π_1, ..., π_k) as most similar to the given element name e. This broadened expression will then be evaluated by the EPI. We also use the OI to evaluate similarity conditions on element contents, that is conditions of the form “e ~ t” with e being an element name and t being a term. The XXL query processor first determines similar terms for the given query term t, and then the ECI retrieves relevant element instances for the broadened set of terms. The relevance score of a matching element is the product of the relevance score provided by the OI and the relevance score provided by the ECI.

As an example, consider a query with the search condition ~region ~ “India”. The element name “region” will be expanded into “region | country | continent” and the ECI lookups for “India” will consider also related terms such as “Asia”, “Bangladesh”, “Tamil Nadu”, etc., provided the similarity scores returned by the OI are above some threshold.

The OI is constructed and maintained as follows. When the crawler passes an XML document to the indexer, all element names that are not yet in the OI are added to the index. Their positions in the ontology graph are determined by calling WordNet [WordNet], which is a comprehensive thesaurus (or linguistically oriented ontology) put together by cognitive scientiests. Specifically, we retrieve the concept description (i.e., the “word sense” in WordNet terminology), all synonyms, and all hypernyms and hyponyms for the given word from WordNet. In our current prototype the OI itself is implemented using the user-defined thesaurus functionality of Oracle8i interMedia [Ora8i]. The information obtained from WordNet is directly mapped onto terms and synonym, hypernym, and hyponym relationships between terms. All this information is dynamically inserted into the thesaurus using the ctx_thes.create_phrase function of interMedia. For query expansion the information is retrieved using the Contains predicate of Oracle’s SQL extensions.

4. Query Processing

The evaluation of the search conditions in the Where clause consists of four main steps:

1. The XXL query is decomposed into subqueries, and each subquery is internally represented in the form of a query graph, which is essentially a finite state automaton.
2. The order in which the various subqueries are evaluated is chosen (global evaluation order).
3. For each subquery, the order in which the components of the corresponding path expression are tested is chosen (local evaluation order).
4. For each subquery, subgraphs of the data graph that match the query graph are computed, exploiting the various indexes to the best possible extent.

4.1 Query Decomposition

The Where clause of an XXL query is of the form “Where P_1 AS V_1 And ... And P_n As V_n” where each P_i is a regular path expression over elementary conditions and the V_i are element variables to which the end node of a matching path is bound. As in other XML query languages, variables can occur in the place of an element within a path expression in multiple subqueries; we assume, however, that there is no cyclic use of variables (i.e., there is no cycle in the variable dependency graph with V_i depending on all variables that occur in the path expression of P_i.

The regular path expression of a subquery can be described by an equivalent non-deterministic finite state automaton (NFSA). For subquery $Q_i = P_i$ AS V_i, NFSA(Q_i) is constructed and represented as a *query graph* QG_i = QG(NFSA(Q_i)). Fig. 5 shows the query graphs of the subqueries of our example query Q1 from Section 2.1. Nodes with dashed ovals are final states of a regular path.

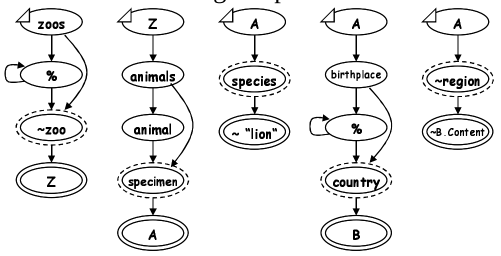


Fig. 5: Query graph representations of subqueries

4.2 Evaluation Order

The global evaluation order specifies the order of executing the query graphs QG_1, …, QG_n. Ideally, this would take into account selectivity estimates for the various subqueries, but note that the order is constrained by the possible use of element variables in the path expressions. So before evaluating a query graph that uses certain variables in the place of element names, all subqueries whose results are bound to these variables must be evaluated. For simplicity, our current prototype simply evaluates subqueries in the order in which they appear in the original query; we assume that variables are first bound by an As clause before they are used in the path expressions of subsequent subqueries.

The local evaluation order for a subquery specifies the order in which it is attempted to match the query graph's nodes with elements in the data graph. The XXL prototype supports two alternative strategies: in top-down order the matching begins with the start node of the query graph and then proceeds towards the final state(s); in bottom-up order the matching begins with the final state(s) and then proceeds towards the start node. These options are similar to the left-to-right and right-to-left strategies for evaluating path expressions in the object-oriented database query language OQL.

4.3 Index-based Subquery Evaluation

A *result path* is a path of the XML data graph that satisfies the path expression p of a subquery. The nodes of this path are annotated with local relevance values as described in 2.2. This result for a single subquery (query graph) determines a local variable binding, which is part of a global variable binding. Taking one result path from each subquery yields a *result graph*, which is a subgraph of the underlying XML data graph, by forming the union of the result paths for a given variable binding. The global relevance score for this result graph is computed as outlined in Section 2.2. Here it is important to emphasize that the same variable binding must be used for all subquery results, as the variables link the various subqueries.

For a local variable binding the evaluation of a subquery always produces a single path of the XML data graph annotated with appropriate local relevance scores for each node according to the elementary conditions given by the considered subquery.

As mentioned before we exploit the various index structures in finding matching subgraphs for a given query graph. This is done by identifying subgraphs of the query graph for which the matching subgraphs from the data graph have been precomputed and stored in one of the indexes. This is feasible in many, albeit not in all cases, but we expect that most applications would use a rather simple structure for XXL subqueries, namely, path expressions of the form ~e_1.#.~e_2.#. … .#.~e_n ~ t where e_1 through e_n are element names and some or all of the ~ symbols in front of the element names and the path wildcards # may be omitted. This subquery type does not use the full expressiveness of regular expressions (e.g., there is no Kleene star recursion over nested subexpressions) and can therefore be efficiently evaluated. As an example consider the subquery zoos.#.~zoo.animals.species ~ "lion" with the query graph shown in Fig. 6.

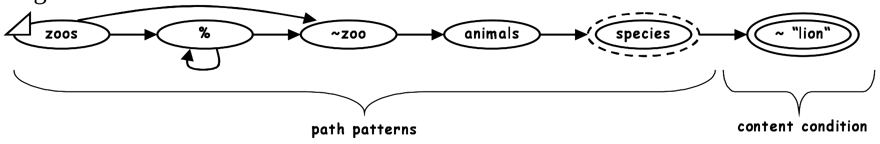


Fig. 6: Example query graph for XXL subquery

A *path pattern* is a path in the query graph from a start node to a final node. Each node of a path pattern is an elementary path condition (see Section 2.2), that is an element name, possibly with lexical wildcards %, or a name prefixed by the unary ~ operator. A *content condition* is an elementary content condition (see Section 2.2) to be met by the content of the end node in a matching path. A *result path* for a query graph is a path of the XML data graph that satisfies both the path pattern and the content condition of the given query graph.

Path patterns are evaluated using the following methods on the element path index (EPI) and the ontology index (OI), and content conditions are evaluated using the element content index (ECI) and the ontology index (OI). Recall from Section 2.2 that n-nodes are nodes of the XML data graph containing element names and c-nodes are nodes of the XML data graph containing the content of an XML element.

1. For a given elementary path condition without the unary ~ operator, e.g., "zoos" or a version with lexical wildcards such as "zoo%", the EPI returns a set of n-nodes of the XML data graph that satisfies this condition with relevance $\pi = 1$.
2. For a given elementary path condition with the unary ~ operator, e.g., "~zoo", the OI returns a set of similar terms with relevance $\pi > 0$, e.g., "animal_park", and then the EPI returns a set of n-nodes of the XML data graph that corresponds to one of these terms.
3. For two concatenated elementary path conditions "$c_1.c_2$" and a given n-node that satisfies condition c_1 with relevance π_1 computed by method 1 or 2, the EPI returns a set of n-nodes of the XML data graph that satisfies the condition c_2 with relevance π_2 using method 1 or 2 such that these are children of the given n-node. Finding parents is analogous.
4. For two given n-nodes the EPI can test the existence of a path between the given nodes, e.g., solving „zoos.#.~zoos".

With the following three methods supported by the ECI and the OI we evaluate content conditions for a given c-node of the XML data graph.

5. For a given content condition without the binary ~ operator, e.g., "= lion", and a given n-node the ECI returns a c-node that satisfies the given search condition with relevance $\pi = 1$ and this c-node is a child of the given n-node.

6. For a given content condition with the binary ~ operator, e.g., "~ lion", and a given n-node the OI returns a set of similar terms with relevance $\pi_1 > 0$, and then the ECI returns a c-node that corresponds to one of these terms with relevance π_2 such that the c-node is a child node to the given n-node. The final relevance is $\pi_1 * \pi_2$.
7. For a given elementary content condition of the form 1 and 2 and no information about the n-node, the ECI returns a set of c-nodes that satisfies the given condition with relevance $\pi > 0$.

4.4 Result Composition

The result paths for the various subqueries of a given XXL query are composed into a global result, which is a subgraph of the underlying data graph, by forming the union of the result paths for a given variable binding. Relevance scores of this result graph are computed as outlined in Section 2.2. Here it is important to emphasize that the same variable binding must be used for all subquery results, as the variables link the various subqueries. The exhaustive search algorithm computes all results for all possible variable bindings according to the global evaluation order for the subqueries and their As clauses.

5. Architecture of the XXL Search Engine

Our prior work [TW00] already described a preliminary and incomplete implementation of XXL based on Oracle. That implementation was fairly limited in that all XML data had to be loaded into an Oracle database upfront and all XXL queries were mapped onto SQL queries with some use of the Oracle interMedia text retrieval engine. The full-fledged prototype that we refer to in the current paper has been completely re-implemented. It now includes the three index structures described in Section 3, relies on Oracle only as a storage manager for index data (but not for the actual XML documents themselves), and has a full-fledged query processor implemented as a set of Java servlets (i.e., running under the control of a Web server such as Apache) and does no longer rely on an underlying SQL engine.
The architecture of the XXL search engine is depicted in Fig. 6. It consists of three types of components:

1. Service components: the crawler, the query processor (both Java servlets), and a Visual XXL GUI (a Java applet)
2. Algorithmic components: parsing, indexing, word stemming, etc.
3. Data components: structures and their methods for storing various kinds of information such as DTD/schema files, the EPI, the ECI, query templates, etc.

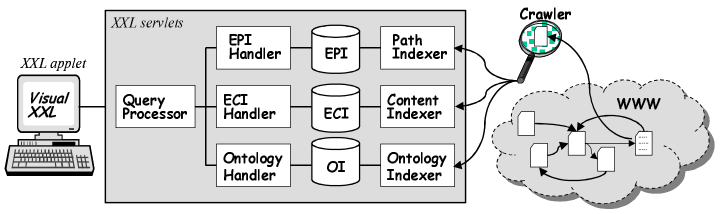


Fig. 7: Architecture of the XXL search engine

The ECI and the OI are stored as Oracle tables. So index lookups to these two structures involve SQL statements. These are very simple statements that can be executed very efficiently, but the query processor may invoke them very frequently. To avoid repeated lookups for the same element name or term and to minimize the number of JDBC calls to Oracle, both ECI and OI entries can be cached in the query processor servlet.

6. Experimental Evaluation

For the experimental setup we crawled four collections of XML documents and constructed the corresponding index structures: a collection of religious books (Bible, Koran, etc.), a collection of Shakespeare plays, bibliographic data about ACM Sigmod Record, and synthetic documents with the structure of bibliographic data. Altogether this data collection contained 45 XML documents, some of which were extremely long and richly structured, with a total number of 208 449 elements (with 62 different element names). Note that the diversity of this data posed a challenge for the search engine that would not be present with a homogeneous XML collection that follows a single DTD or XML schema.

6.1 Ontology-based Similarity Search for Element Names and Contents

Table 1 shows the results for simple similarity queries with both the unary and binary ~ operator. Note that the large number of results is explained by the fact that the queries retrieved all matching paths, and there were many different matching paths within the same document. The “Select URL …” option would have condensed this result into a small number of matching documents, but we were especially interested in evaluating XXL’s capability to find and rank all semantically relevant pieces of information.

The variants “top-down” and “bottom-up” refer to the two strategies for the local evaluation order, either starting from the start nodes of the query graph (top-down) or from the final nodes (bottom-up). As the run-time figures in the table show, there is no clear preference for one of the two strategies; the discriminating factor is the selectivity of the elementary conditions (i.e., element names to be approximately matched) at the begin and end of the path patterns.

For the third and fourth queries the relevance scores were relatively low as a result of multiplying a relevance score from the OI (for the conditions with the unary ~ operator) and a relevance score from the ECI (for the conditions with the binary ~ operator), which in turn stems from an independence assumption in our probabilistic model. Note, however, that this does not distort the relative ranking of the results; the situation is similar to that of Web search engines, and we could as well have applied their standard trick of re-normalizing the scores in the final result list.

Where clause of	*#*	*Runtime (sec)*		*Relevance*	*Example results*
query	*results*	*td*	*bu*	*Score*	
~headline	2531	0.1	0.09	1.0	headline="DBS" (book.xml)
				0.8	title="The Quran" (quran.xml)
				0.8	title="The New Testament" (nt.xml)
				…	…
				0.6	subhead="The Song" (a_and_c.xml)
~publication. ~headline	1509	0.45	0.2	0.8	publication.heading="…" (publication.xml)
				0.64	book.title="…" (book.xml)
				0.64	article.title="…" (SigmodRecord.xml)
				…	…
				0.48	work.heading="…" (book.xml)
~headline ~ "XML"	36	2.5	2.4	0.208	Heading="XML-QL: …" (publication.xml)
				0.108	headline="DBS" (book.xml)
				0.108	title="XML and …" (SigmodRecord.xml)
				…	…
				0.082	heading="Draft…in Java" (book.xml)
~publication. ~headline ~ "XML"	35	0.83	2.5	0.208	publication.heading="XML-QL:…" (publ.xml)
				0.083	article.title="XML and …" (SigmodRec.xml)
				0.083	article.heading="Adding…XML" (article.xml)
				…	…
				0.049	work.heading="Draft…in Java" (book.xml)

Table 1: Experimental results for XXL similarity search (td=top down, bu=bottom-up)

6.2 Queries with Path Expressions

Table 2 shows the results for more sophisticated path expressions in the Where clause of a query. All run-times were in the order of seconds; so the XXL search engine is fairly efficient. Note that even queries whose Where clause starts with a path wildcard # (e.g., the second query and the fifth query in Table 2) could be evaluated with reasonably short response time. This is because the XXL query processor first evaluates the specified element names in the path pattern and only then tests whether the retrieved paths are connected to the roots specified in the From clause.

In this set of queries the bottom-up evaluation strategy mostly outperformed the top-down strategy. This is because the element names at the end of the path patterns tended to occur less frequently in the underlying data, so the corresponding elementary conditions were more selective. For queries with multiple subqueries either all subqueries were evaluated top-down or all of them were evaluated bottom-up.

For the example queries of Table 2 we measured the number of calls to the various index structures, and for the ECI and OI, which are based on Oracle tables, also the number of DBS calls that resulted from the index methods. The number of DBS calls, which were simple SQL queries with many Fetch calls, incurred significant overhead, simply by having to cross a heavy-weight interface many times, in some queries more than 100 000 times. With caching of index entries enabled in the query processor, the number of DBS calls were drastically reduced from down to the order of a few hundred calls. This optimization resulted in acceptable response times shown in Table 2.

Where clause of query	*# results*	*Run-time (sec)* *top-down*	*bottom-up*
~headline	2531	0.10	0.10
#.~headline	11335	19.80	2.88
~author ~ “King”	10	0.43	1.61
~author AS A AND A ~ “King”	10	0.51	0.51
#.~author AS A AND A ~ “King”	60	16.21	0.83
~headline ~ “XML”	36	2.45	2.45
~headline AS A AND A ~ “XML”	36	2.51	2.51
#.~headline AS A AND A ~ “XML”	165	20.19	6.49
~title ~ “Testament”	2	0.70	1.37
~title AS A AND A ~ “Testament”	2	0.77	0.77
#.~title AS A AND A ~ “Testament”	6	7.72	1.22
~figure ~ “King”	62	3.66	4.21
~figure AS A AND A ~ “King”	62	3.68	3.76
#.~figure AS A AND A ~ “King”	190	18.15	4.42
#.(pgroup)?.~figure ~ “King”	190	20.79	3.69
#.~chapter.(v|line) ~ “Testament”	52	8.71	2.65

Table 2: Experimental results for XXL queries with path expressions

6.3 Complex Queries

Finally we considered two more complex queries:

```
Query 1: Select * From Index
         Where #.~publication AS A
           And A.~headline ~ “XML”
           And A.author% AS B
           And B.title AS C

Query 2: Select * From Index
         Where #.play AS A
           And A.#.personae AS B
           And B.~figure ~ “King”
```

Table 3 shows the run-times for these more challenging queries, which were again in the acceptable time range of a few seconds, provided the proper evaluation strategy was chosen. In addition to the top-down and bottom-up strategies for the local evaluation order, we also tested an optimization heuristics, coined *start optimization (so),* for automatically deciding on the most selective node of the query graph based on statistics about the frequency of the corresponding element name(s) in the indexed data. Once the most selective node is determined the evaluation proceeds both ways, top-down and bottom-up, from this node; with the top-down direction being inspected first, we refer to this strategy as td+so, otherwise we refer to it as bu+so. Furthermore we use a simple heuristic (opt. heur.) algorithm that chooses the evaluation order of the subqueries, i.e., 2,1,3 for query Q1 and 1, 2, 3, 4 for query Q2, and the local evaluation strategy (t=top down or b=bottom-up) for each subquery individually. In Table 3 we give the actual query execution times (exec) and the plans (plan) obtained from the optimization. The optimization time itself was about 0,3 sec for each of the two queries. The heuristics are based on considering the syntax and the selectivity of all start nodes and end nodes of each subquery. For example, as the first subquery of Q1 starts with # and ends with a similarity condition involving the relatively frequent term “publication”, it is considered less selective than the second subquery; so our heuristic optimizer decides to evaluate the second subquery first.

	#	td	bu	td + so	bu + so	opt. heur.		opt.heur. + so
	results	(sec)	(sec)	(sec)	(sec)	exec (sec)	plan	exec (sec)
Q1	131	14,30	694,62	14,20	3,24	2,69	**2**bu,**1**bu,**3**td	2,68
Q2	58	8,56	3,76	8,52	3,69	4,63	**1**bu,**2**td,**3**td,**4**td	4,64

Table 3: Experimental results for complex XXL queries
(td = top down, bu = bottom-up, so = start optimization)

7. Ongoing and Future Work

Ongoing and future work includes a more advanced kind of ontology index, performance improvements of the query processor algorithms, and the broad and challenging issue of query optimization. More specifically, we plan to extend the OI so as to capture more semantic relationships between concepts. This also requires extending our similarity metric. As for query evaluation performance, we are considering various heuristics for finding some good approximate matches in the result list as quickly as possible with possibly deferred or slowed-down computation of the (nearly) complete result. This way we strive for the best ratio of retrieval effectiveness (i.e., search result quality) and efficiency (i.e., response time). Finally, we plan further studies on query optimization heuristics for both global and local evaluation ordering of subqueries and elementary search conditions based on selectivity estimations.

References

[ABS00] S. Abiteboul, P. Buneman, D. Suciu: Data on the Web – From Relations to Semistructured Data and XML. Morgan Kaufmann Publishers, 2000.

[BAN+97] K. Böhm, K. Aberer, E.J. Neuhold, X. Yang: Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM, VLDB Journal Vol.6 No.4, Springer, 1997.

[BP98] S. Brin, L. Page: The Anatomy of a Large Scale Hypertextual Web Search Engine, 7th WWW Conference, 1998.

[BR99] R. Baeza-Yates, B. Ribeiro-Neto: Modern Information Retrieval, Addison Wesley, 1999.

[BR01] T. Boehme, E. Rahm: XMach-1: A Benchmark for XML Data Management. 9th German Conference on Databases in Office, Engineering, and Scientific Applications (BTW), Oldenburg, Germany, 2001.

[CK01] T. T. Chinenyanga, N. Kushmerick: Expressive and Efficient Ranked Querying of XML Data. 4th International Workshop on the Web and Databases (WebDB), Santa Barbara, California, 2001.

[Coh98] W.W. Cohen: Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity, ACM SIGMOD Conference, Seattle, Washington, 1998.

[Coh99] W. W. Cohen: Recognizing Structure in Web Pages using Similarity Queries. 16. Nat. Conf. on Artif. Intelligence (AAAI) / 11th Conf. on Innovative Appl. Of Artif. Intelligence (IAAI), 1999.

[CSM97] M. Cutler, Y. Shih, W. Meng: Using the Structure of HTML Documents to Improve Retrieval, USENIX Symposium on Internet Technologies and Systems, Monterey, California 1997.

[FG00] N. Fuhr, K. Großjohann: XIRQL: An Extension of XQL for Information Retrieval, ACM SIGIR Workshop on XML and Information Retrieval, Athens, Greece, 2000.

[FK99] D. Florescu, D. Kossmann: Storing and Querying XML Data using RDBMS. In: IEEE Data Eng. Bulletin (Special Issues on XML), 22(3), pp. 27-34, 1999.

[FKM00] D. Florescu, D. Kossmann, I. Manolescu: Integrating Keyword Search into XML Query Processing, 9th WWW Conference, 2000.

[FM00] T. Fiebig, G. Moerkotte: Evaluating Queries on Structure with Extended Access Support Relations. 3rd International Workshop on Web and Databases (WebDB), Dallas, USA, 2000, LNCS 1997, Springer, 2001.

[FR98] N. Fuhr, T. Rölleke: HySpirit – a Probabilistic Inference Engine for Hypermedia Retrieval in Large Databases, 6th International Conference on Extending Database Technology (EDBT), Valencia, Spain, 1998.

[GW97] R. Goldman, J. Widom: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases, Very Large Data Base (VLDB) Conference, 1997.

[HTK00] Y. Hayashi, J. Tomita, G. Kikui: Searching Text-rich XML Documents with Relevance Ranking. ACM SIGIR 2000 Workshop on XML and Information Retrieval, Greece, 2000.

[Kl99] J.M. Kleinberg: Authoritative Sources in a Hyperlinked Environment, Journal of the ACM Vol. 46, No. 5, 1999.

[Kos99] D. Kossmann (Editor), Special Issue on XML, IEEE Data Engineering Bulletin Vol. 22, No. 3, 1999.

[KRR+00] S.R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, E. Upfal: The Web as a Graph, ACM Symposium on Principles of Database Systems (PODS), Dallas, Texas, 2000.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. SIGMOD Record, 26(3): 54-66 (1997).

[MJK+98] S.-H. Myaeng, D.-H. Jang, M.-S. Kim, Z.-C. Zhoo: A Flexible Model for Retrieval of SGML Documents, ACM SIGIR Conference on Research and Development in Information Retrieval, Melbourne, Australia, 1998.

[MWA+98] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, A. Rajaraman: Indexing Semistructured Data. Technical Report 01/1998, Computer Science Department, Stanford University, 1998.

[MWK00] P. Mitra, G. Wiederhold, M.L. Kersten: Articulation of Ontology Interdependencies Using a Graph-Oriented Approach, Proceedings of the 7th International Conference on Extending Database Technology (EDBT), Constance, Germany, 2000.

[NDM+00] J. Naughton, D. DeWitt, D. Maier, et al.: The Niagara Internet Query System. http://www.cs.wisc.edu/niagara/Publications.html

[Ora8i] Oracle 8i interMedia: Platform Service for Internet Media and Document Content, http://technet.oracle.com/products/intermedia/

[Ra97] Raghavan, P.: Information Retrieval Algorithms: A Survey, ACM-SIAM Symposium on Discrete Algorithms, 1997.

[SemWeb] World Wide Web Consortium: Semantic Web Activity, http://www.w3.org/2001/sw/

[TW00] A. Theobald, G. Weikum: Adding Relevance to XML, 3rd International Workshop on the Web and Databases, Dallas, Texas, 2000, LNCS 1997, Springer, 2001.

[WordNet]http://www.cogsci.princeton.edu/~wn

[XLink] XML Linking Language (XLink) Version 1.0. W3C Recommendation, 2001. http://www.w3.org/TR/XLink/

[XMLQL] XML-QL: A Query Language for XML, User's Guide, Version 0.9, http://www.research.att.com/~mff/xmlql/doc

[XPointer] XML Pointer Language (XPointer) Version 1.0. W3C Candidate Recommendation, 2001. http://www.w3.org/TR/xptr/

[XQuery] XQuery 1.0: An XML Query Language. W3C Working Draft, 2001. http://www.w3.org/TR/xquery/