Supporting regular expressions in queries to distributed XML documents

© Dmitry Barashev

University of St. Petersburg, Russia
db2@acm.org

Abstract

XML data are often distributed over the heterogeneous networks. The modern technologies like Web services allow to build a distributed virtual document spread over cooperating network nodes. The goal of this research is to provide regular expression search facilities in such documents. Construction of a network model of a distributed XML document and a process of queries evaluation are described.

1 Introduction

Evolution of XML [17] related technologies increasingly opens the new levels of applications development. A flexibility of semistructured nature of XML, rich schema definition facilities [15] supported by the numerous world accepted XML applications, ease of XML document transformations [16], provide a good environment for development of flexible cooperating applications. The appearance of Web services [10] was a great step towards rapid building of applications able to work both in intranet and internet environments.

Applications dealing with XML data usually need to store XML documents in databases like any other data. They also would like to query stored documents using one of the XML query languages like XPath [18] or XQuery [19]. Modern database systems support various XML storage and querying options. The industrial relational DBMSs feature decomposition of XML documents into a set of relational tables, storage of XML documents as LOBs, exposition of relational data in the XML format and SQL language extensions which enable querying XML data [6, 9, 12]. Querying facilities are also implemented in so called "native" XML databases which store XML data using some proprietary storage techniques [1].

Efficient evaluation of queries of the locally stored XML data has been a hot research topic over the last several years. A number of papers investigate opportunities of relational databases to store XML documents [3, 13]. Significant efforts have been also dedicated to development of native XML databases [8]. During the last two years, several papers introduced numerous indexing schemes based on the well known secondary index structures [7, 5, 2].

However, the growth of distributed XML data is a new challenge for database community. Perhaps, very soon the data exposed by various Web services running on different software and hardware platforms ranging from high-end servers to mobile devices will be considered as a virtual distributed XML document which also needs to be searched and queried efficiently.

Evaluation of the queries to the distributed semistructured data has some specifics which make traditional indexing and query evaluation techniques relatively inefficient. The operation costs change; navigation between nodes of document becomes the most expensive operation while disk access costs in the local processing become cheaper. Paths which share the same prefix may finish on different network nodes. Different parts of a single document may lack any directly represented relatioships.

The problem of the distributed query evaluation on semistructured data was studied in [14]. We considered this work as a starting point of our research.

This paper is organized as follows: in section 2 we introduce a network model of a distributed document. Section 3 introduces some criteria of a query evaluation algorithm efficiency and in sections 4.1, 4.2 we describe query evaluation algorithms able to answer to regular expression queries in different kinds of distributed documents. In section 5 we provide an analysis of the algorithms with respect to the efficiency criteria.

2 Network model of the document

We will use the definition of a distributed document which closely resembles a definition of distributed database in [14]

Definition 1. A *distributed document db* is a tree whose vertices are partitioned into m sets, called *partitions* and denoted $db_i, i = 1, m$. Each partition db_i is stored on a separate network node N_i.

Definition 2. A *cross link* is an edge between vertices $v \in db_i$ and $u \in db_j$ where $i \neq j$. A cross link must not create a cycle in a document.

Proceedings of the 5th Russian Conference on Digital Libraries RCDL2003, St.-Petersburg, Russia, 2003

Definition 3. A tree with vertices representing network nodes and edges representing relationships between nodes is called a *network tree* of a document.

2.1 Prefix trie

If we consider a virtual XML document as a bag of paths we will find that paths usually have common prefixes and that there may be two or more equal paths which physically end on different network nodes. Querying a document we probably wouldn't desire to distinguish equal paths stored on different nodes or equal prefix instances which belong to different paths. In order to achieve this goal we model a document as a trie [11] which is distributed over the network and is supported by the special enumeration of vertices described in section 2.2.

2.2 Vertices numbering scheme

Very often when searching in an XML document it is necessary to determine whether two nodes v and u are in parent-child or ancestor-descendant relationship. This problem which is called hereafter *Ancestor-Descendant Relationship Determination (ADRD)* problem is more or less easy to solve when the whole document instance is stored in the main memory. In such case we may just build two paths from u and v to the root of the document and check whether one path is a prefix of another. If the document resides on a secondary storage like hard disk the solution becomes more time consuming because it involves a number of expensive I/O operations. When the document is distributed such approach becomes unacceptable due to high cost of the network communication operations.

A popular solution of ADRD problem is to assign some numbers to the vertices of the document tree according to some rules. If we know the numbers assigned to vertices v and u we can solve the ADRD problem with the simple arithmetic calculations.

So far, several numbering schemes have been proposed [7, 5]. They are mostly based on the Dietz enumeration [4] and suffer from the common problem: they have to be rebuilt after a single insert or delete operations in the most primitive case or after a series of such operations in more sophisticated solutions.

We have developed our own numbering scheme which has some important features:

- Insertions and deletions of vertices do not cause renumbering
- Enumeration is *relocatable* in the sense that two parameters necessary to enumerate all vertices on some site are the local root number and the alphabet size
- Each local enumeration can participate in different distributed enumerations

Vertices are assigned the rational numbers recursively according to the formula:

$$q(v) = q(parent(v)) + \frac{Num(\alpha)}{(|\Sigma| + 1)^{depth(v)}}$$

where α is a label of the incoming edge, $Num(\alpha)$ is the ordinal number of α in the alphabet Σ. The root of the document is assigned a number $q(root) = 0$

Proposition 1 (ADRD problem solution). *Two vertices v and u are in ancestor-descendant relationship iff*

$$q(v) < q(u) < q(v) + \frac{1}{(|\Sigma| + 1)^{depth(v)}}$$

A geometric sense of this enumeration is a partition of some interval into disjoint subintervals each corresponding to a symbol from the alphabet. Each subinterval in turn is partitioned by subintervals of the next level, etc.

2.3 Relationships between nodes

Relationships between network nodes which are included into the document may vary differently. We assume that a root node of the document is *navigable* (that is, there is a kind of pointer) from any node of the document. We also assume that destination nodes of cross links are navigable from source nodes.

Concerning other relationships, we consider two options:

- Loosely connected system: there are no other relationships between nodes
- Tightly connected system: any node is navigable from the root node

We will consider query evaluation algorithms for loosely and tightly connected systems in the rest of the paper.

3 Query evaluation algorithm efficiency

It is clear that query evaluation algorithms behave differently. In order to compare them we need to give some definitions of an algorithm efficiency. The definitions given below are motivated by our goal to minimize query evaluation time. They take into account that connection setup cost is high so it is necessary to decrease the total count of connections; huge amounts of data transferred between nodes also affect performance so an efficient algorithm should transfer only relevant data.

The first definition which is stronger than the second one was introduced in [14]. It involves a notion of *communication step* which is a set of network connections established simultaneously between a *dispatcher* (see section 4) and other nodes. However, in a loosely connected system not all nodes are navigable from dispatcher. We provide a relaxed definition of efficiency which still bounds the total number of connections but doesn't require them to be established simultaneously.

Definition 4. An evaluation algorithm is *efficient* if

1. The total number of communication steps is constant
2. The total amount of data transfered during query evaluation depends only on the size of the answer to the query and number of cross links

Definition 5. An evaluation algorithm is *scalable* if

1. The total number of connections established during query evaluation depends on the total number of network nodes only
2. The total amount of data transfered during query evaluation depends only on the size of the answer to the query and number of cross links

An efficient algorithm, obviously, is scalable. At the same time, the total number of network connections which are established by an efficient algorithm is not necessarily less than the number of connections which are established by a scalable, but not efficient algorithm; the efficiency is achieved due to parallelizing the connections.

4 Query evaluation

We will describe query evaluation algorithms for loosely and tightly connected systems in sections 4.1 and 4.2. Both algorithms are modeled as state machines and operate with some objects which are passed between nodes.

A node propagates a query to descendant nodes packing it into *context object*. This object has the following attributes:

A Automaton corresponding to the query

S A subset of the automaton states

pcv A context vertex represented by a valid number in our numbering scheme.

If some node decides to propagate a query to descendant nodes, it creates an intermediate table which stores information necessary to join results obtained from descendants. The table is defined as follows:

```
CONTINUATIONS (
  ENTRY_VERTEX,
    // Number of the vertex serving as
    // entry point for the query on this
    // node
  ENTRY_STATE,
    // State of the automaton when query
    // enters this node
  CONTINUATION_VERTEX,
    // Number of a vertex on a remote node
    // where the query continues
  CONTINUATION_STATE
    // State of the automaton when query
    // leaves this node
)
```

Each node involved into evaluation returns an instance of results table to ancestor node:

```
RESULTS (
  ENTRY_VERTEX,
  ENTRY_STATE,
    // These attributes have the same
    // meaning as corresponding in
    // CONTINUATIONS table
  RESULT_VERTEX
    // Number of a vertex where automaton
    // comes to accepting state
)
```

We distinguish a *dispatcher node* whose responsibility is to start the evaluation of submitted query and to create the final result. This node is likely to be a root node although in a tightly connected system it can be any node of the network.

4.1 Algorithm for a loosely connected system

State machine of the query processor is shown on the fig. 1.

4.1.1 State 1A: standby mode

System is waiting for queries. Query Q submitted on some node moves the system to State 1B and passes a context object

```
Q = {S=''start'', pcv=``0/0''}
```

to the dispatcher.

4.1.2 State 1B: Evaluation on a local node

In this state the system evaluates the query on a local node N_i owning vertex `pcv`.

For each $s \in S$ the local query processor runs the automaton A starting from the state s on the local partition db_i data rooted by `pcv` vertex. During evaluation the processor changes the local variables `currentVertex` and `currentState` which indicate currently processed vertex and state of the automaton and initially are assigned values `pcv` and `s` correspondingly. Local query processor inserts new records into tables `CONTINUATIONS` and `RESULTS` according to the following rules:

1. If the automaton comes to the accepting state or if there are no outgoing edges from `currentVertex` then the processor runs

   ```
   INSERT INTO RESULTS VALUES (pcv, s,
               currentVertex)
   ```

2. If the next transition is $.*$ or the outgoing edge from `currentVertex` is a cross link then the processor runs

   ```
   INSERT INTO CONTINUATIONS VALUES (pcv,
             s, u, currentState)
   ```

   for each `u` $\in$ `crosslinktargets(db`$_i$`)`, `u` $\in$ `descendants(currentVertex)`.

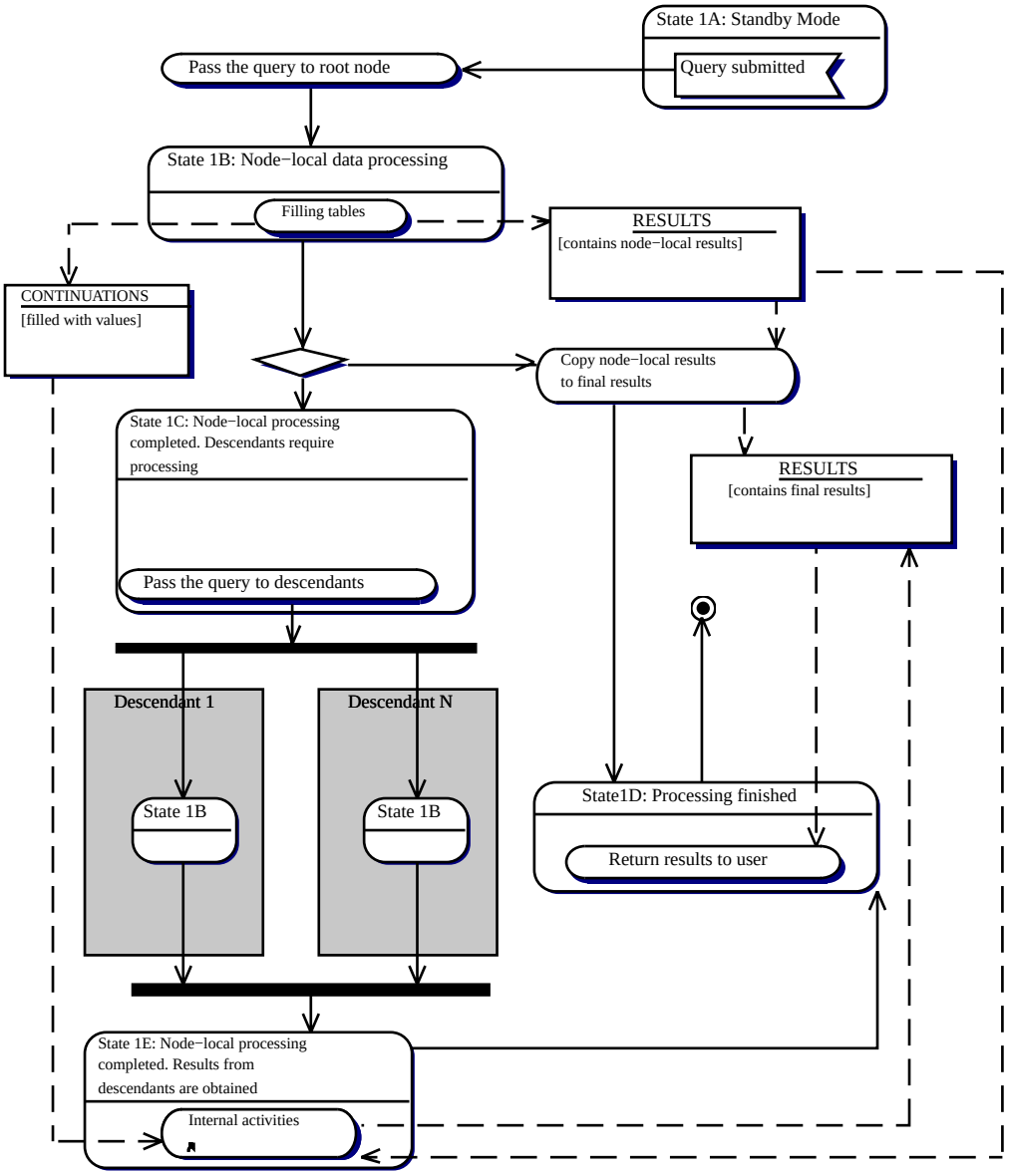

Figure 1: Algorithm in a loosely connected system

If in the end of this activity CONTINUATIONS table contains some tuples then the system moves to State 1C, otherwise it moves to State 1D

4.1.3 State 1C: Passing the query to the descendant nodes

. The system reads the data from CONTINUATIONS table and for each distinct value u of CONTINUATION_VERTEX attribute constructs a new context object

O = {pcv=‘‘u’’, S=‘‘select CONTINUATION_STATE from CONTINUATIONS where CONTINUATION_VERTEX=u’’}.

Then the object O is passed to the node N_j which owns the vertex u. The system creates a swimlane for each N_j and forks a State 1B.

4.1.4 State 1D: Returning results to the ancestor node.

The active node analyzes its RESULTS table and for each value of ENTRY_STATE eliminates vertices lying on the same path retaining only the deepest one. Then the table RESULTS is passed back to caller node and system moves to state 1E.

4.1.5 State 1E: Processing results obtained from descendant nodes

The caller collects all results obtained from the descendants making a union UNITED_RESULT(CONTINUATION_VERTEX, CONTINUATION_STATE, RESULT_VERTEX) of returned RESULTS tables. Then it makes a join CONTINUATIONS $\bowtie$ UNITED_RESULT, unites the result of join with its local RESULTS table and moves to State 1D.

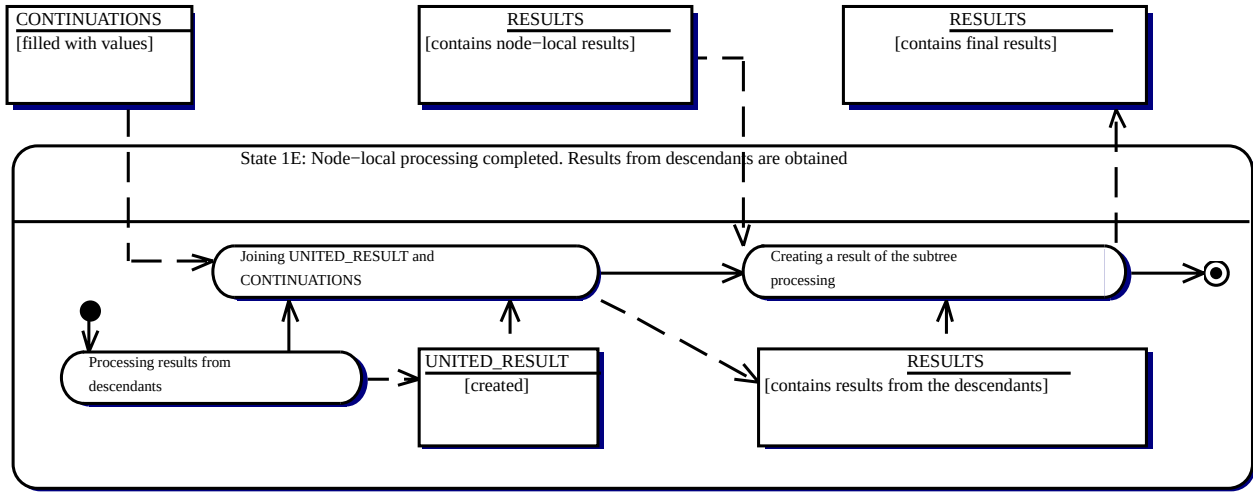

Figure 2: Processing results in a loosely connected system

4.2 Algorithm for a tightly connected system

In a tightly connected system it makes a sense to broadcast a query Q to all nodes, evaluate it on each node and return results to dispatcher who constructs the final result. Such approach possibly establishes more network connections between nodes because each node is accessed even if it doesn't store relevant data. However, in contrast to precedence order between network connections in loosely connected system, we can establish all connections concurrently and thus the time spent to evaluate a query may be lesser.

State machine of this algorithm is very similar to one in loosely connected system. The difference is that dispatcher initiates node-local evaluation simultaneously for all nodes passing the context object $\mathtt{O} = \{\mathtt{S} = \mathtt{allstates}(\mathtt{A}), \mathtt{pcv} = \mathtt{root}(\mathtt{N_i})\}$.

Each node calculates tables `RESULTS` and `CONTINUATIONS` and returns them to the root node. The system moves to state 2E and calculates the final result.

4.2.1 State 2E: Calculation of the final result

In this state (fig. 3) the system unites all `CONTINUATIONS` and `RESULTS` table returned from nodes and makes a join $\mathtt{CONTINUATIONS} \bowtie \mathtt{UNITED\_RESULTS}$. If this join is not empty then it is considered as a new instance of `UNITED_RESULTS` table and join is performed again. Otherwise the selection

$$\sigma_{\mathtt{ENTRY\_STATE}='\mathtt{start}'}(\mathtt{UNITED\_RESULTS})$$

gives the final result.

5 Analysis of the algorithms

During a query evaluation in a loosely connected system, each node may contact its children and wait for results from them. As we assume that a document is a tree, we may say that each node may be called by the ancestor and only by the ancestor and whenever the node is called it returns a result to ancestor. Thus, the total number of connections is bounded by $2 * m$, where m is a number of network nodes. However, the number of communication steps is as big as $2 * H$ where H is the height of the network tree. Thus, although being efficient in terms of definition 5 this algorithm violates item 1 of definition 4.

Each node which participates in the evaluation process receives a small context object and sends back an instance of `RESULTS` table whose size is $O(r)$ where r is the size of the query answer. These data are passed from the leaf nodes to ancestors and finally reach the dispatcher. Hence, the total amount of data transferred during the query evaluation is $O(m) * O(r)$ and this algorithm does not violate item 2 of both definitions

In a tightly connected system a dispatcher accesses every node of the network only twice and connections are established simultaneously. Thus, the number of communication steps is constant. The dispatcher sends to nodes a small context object and receives a result object whose size depends on the query result size only. Thus, this algorithm is efficient in terms of both definition 4 and definition 5.

6 Conclusions

In this paper we have presented a network model of the distributed semistructured data which allows a set of network nodes storing parts of XML document to cooperate in building a virtual distributed XML document. We also have described algorithms of a regular expression query evaluation for two types of the network nodes relationships. These algorithms behave efficiently in terms of network connections count and amount of data transferred.

References

[1] Apache Software Foundation. *XIndice User Manual.*

[2] Brian Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In

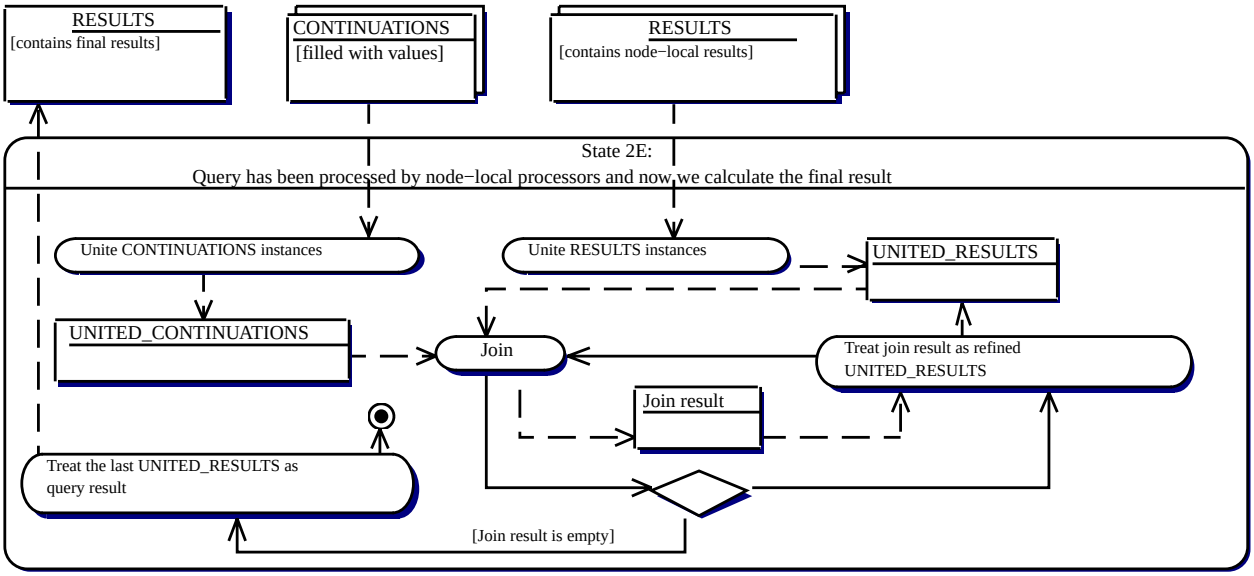

Figure 3: Processing results in a tightly connected system

The VLDB Conference, pages 341–350. VLDB, 2001. citeseer.nj.nec.com/cooper01fast.html.

[3] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 431–442. ACM Press, 1999. http://doi.acm.org/10.1145/304182.304220.

[4] Paul F. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 122–127, 1982.

[5] Torsten Grust. Accelerating XPath location steps. In *Proceedings of ACM SIGMOD 2002, June 4-6, Madison, USA, 2002*, 2002.

[6] IBM. *XML Extender Brochure*, 2000. http://www-3.ibm.com/software/data/db2/extenders/xmlext/index.html.

[7] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *Proceedings of 27th International Conference on Very Large Data Bases*. Morgan Kaufmann, 2001.

[8] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallas Quass, and Jennifer Widom. Lore: a database management system for semistructured data. *ACM SIGMOD Record*, 26(3):54–66, 1997. http://doi.acm.org/10.1145/262762.262770.

[9] Microsoft Corp. *Microsoft SQL Server 2000 Books online*, 2000.

[10] Sun Microsystems. *The Java Web Services Tutorial*. http://java.sun.com/webservices/docs/ea1/tutorial.

[11] Donald R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *JACM*, 15(4):514–534, 1968. http://doi.acm.org/10.1145/321479.321481.

[12] Oracle Corp. *Application Developer's Guide - XML*, 2001.

[13] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. de Witt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. of the VLDB'99*, pages 302–314. Morgan Kaufmann, September 1999.

[14] Dan Suciu. Distributed query evaluation on semistructured data. *ACM Transactions on Database Systems (TODS)*, 27(1):1–62, 2002. http://doi.acm.org/10.1145/507234.507235.

[15] W3C. *XML Schema*. http://www.w3c.org/XML/Schema.

[16] W3C. *XSL*. http://www.w3c.org/Style/XSL.

[17] W3C. *Extensible Markup Language (XML) 1.0*, 1998. http://www.w3.org/TR/REC-xml.

[18] W3C. *XML Path Language (XPath) Version 1.0*, 1999. http://www.w3.org/TR/xpath.

[19] W3C. *XQuery 1.0: An XML Query Language, W3C Working Draft*, December, 2001. http://www.w3.org/TR/2001/WD-xquery-20011220/.