Fusing Structural Properties to Node and Path Constraints in XML Path Retrieval

Gildas Ménié, Pierre-François Marteau and Nicolas Bonnel

1 Université de Bretagne Sud, 56000 Vannes, France

Abstract. We describe a path approximate search process based on an extended edit distance designed to manage ‘don’t care characters’ (*) with variable costs (\(\alpha\)-\(\beta\)) in a path matching scheme extending XPath. The structural path is extended to conditional properties using variables whose values are retrieved through a backtracking processed on the editing distance matrix. This system provides a dedicated iterator for a XML query and processing scripting language that features large XML document collection management, joint joins and extraction features.

Keywords: XML, data management, query and processing language, approximate search, editing distance, XPath, XQuery

1 Introduction

Many XML systems use path processing instead of tree matching, both to lower the processing cost and because the fragment [13] approach is often enough satisfying.

Some users may not have enough knowledge about the structural properties of the documents and semi-structured documents, it seems acceptable to provide an approximate search scheme that does not strictly rely on exact path matching, but also matches semi-matching. Many extensions have been made to XQuery [32] or to XPath, from FleXQuery [4, 7], XQuery/IR [8], ELIXIR [9], XIRQL [14], XXL [25], Xightly [26] and XIRCUS [20] for instance) to meet partial matching. FleXPath [3] is one of the most interesting extensions proposed to the field. XML-QL [30] is a recent W3C standard for query that involves both an exact matching and an approximate search extension proposed by INRIA. One of the most delicate things to handle is the expressiveness of the request language: it should help the user to define precisely his request, where the approximation is acceptable and where it is not; then express requests that vary from ‘very’ approximate to ‘strict’ exact in the same document base discovery. In this view, a query language should be
able to morph from an exploring tool to a strict search engine for content and structure [5].

Many string processing have been studied in the field of path matching. Many of them, dealing with approximate string matching have been used to provide approximate path search. In the Levenshtein [17] editing distance for instance, a global cost can be used to compute a distance. This has been studied for approximate path matching in XML search [19] and leads to encouraging results [22] – even if the computation of joined XML match and attribute conditions remains delicate to express as editing costs. In this scheme, it is uneasy to express which part of the path should strictly match and which part doesn’t have to.

The use of suffix trees [29] for efficient indexing or management of ‘*’ or ‘don’t care characters’ is another way to manage imprecise path matching [10]. The expression of unknown regions lays then in the use of these ‘*’, the other parts having to match strictly. This kind of algorithm has been studied extensively [12].

In the part 2 ‘problem’, we will state the problem of path search in relation to elements and attributes properties. Then, in part 3 ‘Q’Path structural distance computation’, we describe a matching operator able to express local approximation by the mean of both ‘variable length don’t care’ and editing distance operators. We present the algorithm used beneath this process. In part 4 ‘Retrieval of structural bonds: S[0]’, we show how the binding to DOM Tree nodes property constraints is managed. We explain an indexing process to deal with this search and introduce then (in part 5 ‘Processing System’) the Alpha system which features this dedicated distance and t-uple assignment. We introduce search and processing results and then conclude.

2 Problem

In the common query languages, most of the semi structured requests involve both a path and properties on the elements in the path. These properties can be seen as constraints or conditions on attributes or words that may be found in the node or path context. This is the case for instance for XPath [31], or NEXI language [26]. Most of the time, the search involves an inverted list which gives access to the list of paths that satisfies the conditions. This approach achieves a high speed retrieval but cannot always apply when only structural properties are formulated (an inverted list entry is not always provided).

In this work, we focus on the search of a node structure first, and then, on the fulfillment of conditions (if any) on the node unification: this ordering may not be always optimal in specific cases but it ensures that any conditions can be expressed and generalized.

Each XML document ‘d’ can be depicted as a tree $T_d$ in which each node $N(T_d)$ is represented by a tag – or a XML element.

Let Path($T_d$) be a path of $T_d$, also a sequence of Nodes $N(T_d)$.

Let $S_{path}(T_d)$ be the set of all possible path in this tree $T_d$. 

- 47 -
Let $SC_{path}(C)$ be the union of all $S_{path}(T_k)$ for $T_k$ belonging to a collection $C$ of XML documents.

Let $S_{prop\_node}(N)$ be the set of all computable properties $Prop\_node(N)$ on a node $N$. This set contains (for instance) properties on attributes (an attribute exists or not), on attribute values, on text (if the node has some textual data of course).

As stated before, a path $Path(T_0)$ is a ‘word’ built on the set of ‘letters’ $D(T_0)$ of the elements nodes (empty or not) of $T_0$. Thus, any path can be written as a sequence $N_0/N_1/\ldots/N_i/\ldots N_{n\_end}\ N_i \in D(T_0)$. 

In the same way as for a node, a path $Path(T_0)$ has some properties. For instance, its length can be computed. As the path belongs to a specified document, this document’s name or path can be found etc.

Let $S_{prop\_path}(P)$ be the set of all computable properties $Prop\_path(P)$ for path $P$.

Let $S_{Path}$ be the description of a subset of $Path$ in (C): A path $P$ belongs to $S_{Path}$ if, and only if, it features the conditions described in $QPath$. $QPath$ is a sequence of conditions $Cond()$ both on node - $Cond_{node}(N)$ - and on path/ subpath - $Cond_{path}(N_i/N_i+1/\ldots N_j)$ -.

$QPath$ is practically speaking a pattern with conditions on the related parts of the path. For instance, $QPath_1 = \{ node \ with\ name=\"article\"\}/ [ a \ path \ with \ 3 \ undefined \ elements ] \ / [ node \ with\ attribute\ date ]$ 

is a sequence of three conditions, the first and the last involve properties on nodes, whereas the second involves a property on a sub path. For the following document (and the following nodes):

```xml
<article author="bob">
  <document>
    <chapter num="3">
      <text>
        <paragraph date="3 sept 2000">
        </paragraph>
      </text>
    </chapter>
  </document>
</article>
```

The following path would belong to $S_{Path1}$:

```
/document/chapter/text/paragraph
```

The subpath `/document/chapter/text` has a length of 3. `<article>` has a property ‘name’ (its own name) with a value of ‘article’. `<paragraph>` has an attribute named ‘date’.

In the most obvious cases, the set of properties for a node is strongly related to the node’s name. If there is an imprecise match for an element’s name, there will probably be a bad matching for its attributes names, and of course, values: but even if there is a perfect match for the element’s name, one cannot systematically assume that
the semantic of the element is the same. Nevertheless, we made the assumption that
the most critical approximation in such a path remains the node’s name. Thus, we
focus on the name’s property first.
As for the sub path, since a sequence of nodes ‘inherits’ a sequence of nodes
properties, we focus on a subset of path specific features, and especially on the length
feature.
For a given QPath, we also want to find an approximate set of path-solutions that
match ‘as best as possible’ the conditions Cond, with eventually a rejection limit
given by the user.
In the scope of these assumptions and restricting the QPath to the features we want to
focus on, we introduce the following sub path condition *i–j that means ‘any
sequence of undefined elements that have at least i elements and at most j elements’.
We also define its extension: * or *0–∞ (respectively for a path with zero elements
and for any number of undefined elements), ? or *1–1 (one undefined element, or an
undefined path of length 1). (Compared to XPath for instance, the // axis would be
expressed as *0–∞. The *1–1 should match the * from XPath)
Thus, QPath1 can be rewritten as QPath1:

QPath1 = [article] [*2–3] [node with attribute ‘date’]
Or
QPath1 = [article] [*2–3] [node := ? (node has an attribute ‘date’)]
Or
QPath1 = [article] [*2–3] [thisnode := ?]
with the condition (thisnode has an attribute ‘date’)

Under these assumptions, QPath1 becomes a sequence of structural properties in
[element’s name, *i–j] with a set of local conditions (in the example, see after with the condition). The structural properties are bound to the condition, using references –
here ‘thisnode’ is a reference bound to the node found for the match of ‘?’.
So we practically split QPath into QPathstructural(SB).QPathconditions(SB) where SB is
the set of bounding references. In this example SB = \{bound (thisnode,?)\}

In the example provided above,

QPath1 =

QPathstructural({bound(thisnode,?)}) . QPathconditions({bound(thisnode,?)})
with

QPathstructural({bound(thisnode,?)}) = [article] [*2–3] [?]

and

QPathconditions({bound(thisnode,?)}) = thisnode has an attribute ‘date’

To evaluate a match between a path candidate - Pcandidate - and QPath, we intend
to find at first an approximate matching of the structural properties QPathstructural,
retrieve the structural bonds SB (ie solve the unification problem of the bound name
with the value found) and then deal with the conditions QPathconditions on SB.
In the following part, we’ll introduce the algorithm used to perform the match and distance evaluation to find the best structural match. This algorithm is based on an editing distance computation, designed to take the $^*i-j$ into account.

3 Q’Path$_{structural}$ Distance Computation

3.1 Editing Distance with $^*i$-j

The editing distance has been extensively studied [12] and many algorithms have been developed to compute a distance as the sum of elemental editing [17] cost (mainly insert, delete or replace) to get from a string to another string [16]. This involves mainly the computation of a complete or partial cost matrix, thus leading to a complexity of $o(nm)$ with $n$ and $m$ the length of the two strings. Unlike more traditional approaches, the editing distance takes into account the insert operator that may be discarded in Hamming distances for instance.

The use of * has been also extensively studied and many solutions [10, 20] have been developed and lead to high performance algorithms. These solutions deal mainly with exact match of part of strings (separated by *) and don’t fit well to our problem.

We intend to use an implementation of an editing distance merged with variable length * management that takes into account a ‘don’t care path’ extension ($^*i$-j). We use then this *-editing distance to solve the Q’Path$_{structural}$ problem stated above.

Few works [18, 34, 1] have addressed the merging of the two problems. In our case, the approximation has also to take into account a possible error on the $i$-$j$ extends provided by the user. This error, as insertion or deletion (but of course no substitution) should be expressed as extra or missing symbols to fit into the $i$-$j$ interval.

The cost computation in the (standard) editing distance matrix involves the evaluation of a minimal cost path at each position $(o, p)$. For standard editing distance, the cost is computed as follows (1):

$$\text{Dist}(o,p) = \min\{ [o-1,p]+1, \ [o,p-1]+1, \ [o-1,p-1]+\text{sub}(o,p) \}.$$  \hspace{1cm} (1)

where sub(o,p) is 0 if the current string character (position o) matches the pattern character (at position p), 1 otherwise. In standard editing distance, the pattern symbol is a letter. In our case, it may be a $^*i$-j. In this last case, the computation of the Dist(o,p) involves the computation of a minimum cost for the set of hypothesis unfolding the set of $j-i$ possible match. Let call ‘?’ the ‘any’ symbol that can stand for $^*1-1$. This special symbol has no substitution cost for any symbol (character in the case of a string) of the string to match. So the computation of the matrix related to $^*i$-j involves the computation of the collection of cost sub matrixes with i occurrences of $?, \ (i+1)?, \ (i+2)?, \ ...(j-1)?, \ j?$. Then these sub matrixes are merged to provide the path with a minimal cost possible.

A starting inserting cost very high for the pattern $^*i$-j has to be set to prevent the algorithm to discard the * constrain (inserting) ‘$^*i$-j’ in the string to analyze.
D[] is defined as an array of integers;
Q'Path(structural) is an array of symbols that contains element’s names
or ‘*’ symbols
P(candidate) is an array of symbols that contains only element’s names:
it codes a path that could match the pattern Q'Path(structural).

n = length of P(candidate); m = length of Q'Path(structural);

ip=1; while((ip<=m) and (Q'Path(structural) [ip-1]<>'*')) ip++;
// finds the position ml of the first ‘*’ pattern side
if (ip<m) ml = ip; else ml = ¥; // ml is this start, ¥ means no ‘*’
for(jp=0; jp<n; jp++) D[0][jp] = 3p; // initialize the cost matrix
// for this column, this is the same as for a standard cost matrix
jp = MIN((ml-1),m); for(ip=0; ip<jp; ip++) D[ip][0] = ip;
// This computes the cost for the first row: it takes into account
// the don’t care characters
for(ip=1; ip<=m; ip++) { // cost matrix computation main loop
  if (Q'Path(structural)[ip-1]=='*i~j') { // a ‘*i~j’ in the struct.
    for(jp=0; jp<n; jp++) D[ip][jp] = ¥;
    // prevents the discarding of ‘*’ - using a high cost.
    // the following inner double loop computes the result of the
    // merging of all the cost sub matrix computed for each hypothesis of
    // unknown symbols
    for(jp=0; jp<n; jp++) {
      for(h=(jp+i); h<=(jp+j); h++) {
        // merges the unfolded computations for hypothesis of
        // i ?, or i+1.? or i+2.? etc to j.? 
        // only keeps the minimal path cost:
        if ((h<= n) and (D[ip-1][jp]<D[ip][h]))
          D[ip][h] = D[ip-1][jp];
      }
    }
  } else { // a legal element in the structural pattern (< *')
    // Here we have the common editing distance algorithm
    for(jp=1; jp<=m; jp++) {
      D[ip][jp] = MIN( D[ip-1][jp]+1 ,
                       D[ip][jp-1]+1 ,
                       D[ip-1][jp-1]+
                       SUBST(Q'Path(structural) [ip-1], P(candidate) [jp-1]) );
      // SUBST is the cost (0 or 1) of the substitution of an element of the
      // candidate by an element of the pattern
    }
  }
}
return D[m][n]; // this value is the minimal path cost computed
Given a Q’Pathstructural and a path candidate Pcandidate, this pseudo-editing distance computation scheme provides a minimal cost or number of transformation to get from the path Pcandidate to the Q’Pathstructural taking into account the *i–j symbols. Note that this isn’t really a distance anymore since it lacks symmetrical feature.

The user may provide a maximum distance allowed: in this case, the computation is aborted as soon as the minimum distance computed reaches this limit. In this case, the pseudo-distance is ∞ (Pcandidate is rejected).

When no ‘*’ is present in Q’Pathstructural, then the above algorithm reduces to the standard editing distance algorithm of O(nm) complexity. In the worst case, when Q’Pathstructural is only made of ‘don’t care characters’ *0→∞ (which is not realistic since it can reduce to a single *i–j) then the complexity is of O(m.n.max(n,m)). So if the length of Path is of order L, the complexity is under O(L^3).

Using a suffix tree to sort the ‘don’t care column’ and merge the ‘unfolded *’, it is possible to simplify the inner double loop, reducing this complexity to O(L^2.log(L)). In the worst case, the mean length of path is 8-10, so in the very worst cases, L^3 is between 512 and 1000 and L^2.log(L) is between 60 and 100, that is acceptable since the most important time cost for the processing of nodes remains the storage input/output bottleneck of the implementation.

3.2 Indexing and Search Process

In order to reduce the processing of this editing distance, we have to put aside the path candidates that won’t obviously match the search. In this view, we use a path signature to prune the path candidates search. What we call a path signature is actually a string made of the sequence of the first letter of each element in the path. For instance, the path [article/document/chapter/part] has the signature Signature(article/document/chapter/part)="adcp". We proceed similarly for the Q’Pathstructural that provides a kind of pattern signature: For instance, [article] [*2–2] [*?] has the pattern signature ‘a*2–2?’ – where ‘*2–2’ is considered as one signature’s symbol – just as ‘a’.

We use a similar editing-distance to compute a pseudo distance between ‘adcp’ and ‘a*2–2?’. Of course, one cannot assume that the result would be the same as for the full path computation even if the * are legally taken into account (as a character stands for an element). In fact, if there is a difference between a first letter and its ‘first-letter’ counterpart in pattern, then, one can be sure that the ‘extended’ element’s name are different (as their first letters are not the same). This obviously rejects the element’s names that differ only by the first letter. Hence:

Pdistance(Q’Pathstructural, Pcandidate) ≥ Pdistance(Signature(Q’Pathstructural), signature(Pcandidate)).

SCpath(C) (the set of all possible paths in the base) is clustered between classes represented by the different signatures. When a search is performed, a pseudo distance is computed between each of the class signatures and the signature of Q’Pathstructural. Given a maximum acceptable pseudo distance MaxPdistance, it is possible to discard all the Pcandidate that belong to class which signature has been rejected.
4 Retrieval of structural bonds: $S^B$

After a pseudo-distance between a QPath$_{structural}$ and a P$_{candidate}$ has been computed, the structural/conditional bond references must be retrieved to evaluate the conditions QPath$_{conditional}(S^B)$. For instance, in `{article}[*3-3] [thisnode := ?]` the element matching “?” should be retrieved and referenced by ‘thisnode’. ‘thisnode’ is then used as an alias/variable to evaluate conditions (if expressed) on the ‘?’ element.

The matrix used by the *-pseudo-editing distance computation is stored and used to backtrack the possible matching paths. The algorithm starts by the last value computed and back-searches each of the possible paths leading from the first cell (top-left) to the bottom-right cell. This algorithm takes into account the (i,j) values for each of the ‘don’t care character *’. Each diagonal move $(x,y)$-$>(x+1,y+1)$ relates to a substitution between the member x of the pattern (an element or a *i-j) and an element of the path candidate.

If the member ‘x’ of the pattern has a bond reference (‘b:’ in …/b:x/…), then this reference is linked to the substitution retrieved – or to the sequence of substitutions for a path.

For instance: `/article/document/*/chapter/?!` has no bonds. Thus, no retrieval has to be performed since no condition QPath$_{conditional}$ can be expressed.

`/first:article/document/list/*/chapter/element/?!` has 3 bonds {bound(first,<article>), bound(list,*), bound(element,?)}.

Each matching path can be translated into a path of node references (that points to an entry in the index base). For a given acceptable path P$_{candidate}$, it is hence possible to find node reference values for first, list and element. In this above example, first and element are related to nodes, whereas list is a reference of a list of node references (eventually empty).

This backtracking process generates all the possible sets of values for {first, list, element}: for instance this may be {((first->node324, list-> node345/node654/node4456, element->node45), (first->node32, list->empty, element->node456))} is an example of set of substitutions computed for a given cost matrix.

Because the conditions have to be evaluated on the nodes properties pointed by these bonds, no errors are allowed on elements that carry a bond. This is a way to specify, in the QPath$_{structural}$, whether an error should be acceptable or not (it is legal to use a bond without any further condition: this rejects the P$_{candidate}$ with a substitution cost on this element).

This processing is only performed on the final accepted P$_{candidate}$. The retrieving of the tuple of variables is also of a complexity under $o(L_{path}.log(L_{path}))$, with $L_{path}$ the length of a path.
The final evaluation of a path is performed on the resolved bonds, using the properties $\text{prop}_\text{node}(\text{Node})$ for the nodes or/and $\text{prop}_\text{path}(\text{Path})$ for the bonds relating to list of nodes (such as ‘list:*2–4’).

5 Processing System

Many standards of query languages for XML are and have been proposed with few implementations actually available. Each of them has unique features to query and process but few can scale to a large amount of documents (see Exist for instance [33]).

We decided to design our own system, similar to XQuery (and XPath), but able to manage a large collection of document, both for the search, navigation, extracting and processing. The system (called Alpha [2]) manages a cluster of computers for the indexing and the retrieving of a large collection of documents (the system has been tested with 10 Gb XML documents). The indexing scheme uses several QDBM databases on disk [26] as well as a client/server built in Visual Prolog [23, 28].

The scripting language – alphaScript - provides many access levels starting with low-level Document Object Model access. It provides string processing functions, file access function, internet functions as well as a set of functions designed for information retrieval. It features many kinds of iterators to manage a file, a collection of file or index databases. The scheme described in this paper is used to provide a node level location access inside an iterator designed for processing and extraction.

The scripting language describes the processing to be performed in a procedural way. This processing can involves document management as well as indexing, XML document generation from other databases or from Internet.

For instance, \_system.processdir("xml","*.xml","memidx"); triggers the indexing of the collection of XML files in the directory xml to build an index database called memidx.

The following example features a script that generates a new XML document using an already indexed database ‘memidx’:

```xml
_io.setExtractFile("travel92.xml");
_io.extract("<?xml version='1.0' encoding='UTF-8'?>");
_io.extract("<travels>");
foreach /a:article/*2~3/b:chapter/txt/part/abstract/c:? max 1 in "memidx"
   where ( a.date>"1992")
   and (b.num==5)
   and (c.containsWord("travel"))
   {
      _io.extract("<desc document="", a.sourceFile,"">", c.text, "</desc>");
   }
_io.extract("</travels>");
```
This script produces a new XML document with information extracted from the index database `memidx`. It is a feature already offered by (for instance) XML-QL, or XSLT. It is possible to produce any kind of text document (HTML, SVG or XML) and have the script produce another index database from the collection of documents made. This way, starting for instance with a collection of document on movies with all details on actors, soundtrack etc, it is possible to generate or extract a new XML collection of document dedicated to the actors only and build a more specific index database for faster search and processing.

Note that each variable is a reference of the corresponding DOM node: it is hence possible to gain access to the node properties. For instance, the variable ‘c’ is set to the matching element for ‘?: c.name gives also the name of the unknown element. Unlike XQuery (but like in XML-QL [30]) it is possible to express queries based on the name (or attributes) of elements and not only on their contents.

This XPath-like `foreach` is not limited to non branching path management: not only it is possible to use the DOM-like access on the nodes (scanning thru any connected node), but `alphaScript` supports also joint-like operations on nodes as well (or based on any node or path properties):

```xml
foreach /*/root1:?/aaa in "idxmem" {
    foreach /*/root2:?/bbb in "idxmem" where (root1==root2) {...}}
```

describes two elements, aaa and bbb that have the same ancestor.

```xml
foreach /path:*/aaa in "idxmem" {
    foreach /*/root2:?/*/bbb in "idxmem" where (path.contains(root2)) {...}}
```

constrains bbb to have one ancestor that belongs to the list of ancestors of `aaa`. A cache evaluation system prevents a full scan to be performed for each loop: it is also possible to use many levels of joint-processing without lost of performance.

In the example above, only the `idxmem` index database is used, but each `foreach` doesn’t have to be performed on the same index database: it is therefore possible to perform joint on different index databases, unlike most actual XML query languages. This can be used to split or merge different index databases to merge or produce new XML documents or even index databases.

The XML Wikipedia [11] and the INEX2005 DB has been used to evaluate the system. The index size is 6.5 times the size of the XML collection. The indexing speed is of 3Mib mn of source data on a P4 3 GHz. For each node and path, an entry in a QDBM hashable on hard drive is created and a defined set of $S_{prop\_node}(N)$ (respectively $S_{prop\_path}(P)$) is added. The signatures of the paths are computed and inverted lists of path references are created using the QDBM’s depot model. For precise requests (ie – with no *) and requests of a length of 8, 87% of $S_{path}(C)$ is discarded by the signature’s test for $MaxP_{distance}$=0. Of course, for ‘very’ imprecise requests (for instance ‘/*’), none of the $P_{candate}$ are discarded (as awaited).

For the backtrack computations and paths of length n, experimentally we found that the mean operation’s number stays under 5.n (under 40 D[]-cell access for a pattern/path mean length of 8): that remains quite acceptable in regards to the matrix computation complexity.
Since a request can provide a large amount of solutions and even enumerate all the paths in the database (see ‘foreach /a:*’ for instance), we implemented an asynchronous interface system to provide the answers to the user as soon as they are available. The mean processing speed is of 2500 nodes evaluated per seconds. The answer speed depends of course on the complexity of the request and processing. When no processing is involved, a maximal distance of 0, and don’t care characters used (such as ‘foreach /article/chapter/b:text/ where (b.containsWord("travel"))’), the system returns the answers within a tenth of second. The use of length undefined ‘don’t care’ symbols slows down the search, as well as the use of a maximum accepted distance. For instance, the use of a path of length 10 with 2 ‘*’ (no min/max) and with a max distance of 2 leads to a mean answer time of 4s on the whole database (between 2s and 6s). Using min/max for the don’t care characters speeds up the search, because more signatures are rejected – and more path candidates as well.

6 Conclusion

We introduced a mechanism based on a dedicated editing distance and a t-uple matching process to address the matching of imprecise paths. This computation is used in a scripting language (alphaScript) to retrieve DOM node references in an approximate way. The assignment of the t-uple of variables is used to evaluate condition and to perform joints filtering in the same index database or even between two (or more) databases. This crafts one of many iterators used in the script language of alpha, a system designed to manage a large collection of XML documents.

References

2. Alpha : http://www.arsanit.com
13. Fragment Description: [http://www.w3.org/TR/WD-xml-fragment#terminology](http://www.w3.org/TR/WD-xml-fragment#terminology)
23. Prolog Development Center (PDC): [http://www.pdc.dk](http://www.pdc.dk)
30. XML-QL: [http://www.w3.org/TR/NOTE-xml-ql/](http://www.w3.org/TR/NOTE-xml-ql/)
31. XPath Reference: [http://www.w3.org/TR/xpath](http://www.w3.org/TR/xpath)
32. XQuery Reference: A Query Language for XML. Feb 2001 [http://www.w3.org/TR/xquery](http://www.w3.org/TR/xquery)