# An XML Query Engine for Network-Bound Data

Zachary G. Ives* Alon Y. Halevy Daniel S. Weld

{zives, alon, weld}@cs.washington.edu

University of Washington
Seattle, WA USA

**Abstract**

The rapid and nearly universal adoption of XML as a *de facto* standard for data interchange provides new opportunities for querying and integrating data. It is now possible to develop an environment in which users can pose queries over and across Internet, intranet, or extranet data sources — without having to first develop custom wrappers and schema translators for each data source. An important research focus must now be in the development of efficient query processing techniques for network-based XML data sources, for both batch-oriented and interactive query applications.

In this paper, we present *NetXML*, the first XML query engine designed specifically for efficient processing of network-bound XML data streams. The *NetXML* query engine builds upon many well-understood aspects of a traditional relational database engine — allowing it to leverage existing implementation techniques such as pipelining and adaptive query operators — but it includes key extensions for the XML data model, such as support for object- (in addition to value-) based data semantics, the *ScanXML* query operator, and XML nesting and construction operators. We experimentally demonstrate that, compared with existing approaches, *NetXML* provides better overall query performance, faster initial answers, and has excellent scalability.

## 1 Introduction

XML, the eXtensible Markup Language standard from the World Wide Web Consortium [XML98], is increasingly being used as a format for the dissemination and exchange of information from all types of data sources and applications. XML is quickly becoming the *lingua franca* for data exchange, and nearly every vendor of data management tools has been racing to adopt it. An XML document (e.g, Figure 1) consists of pairs of matching open- and close-tags (elements), each of which may enclose additional elements or uniquely valued attributes. The strengths of XML lie in its simplicity, self-describing nature, and flexibility — particularly in its ability to represent a graph structure (via `ID` and `IDREF` attributes, which serve to uniquely identify elements and to form links to them, respectively), which allows it to encode both structured and semi-structured data.

Recent development of standard XML vocabularies (i.e., DTDs and XML Schemas)[1] attests to the growing adoption of XML for publishing and exchanging data. Indeed, XML is a natural medium for the emerging area of querying network-based data. Efficient network-based query capabilities over XML data are especially important to the document community — which wants to efficiently query and restructure

*Supported in part by an IBM Research Fellowship. Thanks to Phil Bernstein, Daniela Florescu, Hartmut Liefke, David Maier, Rachel Pottinger, and Dan Suciu for their comments on earlier versions of this paper.

[1]See the OASIS web site xml.org for an online catalog of such schemas.

```
<db>
  <book publisher="mkp">
    <title>Readings in Database Systems</title>
    <editors>
      <name>Stonebraker</name>
      <name>Hellerstein</name>
    </editors>
    <isbn>123-456-X</isbn>
  </book>
  <book publisher="mkp">
    <title>Transaction Processing</title>
    <authors>
      <name>Bernstein</name>
      <name>Newcomer</name>
    </authors>
    <isbn>235-711-Y</isbn>
  </book>
  <company ID="mkp">
    <name>Morgan Kaufmann</title>
    <city>San Mateo</city>
    <state>CA</state>
  </company>
</db>
```

Figure 1: Sample XML document representing book and publisher data

documents while preserving their semantic meaning — and the data integration community — which queries, transforms, and integrates multiple heterogeneous data sources.

Of course, a network-based XML query system requires a new and expressive query and transformation language (a "SQL for XML"). This language is still under construction: various XML query language proposals have been made, such as [RLS98, DFF+99, CCD+98, GMW99, XSL99, CRF00], and the World Wide Web Consortium is attempting to consolidate them into a standard. The proposed languages vary somewhat in syntax and, to a lesser extent, in capabilities, but share many important features. They represent an XML document as either a tree (equivalent to the parse tree) or graph (equivalent to the parse tree with references). Subtrees or subgraphs of the XML document are selected via *regular path expressions* describing traversals over the document structure. These subgraphs form the basic operands within the query language, and are combined and restructured to construct new XML output.

Thus, even before the final query language emerges, it is reasonable to adopt a basic feature set from the existing proposals and to develop basic techniques for efficient network-based XML query processing. The query processor's front-end may change to reflect a new language syntax, but its core architecture should remain similar or even the same. Core requirements of such a system are as follows:

- The ability to query, combine, and restructure the content of one or more XML documents of arbitrary size.
- Support for "streaming" query output, *i.e.* the query processor should output results as it is still reading input, in order to produce early answers even with slow network connections.
- Efficient support of queries over XML data that is stored remotely and subject to change at any time.

Currently, we are not aware of any existing systems that meet these requirements. Instead, most XML query engines have fallen into one of two classes. The first, exemplified by XSLT [XSL99] processors and the Niagara system at the University of Wisconsin [CDTW00], are web-oriented, main-memory systems. They take each source document, parse it and construct an in-memory representation, and finally perform query operations on the subtrees. This has two drawbacks: first, document size is limited to the amount of memory available, and second, each document must be fully read and parsed before query processing can begin.

The second approach, which is not really intended for integrating data, but simply for storing it, is repository-oriented: systems of this class map the XML document to a more traditional data model and storage subsystem. This typically requires loading the data into the system and indexing the repository before queries can be posed. Different techniques have been developed for storing XML in a relational database [FK99, SGT$^+$99, MFK$^+$00], an object-oriented database [vZAW99, LAW98] or a repository for semi-structured data [GMW99]. This approach does support arbitrarily large XML documents. However, it must not only read and parse entire documents before processing them, but in fact it must load and index them on disk. Furthermore, since network-bound documents are often subject to frequent updates at their sources, a single load operation cannot generally be amortized across multiple queries.

Clearly, neither of the existing approaches is well-suited to the domain of network-based querying; hence, we have developed the *NetXML* system with a novel architecture. Building on a regular path expression evaluation operator called *ScanXML* and mechanisms for constructing XML in a relational-style pipeline, the *NetXML* system provides efficient querying of network-based XML data with short time-to-first-results. Moreover, it features nonblocking operators as in [IFF$^+$99, UF00a] and is capable of leveraging techniques and components developed for object-relational systems — particularly useful extensions would include adaptive query processing techniques, such as query operator and plan rescheduling [BFMV00, UFA98, AH00] and interleaving of planning and execution [KD98, Col00, IFF$^+$99]. In this paper, we present the first detailed description of a query execution engine for network-bound XML data, and demonstrate that it provides superior performance to existing XML query systems within this domain. This leads to the following specific contributions:

- *ScanXML*, an operator that progressively maps selected portions of an XML document into tuples
- Basic operators for combining and restructuring tuples of XML subtrees into new XML content
- An architecture that extends tuple-oriented, relational techniques — such as pipelining, as well as recently developed adaptive query processing techniques for network-based relational data — to work directly on XML.
- A set of experiments that show that *NetXML* provides superior performance compared to previous XML query processing systems when applied to network data, and scales up gracefully to large XML data sets.

To set the context, we begin in Section 2 with a discussion of the XML data model and how it is queried. Section 3 describes the *NetXML* architecture for XML query processing, highlighting the differences from a typical relational engine. We then describe the XML query operators, and in particular the ScanXML [SCN] operator that forms the basis of our architecture, in Section 4. Section 5 provides experimental validation of our work. We discuss related in Sections 6, and conclude in Section 7.

# 2 Data Model and Query Language

We begin by briefly discussing the issues in choosing an XML data model and XML-QL, the language we use for querying XML[2].

## 2.1 Data Model for XML

Several proposals have been made for data models for XML. They are all based on representing XML as a graph, and differ on whether they consider the order of the XML document, how they distinguish between

[2] We have chosen XML-QL as our example language because it is one of the most expressive languages and because it appears to be the most "stable" and well-known language specification. Other XML query languages, particularly Quilt [CRF00], are also well-suited for our system.

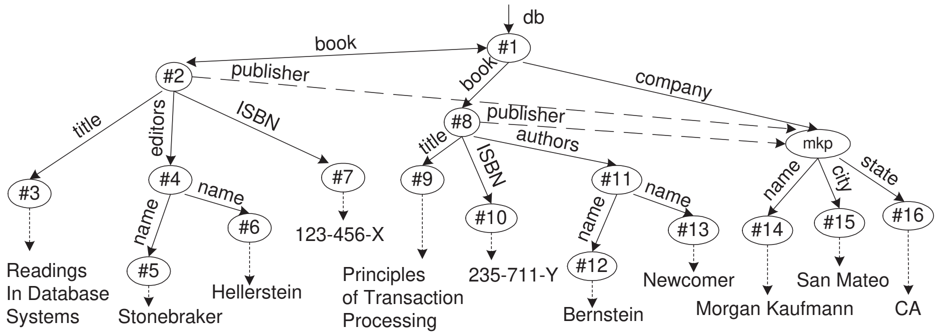



Figure 2: XML-QL graph representation for Figure 1. Dashed edges represent `IDREF`s; dotted edges represent `PCDATA`.

subelements and attributes, and how they represent `IDREF`s in the graph. In our discussion, we follow the XML-QL data model, in which we represent XML data as a graph, where each XML tag is an edge (labeled with the tag name) that is directed towards a node (with a label equal to the tag's `ID`). A given element node may be annotated with attribute-value pairs; it has labeled edges directed to its sub-elements and any other elements that it references via `IDREF` attributes. Figure 2 shows the graph representation for the sample XML data of Figure 1. Note that `IDREF`s are shown in the graph as dashed lines and are represented as edges labeled with the `IDREF` attribute name; these edges are directed to the referenced element's node. In order to allow for intermixing of "parsed character" (string) data and nested elements within each element, we create a `PCDATA` edge to each string embedded in the XML document. These edges are represented in Figure 2 as dotted arrows pointing to leaf nodes.

In this paper we consider execution over both ordered XML graphs and unordered ones, in the former case following the established semantics of processing order in XML. There are certain cases in which XML ordering semantics are undefined or ambiguous (*e.g.* how data from different sources should be ordered when it is combined); we do not attempt to address these issues. Moreover, we observe that XML considers subelements to be ordered but attributes to be unordered; in our model, we *preserve* order across both attributes and subelements, but only allow queries to test and constrain ordering among subelements.

## 2.2 Querying XML

A variety of XML query languages have been proposed, mostly based on languages for querying semi-structured data (XQL [RLS98], XML-QL [DFF+99], XML-GL [CCD+98], Lorel [GMW99]). These languages are driving the current W3C Query Language Committee whose final recommendation is likely to encompass features from each. One fairly recent proposal to the W3C, the Quilt query language [CRF00], attempts to "patch together" elements of the aforementioned languages. The key features all of these languages have in common are that they enable a user to match *regular path expressions* over the data, and, to varying extents, they all have the ability to construct XML documents as a result of the query. In this paper we use XML-QL, but the features of the language that are relevant to our algorithm are mostly found in the other languages as well.

XML-QL uses a `WHERE` *pattern1* `IN` *source1*, *pattern2* `IN` *source2*, ... `CONSTRUCT` *result* syntax, in which the *pattern* template is matched against the input XML data graph from *source* (a URI) and the *result* defines the desired structure of the query output graph. An XML-QL pattern is expressed as a set of nested

```
CONSTRUCT <result>
{
  WHERE <db>
          <book>
               <title>$t</>
               <editor|author><name>$n</></>
          </>
        </>
        IN "books.xml"
  CONSTRUCT <item> <person>$n</>
                   <pub>$t</>
            </>
} </>
```

Figure 3: XML-QL query that finds the names of people who have published. The WHERE clause specifies a graph-structured pattern of nested tags. Variables are prefixed with a dollar sign, and | represents disjunction (*i.e.*, follow either editor or author edges). Although not used in this example, wildcards and Kleene star are also allowed.

tags with embedded variable names (prefixed by leading dollar-signs) that specify *bindings* of graph nodes to variables. Queries can be nested in a manner mirroring the nesting of XML structure (*i.e.*, the CONSTRUCT clause can contain a nested query). Continuing the example of Figures 1 and 2, we can issue the query of Figure 3, which returns a list of author and editor names and their book titles.

More precisely, this query begins by constructing an XML root tag called result, and then defines its contents with a subquery. The subquery searches for all book elements that are immediately inside a db element, that have a child title element and a child element that is either editor or author, but in either case are followed by a name subelement. The inner CONSTRUCT clause returns item elements, whose subelements are person and pub. Note that in XML-QL, we can abbreviate each close-tag with a </>. The WHERE template can be thought of as a set of tree-structured path expressions that get "matched" across the input graph. Each variable name (*e.g.*, $t$ and $n$) is bound to the matching node at the end of the path. In our example, we can take a db edge from the document root and then follow the book edge to node #2 of Figure 2. From here, we traverse a title edge to node #3 whose value we assign to the variable $t$. Also starting from the book node #2 we traverse an editor or author edge followed by a name edge, yielding node #5, which is assigned to $n$.

The result of the WHERE clause of the query is a set of bindings for every possible combination of path expression matches. In our example, a pair of $t$ and $n$ values will be returned for each combination of possible title and name edges under a common book node; {Readings in Database Systems, Stonebraker} is the first of four such pairs. Note that a WHERE clause can consist of several patterns, and each one can be posed over a different document. The result of the WHERE clause in this case is the join of the binding tuples produced by each of the patterns.

The CONSTRUCT clause specifies a tree-structured template of edges and nodes that should be added to the output graph for each tuple of variable bindings. Wherever a variable appears in the CONSTRUCT clause, its associated node-value is inserted into the output. Additionally, we also "carry forward" all other nodes transitively connected by edges radiating from the value node. In essence, an XML-QL variable bound to an XML graph node represents not simply the node, but the *entire subgraph* to which the node transitively connects via outgoing edges. The constructed output for query of Figure 3 over the data in Figure 1 is shown in Figure 4. Note that the outermost (result) tag in the CONSTRUCT clause only appears once in the output;

```
<result>
  <item> <person>Stonebraker</person> <pub>Readings in Database Systems</pub> </item>
  <item> <person>Hellerstein</person> <pub>Readings in Database Systems</pub> </item>
  <item> <person>Bernstein</person> <pub>Transaction Processing</pub> </item>
  <item> <person>Newcomer</person> <pub>Transaction Processing</pub> </item>
</result>
```

Figure 4: The result of applying the query from Figure 3 to the XML data in Figure 1.

this is because XML syntax requires a single "root" element enclosing all remaining content.

# 3 An Architecture for XML Processing

In this section, we describe the *NetXML* architecture for processing XML in streaming fashion. Our approach offers a number of improvements over previous work in XML query processing.

## 3.1 Architectural Goals

The basic goal of the *NetXML* system is to meet the requirements of network-based XML query applications:

- the ability to transform, integrate, and query arbitrary XML documents,
- support for output of query results as early as possible, *i.e.* a "streaming" model of query execution,
- and efficient execution over remote sources that are subject to frequent updates, and therefore their contents cannot be cached locally.

These requirements are not met by main-memory XML query systems, such as Niagara [NDM+00], Kweelt [Sah00], MIX [BGL+99], and a wide variety of XSLT [XSL99] processors — these systems need to fully read and parse documents before executing queries over them, and are also limited by memory. Likewise, XML repository systems [XLN, Tam, FK99, SGT+99, DFS99, GMW99, vZAW99] must map XML documents into a local store before querying them — this greatly impedes performance, and load costs generally cannot be amortized across queries since data sources are subject to frequent updates.

A key observation is that relational engines and relational data integration systems have addressed the some of our desiderata (except, of course, for the XML data model). The relational model of pipelining execution throughout a query plan allows for fast output of results even as input tuples are being read. Recent data integration systems have been based on a model of *adaptive query processing*, where the query processor can change its operator scheduling or query plan to accommodate unanticipated behavior in the incoming data streams. Systems such as Tukwila [IFF+99] and Telegraph [AH00], and other works (e.g., [KD98, UFA98, UF00b, BFMV00, Col00]) integrate data from remote sources and attempt to adapt to changing throughput rates and unanticipated data sizes.

Also notable is the fact that XML query languages are based on a model in which a *pattern* is matched against an input XML document, and for each match, variables are bound to subtrees or subgraphs within the document. These bound variables are then combined (perhaps with variables from other documents) and structured to produce new XML output. A single instance of a pattern match can essentially be viewed as the production of a single "binding tuple": each variable gets bound to a subgraph within the document, and that particular binding tuple gets processed into a new XML output tree.

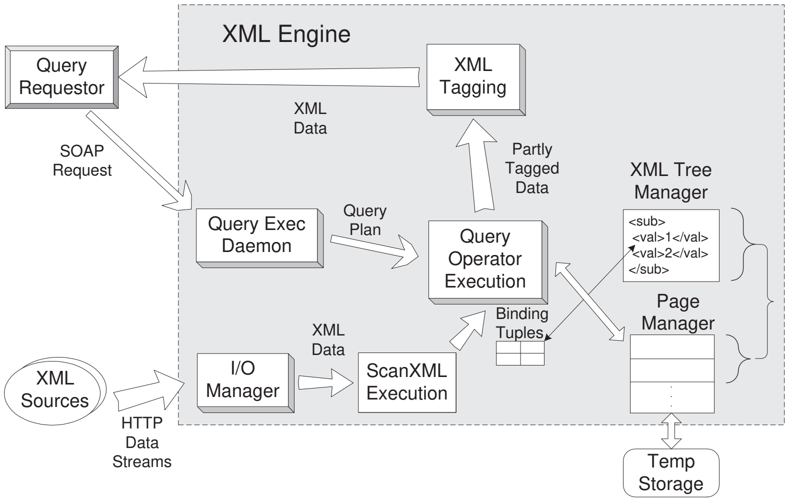



Figure 5: Architecture of the *NetXML* query execution engine. The ScanXML modules convert from XML data to tuples that are fed into a relational-style engine; tuples contain references to subtrees within the Tree Manager. Tuples of XML data are fed into an XML tagging component that returns the final XML output to the requestor.

Our basic goal with *NetXML*, therefore, is to leverage the existing adaptive query processing techniques and components of a relational or object-relational system, and to extend them to support the XML query execution model. This requires pipelined execution not simply of traditional tuples, but tuples of subtrees or subgraphs. It also requires operators that take XML and output binding tuples, that structure binding tuples into new XML content, and that return XML results. Throughout this work we assume that the majority of queries will be over tree-structured data, but we also support querying over graph structures (with some small additional cost).

### 3.2 *NetXML* Query Engine

As suggested by the previous section, the *NetXML* system is designed over an object-relational core, with a number of important extensions to support XML. In particular, it includes:

- a *ScanXML* operator that maps XML data to tuples of subtrees,
- support for efficient memory representation of subtrees, using references within tuples, rather than replication,
- special operators for combining and structuring bindings as XML output

A high level view of the *NetXML* architecture is illustrated in Figure 5. The Query Requestor (which, in this simplified diagram encompasses the *NetXML* user interface and optimizer) makes a request to the execution engine, and the Query Execution Daemon invokes a new thread to service the request. Attached to the request is a query plan, which contains at its leaves the ScanXML operators described above. The ScanXML operators (1) retrieve XML data from the data sources, (2) parse and traverse the XML data, matching regular path expressions, (3) store the selected XML subtrees in an XML Tree Manager, and

```
WHERE
      <db>
        <book publisher=$pID>
          <title>$t</>
        </> ELEMENT_AS $b
      </> IN "books.xml",
      <db>
        <publication title=$t>
          <source ID=$pID>$p</>
          <price>$pr</>
        </>
      </> IN "amazon.xml",
      $pr < 49.95
CONSTRUCT
      <book>
        <name>$t</>
        <publisher>$p</>
      </>
```

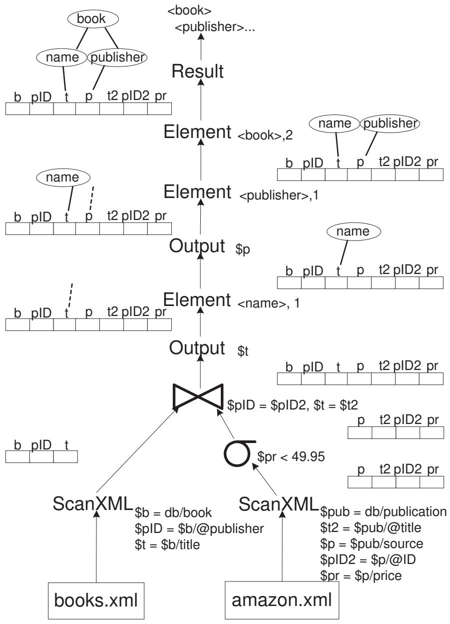



Figure 6: Example XML-QL query joining books by publisher ID and title with Amazon price lists, for publications priced less than $49.95.This outputs an XML subtree for each book with its nested title and publisher information. A typical query plan for this is a pair of ScanXML operators to compute the input bindings, a join over tuples from the two inputs, and a series of Output and Element operators that copy the desired variables to the output and construct XML elements around them. The Result operator outputs XML by traversing the output tree in right-to-left order.

(4) output tuples of references to subtrees[3]. The tuples are fed into the remaining operators in the query execution plan, where they are combined and restructured. XML output structures are built within each tuple as it flows through the query plan, and these are ultimately fed to the XML Tagger for conversion back into a standard XML stream that is fed to the user.

A key component that distinguishes our architecture from a standard object-relational architecture is the *XML Tree Manager*, which is a virtual memory manager for subtrees. The tuples being pipelined through the operations in the system contain references to subtrees within this memory manager, thereby saving the cost of replicating entire subtrees as they are passed from one operator to another. When appropriate, tuples may also contain PCDATA or XML strings, rather than references to the structure. Our operators can compare both references and values, so both object-based and value-based operations are possible — including grouping, nesting, and aggregation. XML subtrees can be reference-counted and garbage-collected when all tuples referring to them have been processed by the system.[4]

Of course, XML query processing does require a number of specialized operators, beginning with *ScanXML* (described in Relational operators such as select, project, and join operate based on these reference-based attributes (so, for example, two tuples can be joined based on PCDATA content). Other operators are responsible for constructing and structuring the XML for the query output.

An example XML-QL query and its physical query evaluation plan are shown in Figure 6. The ScanXML operators at the leaves are responsible for converting XML to streams of tuples by binding variables to the

[3]If a path expression matches PCDATA rather than a full subtree, we may "inline" the data in our output tuple, rather than referencing to an entry within the Tree Manager, for better performance.

[4]Our current implementation does not yet "garbage collect," but the principle fits into our framework and should result in an improved memory footprint.

nodes matched by regular path expressions. General query operators such as selects and joins are performed over these tuples: first we select Amazon `publications` priced less than $49.95, and then we join the results with `books` on the `publisher` and `title` values. Once the appropriate binding values have been selected and joined, an output XML tree must be generated with the variables' content. The **output** operator is responsible for replicating the subtree value of a given binding to the query's CONSTRUCT output. The **element** operator constructs an element tag around a specified number of XML subtrees. In the figure, the output subtree is shown at different stages of construction — first we output $t and insert a `name` element above it; then we output $p and a `publication` element tag around it; finally, we take both of these subtrees and place them within a `book` element. As a last step, the **result** operator converts a stream of tuples with embedded XML subtree into XML text.

# 4 Query Operators for XML

In this section, we describe the basic query operators of the *NetXML* system, which are combined with traditional relational operators to form XML query plans. We begin with the ScanXML operator, which can be viewed as a converter between an XML document and our internal tuple-oriented, pipelining execution model. We then describe the structuring operators responsible for constructing new XML output.

## 4.1 The ScanXML Operator

Given the text stream of an XML document and a set of regular path expressions (equivalent to an XML-QL pattern) as inputs, ScanXML incrementally outputs a stream of tuples assigning binding values to each variable. ScanXML supports traversal of both tree- and graph-structured XML data models. We first describe the data structures used by ScanXML, and then describe its operation (first on tree data, and then on graph data).

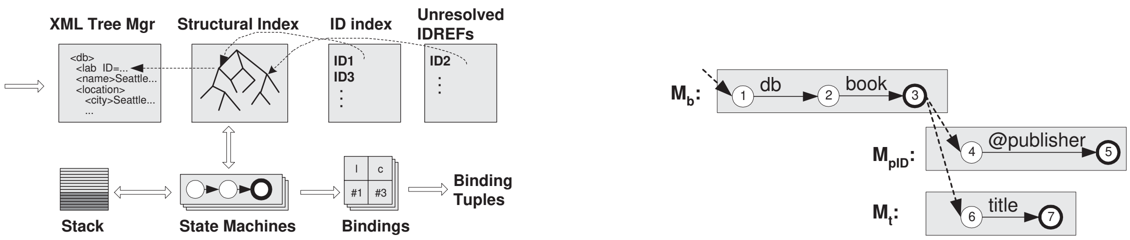
### 4.1.1 Data Structures

The data structures of ScanXML are illustrated in Figure 7(a). As the data streams into the system, we store relevant portions in the XML Tree Manager. If the data is graph structured, we also build the following three data structures:

- Structural index: provides an index of the XML graph, corresponding to Figure 1(b), but without the data values at the leaves. It facilitates ScanXML's traversal through the graph in memory.
- `ID` index: records the `ID`s of all elements and their matching locations in the structural index. It is used to facilitate resolution of `IDREF`s in the graph.
- Unresolved-`ID` index: maintains a list of references to not-yet-seen element `ID`s (to be resolved as they are appear in the input).

The key structure driving the operation of ScanXML is a set of finite state machines (one per regular path expression/variable) that perform a depth-first search over the XML tree or graph. When a machine reaches an accept state, a new value is added to the binding-value table associated with the machine. These values are then combined to produce the binding tuples for the query.

State transitions in these machines correspond to edge traversals in the XML data graph (see Figure 7(b) for the state machines created for the example query of Figure 3). The end of a path expression yields an




(a) Data structures (b) State machines for Fig. 3 query

Figure 7: ScanXML takes an XML document and generates an index of its structure, tracking `ID`s and filling in unresolved `IDREF` targets as they are encountered. Simultaneously, it runs state machines over the graph structure (using a stack to save previous states) and generates tables of variable bindings. Solid arcs in (b) denote state transitions on the label; dashed arcs denote dependencies between machines.

accept state in the machine, which outputs an instance of the corresponding variable. The different state machines are related according to the dependencies of the corresponding variables: because $pID$ is dependent upon $b$, machine $M_{pID}$ is dependent on $M_b$; this means that $M_{pID}$ is only enabled once $M_b$ reaches an accept state.

#### 4.1.2 ScanXML General Operation

In the common case, although XML can encode a graph, most queries are likely to be posed over the tree structure of the document; in fact, many XML query languages only have limited support for graph-based queries. Hence, we begin by describing the operation of the ScanXML operator under a tree-based data model.

ScanXML functions by executing a set of *active* state machines as it encounters subelements and attributes during document parsing. Initially, only the top-level machine ($M_b$ in our example) is active. When a machine $M$ reaches an accepting state, it produces a binding $b$ for the variable associated with it. It then activates all of its dependent state machines, and they remain active while ScanXML is scanning $b$ or any element accessible by a path from $b$. In our example, the machines $M_{pID}$ and $M_t$ remain active while we scan children of $b$.

Associated with each machine is a table for storing binding values. As a machine reaches an accept state, it writes into this table an entry containing its bound subtree value as well as the value of the binding for its parent variable (thus providing a means of associating the variable and its parent)[5]. In our example, $M_b$'s table would just store values of $b$, while $M_{pID}$ and $M_t$ would store publisher names and titles, respectively, paired with their corresponding $b$ values. The final output of ScanXML is the natural join of the tables maintained by the machines (this is done in a push-oriented manner as with a pipelined hash join [WA91]).

We illustrate the execution of ScanXML on our example data of Figure 2. Suppose $M_b$ is initialized to machine state 1, which takes the XML `root` as its start position. ScanXML follows the `db` edge, pushing $M_b$'s old value on the stack and setting $M_b$ to state 2 with its position at input graph node `#1`. Next ScanXML can follow one of two outgoing `book` edges. It chooses the first one, taking it to node `#2`, and setting $M_b$ to state 3. $M_b$ is now in an accepting state, so ScanXML writes the reference `#2` into $M_b$'s table, suspends $M_b$, and activates $M_{pID}$ and $M_t$. The next (`publisher` attribute) edge takes $M_{pID}$ from state 4 to 5, which is an accepting state for $M_{pID}$. Hence, ScanXML writes ⟨`"mkp"`, `#2`⟩ into $M_{pID}$'s table. In the meantime, $M_t$

[5]The implementation can store subtrees by value or reference. For expository simplicity, we write as though nodes are stored by `ID`-based reference and attributes are stored by value.

follows the `title` arc to its state 7, putting it also in an accepting state, and therefore writing $\langle$#3, #2$\rangle$ into $M_t$'s table. From this node, no (non-PCDATA) edges remain for exploration, so ScanXML pops the stack and backs up the state machines. It sets $M_b$ to state 2, where it can continue to explore the second `book` edge, proceeding as before. □

#### 4.1.3 ScanXML Operation on Graphs

In the previous section, we described the general operation of ScanXML on tree-structured data. A number of extensions are necessary to support evaluation of regular path expressions across a full graph model, as IDREFs may reference arbitrary elements anywhere within a document.

When ScanXML is run on a graph-structured XML source, it generates the structural index, which is a trie-like index of the XML graph structure (*i.e.* the subelement and IDREF links). This index allows ScanXML to quickly traverse back through XML structure in evaluating references. In addition, as we explain below, the construction of the structural index continues even when we need to suspend the state machines because of unresolved IDREFs.

Each node in the index contains information about an element (its ID and an offset into the original XML data file so that the node's source can be accessed quickly) as well as pointers to all subelements, attributes, and IDREFs of the element. Essentially, the index structure looks like the graph of Figure 2 except that data values such as those in the leaf (PCDATA) nodes are not stored.

In addition, ScanXML creates the ID index, which records all the IDs that it has encountered so far, mapping from ID to entry in the structural index, and the unresolved-ID index which records all IDs that have not yet been seen in the input, and lists all referrers to each such ID.

ScanXML's general execution proceeds similarly to the tree-structured case, except when an element with references is encountered, and the references are to be traversed by the regular path expression. If the reference is to an element that has already been parsed (a backward reference), the state machines are run over the reference's target in the structural index, and then parsing continues.

**Forward References:** On occasion ScanXML will encounter an IDREF edge which points "ahead" to a node which has not yet been parsed. When ScanXML hits a forward reference, it pauses all state machines and adds an entry to the list of unresolved IDREF symbols, specifying the desired ID value and the referrer's address. However, ScanXML continues reading the XML source and building the structural index. Once the target element is parsed, ScanXML fills its address into each referring IDREF in the structural index, removes the entry from the list of unresolved IDREFs, and awakens the state machines and proceeds.

**Cycles:** When the input XML document contains cycles, care must be used to ensure that ScanXML returns all possible binding tuples without getting trapped in an infinite loop along the cycle. The solution is to save a history of nodes visited by each automaton state in a given path traversal. ScanXML uses deterministic automata, so if a machine re-visits a node that was visited by the same state, it is in a cycle that cannot produce any new and useful output.

**Querying Order:** ScanXML can also match path expressions with restrictions on order. In particular, the algorithm can allow constraints over subelements' index numbers (e.g. "paragraph immediately followed by a figure") in forming binding tuples. Similarly, counters can be used to find the $i$th subelement (*e.g.* "second paragraph"). We can even support the BEFORE ($a BEFORE $b) and AFTER clauses of Quilt, which are generally challenging to implement in a non-native XML query processor, by simply comparing the addresses of bindings within our XML Tree Manager (lower addresses correspond to earlier occurrences in the XML document).

**Selection over Results:** Another useful capability in ScanXML is the ability to apply certain selection predicates over the variable bindings and their subtrees. These can be simple predicates over values (e.g. "bind $b to book title when it has the value 'Transaction Processing' "), or they can optionally include existential or universal quantification over a path. Existential quantification is supported in the same fashion as a typical ScanXML variable binding to a path; however, the path does not get a binding and is not returned. For universal quantification, we build a state machine for the path, and ensure it always reaches an accept state for each possible traversal over the subgraph.

**Handling memory overflow:** Often, ScanXML needs very little working space — it outputs a stream of binding tuples (*i.e.*, sets of subtrees) and little state needs to be maintained between the production of any two tuples. However, there are three specific cases where we may run out of memory.

The first, and most obvious, is when the trees themselves are large, and the query plan is sufficiently expensive that these trees cannot be quickly garbage-collected as they exit the pipeline. This presents little problem for ScanXML, as the XML Tree Manager is built over the *NetXML* page manager, and some subtrees may be paged out. Most queries do little manipulation of the trees themselves — items such as join keys and selection attributes can be directly inlined within tuples — so this is generally not a costly operation.

The second case, which may occur for trees with high fanout, is when several sibling variables within the pattern each bind many times, *e.g.*, if there were many titles and editor or author names for each book in Figure 3. In this case, XML-QL semantics specify that we must return a tuple for each combination of $t$ and $n$ variables in a given book. Internally, ScanXML must maintain a table of $n$ bindings and a table of $t$ bindings, and perform a pipelined cartesian product on them; it flushes these tables each time a new book is encountered. In an extreme case, it is possible that these binding tables might exceed memory — we handle this case in a manner similar to the pipelined hash join overflow strategies of [UF00a, IFF$^+$99].

Finally, for graph-structured queries, it is possible that one of the primary data structures may overflow memory. These are fairly easily handled. The structural index is already a paged structure and the ID indices may be converted into multilevel, paged hash tables.

## 4.2 Relational Operators

Since *NetXML* was designed to utilize existing relational components, it naturally makes use of the standard set of relational operators: select, project, join, sort, group-and-aggregate, union, and so forth. Since we are interested in a network query domain, we have included a number of adaptive query processing operators and features, such as a pipelined hash join [WA91, UF00a, IFF$^+$99].

The operator implementations are generally unchanged from their corresponding relational versions, except that they have been extended so that they can process values and references to XML subtrees. In general, for comparison and join predicates over trees, we tested for structural equality (of a canonical string-ified version of the XML subtree). This model seemed the best match to the XML model, since it does an order-based comparison. We do believe that an unordered comparison may also be useful, but most XML query languages only support comparisons of primitive values, so this extension has not yet been implemented.

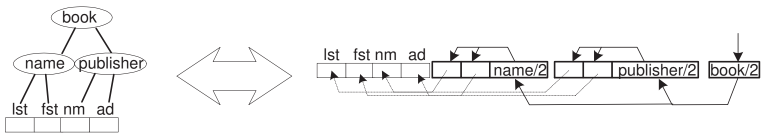



Figure 8: Encoding of a tree within a tuple. The encoding is done in bottom-up fashion, so each parent element needs only to specify its number of children. The tree "leaves" are references to attributes containing bindings.

An interesting and potentially useful benefit of our hybrid architecture is that *NetXML* can not only process both relational and XML data, but it can actually combine data from both types of sources, and output the results either as structured XML, or as relational tables with string-ified XML subtrees. This enables us to provide the ability to integrate arbitrary XML content with relational data without incurring the overhead of transporting the relational data as XML.

## 4.3 XML Output and Tagging Operators

A query in XML-QL consists of two portions, a `WHERE` and a `CONSTRUCT` clause (there is a similar basic form in languages like Quilt). Within *NetXML*, the ScanXML operator performs the `WHERE` pattern-matching operation and establishes tuples of bindings. Join and selection operators express predicates within the `WHERE` clause. Once these operations have been performed, there needs to be a mechanism for generating the XML-QL `CONSTRUCT` output template and returning it with the appropriate variables' values inserted in each binding tuple.

Logically, *NetXML* constructs output by building an XML tree or forest "over" each tuple flowing through the system, as is suggested by Figure 6. In reality, nodes within the tree are embedded within the tuple as a series of hidden attributes in an array-based tree encoding. The tree encoding is constructed bottom-up — every subtree within the tree is stored contiguously, and every non-leaf node only needs a count to specify how many children it has. See Figure 8 for an example encoding that constructs a tree of height 3 with each intermediate node having a fanout of 2.

Our operators, each of which has a correspondence within an XML-QL CONSTRUCT clause, are as follows:

- **output(***var***)** corresponds to the use of a variable binding in the XML-QL template. This operator takes a specified variable binding *var* and designates that it should be copied to the XML output, *i.e.*, it appends an output attribute to the tuple. In Figure 6, for example, we can see an **output** operator with $t as a parameter, and this copies the value of $t to the output.

- **literal(***val*|*expr***)** produces a literal string (*i.e.*, new PCDATA) in the CONSTRUCT clause. When called with an *expr* argument, **literal** computes the value of a scalar expression and appends an output attribute to the tuple. Note that there is overlap between **literal** and **output**, since a variable may be bound to PCDATA. However, we separate the two operators for efficiency purposes (**output** does not require expression evaluation).

- **element(***name*|*var*, *count***)** corresponds to an XML-QL tag in the CONSTRUCT clause, and creates an element (with either a literal or variable name) around the content inside it. Physically, this corresponds to appending to the tuple an element-typed attribute specifying *count* children (see Figure 8).

- **attribute(**$name|var$**,** $val|expr$**)** corresponds to an attribute specifier in XML-QL. As with an element, the attribute may be named with a literal or a variable; likewise, its value can be a literal or the result of a scalar expression.

These operators are sufficient to represent the **CONSTRUCT** clause of a single XML-QL query, provided Skolem functions are not present or have been remapped to nested queries. However, the above operations can construct an XML forest, not a tree — output of well-formed XML requires the ability to hierarchically nest data, the subject of the next section.

### 4.4 XML Nesting Operators

There are generally two distinct cases in which hierarchy is added in an XML query. The first instance is when we perform an aggregate operator over a group (in SQL, a "GROUP BY"). The other is when an XML-QL query contains a nested (and generally correlated) subquery. *NetXML* manages hierarchy with the coordinated effects of three operators: **group**, **nest**, and **result**. This Section explains how the first two operators create hierarchical structure "across" tuples, while Section 4.5 describes how **result** converts the tuple encodings into actual XML content.

The unary **group** operator hierarchically restructures tuples relative to a set of grouping attributes as follows. For each set of tuples with identical grouping attributes, the operator conceptually outputs a single tuple with these grouping attributes, *plus* an embedded subtable with tuples of the non-grouped attributes. Physically, we actually use an encoding in which a set of "flat" tuples represents a combination of grouped and nested content; we describe this encoding in greater detail below, with our **nest** operator. **Group** can be used for restructuring, or an aggregate function can be applied across one of the non-group attributes to return an aggregate value.

To execute a nested XML-QL query, we can left-outer-join the parent and child queries, then execute a **group** operation over the attributes of the parent query[6]; however, we have optimized for this frequently-used operation with a binary **nest** operator, as described below.

Our **nest** operator has semantics very similar to a relational left outer join. **Nest** takes a parent and a child tuple stream, plus a nesting predicate. For each parent tuple, **nest** finds the set of child tuples meeting the predicate and groups them with the parent tuple. At the same time, it groups the parent's XML subtree together with all of the children's XML subtrees. We note that many nested relational algebras (and XML algebras derived from them, *e.g.* the operators in the Niagara system [CDTW00]) include a unary operator called "nest" which is closer in nature to our **group** operator than our **nest**.

Note that while join can support many possible orders, **nest** requires a nested loops join-like ordering, and **nest**'s output must somehow encode groups or sets in a form that can be decoded. Since *NetXML* is based on a tuple-oriented execution model (where all tuples in a given stream have the same schema), we encode a parent-and-children group as a succession of tuples, rather than as a single tuple with a set-valued attribute. However, we do this in a way that avoids comparing a large number of attributes to see if they are identical (and thus a group).

Our solution is to have the **nest** operator embed a hidden attribute within its output. Each output tuple from **nest** contains the parent attributes, a "group" attribute containing the tuple ID of the parent tuple,

[6]Note that in this description, we are assuming that our tuple consists of references, so if we group by value, we are grouping by the values of the references, and thus by identity. Pure value-based grouping requires a few obvious changes to our comparison routines.

and all of the attributes from a child tuple. Now, regardless of whether all operators within our system are order-preserving, we can always cluster parent-child groups together by sorting by the hidden parent tuple ID. The XML **result** operator, described in the next section, is then responsible for creating the XML in its final hierarchical parent-children form, rather than as repetitive parent-child structures.

*NetXML* has both a hash join and nested loops join implementation of **nest**. For cases with an equijoin predicate, the hash version is used as long as the hash table fits into memory (hash join fully preserves ordering in this case). In all other cases, the nested loops implementation is used.

## 4.5 XML Result Operator

The **result** operator is a transducer from tuple streams to XML strings. Hence, **result** must be able to (1) traverse the XML CONSTRUCT tree structure embedded in a tuple, outputting the appropriate data, (2) decode XML subtree content (which is tokenized) from the XML Tree Manager, and (3) correctly output hierarchical XML structure which may span multiple tuples.

The first step, decoding the tree structure embedded within a tuple, simply consists of starting at the rightmost output attribute and recursively traversing the array-encoded tree. The resulting XML content is created via a preorder right-to-left traversal.

Once a leaf node within the embedded tree is reached, the **result** operator must follow a reference to content within the Tree Manager. The *NetXML* Tree Manager does not store XML tags directly; it tokenizes tags and attribute names and stores the tokens. This has two benefits: it reduces the memory footprint for storing a document, since tokens tend to be shorter than element names, and it also makes path expression evaluation more efficient, since numeric token-value comparisons are faster than string comparisons.

The description above assumed that all values encoded within a tuple were to be output; this is not necessarily the case if grouping or nesting attributes are present. If a **group** or **nest** operation has been performed, the grouping attributes should only be output for the first tuple in which they appear, and they should be ignored for all other tuples in the group. Suppose, for example, that we are nesting books inside their publishers. The **nest** operator would output a tuple joining publishers and books, with a group ID attribute for determining when two consecutive tuples belong to the same publisher. Now suppose we have two consecutive books by the same publisher. For the first tuple, **result** will output the publisher (*i.e.* parent) open-tag and related data; then it will output the book data in the tuple. The next tuple will have the same group ID as the previous one, indicating that it "belongs" to the same publisher. Thus **result** will omit the tuple's publisher data from the output, and will only output the book data. This continues until a new publisher is encountered or the data streams end — at this point, a close-tag for publisher will be issued.

Groups can be grouped or nested, so this process scales to arbitrary depth. Additionally, **result** allows for XML-QL-style outer-join nesting semantics — one can have a publisher with no books. In this case, it is possible to have a tuple representation in which the book attributes are all null values.

Note that all of the construction operators in *NetXML* are order-preserving, which enables the **result** operator to be fully pipelined — it only needs to check for a match between group IDs of consecutive tuples. If we were to support an approach that allowed non-order-preserving operators during construction, **result** would need to sort by group ID(s) before outputting results.

## 5 Experimental Results

We first validate our approach by comparing our *NetXML* implementation with a number of existing systems over various data sets and query loads. Next, we examine the scalability of our approach to larger documents and to more complex path expressions. Finally, we evaluate the overhead of supporting path expressions over graph-structured data, and the cost our order-preserving **nest** operator has when compared to a conventional, non-order-preserving join.

### 5.1 Evaluation Details

Our implementation of the *NetXML* execution engine is written in Microsoft Visual C++, and our ScanXML implementation was built over Apache's Xerces-C 1.3 parser. Our system has a client-server architecture, with a Java client that submits queries, then reads, times, and materializes the XML results. All experiments were run on an 866MHz Pentium III machine with 1GB RAM, a 20GB ATA hard drive, and Windows 2000 Server. XML documents were served via HTTP from our web server, a dual Pentium II 450MHz system with 512MB RAM, running NT Server 4 with IIS 4. The two machines communicated via standard 10Mb Ethernet, with each machine on a separate subnet within a larger-scale network.

Our initial experiment compares *NetXML* with several systems written in Java (which we ran in Sun's JDK 1.3 with HotSpot). Although Java applications tend to generally perform more slowly than comparable C++ systems, it should be noted that a Java-based system, XT, is generally acknowledged to be one of the fastest XSLT engines available in any language.

Experimental data sets were chosen to encompass a range of different XML data classes, and include the following:

- Documents: religious texts (Quran, 898KB, and Book of Mormon, 1.5MB) from Jon Bosak's XML collection.
- Synthetic data: automatically generated XML structure with random variation in depth and in distribution of IDREF attributes. These ranged from 100KB to 28MB in size.
- Semistructured data: Conference proceedings and publications data from DBLP, consolidated into a single file for each (155KB and 8.9MB, respectively).
- Relational tables: TPC-H data, scale factor 0.01 (10MB). When stored as XML, these files grow considerably, *e.g.*, the LineItem table grows from 7MB to 31MB.

### 5.2 Comparison

Since we are proposing a new model for query execution, we begin by comparing *NetXML* with systems using more traditional approaches. Our comparison systems included two popular XSLT processors (which tend to be centered around document extraction and transformation), plus the Niagara system from the University of Wisconsin (an XML data integration system). Early in the development of *NetXML*, we also compared it to the Lore System [GMW99], an XML repository; at the time, *NetXML* significantly outperformed Lore. Unfortunately, Lore is no longer being distributed, and therefore we omit Lore from our comparison, because it would be unfair to compare with an outdated version of Lore. See Table 1 for details on our systems; all except for *NetXML* are main-memory-only XML engines.

| Name | Implemented | Domain | Description |
|---|---|---|---|
| Xalan | C++ | Doc | Apache XSLT processor, built over Xerces-C parser |
| XT | Java | Doc | James Clark's XSLT processor (generally rated #1 in benchmarks) |
| Niagara | Java | XML-DB | University of Wisconsin XML integration system |
| NetXML | C++/Java | XML-DB | XML engine described in this paper |

Table 1: Systems compared in Figures 9 and 10

| Nbr. | Class | Input | Query |
|---|---|---|---|
| Q1 | Extract | 0.9MB | Suras in Quran with "The" in title (large trees) |
| Q2 | Extract | 1.5MB | Chapters after Chapter 7 in Book of Mormon (medium trees) |
| Q3 | Extract | 1.5MB | Sura titles with "Mormon" from Book of Mormon (single result) |
| Q4 | Extract | 39MB | DBLP papers that reference conferences (select by existential) |
| Q5 | Integrate | 200KB x 9MB | Nest DBLP papers in their conference entries ($1 : n$ nesting) |
| Q6 | Rel. Sel. | 5MB | TPC-H Orders for Customer "1234" |
| Q7 | Rel. Sel | 31MB | TPC-H LineItems with Quantity $> 32$ |
| Q8 | Rel. Join | 5MB x 0.5MB | Join TPC-H Orders for Customer key $<$ "1234" with all Customers |
| Q9 | Rel. Join | 31MB x 7MB | Join TPC-H LineItems with Orders |

Table 2: Queries for extraction (Q1-4) and combination (Q5) of XML, and those for selection (Q6-7) and joining (Q8-9) of relational data

We ran a mix of document-oriented (search and extraction of XML content), "XML integration" (nesting and structuring), and relational (join-based over limited structure) queries. See Table 2 for a list of queries, which we classify as queries over hierarchical (semistructured) XML (Q1-Q5) and relational data mapped into XML (Q6-Q9).

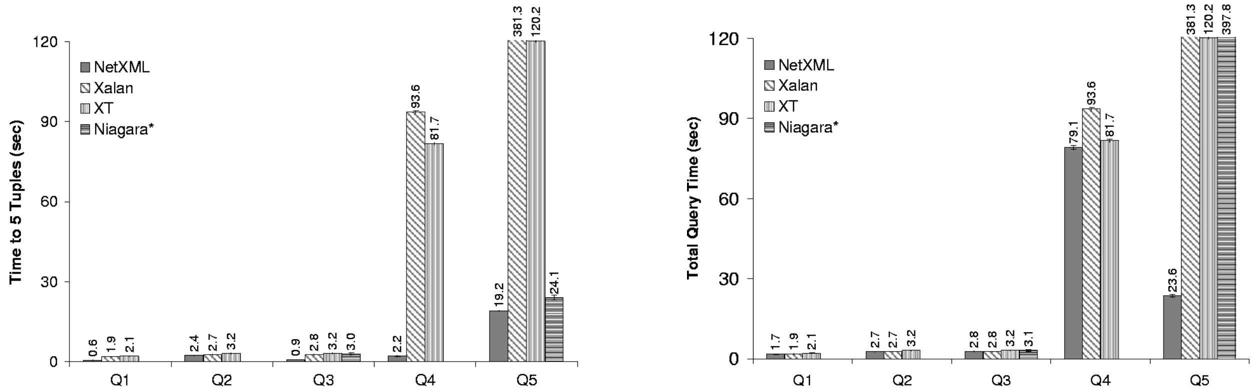
Figure 9 shows the results for queries over semistructured XML. The first queries, Q1-Q3, are simple extraction (selection) queries over small (up to 1.5MB) data files, and here parse time dominates querying time — hence, *NetXML* has similar overall running times to the other systems[7], with better time to initial results. There is a noticeable performance benefit in the more data-intensive query Q4 — particularly in the time to initial results. Query Q5, which hierarchically structures papers within proceedings, shows dramatically better performance for *NetXML*.

Overall, ScanXML's support for pipelined operation over data streams results in much better time to initial tuple (in general returning 5 answers in approximately 2 seconds, except when nesting is required), and in fact improves overall execution time as well. We also observe that the Niagara system, which has largely focused on producing partial answers in order to return early results, can *only* produce those results after it has finished loading and parsing an XML document — Niagara would benefit significantly from the ScanXML operator.

One of the major sources of data is, of course, relational databases, and there is significant interest in sharing relational data in the XML format. An important concern is the amount of overhead incurred by "adding XML into the loop." Do we lose a great deal by querying over an XML view, rather than over traditional relational data? To answer this question, we compared three different means of processing selection and join queries:

- **RDB/XML**, where we sent a SQL selection or join query to a database at the server (in our case,

[7]Note that Niagara has no "LIKE" or "substring" predicate, and thus could not run queries Q1 or Q3; we also were unable to get it to successfully execute query Q4. Niagara does not support nesting of XML-QL queries, so we substituted a join operation for the nesting operation of Q5.




(a) First 5 tuples (b) Total query time

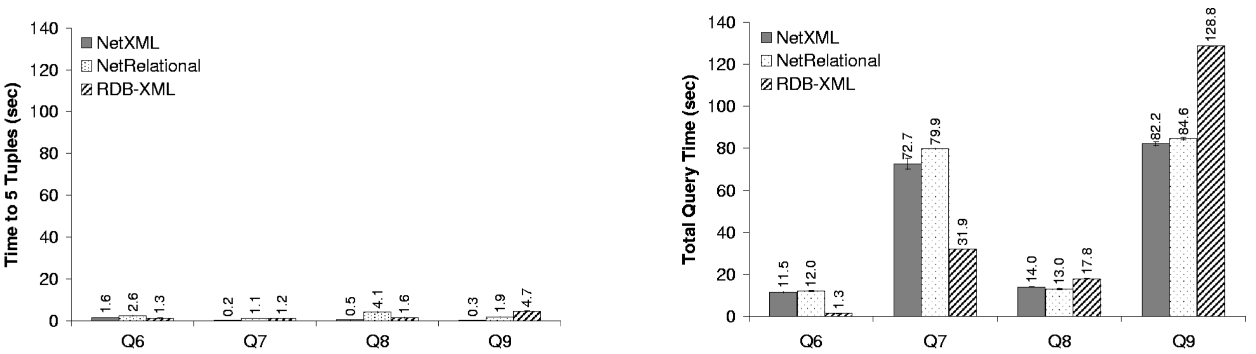
Figure 9: Experimental comparison of XML queries shows that *NetXML* has equally good total running time (and better time to first tuples) for simple extraction of small documents (Q1-Q3), and it scales better to large documents (Q4). *NetXML* performance is dramatically better than other systems for combining and restructuring XML data (Q5).

DB2 UDB 6.1, running on our 450MHz web server), read the data via Java JDBC and sent it as tuples across the network to our mediator, which added XML tags around the tuples and returned the results to our client. The relational database was indexed on all join and selection attributes. This approach is similar to those adopted by the SilkRoute [FTS99] and XPERANTO [CFI+00] mediator layers, which wrap an XML view interface over relational systems, except that we posed our queries directly in SQL and relied on a default mapping to tables.

- **NetRelational**, in which all tuples from the tables were simply scanned through JDBC and returned to our mediator, which queried them internally and converted them into XML results for our client. (This only utilized the relational aspects of our mediator.)
- **NetXML**, which took materialized XML views of the relational tables, read them via HTTP, and did XML query processing over the data.

As Figure 10 illustrates, the NetXML and NetRelational approaches tended to have very comparable running times, despite the fact that the XML-ified input tables were considerably larger. Moreover, the overhead inherent in JDBC and Java socket I/O appear to be more substantial than we had anticipated, so processing the query at the server was not necessarily a win. As expected, selection queries are significantly faster when done within the database engine. However, both join queries execute more slowly when done inside the relational engine. For query Q9, the join results are somewhat larger than the original sources when transferred relationally, but are far smaller than the XML-ified versions — the Java JDBC-based wrapper appears to be a bottleneck[8]. For query Q8, the results of the join are similar in size to the sum of the inputs, and yet the relational-mediator approach (which also uses JDBC and wrappers) is noticeably faster than the RDB/XML approach. We attribute this to the fact that the relational mediator can fetch the two data sources in parallel, whereas the in-database approach must serialize all of the join results through a single connection. Many data integration systems have tended to push down operations to the underlying

[8] We note that, although the database was running on a slower machine than our mediator, the machine was never CPU-bound. Moreover, actual query processing times for all queries were on the order of a few seconds when output of the query results was not required.




(a) First 5 tuples (b) Total query time

Figure 10: Experimental comparison of relational queries shows that *NetXML* performs nearly as well over data mapped into XML as a comparable relational-model data integration system does over relational data through a wrapper.

data sources as a general heuristic; we can see from this experiment that, as in distributed databases, such operations should really be considered in a cost-based framework.

Since mediator-based query processing of the materialized XML views is competitive with relational-model, wrapper-based query execution, it may be advantageous for data providers to materialize their data directly as XML, and to serve the XML for query processing at the client — rather than providing a virtual XML view over the underlying database and executing queries on each request. In our discussion of future work, we propose a further extension, which is to provide the ability to provide ScanXML-like services over such XML documents at the server, and retrieve only the desired XML subtrees.

## 5.3 Scalability of *NetXML*

A point of emphasis in our design of the *NetXML* architecture has been scalability to large XML documents. Most current systems tend to assume small XML files, so main-memory execution (and even unoptimized performance) are generally acceptable. However, as XML matures, an important application domain is expected to be querying and integration of data exchanged between groups or enterprises — such data may not be small. We observe from Figure 9 that *NetXML* is markedly better than the other systems at combining large XML structures. Equally important, it supports out-of-core execution, so it can (perhaps with reduced performance) execute with limited resources, as might occur when a single server is executing multiple client requests. The architectures of other systems result in significant memory footprints — *e.g.*, the XT processor required over 260MB of memory to load and scan the 39MB XML file of DBLP papers. Even a moderately high-end server is unlikely to be able to handle a large number of such requests without thrashing.

Many aspects of the *NetXML* architecture (*e.g.*, external sorts, grouping operators, hash and pipelined hash joins) will scale in predictable ways, as they are well-understood components of relational query engines. During general execution, the major difference between the *NetXML* architecture and a purely relational system is that some of the attributes within a tuple may be references into the tree manager, rather than values inside the tuple. However, any binding values needed for join, sort, and selection predicates are likely to be PCDATA rather than full XML subtrees[9] — these types of bindings will typically be inlined within the tuple. Hence, general query execution within *NetXML* is similar to an object-relational system in which

[9]Most query languages do not even define notions such as equality between subtrees. While some, such as Quilt, do allow path expressions over existing bindings, such operations can typically be pushed into ScanXML.

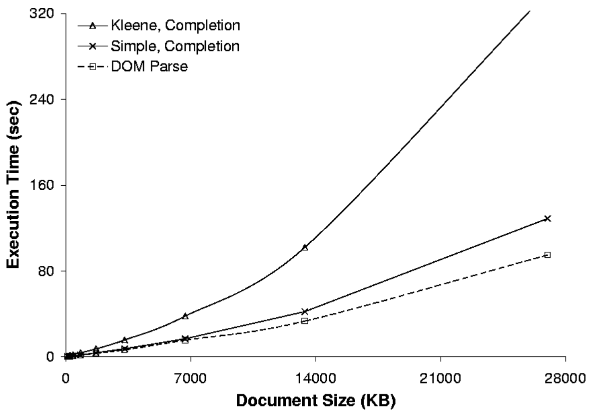



Figure 11: Scale-up results for query completion time on synthetic data for simple path query, Kleene-* query, and DOM parse. (Time to first 5 tuples in both queries was under 2 seconds.)

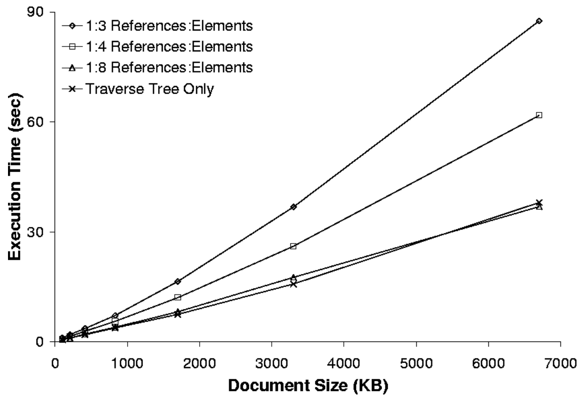



Figure 12: Scale-up results for graph query on synthetic data for simple path query, Kleene-* query, with tree query shown for reference.

BLOBs or other large ADTs are being carried through to the output, but are not used within the query plan.

The main concern for scalability, then, is the ScanXML operator and the components it uses (the XML Tree Manager and the various data structures required for supporting graphs). Thus we studied the performance of ScanXML for both simple path expressions and more complex ones (*i.e.*, those with more bindings and a Kleene-star operator in them), across a variety of document sizes. An experiment with subsets of DBLP data scaled in an entirely predictable way and is thus omitted. More interesting are our results over synthetic data, in which we could control more factors; see Figure 11. We observe that the time required to process a simple query grows at a rate only slightly faster than it takes to parse the XML and build a DOM tree (the approach taken by previous systems); ScanXML state-machine operation and Tree Manager overhead within *NetXML* is fairly low. Kleene query execution times grow at a significantly faster rate than the simple query, but this query produces many more tuples because it consists of two sibling Kleene-star path expressions — the cartesian product of these two bindings must be returned for each common subtree. The increase in execution times is closely approximated by the growth rate in the number of tuples produced.

## 5.4 Supporting Graph-Structured Data

Although many query languages tend view XML as a tree, there are many potentially interesting uses of XML as a representation for semistructured graphs, encoding edges as both elements `IDREFs`. Thus, the ScanXML operator has a number of features designed for traversing `IDREFs` as well as subelements. In Figure 12, we compare execution times of ScanXML across synthetic documents of different sizes. The different lines represent execution times when the ratio of `IDREFs` to elements is 1:8, 1:4, 1:3, and for comparison we include the execution time for a typical tree-traversing query over the mid-sized (1:4-ratio) documents. For proportionately low numbers of references, we see that the overhead in supporting graphs is relatively low; and even with fairly high numbers of traversed `IDREFs`, running times are reasonable, especially since initial results are output quickly. With a 1:3 ratio of `IDREFs` to elements, *NetXML* takes 90 seconds to return 193,000 leaf nodes from a 7MB synthetic graph. In contrast, the tree version of the same query yields only

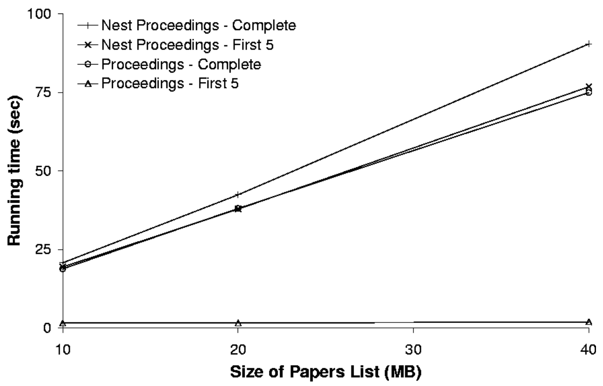



Figure 13: Comparison of nest and join operations combining DBLP papers and proceedings. Nest requires the (larger) inner relation to be read first, thus it has much longer time-to-first-results and slower overall time than the optimal join.

55,000 leaf nodes.

## 5.5 Performance of Nesting vs. Joining

In our final experiment, we examine the overhead incurred by the order-preserving **nest** operator when compared with a traditional hash join operator, which does not need to maintain any particular ordering. A general practice in query optimization, especially for network-based data, is to use the *smaller* join relation as the inner relation, and the larger as the outer relation. Not only does this reduce memory overhead in algorithms such as the hash join, but it also produces initial results earlier (assuming roughly equivalent transfer rates between sources). because the hash join must block until it has finished reading its inner relation.

Unfortunately, since **nest** is used to create a $1 : n$ hierarchical relationship, it must place the *larger* join relation as the inner relation. We can see in Figure 13 that as a result, **nest** not only performs more slowly than the hash join, but in fact the hash join completes its execution in the same amount of time as **nest** takes to output the first 5 tuples.

This suggests that performance would be considerably improved if it were possible to do the **nest** the same manner as the join, *i.e.*, if we did not have to maintain the parent-based ordering constraints on its output tuples. In situations where overall document order is unimportant but grouping is still required, we could indeed do this. We would use the smaller (parent) relation as the inner relation, and would still tag our output tuples with a grouping attribute corresponding to the parent tuple. However, the tuples would be output not in grouped order, but rather in the order of the child relation, except in cases of memory overflow. Thus we would need to perform a sort by grouping attribute at the end — and sorts are typically blocking operations. However, for certain types of interactive query applications where the final result is small enough to fit in memory, it should be possible to directly send the *NetXML* tuple stream (rather than the XML output of a **result** operator) to the client application. The client would be responsible for converting from tuples to XML output, and it might use an insertion sort or hash table to correctly cluster groups of nested data and display them.

## 6 Related Work

As XML has emerged as a medium for representing data as well as documents, a number of approaches have been proposed for evaluating XML queries. Most of these involve mapping XML to an existing database model and utilizing conventional query engine to do the core work. XML mapping strategies for relational [FK99, SGT+99, DFS99], object-oriented [vZAW99, LAW98], and semi-structured [GMW99] databases have all been implemented. The system's particular storage mapping may actually simplify certain path expressions, *e.g.* if a set of path expressions includes multiple data items that are mapped to the same tuple in a table. However, in the general case, indexing techniques such as join indices [Val87], access support relations [KM90], DataGuides [GW97], and t-indices [MS99] must be used to speed the processing of path expressions. These index structures describe the nodes reachable by certain classes of path expressions. The t-index, and to some extent the DataGuide, are particularly powerful structures that allow efficient computation of a wide range of regular path expression types. However, the actual index generation tends to be fairly complex and time consuming: DataGuides can be exponential in the size of the data and t-indices, while not exponential, are also costly to generate, especially for more complex path expression types. All of these techniques are ill-suited for a network-based query domain with autonomous data sources, because they invest a great deal of time into mapping, storing, and indexing data before it can be queried — the expectation is that this time will be amortized across multiple queries, but this assumption does not hold if the data must be frequently refreshed.

In contrast, XSLT processors such as XT and Xalan, as well as XML data integration systems such as MIX [BGL+99] and Niagara [CDTW00], perform operations over a main-memory representation of the XML parse (DOM) tree. These engines generally give faster response times to queries over small documents, but do not scale to large documents. In particular, they do not output results until they have parsed their entire input document(s), and they can only operate over in-memory structures. To make an analogy to relational systems, they may support pipelining during query execution, but the leaf-node "scan" operator for XML data is a blocking operation.

Our approach in *NetXML* has been to design an architecture that supports fully pipelined execution and leverages techniques developed for relational query processing over the network, such as those developed for data integration and distributed databases [KD98, UFA98, UF99, BFMV00, Col00]. Our internal execution model bears similarities to an object-relational engine, including the use of references for (potentially out-of-core) large objects — in our case, XML subtrees. This approach bears some similarities to the text ADT for structured text described in [BCD+98]. Our physical algebra has been designed with a "minimalist" philosophy, borrowing certain features from the nested relational algebra [RKS88], but avoiding much of the complexity present in many XML algebras such as [FSW00].

The key component responsible for outputting pipelined tuples from an input document is the ScanXML operator, which uses a hierarchical set of finite state machines to traverse the input as it is streaming into the system. Unlike a repository system, ScanXML does not break the XML document into components requiring indexing and later reassembly; unlike existing main-memory systems, it scales beyond memory and it supports efficient traversal of `IDREF`s for graph-structured input.

The ScanXML pattern matching approach bears some similarity to the Knuth-Morris-Pratt substring-matching algorithm, which also uses finite state machines to perform matches — however, our algorithm supports hierarchies of bindings and path expressions, traverses graph structure, and avoids cycles. The XFilter operator [AF00] also uses a similar technique for filtering XML documents according to an XPath expression, but not for query processing. Finally, ScanXML has similar goals to the *scan* logical operator proposed by Cluet and Moerkotte for tree-structured data in [CM97], but our work includes a specific algorithm, support for graph-structured data, and an experimental evaluation.

## 7 Conclusions and Future Work

Technology trends in networking and data exchange have increased the need for an XML query processor for network-bound data. Applications such as integration of intranet or Internet-based data, query and transformation systems for XML documents, "live" data analysis tools, and electronic commerce all require the ability to query, combine, and restructure the content of one or more XML documents of arbitrary size. In addition, applications need to support a "streaming" query processing model that produces results as soon as possible. In many cases, the sources of data for such applications are autonomous data that is stored remotely and subject to updates at any point.

This paper described *NetXML*, the first XML query processing system that satisfies the above requirements. *NetXML* is based on an architecture which extends tuple-oriented, relational techniques such as pipelining, as well as recently developed adaptive query processing techniques for network-based relational data, to work directly on XML. At the core of *NetXML* is ScanXML, an operator that maps selected portions of an XML document into tuples in a streaming fashion. In addition, *NetXML* incorporates basic operators for combining and restructuring tuples of XML subtrees into new XML content. We described a set of experiments that demonstrate that *NetXML* provides superior performance to existing XML query systems when applied to network-bound data. In conclusion, our results suggest that it is indeed possible to construct a native query processor for XML data that rivals the efficiency of a relational query engine.

The architecture of *NetXML* suggests several directions for future research. Clearly, a next step must be to build a query optimizer for a system such as *NetXML*. Such an optimizer should support the interleaving of optimization and execution, and therefore should be able to use statistics obtained during query execution to change its behavior (in the spirit of [BFMV00, UFA98, IFF+99, KD98, Col00]). We also plan to incorporate adaptive features such as eddies [AH00] and the ability to output partial or approximate answers from blocking operators [STD+00] to our system.

Another important avenue of research is in the area of processing XML as graph-structured data — the graph model is extremely useful for many applications that do not map well into traditional databases (in particular, we are looking at domains such as cellular biology). Our current implementation can process graph-structured input, but it assumes that the CONSTRUCT portion of the query produces tree-structured output. We anticipate generating graph-structured output with use of Skolem functions and a new "fuse" operator that consolidates element structures across the query output. Another area of future investigation concerns the performance implications of ordered versus unordered execution; our comparison of the ordered nest versus unordered join suggests that there are tradeoffs between these two models.

Finally, we note that another useful application of our ScanXML operator may be as a simple "XML selector" interface over XML files (which may be documents or materialized views). A mediator would send a document identifier and path expression to the XML selector, which would return the set of XML subtrees matching that expression. Since the XML selector would run locally at the server, this could greatly reduce the amount of XML data transferred across the network.

## References


[AF00] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, Cairo, Egypt*, 2000.

[AH00] Ron Avnur and Joseph M. Hellerstein. Continuous query optimization. In *SIGMOD 2000, Proceedings ACM SIGMOD International Conference on Management of Data, May 14-19, 2000, Dallas, Texas, USA*. ACM Press, 2000.

[BCD+98] L. J. Brown, Mariano P. Consens, Ian J. Davis, Chris R. Palmer, and Frank Wm. Tompa. A structured text ADT for object-relational databases. In *TAPOS*, volume 4(4), pages 227–244, 1998.

[BFMV00] Luc Bouganim, Francoise Fabret, C. Mohan, and Patrick Valduriez. Dynamic query scheduling in data integration systems. In *ICDE 2000, San Diego, CA USA*, pages 425–434, 2000.

[BGL+99] Chaitanya K. Baru, Amarnath Gupta, Bertram Ludäscher, Richard Marciano, Yannis Papakonstantinou, Pavel Velikhov, and Vincent Chu. XML-based information mediation with MIX. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 597–599, 1999.

[CCD+98] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: A graphical language for querying and reshaping XML documents. W3C Query Language Workshop, http://www.w3.org/TandS/QL/QL98/pp/xml-gl.html, December 1998.

[CDTW00] Jianjun Chen, David DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD 2000, Proceedings ACM SIGMOD International Conference on Management of Data, May 15-18, 1999, Dallas, TX, USA*, 2000.

[CFI+00] Michael J. Carey, Daniela Florescu, Zachary G. Ives, Ying Lu, Jayavel Shanmugasundaram, Eugene Shekita, and Subbu Subramanian. XPERANTO: Publishing object-relational data as XML. In *WebDB*, 2000.

[CM97] Sophie Cluet and Guido Moerkotte. Query processing in the schemaless and semistructured context. Unpublished manuscript, 1997.

[Col00] Richard L. Cole. A decision theoretic cost model for dynamic plans. *IEEE Data Engineering Bulletin*, 23(2):34–41, 2000.

[CRF00] Donald D. Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB (Informal Proceedings) 2000*, pages 53–62, 2000.

[DFF+99] Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. In *Proceedings of the International Word Wide Web Conference, Toronto, CA*, 1999.

[DFS99] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 431–442, 1999.

[FK99] Daniela Florescu and Donald Kossman. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report 3684, INRIA, March 1999.

[FSW00] Mary Fernandez, Jerome Simeon, and Philip Wadler. An algebra for xml query. Available from www.cs.bell-labs.com/ wadler/topics/xml.html#xalgebra, 2000.

[FTS99] Mary Fernandez, Weng-Chiew Tan, and Dan Suciu. SilkRoute: Trading between relations and XML. In *Ninth International World Wide Web Conference*, November 1999.

[GMW99] Roy Goldman, Jason McHugh, and Jennifer Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *ACM SIGMOD Workshop on the Web (WebDB), Philadelphia, PA*, pages 25–30, 1999.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 436–445, 1997.

[IFF+99] Zachary G. Ives, Daniela Florescu, Marc T. Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 299–310, 1999.

[KD98] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 106–117. ACM Press, 1998.

[KM90] Alfons Kemper and Guido Moerkotte. Access support in object bases. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 364–374, 1990.

[LAW98] Tirthankar Lahiri, Serge Abiteboul, and Jennifer Widom. Ozone: Integrating structured and semistructured data. Technical report, Stanford University, October 1998.

[MFK+00] Ioana Manolescu, Daniela Florescu, Donald Kossman, Florian Xhumari, and Don Olteanu. XML and relational: How to live with both. In *VLDB 2000, Cairo, Egypt*, September 2000.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999*, pages 277–295, 1999.

[NDM+00] Jeffrey Naughton, David DeWitt, David Maier, Jianjun Chen, Leonidas Galanis, Kristin Tufte, Jaewoo Kang, Qiong Luo, Naveen Prakash, Feng Tian, Jayavel Shanmugasundaram, Chun Zhang, Ravishankar Ramamurthy, Bruce Jackson, Yuan Wang, Anurag Gupta, and Rushan Chen. The Niagara internet query system. Available from www.cs.wisc.edu/niagara/papers/NIAGRAVLDB00.v4.pdf, 2000.

[RKS88] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases. *TODS*, 13(4):389–417, 1988.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html, September 1998.

[Sah00] Arnaud Sahuguet. Kweelt, the making-of: Mistakes made and lessons learned. Available from db.cis.upenn.edu/DL/kweelt-TR.pdf, 2000.

[SCN] Reference removed for double blind reviewing. Technical report.

[SGT+99] Jayavel Shanmugasundaram, H. Gang, Kristin Tufte, Chun Zhang, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, Scotland*, pages 302–304, 1999.

[STD+00] Jayavel Shanmugasundaram, Kristin Tufte, David J. DeWitt, Jeffrey F. Naughton, and David Maier. Architecting a network query engine for producing partial results. In *WebDB (Informal Proceedings) 2000*, pages 17–22, 2000.

[Tam] Tamino: The information server for electronic business. http://www.softwareag.com/tamino.

[UF99] Tolga Urhan and Michael J. Franklin. XJoin: Getting fast answers from slow and bursty networks. Technical Report CS-TR-3994, University of Maryland, College Park, February 1999.

[UF00a] Tolga Urhan and Michael J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2), June 2000.

[UF00b] Tolga Urhan and Michael J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[UFA98] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 130–141, 1998.

[Val87] Patrick Valduriez. Join indices. *TODS*, 12(2):218–246, 1987.

[vZAW99] Roelof van Zwol, Peter M.G. Apers, and Annita N. Wilschut. Modelling and querying semistructured data with MOA. In *Proceedings of the Workshop on Semi-Structured Data and Non-Standard Data Formats, Jerusalem, Israel*, 1999.

[WA91] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. First International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 68–77, December 1991.

[XLN] eXcelon: The XML application development environment. http://www.odi.com/excelon/main.htm.

[XML98] Extensible markup language (XML) 1.0. http://www.w3.org/TR/1998/REC-xml-19980210, 10 February 1998. W3C Recommendation.

[XSL99] XSL Transformations (XSLT), version 1.0. http://www.w3.org/TR/xslt, 13 August 1999. W3C Working Draft.