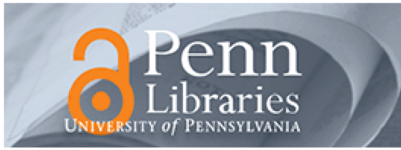


University of Pennsylvania
ScholarlyCommons

Departmental Papers (CIS) Department of Computer & Information Science

December 2002

An XML Query Engine for Network-Bound Data

Zachary G. Ives
University of Pennsylvania, zives@cis.upenn.edu

Alon Y. Halevy
University of Washington

Daniel S. Weld
University of Washington

Follow this and additional works at: http://repository.upenn.edu/cis_papers

Recommended Citation

Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld, "An XML Query Engine for Network-Bound Data", . December 2002.

Postprint version. Published in *VLDB Journal : The International Journal on Very Large Data Bases*, Volume 11, Number 4, December 2002, pages 380-402. The original publication is available at www.springerlink.com.
Publisher URL: http://dx.doi.org/10.1007/s00778-002-0078-5

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_papers/121
For more information, please contact libraryrepository@pobox.upenn.edu.

An XML Query Engine for Network-Bound Data

Abstract
XML has become the *lingua franca* for data exchange and integration across administrative and enterprise boundaries. Nearly all data providers are adding XML import or export capabilities, and standard XML Schemas and DTDs are being promoted for all types of data sharing. The ubiquity of XML has removed one of the major obstacles to integrating data from widely disparate sources –- namely, the heterogeneity of data formats.

However, general-purpose integration of data across the wide area also requires a query processor that can query data sources on demand, receive streamed XML data from them, and combine and restructure the data into new XML output -- while providing good performance for both batch-oriented and ad-hoc, interactive queries. This is the goal of the Tukwila data integration system, the first system that focuses on network-bound, dynamic XML data sources. In contrast to previous approaches, which must read, parse, and often store entire XML objects before querying them, Tukwila can return query results even as the data is streaming into the system. Tukwila is built with a new system architecture that extends adaptive query processing and relational-engine techniques into the XML realm, as facilitated by a pair of operators that incrementally evaluate a query's input path expressions as data is read. In this paper, we describe the Tukwila architecture and its novel aspects, and we experimentally demonstrate that Tukwila provides better overall query performance and faster initial answers than existing systems, and has excellent scalability.

Keywords
XML, query processing, data streams, data integration, web and databases

Comments
Postprint version. Published in *VLDB Journal : The International Journal on Very Large Data Bases*, Volume 11, Number 4, December 2002, pages 380-402. The original publication is available at www.springerlink.com.
Publisher URL: http://dx.doi.org/10.1007/s00778-002-0078-5

This journal article is available at ScholarlyCommons: http://repository.upenn.edu/cis_papers/121

VLDB Journal manuscript No.
(will be inserted by the editor)

An XML Query Engine for Network-Bound Data

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

Department of Computer Science and Engineering, Box 352350, University of Washington, Seattle, WA 98195-2350
e-mail: {zives,alon,weld}@cs.washington.edu

Received: date / Revised version: date

Abstract XML has become the *lingua franca* for data exchange and integration across administrative and enterprise boundaries. Nearly all data providers are adding XML import or export capabilities, and standard XML Schemas and DTDs are being promoted for all types of data sharing. The ubiquity of XML has removed one of the major obstacles to integrating data from widely disparate sources – namely, the heterogeneity of data formats.

However, general-purpose integration of data across the wide area also requires a query processor that can query data sources on demand, receive streamed XML data from them, and combine and restructure the data into new XML output — while providing good performance for both batch-oriented and ad-hoc, interactive queries. This is the goal of the Tukwila data integration system, the first system that focuses on network-bound, dynamic XML data sources. In contrast to previous approaches, which must read, parse, and often store entire XML objects before querying them, Tukwila can return query results even as the data is streaming into the system. Tukwila is built with a new system architecture that extends adaptive query processing and relational-engine techniques into the XML realm, as facilitated by a pair of operators that incrementally evaluate a query's input path expressions as data is read. In this paper, we describe the Tukwila architecture and its novel aspects, and we experimentally demonstrate that Tukwila provides better overall query performance and faster initial answers than existing systems, and has excellent scalability.

* Supported in part by an IBM Research Fellowship. *Present address:* Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

Key words XML, query processing, data streams, data integration, web and databases

1 Introduction

For many years, a wide variety of domains, ranging from scientific research to electronic commerce to corporate information systems, have had a great need to be able to integrate data from many disparate data sources at different locations, controlled by different groups. Until recently, one of the biggest obstacles was the heterogeneity of the sources' data models, query capabilities, and data formats. Even for the most basic data sources, custom *wrappers* would need to be developed for each data source and each data integration *mediator*, simply to translate mediator requests into data source queries, and to translate source data into a format that the mediator can handle.

The emergence of XML as a common data format, as well as the support for simple web-based query capabilities provided by related XML standards, has suddenly made data integration practical in many more cases. XML itself does not solve all of the problems of heterogeneity — for instance, sources may still use different tags or terminologies — but often, data providers come to agreement on standard schemas, and in other cases, we can use established database techniques for defining and resolving mappings between schemas. As a result, XML has become the standard format for data dissemination, exchange, and integration. Nearly every data management-related application now supports the import and export of XML, and standard XML Schemas and DTDs are being developed within and among enterprises to facilitate data sharing (instances of these are published at the BizTalk and OASIS web sites[1]). Language-

[1] See www.biztalk.org and www.xml.org.

and system-independent protocols such as the various web services standards, Microsoft's .NET [NET01] initiative, and Sun's JXTA [JXT01] peer-to-peer protocols use XML to represent transactions and data.

Processing and integrating XML data poses a number of challenges. In many data integration applications, XML is merely a "wire format," the result of some view over a live, dynamic, non-XML source. In fact, the source may only expose subsets of its data as XML, via a query interface with access restrictions, e.g., the source may only return data matching a selection value, as in a typical web form. Since the data is controlled and updated externally and only available in part, this makes it difficult or impossible to cache the data. Moreover, the data sources may be located across a wide-area network or the Internet itself, so queries must be executed in a way that is resilient to network delays. Finally, the sources may be relatively large, in the 10s to 100s of MB or more, and that may require an appreciable amount of time to transfer across the network and parse. We refer to these types of data sources as "network-bound": they are only available across a network, and the data can only be obtained through reading and parsing a (typically finite) stream of XML data.

To this point, integration of network-bound, "live" XML data has not been well studied. Most XML work in the database community has focused on designing XML repositories and warehouses [Aea01, XLN, Tam, GMW99, FK99b, AKJK+02, KM00, BBM+01, SGT+99], exporting XML from relational databases [FTS99, FMS01a, CFI+00], adding information retrieval-style indexing techniques to databases [NDM+01, FMK00], and on supporting query subscriptions or continuous queries [Aea01, CDTW00, AF00] that provide new results as documents change or are added.

Clearly, both warehousing and indexing are useful for storing, archiving, and retrieving file-based XML data or documents, but for many integration applications, support for queries over dynamic, external data sources is essential. This requires a query processor that can request data from each of the sources, combine this data, and perhaps make additional requests of the data sources as a result. To the best of our knowledge, no existing system provides this combination of capabilities. The Web community has developed a class of query tools that are restricted to single-documents and not scalable to large documents. The database community's web-based XML query engines, such as Niagara and Xyleme, come closer to meeting the needs of data integration, but they are still oriented towards smaller documents (which may be indexable or warehoused), and they give little consideration to processing data from slow sources or XML that is larger than memory.

Query processing for data integration poses a number of challenges, because the data is not tightly controlled or exclusively used by the data integration system. For example, query optimization is difficult in the absence of data source statistics. This subjects has been the focus of *adaptive query processing* research discussed elsewhere (e.g., [RS86, WA91, HS93, UF00, HH99, IFF+99, UF01, UFA98, KD98, AH00]). However, adaptive query processing research has generally focused on relational query processing, not on XML. One of the major advantages offered by relational query processing has been a pipelined execution model in which new tuples can be read directly off the network and fed into the query plan. This presents a number of significant benefits for data integration and for enabling adaptivity:

– A single execution pipeline does not require materialization operations, or pre-parsing or preprocessing of an XML document, so initial answers will be returned more quickly. This satisfies an important desideratum for interactive data integration applications.
– A single pipeline provides the most opportunities for exploiting parallelism and for flexibly scheduling the processing of tuples. This enables the use of techniques such as the pipelined hash join [RS86, WA91, HS93, UF00, HH99, IFF+99] as well as eddies [AH00].

Pipelining and adaptive query processing techniques have largely been confined to the relational data model. One of the contributions of this paper is a new XML query processing architecture that emphasizes pipelining the XML data streaming into the system, and which facilitates a number of adaptive query processing techniques.

As described in Section 2, XML queries operate on *combinations of input bindings*: patterns are matched across the input document, and each pattern-match binds an input tree to a variable. The query processor iterates through all possible combinations of assignments of bindings, and the query operators are evaluated against each successive combination. At first glance, this seems quite different from the tuple-oriented execution model of the relational world, but a closer examination reveals a useful correspondence: if we assign each attribute within a tuple to a variable, we can view each legal combination of variable assignments as forming a tuple of binding values (where the values are XML trees or content). In this paper, we describe an XML query processing architecture, implemented in the Tukwila system, which exploits the correspondence between the relational and XML processing models in order to provide adaptive XML query processing capabilities, and thus to support efficient network-bound querying, even in the presence

of delays, dynamic data, and source failures. This architecture includes the following novel features:

- Support for efficient processing of scalar and structured XML content. Our architecture maps scalar (e.g., text node) values into a tuple-oriented execution model that retains the efficiencies of a standard relational query engine. Structured XML content is mapped into a Tree Manager that supports complex traversals, paging to disk, and comparison by identity as well as value.
- A pair of streaming XML input operators, *x-scan* and *web-join*, that are the enablers of our adaptive query processing architecture. Each of these operators transforms an incoming stream of XML data into an internal format that is processed by the query operators. X-scan matches a query's XPath expressions against an input XML stream and outputs a set of tuples, whose elements are bindings to subtrees of XML data. Web-join can be viewed as a combination of an x-scan and a dependent join — it takes values from one input source, uses them to construct a series of dynamic HTTP requests over Internet sources, and then joins the results.
- A set of physical-level algebraic operators for combining and structuring XML content and for supporting the core features of XQuery [BCF+02], the World Wide Web Consortium XML query language specification, which is nearing completion.

In this paper, we describe Tukwila's architecture and implementation, and we present a detailed set of experiments that demonstrate that the Tukwila XML query processing architecture provides superior performance to existing XML query systems within our target domain of network-bound data. Tukwila produces initial results rapidly *and* completes queries in less time than previous systems, and it also scales better to large XML documents. The result is the first scalable query processor for network-bound, live XML data. We validate Tukwila's performance by comparing with leading XSLT and data integration systems, under a number of different classes of documents and queries (ranging from document retrieval to data integration); we show that Tukwila can read and process XML data at a rate roughly equivalent to the performance of SQL and the JDBC protocol across a network; we show that Tukwila's performance scales well as the complexity of the path expressions is increased; and we show that Tukwila's *x-scan* operator can scale well to large (100's of MBs) graph-structured data with `IDREF`s.

The remainder of this paper is structured as follows. Section 2 describes the basics of querying for XML; then Section 3 begins by describing standard techniques for XML query processing, and finishes by presenting the Tukwila architecture and emphasizing its differences. We then describe the XML query operators and cost model in Section 6, and how the operators can be extended to support a graph data model in Section 7. Section 8 provides experimental validation of our work. Section 9 discusses related work, and we conclude in Section 10.

2 Querying XML

During the past few years, numerous alternative query languages and data models for XML have been proposed, including XML-QL [DFF+99] and XSLT [XSL99]. XSLT is a single-document-oriented query language consisting of *rules*: each rule matches a particular path in an XML tree and applies a transformation to the underlying subtree. XML-QL was a data-oriented query language, adapted from the semistructured database community, and could join data across documents, but had few document-oriented features.

Recently the World Wide Web Consortium has combine the features of these languages with its XQuery language specification [BCF+02] and accompanying data model [FMN02]. The XQuery data model defines an XML document as a tree of ordered nodes of different content types (e.g., element, processing instruction, comment, text), where element nodes may also have unordered attributes. For example, the XML document of Figure 1 can be modeled as the tree of Figure 2. In this diagram, we have represented elements as labeled nodes, text content as leaf nodes, attributes as annotations beside their element nodes, and special `IDREF`-typed reference attributes as dashed edges from their elements to their targets (where the target element is identified by an `ID`-typed attribute of the same name).

The XQuery language is designed to extract and combine subtrees within this data model. It is generally based on a `FOR-LET-WHERE-RETURN` structure (commonly known as a "flower" expression): the `FOR` clause provides a series of XPath expressions for selecting input nodes, the `LET` clause similarly defines collection-valued expressions, the `WHERE` clause defines selection and join predicates, and the `RETURN` clause creates the output XML structure. XQuery expressions can be nested within a `RETURN` clause to create hierarchical output, and, like OQL, the language is designed to have modular and composable expressions. Furthermore, XQuery supports several features beyond SQL and OQL, such as arbitrary recursive functions.

XQuery execution can be considered to begin with a *variable binding* stage: the `FOR` and `LET` XPath expressions are evaluated as traversals through the data model tree, beginning at the root. The tree matching the end of

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

Fig. 1 Sample XML document representing book and publisher data

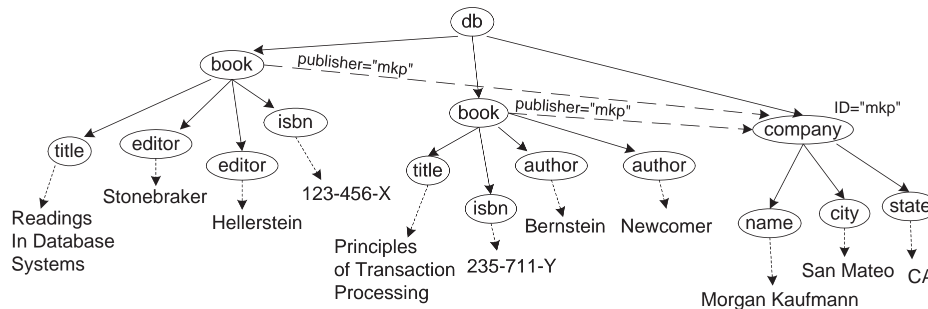


Fig. 2 Graph representation for Figure 1. Dashed edges illustrate relationships defined by IDREFs; dotted edges point to text nodes.

an XPath is *bound* to the FOR or LET clause's variable. If an XPath has multiple matches, a FOR clause will iterate and bind its variable to each, executing the query's WHERE and RETURN clause for each assignment. The LET clause will return the collection of all matches as its variable binding. A query typically has numerous FOR and LET assignments, and legal combinations of these assignments are created by iterating over the various query expressions.

An example XQuery appears in Figure 3. We can see that the variable $b is assigned to each book subelement under the db element in document books.xml; $t is assigned the title within a given $b book, and so forth. Our version of XPath includes extensions allowing for disjunction along any edge (e.g., $n can be either an editor or author), as well as a regular-expression-like Kleene star operator (not shown).

In the example, multiple match combinations are possible, so the variable binding process is performed in the following way. First, the $b variable is bound to the first occurring book. Then the $t and $n variables are bound in order to all matching title and editor or author subelements, respectively. Every possible pairing of $t and $n values for a given $b binding is evaluated in a separate iteration; then the process is repeated for the next value of $b. We observe that this process is virtually identical to a relational query in which we join books with their titles and authors — we will have a tuple for each possible ⟨title, editor|author⟩ combination per book. The most significant difference is in the terminology; for XQuery, we have an "iteration" that produces a binding for each variable, and in a relational system we have a "tuple" with a value in each attribute.

```
<result> {
  FOR $b IN document("books.xml")/db/book,
      $t IN $b/title/data(),
      $n IN $b/(editor|author)/data()
  RETURN <item>
             <person>{$n}</person>
             <pub>{$t}</pub>
         </item>
} </result>
```

Fig. 3 XQuery query that finds the names of people who have published and their publications. The FOR clause specifies XPath expressions describing traversals over the XML tree, and binds the subtrees to variables (prefixed with dollar signs).

The RETURN clause specifies a tree-structured XML constructor that is output on each iteration, with the variables replaced by their bound values. Note that variables in XQuery are often bound to XML subtrees (identified by their root nodes) rather than to scalar values. The result of the example query appears in Figure 4. An item element is output for each possible combination of bindings.

```
<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>
```

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

Finally, we note that some XML data makes use of `IDREF` attributes to represent links between elements (the dashed lines in Figure 2). `IDREF`s allow XML to encode graph-structured as well as tree-structured data. The current XQuery proposal has limited support for `IDREF` traversal, only offering traversal of a fixed number of edges, rather than transitive closure. However, since there are many interesting applications of graph-structured XML data, we investigate querying it in this paper using an extended version of XQuery.

3 Previous Approaches to XML Processing

As discussed in the previous section, the XML data model and XQuery language are considerably more complex than simple relational query processing because of their reliance on path expressions. In particular, the hierarchical nature of XML typically means that a document can be normalized to a single relational table, but a set of tables that have parent-child foreign-key relationships.

People have generally attempted to handle the XML processing problem using one of four methods: (1) focus on techniques for "shredding" XML into tables, combining the tables, and re-joining the results to produce XML output; (2) make a few modifications to object-oriented or semi-structured databases, which also inherently support hierarchy, so they support XML; (3) use a top-down tree-traversal strategy for executing queries; (4) use a custom wrapper at the source end for index-like retrieval of only the necessary content. Before we describe the Tukwila architecture, it is useful to briefly examine these previous approaches, including their relative strengths and weaknesses.

Relational databases A variety of research projects at INRIA [FK99a, MFK+00], AT&T Labs [DFS99, FTS99, FMS01b], IBM Almaden [CFI+00, SKS+01, TVB+02], and the University of Wisconsin [SGT+99] focused on the problems of mapping XML data to and from relational databases. Today, all of the major relational DBMS vendors build upon this work and provide support some form of XML export (e.g., [Rys01, BKKM00]). In general, results suggest that a relational database is generally not ideal for storing XML, but when the XML data either originates from relational tables or is slow to change, it may be an acceptable solution. Significant benefits include scalability and support for value-based indexes; drawbacks include expensive document load times and expensive reconstruction of XML results. The relational query optimizer can improve performance significantly if the XML query maps to simple SQL, but it frequently makes poor decisions for more complex queries, since it does not optimize with knowledge of XML semantics [ZND+01].

Object-oriented and semi-structured databases Several major commercial OODBs, including Poet and ObjectStore, have been adapted to form new XML databases. They provide some benefits over strictly relational engines because their locking and indexing structures are designed for hierarchical data; however, OO query optimizers are still generally relatively weak. The Lore semi-structured database [GMW99], which has a number of special indexing structures, has also been adapted to XML (though performance was shown to be poor relative to a relational system storing XML [FK99a]). Several native XML databases [KM00, BBM+01, AKJK+02, MAM01] are also under development. Most of these systems focus on issues relating to efficiently storing and traversing hierarchical objects, as well as on indexing. For more details, please see the discussion of related work in Section 9.

Web-oriented DOM processors The techniques mentioned above focus on storage and retrieval of XML content. Of course, XML is expected to also be a format for content *transmission* across networks, and some of this content will be of a transient nature — there is a need for systems that format, query, combine, and present it without storing it. For this domain, an entirely different class of query processors has been developed. These processors, such as the XT, Xalan, and MSXML XSLT engines and the Niagara system from the University of Wisconsin [NDM+01] typically work by parsing an XML document into an in-memory DOM tree; then they traverse the tree using XPath expressions, extract the specified content, and combine it to form a new document. For transient data of small size, this performs much better

than storing the data on disk and then querying it; however, it is limited by available memory, and it cannot begin producing answers until after the input document has been parsed. (For a large document over a slow Internet connection, this may be a significant delay.)

Other web-oriented processors The MIX system from the University of California-San Diego [BGL+99] is web-based, but has a pull-based, lazy XML evaluation method where the query processor can request specific regions from the data from the mediator as required. This allows for better scalability than the DOM approach, but suffers from two potential problems. First, it requires a custom wrapper at the source, which processes the pull-based messages. Second, one of the major costs in wide-area communication is round-trip time, and the pull-based method requires considerable communication between data source and consumer.

4 The Tukwila XML Architecture

Our intent in designing the Tukwila system is to provide the scalability and query optimization of a full-fledged database while maintaining the interactive performance characteristics of the web-based systems. We want to be able to support naive producers of XML content, but also to take advantage of more complex systems that can process queries (or portions of queries) directly.

Although the Tukwila project investigates both query optimization and execution for integrating network-bound XML data, in this paper we shall focus on the query execution architecture and operators. A brief discussion of the query optimizer and cost model appears in Section 6.

The Tukwila architecture is based on the following observations:

1. The basic execution model of XQuery is very similar to that for relational databases: XQuery evaluates the `WHERE` and `RETURN` clauses over every possible combination of input bindings, and each combination of bindings can be viewed as a tuple.
2. The `FOR` and `LET` clauses bind input variables using XPath expressions, which typically are traversals over XML parse tree structure, occasionally with selection or join predicates. The majority of XPath expressions traverse in the downward ("forwards") direction, which matches the order in which a parser encounters the XML elements as it reads an input stream.
3. Most selection and join predicates in XQuery involve scalar (text node) data, rather than complex XML hierarchies. Bindings to hierarchical XML data are most commonly used only in the `RETURN` clause.
4. The majority of XML processors use DOM-based parsers, which must construct the entire XML parse tree before query processing begins. Incremental parsing, combined with pipeline-based execution[2] as in relational databases, would produce significant benefits. First, it can reduce the time to first answers, as results percolate through the query plan more quickly. Second, the increased parallelism of pipelined operators allows for *adaptive scheduling*, which allows the query processor to overlap I/O with computation [IFF+99] and prioritize important work [UF01].

Based on these observations, we have designed an architecture that is particularly efficient for common-case query execution.

4.1 The Tukwila Execution Engine

The core operations performed by most queries are path matching, selecting, projecting, joining, and grouping based on *scalar* data items. Our engine can support these operations with very low overhead, and in fact it can approach relational-engine performance on simple queries. Our query execution engine also emphasizes a relational-like pipelined execution model, where each "tuple" consists of bindings to XML content rather than simple scalar attributes. This gives us the time-to-first-tuple benefits cited previously, and it has the benefit of leveraging the best techniques from relational query processing.

A high level view of the Tukwila architecture is illustrated in Figure 5. The query optimizer passes a plan to the execution engine; at the leaf nodes of this plan are x-scan operators. The x-scan 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 the XML Tree Manager, and (4) output tuples containing scalar values and references to subtrees. The tuples are fed into the remaining operators in the query execution plan, where they are combined and restructured. As it flows through the operators near the top of the query plan, each tuple is annotated with information describing what content should be output as XML, and how that content should be tagged and structured. Finally, the XML Generator processes these tagged tuples and returns an XML result stream to the user.

In a sense, the "middle portion" of our architecture (represented by the "Query Operators" box and the Page Manager) resembles a specialized object-relational database core. Tuples contain attribute values that have been

[2] Note that while not all operators are pipelineable, a fairly large class of queries can be answered with pipelined operators.

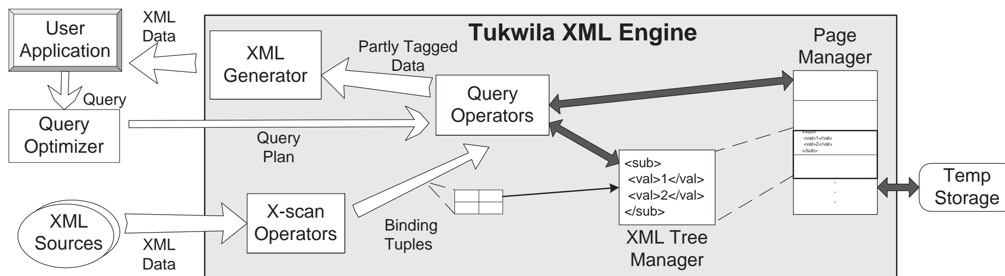


Fig. 5 Architecture of the Tukwila query execution engine. After a query plan arrives from the optimizer, data is read from XML sources and converted by x-scan operators into output tuples of subtree bindings. The subtrees are stored within the Tree Manager (backed by a virtual page manager), and tuples contain references to these trees. Query operators combine binding tuples and add tagging information; these are fed into an XML Generator that returns an XML stream.

bound to variables; these values can be scalar, and stored directly within the tuple, or they can be XML structures, similar to CLOBs (character large objects) in an OR database — XML structures are stored separately from the tuple, in an *XML Tree Manager*, which is a virtual memory manager for XML subtrees. (Note that we do not attempt to support any other object-oriented types, nor do we implement methods.) The tuples being pipelined through the query plan contain references to subtrees within this Tree Manager, so if multiple variables are bound to an XML tree, the data does not need to be duplicated. Our query operators can manipulate both references within the Tree Manager and values embedded within the tuple, so both object-based and value-based operations are possible — including grouping, nesting, and aggregation. XML subtrees are reference-counted and garbage-collected when all tuples referring to them have been processed by the system.

The Tukwila architecture allows us to leverage a number of components from the relational world, such as most of the basic memory management strategies and operators; it is also straightforward to make use of adaptive query processing operators when these are appropriate for the query semantics. We discuss Tukwila's query operators later in this paper.

4.2 Pipelining XML Data

One of the virtues of the flat relational model is its extreme flexibility as a representation. For example, since relations are order-independent, joins can be commuted and non-order-preserving algorithms can be used. Techniques for query decorrelation can be used. Predicates can be evaluated early or late, depending on their selectivity.

A hierarchical data model, such as XML, is often more intuitive to the data consumer (since it centers on a particular concept), but the natural model of execution — breaking a query by levels in the hierarchy — is not necessarily the most efficient. Even more restrictive than hierarchy is ordering: by default, XQuery is very procedural, specifying an order of iteration over bindings, an implicit order of evaluating nested queries, and so forth.

One possible execution model for XQuery would resemble that for nested relations, and in fact "recursive" algebras for nested relations, in which all operators can operate at any level of nesting in the data, have been proposed and implemented (e.g., [HSR91, Col89]). However, we have a preference for mapping XML — even hierarchical XML — into something more resembling the "flat" relational model: an XML document gets converted into a relation in which each attribute represents the value of a variable binding, and position is encoded using counter or byte-offset information. Each such binding may contain arbitrary XML content; but unlike in a nested relational model, the query may only manipulate the top level of the structure. Nested structure must be expanded before it can be manipulated.

This architecture allows us to directly leverage relational query execution and optimization techniques, which are well-understood and provide good performance. Moreover, we believe that, particularly in the case of data integration, we can get better performance from an execution model that preserves structure but has "flat" query operators, for three key reasons. First, many data integration scenarios require significant restructuring of XML content anyway — hence, it makes little sense to spend overhead maintaining structure that will be lost in the end. Second, we can make the unnesting and renesting operations inexpensive: our x-scan algorithm provides a low-overhead way to unnest content, and we can insert additional metadata into every tuple to make it easy to re-nest or re-order values. Third, we believe that there is an inherent overhead in building algorithms that preserve multiple levels of hierarchy, and as a result we feel a "RISC" philosophy is most appropriate.

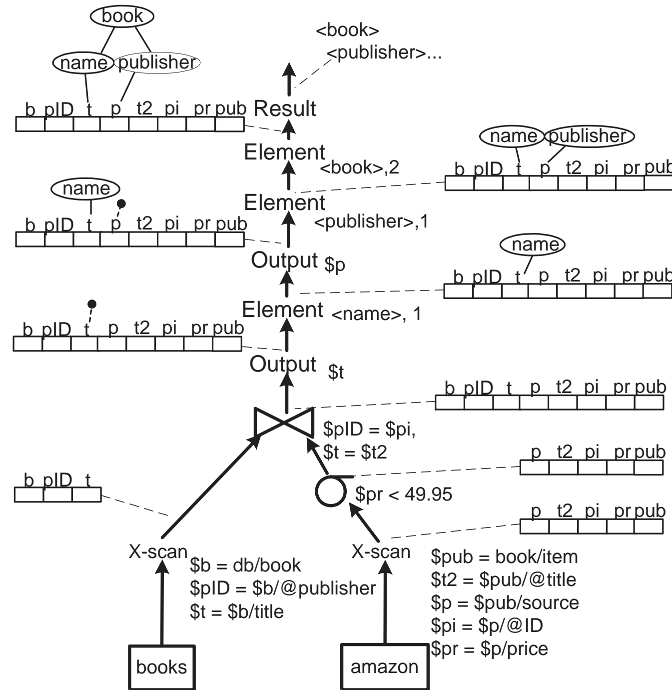


Fig. 7 Query plan for Figure 6 includes a pair of x-scan operators to compute the input bindings, a join across these sources, and a series of **output** and **element** operators that copy the desired variables to the output and construct XML elements around them.

```
FOR $b IN document("books.xml")/db/book,
     $pID IN $b/@publisher,
     $t IN $b/title/data(),
    $pub IN
    document("amazon.xml")/book/item,
     $t2 IN $pub/title/data(),
     $p IN $pub/source,
      $pi IN $p/@ID,
     $pr IN $pub/price/data()
WHERE $pr < 49.95
      AND $pID2 = $pID
      AND $t = $t2
RETURN  <book>
         <name>{ $t }</name>,
         <publisher>{ $p }</publisher>
        </book>
```

Fig. 6 Query returning titles and publishers for books priced under $49.95 at Amazon. The plan for this query is shown in Figure 7.

Example Figure 7 shows a physical query plan and the tuple encoding for the simple XQuery of Figure 6. The x-scan operators at the leaves convert XML to streams of tuples by binding variables to the 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 under $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 constructed 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 `publisher` element tag around it; finally, we take both of these subtrees and place them within a `book` element. As a last step, the stream of tuples is converted back into a stream of actual XML.

In subsequent sections, we describe in detail how Tukwila encodes XML structural information, including tags, nested output structure, and order information.

4.2.1 Encoding XML Tags In XQuery, a single `RETURN` clause builds a tree and inserts references to bindings within this tree. The tree is in essence a template that is output once for each binding tuple.

In Tukwila, we need to encode the tree structure and attach it to each tuple. We do this by adding special attributes to the tuple that describe the structure in a right-to-left, preorder form. The benefit of this encoding is that we do not need pointers from parent nodes to children — instead, each non-leaf node specifies a count of

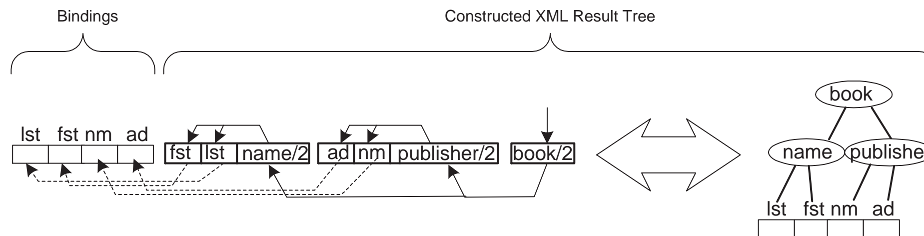


Fig. 8 Encoding of a tree within a tuple, for the query of Figure 7. The encoding is done in bottom-up fashion, so each parent element needs only to specify its number of children. (The arrows are for visualization only.) The tree "leaves" are references to attributes containing bindings.

how many subtrees lie underneath it, and the decoding algorithm simply expands each subtree recursively.

Figure 8 shows this schematically: the tree in the right half of the figure is encoded as the tuple in the left half. The leftmost 4 entries in the tuple are the values of the variable bindings, which contain data values but are not directly part of the XML document under construction. The XML fragment represented by this tuple can be decoded as follows: we start at the rightmost item in the tuple (`book`); this represents a book element with two children (indicated by the "/2" in the figure), and we output a `<book>` tag. We traverse to the leftmost child of the element by moving left by 2 attributes; this yields a `<name>` with 2 children. Again, we traverse the left child – here, we are instructed to output the **fst** attribute. Next we visit the sibling, **lst**, and output its value, and so on.

Of course, the encoding mentioned above assumes that there are no $1 : n$ parent-child relationships in the returned output (every element occurs once for every combination of input bindings). It is very common for XQueries to contain correlated nested subqueries, which embed many results within each iteration of the outer query.

4.2.2 Encoding Nesting As mentioned previously, although we want to capture hierarchical nesting of XML results, we do not actually encode it using nested relations. Instead, we flatten or denormalize the results: a single parent tuple with n child tuples is represented by n "wide" tuples with both parent and child information. The XML hierarchy could be decoded by grouping together all tuples with the same parent content. However, that approach does not support proper bag semantics, since duplicate parents will be combined, and it is fairly costly since all parent attributes must be matched. Instead of adopting that approach, we insert an additional attribute that encodes the parent's sequence ID, and group tuples by this ID to find all of the children with a common parent.

Note that this flattened encoding gives the query processor the opportunity to arbitrarily re-order tuples at any point, potentially distributing consecutive data items anywhere in the tuple stream, as long as it performs a sort at the end. It is worth noting that this tuple encoding approach has some similarities to the "outer union" encoding implemented in [CFI$^+$00, SSB$^+$00] and in Microsoft SQL Server's `FOR XML EXPLICIT` mode; however, we encode the branches of the *subquery hierarchy* rather than the *XML data hierarchy*. As a result, we seldom have null values in our tuple stream.

4.2.3 Encoding Order All of Tukwila's path-matching algorithms can insert attributes that record both the *position* of a binding, by encoding its byte offset within the XML stream, and its *ordering* relative to any other matches for the same variable. Note that these are two distinct concepts, especially when references are used. By adding an ordinal attribute, Tukwila may use non-order-preserving join operators but still maintain XQuery ordered semantics: it simply sorts the data before outputting it.

4.2.4 Generating XML Output Converting from a tuple stream to an XML stream requires several steps: (1) traverse the XQuery RETURN clause constructor embedded within a tuple, outputting the appropriate structure, (2) retrieve and embed any referenced XML subtrees, and (3) correctly output hierarchical XML structure which may span multiple tuples. The first step, traversing the tree structure embedded within a tuple consists of starting at the rightmost output attribute and recursively traversing the tuple-encoded tree, as described in Section 4.2.1. Each time a leaf node is encountered, the second step is performed: the referenced XML subtree is retrieved from the Tree Manager and replicated to the output.

The first two steps above are used when all values encoded within a tuple are to be output; this is not necessarily the case if grouping or nesting attributes are

present. If nested structure is being represented, then each tuple will actually consist of data for the parent relation followed by data for the child relation. Clearly, the parent data should only be output once for the entire group. This is easily determined by testing whether the parent ID attribute has changed between successive tuples.

Groups can be grouped or nested, so this process scales to arbitrary depth. Moreover, XQuery semantics are outer-join-like, so it is possible to have a publisher with no books. In this case, the book attributes in the tuple are set to null values, and the XML decoder simply outputs the publisher attributes with no book content.

In the next two sections, we describe the Tukwila query operators, which make use of this tuple-based encoding. We begin with the operators that produce the tuple stream: the *x-scan* and *web-join* operators.

5 Streaming XML Input Operators

It is Tukwila's support for streaming XML input that most differentiates it from other XML query processors. This support is provided by two different operators that take an input XML stream and a set of XPath expressions, and they return "tuples of trees" representing the combinations of variable bindings that match the XPaths. The simpler operator, *x-scan*, performs XPath matching over a specified input document. The *web-join* operator adds further mechanisms for supporting data-dependent queries: like the dependent join in a relational system, it is provided with a stream of "independent tuples." A web-based (e.g., HTTP) query string is generated by inserting values from the current tuple into a *query generating expression*; this query request is performed, and the resulting XML document is then pattern-matched against XPath expressions. Finally, the matching bindings are combined with the original independent tuple to produce a cartesian product. X-scan is used for querying static or predetermined web sources, and web-join allows Tukwila to dynamically query and combine numerous sources.

The intuition behind the streaming XML input operators is that an XPath expression greatly resembles a regular expression (where the alphabet consists of element and attribute labels), and this can be simulated by a finite state machine. Tukwila uses an event-driven (SAX) XML parser to match input path expressions as an XML stream is being parsed; a variable binding is created each time a state machine reaches an accept state. Bindings are combined to form tuples, which are pipelined through the system, supporting output of XML results as the data stream is still being read.

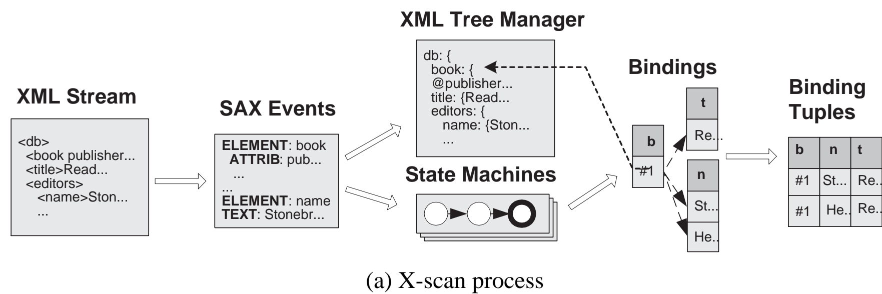
While using a finite state machine to match XPath expressions seems natural, the algorithms for supporting the details of XPath, combining the bindings, and supporting efficient execution are quite complex. To the best of our knowledge, Tukwila is unique in creating pipelinable XML query results directly from a data stream, and in using finite state machines to do so — and as a result it shows significant performance and scalability benefits over other systems. Systems such as Niagara fetch and parse an entire input XML document, construct a complete DOM representation in memory, and finally match path expressions across the tree and pass the results through query operators. XSLT processors such as Xalan, MSXML, and XT are similar, but use a recursive pattern-matching semantics rather than a set of query operators. Most other XML query processors are designed to operate on XML in a local warehouse. One interesting system that is not a query processor but bears some resemblance to Tukwila is the DBIS system and its XFilter [AF00] operator[3]. DBIS takes XML documents and determines whether they meet specific XPath expressions, and it "pushes" those that do to "subscribing" users. DBIS performs document filtering rather than query processing, so XFilter, an operator with a binary (match/non-match) return value, differs substantially from x-scan in its functionality. The XML Toolkit [GMOS02] builds upon the XFilter work, but proposes a "lazy" approach to building deterministic finite state machines from nondeterministic path expressions.

We now present the details of the streaming XML input operators, beginning with x-scan.

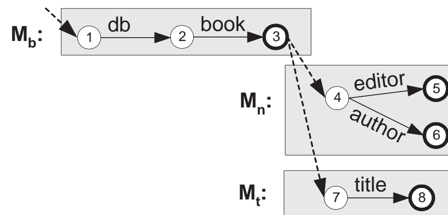
5.1 X-scan Operator

Given an XML text stream and a set of regular path expressions as inputs, x-scan incrementally outputs a stream of tuples assigning binding values to each variable. A binding value is typically a tree — in which case the tuple contains a reference to data within the Tukwila XML Tree Manager — but if it is a scalar value, this value may be "inlined" directly within the tuple. A depiction of x-scan's data structures appears in Figure 9: the XML stream is processed by an event-driven SAX parser, which creates a series of event notifications. The XML data is stored in the XML Tree Manager and is also matched against a series of finite state machines (responsible for XPath pattern matching). These state machines produce output binding values, which are then combined to produce binding tuples.

[3] In fact, the XFilter and x-scan operators were developed concurrently.



(a) X-scan process



(b) State machines for Fig. 3 query

Fig. 9 X-scan takes an XML document and maps it into the XML Tree Manager, while simultaneously running state machines over the parse tree. Each state machine creates variable bindings, and these must be combined to produce binding tuples. Solid arcs in (b) denote state transitions on the label; dashed arcs denote dependencies between machines.

Basic XPath expressions are a restricted form of regular path expressions[4]. Thus x-scan converts each XPath expression into a regular expression and generates its corresponding nondeterministic finite state machine; it later converts this into a deterministic machine, for reasons discussed later in this section. XPath expressions originating at the document root are initialized to the *active* mode, and the active machines' states are updated as x-scan encounters subelements and attributes during document parsing. Figure 9(b) shows the state machines created for the example query of Figure 3.

Initially, only the top-level machine (M_b in our example) is active. When any machine reaches an accepting state, it produces a binding for the variable associated with it. The machine then activates all of its dependent state machines, and they remain active while x-scan is scanning the value of the binding. In our example, the machines M_n and M_t remain active while we scan children of b.

Associated with each machine is a table for binding values. As a machine reaches an accept state, it adds an entry containing its bound subtree value, and also an association with the entry's parent binding (shown in Figure 9(a) as a dashed arrow from parent to child)[5].

[4] We shall discuss additional, non-path-oriented XPath features later in this section

[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.

In our example, M_b's table would just store values of b, while M_n and M_t would store author/editor names and titles, respectively, and these would be associated with their corresponding b values. The final output of x-scan is essentially a join of the entries maintained by the machines, done for matching parent-child pairs (this is done in a data-driven, rather than iterator, model, as with a pipelined hash join [WA91]).

We illustrate the execution of x-scan 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. The `root` node is a virtual node representing the entire document, and its only child is the `db` node. X-scan follows the edge to the `db` node, setting M_b to state 2. Next, x-scan can follow one of two outgoing edges to `book` nodes. It chooses the leftmost one (*Readings in Database Systems*), causing it to set M_b to state 3. M_b is now in an accepting state, so x-scan writes the reference to the current node into M_b's table, suspends M_b, and activates M_n and M_t. The `editor` element takes M_n from state 4 to 5, which is an accepting state for M_n. Hence, x-scan writes "`Stonebraker`" and a pointer from the current book. In the meantime, M_t follows the arc to the `title` element, putting its machine into state 8, which is also an accepting state. Hence, the tuple $\langle$`title1`, `book1`$\rangle$ will be written into M_t's table. From this node, no (non-text) children remain for exploration, so x-scan 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` node, proceeding as before. □

To this point, we have described how x-scan performs simple path expression matching. However, XPath supports capabilities beyond mere path matching, and these features are also provided by x-scan.

Querying order (node indexing): XPath expressions may restrict bindings based on ordering information, such as a constraint on a subelement's index number (e.g., "2nd paragraph subelement") or on the relative positions of bindings (e.g., $a BEFORE $b). X-scan supports both capabilities: the x-scan state machines are annotated with counters to keep track of element indices, and the output of the x-scan can include both a binding and its index or its absolute position. A select operator can then filter out tuples based on order.

Selection predicates: Another useful capability in x-scan 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 titles with the value 'Transaction Processing' ") — similar to "sargable predicates" [SAC+79]. Additionally, x-scan supports existential path tests (e.g., return books only if they have titles). Existential quantification of a path is similar to any other path expression, except that its binding is not returned. (Other forms of existential quantification are possible, and they can be implemented using correlated subqueries and traditional relational techniques.)

Node test functions: XPath expressions often include node tests, which restrict the type of XML node being selected (e.g., `text()`, `comment()`, `processing-instruction()`). Similarly, an XPath edge with an at-sign prefix () represents an attribute node. All of these conditions are expressed within the x-scan state machines as restrictions on the XML nodes to be matched.

Traversing in reverse: Our current implementation of x-scan does not evaluate the XPath "parent" axis, i.e., it does not traverse backwards through the tree. Instead, the Tukwila query optimizer rewrites path expressions with the parent operator by splitting them into a parent-binding and a child-binding. Conditions are evaluated on the child, and if they are met, the parent is used. (While this process may at times be less efficient than supporting a true "parent" traversal, we expect use of the parent axis to be uncommon.)

Efficiency enhancements: In x-scan, we include a number of optimizations to boost XML parsing and processing performance. First, we avoid processing XML content (i.e., handling SAX parser messages) when the state machines are inactive — it is important to avoid unnecessary copying and handling of string data. Additionally, the instant it becomes evident that a subtree cannot satisfy an XPath expression (e.g., it does not meet a sargable predicate or is missing an attribute), we deactivate the state machines until the next subtree is reached.

Expected complexity of state machines: While x-scan uses deterministic finite state machines — which can be exponentially larger than the nondeterministic machines from which they are derived — XPath expressions tend to be short (queries to depth of more than 6-8 seem to be rare). Furthermore, XPath only supports a restricted version of regular path expressions: instead of Kleene closure, XPaths are limited to simpler "wildcard" and "descendent" operations.

Handling memory overflow Typically, x-scan 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 two cases where it may run out of memory.

First, the XML data that is still being referenced may be larger than memory. Since the XML Tree Manager is a paged data structure, segments of this data are swapped to and from disk as needed. Of course, as a result, a large XML file could produce "thrashing" in the swap file during query processing. However, our experiments in Section 8 suggest that this is typically avoided, which we attribute to two factors. First, the system supports "inlining" of scalar values: string, integer, or other "small" data items are embedded directly in the tuple, avoiding the dereferencing operation. Typical query operations in XQuery are done on scalar rather than complex data (e.g., joining or sorting are frequently based on string values); thus these operations often only need data that has been inlined. Large, complex tree data is typically only required at the XML generation stage, when the final results are returned. A second mitigating factor is that many XML queries tend to access the input document in sequential order, and the Tree Manager therefore can avoid re-reading data that has been paged out. For purposes of comparison, we point out that a paged DOM-based approach would have similar behavior to our scheme (except that in-memory representation of XML is larger in a DOM tree, typically at least 2-4 times larger, because of DOM's heavyweight nature); a mapping from XML to relations ("shredded XML") typically requires a significant amount of materialization in the first place, and often incurs heavy costs whenever it needs to perform joins to recreate irregular structure.

The second memory overflow case, which may occur for trees with high fanout, is when sibling XPaths each have many bindings, and x-scan must return all combinations. To take the query of Figure 3 as an example, we might somehow have many authors and alternative titles per book, and x-scan would have to return every possible title-author pairing for each book. To accomplish this, x-scan maintains the current value of b, plus tables for n and t bindings. As values of n are added, they are combined with b and all existing values of t; and the process works similarly for new values of t. Each time a new value of b is encountered, the tables can be flushed and the process restarted. In an extreme case, the tables may grow larger than memory — this case can be handled in a manner similar to the pipelined hash join overflow strategies of [UF00, IFF+99].

5.2 Web-Join Operator

The x-scan operator is analogous to the sequential table scan in relational databases, and to the “wrapper fetch” operation in relational data integration: it allows the query processor to read through an XML document and extract out the relevant content. If the source has more sophisticated query capabilities, certain operations may in fact be “pushed” into it via the x-scan HTTP query request.

In distributed query processing, sometimes it is beneficial to make use of a *dependent join* operator instead of more traditional table scan and join operators. Instead of requesting data independently from two sources and then joining it, the dependent join reads data from one source, sends this data to the other source and requests matching values, and then combines the data from the two sources. This operation is particularly useful in two cases: one is if the join with the second source is highly selective, so much less data is transferred using the dependent join. The second is when the source requires input values before it will return an answer (e.g., the source may be an online bookseller with a web forms interface that requires an author or title), this is equivalent to the notion of “binding patterns” in relational data integration [RSU95, KW96, LRO96].

In a web context, a query to a data source is generally provided using one of two means: via an HTTP request (GET or POST) or via a SOAP call with some form of query (perhaps an XQuery). For both of these domains, we propose the *web-join* operator. Web-join (Figure 10) is intuitively similar to the combination of an x-scan operator with a relational-style dependent join: it receives an input tuple stream and a query generating expression (shown schematically in the lower left of the Figure as a string with two underlined parameters, `$a` and `$b`, although in reality it can be any string expression). The parameters in the query generating expression will be instantiated with values from the input tuple stream, and the resulting query string will be evaluated as a URI string, HTTP POST sequence, or SOAP envelope. The XML resulting from the request is now evaluated against XPath expressions by an embedded x-scan operator. Now, each of the resulting binding tuples is joined with the original tuple and output. The process repeats for each tuple of the original input stream.

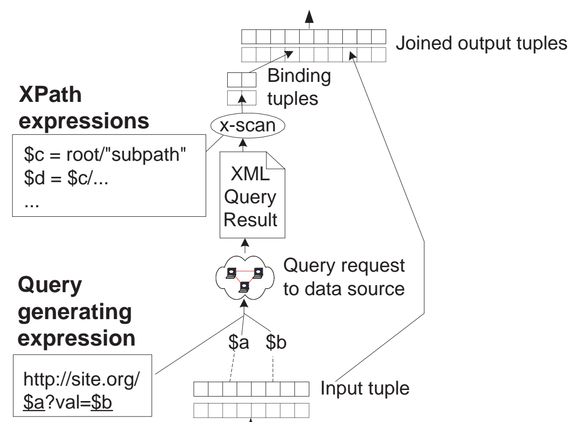


Fig. 10 The web-join operator takes each input tuple and substitutes its values into a query generating expression. This expression becomes a web request that queries a data source; its results are matched against a set of XPath expressions by an x-scan operator. The resulting tuples are joined with the original input tuple to produce a set of results for later query processing.

Web-join is an important operator for querying dynamic sources, especially ones with embedded Xlinks or URIs. It also allows our query processor to do “lazy” evaluation: Tukwila can request some initial data, execute filtering operations on it, and then request additional content for those elements that remain.

6 Tukwila XML Query Operators and Optimization

The previous section presented the query operators that are responsible for mapping an XML data stream into a stream of tuples. Now we describe the query operators that process this data. A logical query algebra usually is designed to be expressive and minimal. In contrast, the set of physical query operators needs to have predictable performance (to make the optimizer’s cost model easier to build) and in efficient implementations for specific contexts (where the optimizer should choose the most appropriate implementation).

As we have constructed the physical algebra for Tukwila, we have focused on providing efficient support for

executing a relatively expressive "core" of XQuery: our focus to this point has not been on supporting the full language. Currently, we do not support recursion, type-checking, or conditional assignments. We have also implemented only a small subset of the proposed XQuery function library. However, we feel that the current implementation is sufficient to demonstrate how common-case queries can be executed quickly, and that it can eventually be extended to include the absent features. The complete list of operators is summarized in Table 1, and we provide more detail below.

6.1 Query Operator Classes

Streaming Input The *x-scan* and *web-join* streaming input operators were already discussed in Section 5.

Path Evaluation The *follow* operator is a path traversal operator. It takes as input a binding tuple, evaluates an XPath (which may involve following an `IDREF` or even an XLink) originating at one of the bindings, and returns a sequence of 0 or more binding tuples. Since x-scan has very little overhead, *follow* is primarily useful when following XLinks or references within a graph-structured document[6].

Combination/filter Most of these operators are almost identical to the standard relational equivalents. One notable exception is the *collector* operator, which we first proposed in [IFF+99]: it starts reading from one or more data sources, but can switch to alternate sources depending on availability and performance. We have one additional operator, *assign*, which adds a new attribute (and binding) to a tuple, assigning it the result of some scalar expression. This expression may be posed in terms of other bindings (e.g., a string concatenation).

Second-order The second-order operators all process sets (or bags) of tuples. The only nonstandard operator is *aggregate*, which takes a stream of tuples representing subquery content nested within parent query content and, for each parent, computes an aggregate value across all of its children. This is very similar to the relational GROUP BY operator, with two exceptions: (1) the grouping information is already present, as the result of a *group* operator as discussed below, and (2) the nested data within the group is preserved rather than discarded.

[6] We expect that XLink reference traversal will be less frequently used than the other operations, and hence we have defined but not yet implemented this operator feature.

Nesting These operators are also second-order, but we separate them because they have a special role in our XML encoding. The *group* operator hierarchically restructures tuples: for each set of tuples that have an identical set of grouping attribute values, the operator conceptually outputs a single tuple with these grouping attributes, *plus* an embedded subtable with tuples of the non-grouped attributes. In Section 4.2.1, we described how this nested structure is encoded within "flat" tuples; we provide each tuple-group with a unique ID, and this becomes the identifier for the "parent tuple," while all non-grouped attributes are the "child tuple." *Group* is primarily useful for providing a relational-style group-by, or for extracting common structure from "flat" XML.

Nested FLWR query expressions are a basic idiom in XQuery, and we handle this case with our *nestChild* operator, which has semantics very similar to a relational left outer join. *NestChild* takes a parent and a child tuple stream, plus a correlation predicate. For each parent tuple, *nestChild* 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 their derivatives include a *unary* operator called "nest" which is closer in nature to our *group* operator than our *nestChild*. Systems with that type of algebra must perform least two operations — join and "nest" — to achieve the same effect as our *nestChild*, and end up doing redundant work.)

Whereas the join operator is typically allowed to output results in any order, *nestChild* semantics require a nested loops join-like ordering, where all child values are returned with their parent. We encode the "hierarchical tuple" as described in Section 4.2.1, which preserves enough information to determine whether any two "flat" tuples contain the same parent tuple. Using this approach, if we use order-preserving operators, we can pipeline the encoded structure all the way to the output result; otherwise, we must use a hashing or sorting algorithm to cluster tuple groups together before we convert them to XML.

Result These operators are responsible for creating the output for the XQuery. They construct the output XML tree and are applied using a postfix ordering. An *output* operator always creates a leaf node in the output; it simply outputs the result of a binding as a string value. *Attribute* wraps the result of the last *output* node within the specified XML attribute name (which may be a literal or the value of a binding). *Element* constructs an XML element around the last k nodes (which may be the result of previous *output*, *attribute*, or *element* operations), where k is a constant specified by the query and

Table 1 Physical query operators and algorithms in Tukwila

Name	Class	Function
x-scan	streaming input	Match input path expression
web-join	streaming input	Query based on bound vars.
follow	path evaluation	Evaluate XPath over binding
select	combination/filter	Filter tuples by predicate
project	combination/filter	Discard bindings
hybrid hash join	combination/filter	Equijoin
pipelined hash join	combination/filter	Equijoin
merge join	combination/filter	Ordered equijoin
nested loops join	combination/filter	Order-preserving join
union	combination/filter	Relational-style union
collector	combination/filter	Union with fail-over
assign	combination/filter	Evaluate expression
distinct	2nd-order	Remove duplicates
sort	2nd-order	Reorder tuples
aggregate	2nd-order	Compute aggregate over group
nestChild	nesting	Correlated nesting of elements
group	nesting	Group and restructure sets of elements
output	result	Output binding to XML
element	result	Create XML element
attribute	result	Create XML attribute

the attribute's label may be either a literal or the value of a binding.

6.2 Optimization and Cost Model

A complete description of Tukwila's query optimizer is beyond the scope of this paper, but to provide a better perspective on the overall query execution process, we briefly discuss the optimizer and cost model here.

Since Tukwila focuses on ad hoc queries over remote data sources, where each source has XML data of arbitrary complexity, our query optimization component must be able work even given *no initial statistics* about the underlying sources. We have developed a technique called *convergent query processing* [Ive02] to enable the query processor to incrementally execute a plan, re-optimize as it gets improved statistics about the data sources (e.g., expected number of tuples produced by the x-scan operator over a particular source, apparent or actual selectivity of a join), and adapt the query plan to a more efficient one — without having to redo work,. Convergent query processing essentially calibrates the optimizer's cost model and statistics to match real-world conditions and performance, so the optimizer can pick a better plan and improve running times.

Our query optimizer is a System-R [SAC+79]-style dynamic programming optimizer, and it optimizes at the physical level. Tukwila's cost model looks very much like that of a relational DBMS: it recursively builds cost and cardinality estimates for increasingly larger subtrees of the query plan, starting with the leaves. We estimate (and periodically re-estimate) the number of tuples that the x-scan operator will produce, including the fan-out at each step of an x-scan path expression. This becomes the "cardinality of the x-scan" from the perspective of the tuple processor and optimizer. The rate of tuple production can also be estimated; it will need to be re-estimated frequently, because it can quickly change due to variations in XML structure or congestion in the network.

Given cardinalities and expected rates of production, we can leverage existing relational query optimizer cost estimation techniques for the remaining operators in the plan. Costs for relational operators like join can be estimated just as in their original context; the remaining XML operators generally have a close equivalent in the relational world (e.g., *nestChild* is implemented much like a join) or an easily predictable cost (e.g., *element* creates a new tag for each input tuple).

Initial query optimization in Tukwila is typically done without statistics, and in this case we typically estimate that each XML element has a fan-out of 1000 at the first level and 10 for every successive level, and we use the standard System-R heuristics for selectivity estimation. Once execution begins, we can obtain more accurate estimates of selectivity and cardinality values and use those to get a better query plan.

A more detailed discussion of the Tukwila query optimizer appears in [Ive02], and we plan to publish an extensive evaluation of the optimizer in the future. Note, however, that many XML queries (and significantly, most

of the queries in this paper) do not depend heavily on query optimizer decisions: the query optimizer focuses on ordering join, nesting, and grouping operators, and queries that only have selection or a single join are minimally impacted by optimizer decisions. Instead, what matters is the performance of the query operators, particularly the x-scan.

With the basic set of operations described in this section, Tukwila can execute the core, database-like subset of XQuery that avoids conditionals, recursive functions, and type information. Additionally, whereas XQuery is a heavily tree-oriented query language, we can also support graph-structured data in Tukwila, as we describe in the next section.

7 Supporting Graph-Structured Data in Tukwila

To this point, we have presented the Tukwila query processing system under the assumption that our data is completely tree-structured and that this structure is mirrored in the XML element/attribute hierarchy. However, the XQuery data model and language do support limited forms of encoding graphs in XML, through the use of `IDREF` attributes (within a document) and XLinks (outside a document). In this section, we briefly describe some of the issues involved in supporting these operations.

7.1 Join-Based Traversal

The conventional way to evaluate an `IDREF` is to use a join operation: for example, suppose we allow only a single `IDREF` in each XPath. To evaluate these expressions, take all XPaths and separate them into "pre-`IDREF`-traversal" and "post-`IDREF`-traversal" steps. Do an x-scan of the input document with the pre-`IDREF` XPaths. In parallel, do an x-scan over the same document for all elements that have `ID`s, and evaluate the post-`IDREF` XPaths. Now join the results when the last `IDREF` of the first x-scan matches the originating `ID` of the second x-scan. Similar techniques can be used to support k `IDREF`s in each XPath. XQuery does not support Kleene closure over `IDREF`s, so a query must have a fixed number of reference traversals and this technique can always be made to work.

The join-based traversal method is effective for following links in many situations, but it has two potential drawbacks. First, standard join algorithms will not "short-circuit" once an `IDREF` is matched to its target `ID`, i.e., they do not "know" that there should be precisely one match to every `IDREF`. Alternative means of traversing `IDREF`s, which we discuss next, can move to the next reference as soon as the current reference has been matched once — fully pipelining the results. Second, the join-based traversal only works for `IDREF`s or XLinks that all belong to the *same* target document.

7.2 Follow-Based Traversal

A second option, which supports both `IDREF`s and XLinks, is to use the Tukwila *follow* operator. In following an `IDREF`, *follow* does an XPath match against an *in-memory* XML document that was output from a prior x-scan and returns a set of bindings. For `IDREF` traversal, *follow* makes use of an index of ID elements that was created by the x-scan operation. This index is further described below.

Follow is intuitively an x-scan that operates on "tuples of trees" rather than on XML documents. Given a set of path expressions and an input tuple stream (as well as the XML trees it references), *follow* adds new variable bindings to each of its input tuples by evaluating the path expressions against the trees within the tuple. If a pattern matches multiple subtrees within the tuple, a set of tuples will be returned, one for each possible binding combination. (This operator is essentially a special case of the *map* operator in some object-oriented algebras.)

Follow is the only reasonable option for evaluating XLinks. At each link, it opens the referenced document and evaluates the XLink path expression to select out the desired XML data, then matches the remainder of the query's XPath against this document fragment, in a manner similar to x-scan or web-join.

7.3 Graph Traversal with X-scan

As we shall see in our experimental evaluation, x-scan's state machine infrastructure adds very little overhead in performing XPath matching against an XML tree. Hence, any XPath traversal across a document's tree structure should generally be done at the x-scan level. Traversals across `IDREF`s can also be done at the x-scan level, and as we shall see later, this performs reasonably for moderately sized documents that do not contain large numbers of references. We now discuss the extensions necessary for traversing `IDREF`s in x-scan.

The first difference is the addition of three new data structures, shown in the upper left corner of Figure 11:

– `ID` index: records the `ID`s of all elements and their matching locations in the XML data. It is used to facilitate resolution of `IDREF`s in the graph.

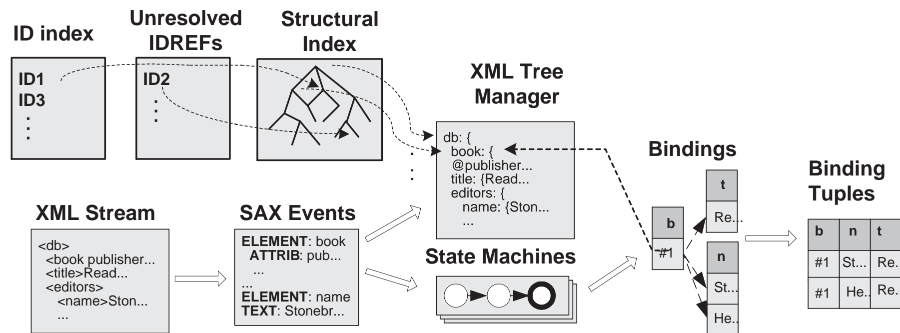


Fig. 11 Graph-based execution of x-scan uses 3 new data structures (upper left). The `ID` index records the positions of each `ID` within the XML data graph; the unresolved index maintains a list of `IDREF`s that have not been resolved; the structural index physically encodes element, attribute, and reference relationships.

– Unresolved-`ID` index: maintains a list of references to not-yet-seen element `ID`s (to be resolved as they are found later in the input).
– Structural index: provides an index of the XML graph, corresponding to Figure 2, but without the data values at the leaves. This is not necessary, but speeds x-scan's traversal through the graph in memory.

When x-scan processes graph-structured XML, it generates a structural index, which is a trie-like index of the XML graph structure (i.e. the subelement and `IDREF` links). This index allows x-scan 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 `IDREF`s. This index is intended only to last for the lifespan of the query, so it is built in memory and paged out only as necessary. (We expect that x-scan will generally only be used to traverse moderately-sized graph data, and will be supplanted by *follow* or joins for larger documents, so paging of this index should seldom occur.)

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 `IDREF`s 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, x-scan creates the `ID` index, which records all the `ID`s that it has encountered so far, mapping from `ID` to entry in the structural index, and the unresolved-`ID` index which records all `ID`s that have not yet been seen in the input, and lists all referrers to each such `ID`.

X-scan'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 x-scan will encounter an `IDREF` edge which points "ahead" to a node which has not yet been parsed. When x-scan hits a forward reference, it pauses all state machines [7] and adds an entry to the list of unresolved `IDREF` symbols, specifying the desired `ID` value and the referrer's address. However, x-scan continues reading the XML source and building the structural index. Once the target element is parsed, x-scan fills its address into each referring `IDREF` in the structural index, removes the entry from the list of unresolved `IDREF`s, and awakens the state machines and proceeds. Although this approach causes x-scan output to stall as it waits for a reference to be resolved, our empirical results have shown that with the help of the structural index, x-scan "recovers" quickly. In the worst case, x-scan should still do at least as well as a DOM-based query processor — as with DOM, it builds a structure in memory that can be quickly traversed; however, unlike the DOM implementations with which we are familiar, x-scan can still execute when this structure must be paged to disk because it exceeds virtual memory.

Cycles In order to avoid cyclic traversals of references, x-scan maintains a history of nodes visited by each automaton state in a given path traversal. X-scan uses deterministic automata, so if a machine re-visits a node that it has encountered *in the same state along the same path*, this is a cycle and can be aborted.

[7] Conceptually, x-scan could continue state machine operation until the reference target is found, then insert the target, return all of the matches found afterwards, and continue normal operation; but for simplicity of coding, our implementation does not do this.

7.4 General Guidelines for Reference Traversal

There are a number of ways of supporting graph-structured data within the Tukwila system. Each of these methods has different capabilities and performance results; we now present a set of guidelines by which an optimizer can choose the best mechanism for evaluating XPaths in the graph context.

We begin by noting that the x-scan operator is very efficient on strictly tree-structured data, so we believe it will seldom make sense to use either the join or *follow* methods to traverse anything but `IDREF`s or XLinks. Thus, the query processor should use x-scan to evaluate the segment of an XPath before (or after) a reference traversal.

The type of reference being evaluated now becomes important: as was noted earlier, the join method does not work for evaluating XLinks. Our x-scan implementation does not follow XLinks, either, because such a traversal is quite expensive and probably should not be done as a leaf-level operation. Thus, for XLinks, the *follow* operator is the only option.

For documents with a low number of `IDREF`s, the x-scan traversal approach works well. Once a large number of `IDREF`s must be evaluated, however, the join and *follow* alternatives look more promising. The *follow* operator is a unary operator, and only requires one scan of a given document; however, it traverses through the XML data (which may result in thrashing if the document is larger than memory). The join operator is less likely to cause thrashing, since it combines tables that are each completed in a single pass — but it requires two separate scans of the input document.

8 Experimental Results

Now that we have seen the details of the Tukwila query engine for both tree-structured and graph-structured data, we move to our experimental validation of the system. Our implementation was written in C++. We originally wrote the system for Windows 2000 using the Apache Xerces-C XML parser at the core of our x-scan implementation. Later, we migrated to a slightly slower Linux machine using James Clark's expat 1.95.1, which performed faster XML parsing. In the experiments below, we used the expat-based implementation for comparing XML pattern matching experiments (Section 8.1), and we relied on the Windows machine for the compute-bound and memory scalability experiments, since it was faster and had more memory.

Our system architecture is based on a client-server model, with a Java client that submits queries using SOAP over HTTP, then reads and times the XML results. Most experiments measured the performance of the Tukwila engine on an 866MHz Pentium III machine with 1GB RAM (of which we allocate only a subset to Tukwila) under Windows 2000 server; but as mentioned above, for the studies of XML pattern matching performance in Section 8.1, we instead ran Tukwila on an 800MHz Pentium III with 256MB RAM under Red Hat Linux 7.1. In all cases, XML documents were served via HTTP from our web server, a dual Pentium II 450MHz system with 512MB RAM, running Windows 2000 and IIS 5. The web server and query processing machine communicated via 100Mb fast Ethernet, with each machine on a separate subnet within a larger-scale network. Experiments were run once for "warm-up" and repeated at least 7 times, and error bars are included for queries where the confidence interval is less than 95%.

Experimental data sets were chosen to encompass a range of different XML data classes, and are listed in Table 2. They include real documents, real semistructured data, semistructured data generated with the recent XMark XML query benchmark [SWK$^+$02], synthetic data with references, and relational tables saved in XML format. The synthetic data with references was the only data set that we created ourselves; it was designed to have random variation in depth and distribution of `IDREF`s. The data set was generated using the following process: replicate a "core" XML subtree a specified number of times, and then randomly attach it to different points within the current document, with probability 15% that it attaches to the root. Afterwards, the designated number of `IDREF` edges were added between random pairs of endpoints.

Since we are proposing a new model for query execution, we begin by comparing Tukwila's performance with that of systems using more traditional approaches. Later, we look at scalability and the performance of Tukwila on database-style operations including join; we examine how hierarchically nesting XML content limits performance because it restricts order; and we look at how Tukwila's x-scan algorithm can be used to support `IDREF` traversal for graph-structured data.

8.1 XML Extraction Queries

Clearly, the core operation at the heart of any XML processor is the pattern-matching and XML content extraction step, and in fact this is where Tukwila's approach differs from other implementations. Our first set of experiments focuses on comparing the relative performance of Tukwila with other systems when extracting XML content with XPath expressions. Our suite of queries is described in Table 4, and consists of a mix of text-

Table 2 Data sets used in experiments.

Name	Size	Description
religion	7MB	Concatenation of Bosak's collection of religious texts (bible, quran, Book of Mormon)
xmark-50	59MB	0.5-scale-factor XMark auctions file
xmark-1000	118MB	1.0-scale-factor XMark auctions file
xmark-500	596MB	5.0-scale-factor XMark auctions file
dmoz	341MB	Open directory (dmoz) RDF hierarchy
dblp-proc	155KB	DBLP list of conference proceedings
dblp-pubs	8.9MB	DBLP list of conference publications
dblp-conf	39MB	DBLP complete conference information
dblp-cj	61MB	DBLP complete conference and journal information
customer-10	0.5MB	TPC-H 10MB (0.01-scale-factor) customer table in XML
orders-10	5.4MB	TPC-H 10MB (0.01-scale-factor) orders table in XML
lineitem-10	32MB	TPC-H 10MB (0.01-scale-factor) lineitem table in XML
customer-100	5.2MB	TPC-H 100MB (0.1-scale-factor) customer table in XML
orders-100	53MB	TPC-H 10MB (0.1-scale-factor) orders table in XML
lineitem-100	324MB	TPC-H 100MB (0.1-scale-factor) lineitem table in XML
synth	100K-100MB	Data from synthetic generator (see text)

oriented and path-oriented queries over different types of hierarchical documents and semistructured data. (We examine performance on more regular XML data from relational systems in the next section.)

See Table 3 for details on the systems in our comparison; all except for Tukwila are main-memory-only XML engines. We included three popular XSLT processors in our study: the Apache Xalan-C system, James Clark's XT engine (which was generally rated as one of the faster XSLT engines), and the XSLT processor in Microsoft's MSXML 4.0 toolkit (which has been heavily optimized and is considered to have the fastest parser and XSLT engine available). We also wanted to compare with data integration systems, so we included the December 2000 version of the Niagara system (as of this time, the latest version that is publicly available). Early in the development of Tukwila, we also compared our performance against the Lore System [GMW99], an XML repository; at the time, Tukwila significantly outperformed Lore. Unfortunately, Lore is no longer being distributed, and therefore we omit it from our comparison, because it would be unfair to compare with an outdated version of Lore.

Figure 12 shows the results for the queries in two graphs: part (a) shows the time to the initial 5 answers, as a way of measuring quick feedback to the user; part (b) shows the overall query completion time. Note that queries Q3, Q5, and Q6 all had fewer than 5 answers, so they have identical timings.

We make several observations about the results. First, although Tukwila was run on a slower machine (800MHz vs. 866MHz) with less memory (256MB vs. 1GB) than all of the other systems, it nearly matched or significantly outperformed all of the other engines documents across the entire suite of queries. Microsoft's MSXML processor lives up to its reputation as being a very fast engine, and it is actually faster by a margin of half a second for the queries over the relatively small `religion` document — we attribute this to the additional overhead Tukwila incurs to optimize its queries. For larger documents, however, such as the XMark document, Tukwila is substantially faster overall, and is especially faster for Query Q7. Q7 clearly demonstrates that Tukwila is the only processor to scale to large XML data files: our system comfortably processed the 324MB `dmoz` XML document on a 256MB machine in less than a quarter the time that MSXML (needing most of the 1GB of RAM in its experimental configuration) did. No other systems were able to accommodate the large document.

Table 4 List of queries used for comparing pattern-matching performance.

Nbr.	Input	Query
Q1	religion	Chapter 5's (medium trees)
Q2	religion	Chapters ≥ 8 (medium trees)
Q3	religion	Sura titles with "Mormon" (single result)
Q4	religion	Suras with "The" in title (large trees)
Q5	xmark-50	XMark query Q1 (person0's data)
Q6	xmark-50	XMark query Q2 (bidder 1's bid increases)
Q7	dmoz	Return all topic IDs

Surprisingly, although our suite of queries was relatively simple, some of the queries could not be executed on all systems. Niagara does not support the XML-QL **LIKE** predicate or index variables, so we could not ex-

Table 3 Systems compared in Section 8.1.

Name	Implemented	Domain	Description
Xalan 1.1	C++	Doc	Apache XSLT processor, built over Xerces-C parser
XT 19991105	Java	Doc	James Clark's XSLT processor
MSXML 4.0	C++	Doc	Microsoft parser and XSLT processor toolkit
Niagara 1.0	Java	XML-DB	University of Wisconsin XML integration system
Tukwila 1.0	C++/Java	XML-DB	XML engine described in this paper

press queries Q3, Q4, and Q5. MSXML executed query Q2 with incorrect results (returning no answers). Several query processors failed with the XMark document (generating what appear to be spurious parse errors), and nearly all failed on the large `dmoz` document (running out of memory even on a 1GB system).

Overall, x-scan'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 in the cases where there were fewer than 5 answers to be returned), and in fact the incremental processing model 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 x-scan operator.

8.1.1 Slow links Our first experiment measured general query processing performance across a local area network; however, wide-area query processing is one of the focal points of the Tukwila project. Thus our second experiment repeats the previous queries in a bandwidth-constrained environment. We simulated these conditions by artificially adding a 50ms delay to the initial request for a document (representing a slightly longer round-trip time), plus a 15ms delay per 16KB of data sent (limiting the throughput of the connection). This delay was sufficient to inject 960 msec of delay per MB of data transferred, giving us about 1MB per second or 8Mbits per second as our approximate transfer rate. We repeated all of the queries of the previous section except for the `dmoz` query, which we judged to be too huge for anyone to want to transfer in this situation.

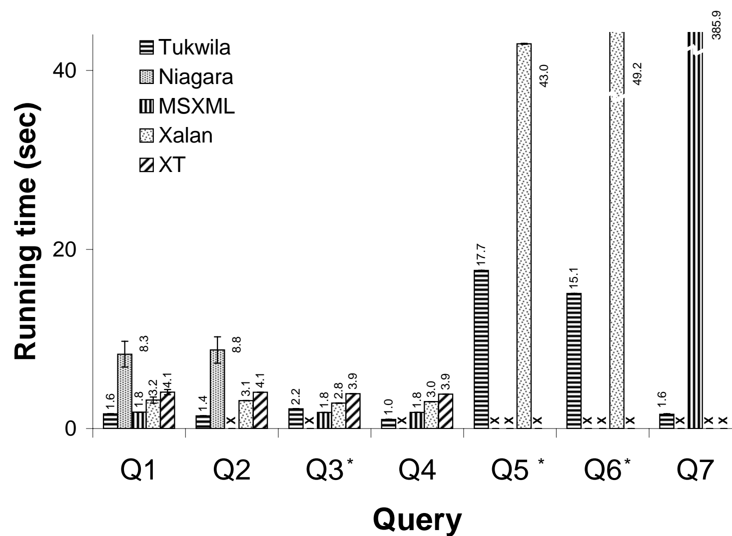
Performance results are in Figure 13. As expected, Tukwila's incremental output greatly improves the time to initial answers, but the overall query completion time also shows a relative performance gain versus the other query processors. Since Tukwila does filtering and construction of content in parallel with reading, it manages to use the network delay times to help compute answers; in contrast, the other query engines are idle during delays, since they cannot process results until after the parse is complete.

8.1.2 Scale-up A point of emphasis in our design of the Tukwila architecture has been scalability to large XML documents. While most XML files on the Web are currently only tens of KB in size, as XML matures, querying and integration of data between groups or enterprises is expected to become commonplace — and such data will be considerably larger. In many of these situations, the query processor may be servicing many outstanding requests simultaneously, so each query must run with limited resources. Moreover, current query processors' in-memory representations of XML data are substantially larger than the original XML data — e.g., the XT processor required over 260MB of memory to load and scan the 39MB DBLP XML file in query Q4 of the previous subsection; even a server with 1GB of memory cannot handle many such queries simultaneously.

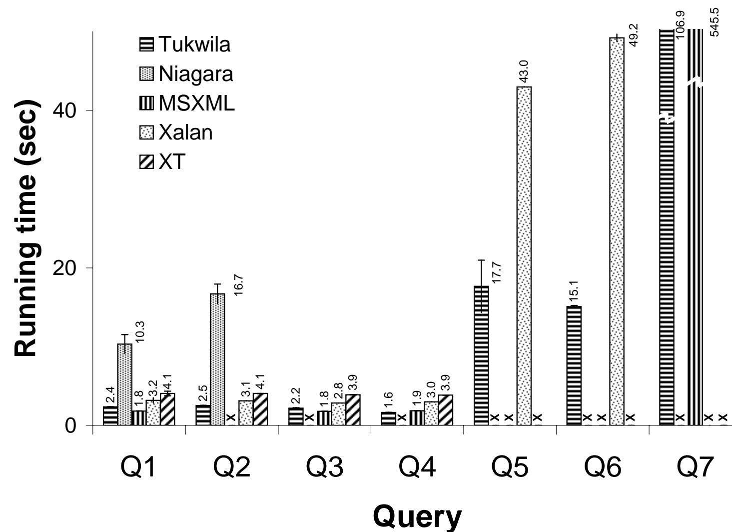
Tukwila avoids this pitfall by supporting out-of-core execution. Many aspects of the Tukwila architecture (e.g., external sorts, grouping operators, hash and pipelined hash joins) will scale in predictable ways, as they are well-understood components of relational query engines. As observed in Section 6, most query operations take place over scalar data values rather than subtrees, and these values are likely to be inlined within the tuple — hence page faults in the XML Tree Manager are not likely to greatly affect performance.

The main concern for scalability, then, is the x-scan operator and the data structures it uses. We investigated the performance of x-scan 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.

We took all of the queries from the previous section, plus two selection queries over relational data and plotted the running times versus the data sizes in Figure 14. We note that an interesting dichotomy emerges: the relational tables, which are quite "dense" with many tuples and many XPath matches, seem to yield running times that all fall on the approximately same line at the left of the plot. Likewise, the other queries over sparser semistructured data seem to follow a different line with a lower slope. As we would hope, Tukwila's performance appearance appears to scale approximately linearly, with



(a) First 5 tuples returned (queries marked with a * have fewer than 5 tuples total).



(b) Total query time

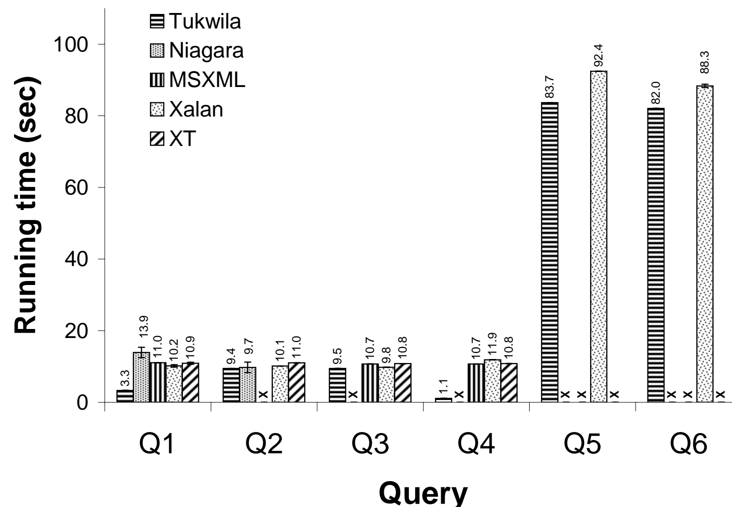
Fig. 12 Experimental comparison of XML queries shows that Tukwila has equal or better total running time (and better time to first tuples) for a variety of XML extraction queries.

the slope determined by the number of pattern matches that occur.

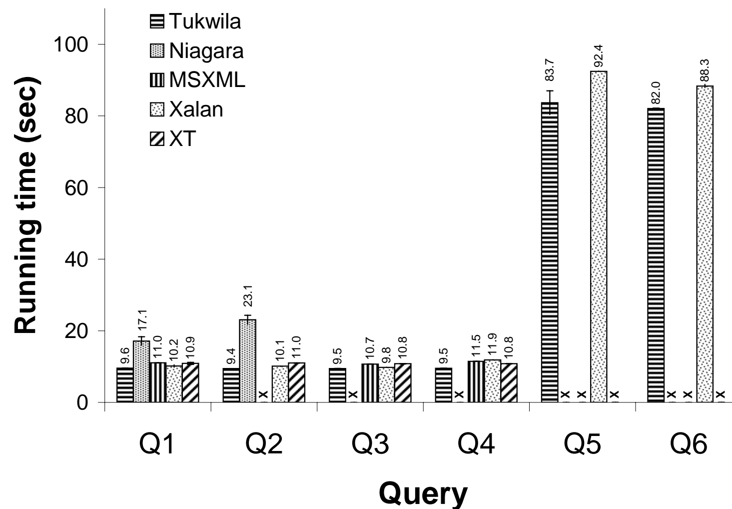
Figure 15 shows performance over a range of synthetic data, generated as described at the beginning of this section. We observe that the time required to process a simple query grows at a rate only slightly faster than it takes to parse the XML and build a DOM tree (the approach taken by previous systems); x-scan state-machine operation and Tree Manager overhead within Tukwila 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.

8.2 Database-Style Operations

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



(a) First 5 tuples



(b) Total query time

Fig. 13 In the wide-area context, Tukwila's architecture provides even greater performance improvements when compared to the other systems.

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:

– **XMLified SQL**, where we sent a SQL selection or join query to a database at the server (DB2 UDB 6.1 running on a 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 fully indexed. 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 do not translate queries.
– **Relational Mediator**, in which all tuples from the tables were simply read from JDBC and returned to the original Tukwila system, which executed a relational query and then converted the data to tagged XML.
– **Tukwila**, which took materialized XML views of the relational tables, read them via HTTP, and did XML query processing over the data using the techniques described in this paper.

As Figure 16 illustrates, the Tukwila and Relational Mediator approaches tended to have very comparable

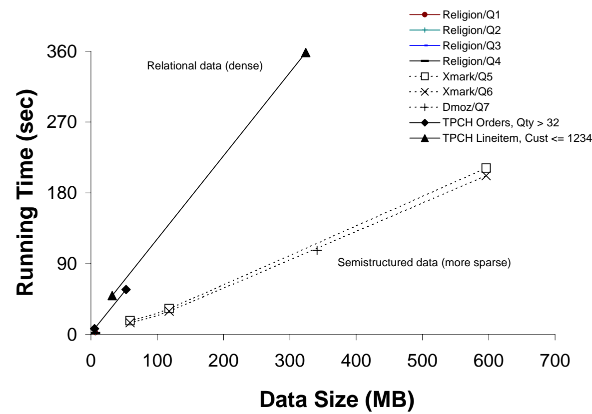


Fig. 14 An X-Y plot of running times versus data sizes shows that Tukwila yields relatively consistent and linear performance. Note that queries over relational data, which is typically more "dense," result in a higher slope than more sparse semi-structured data.

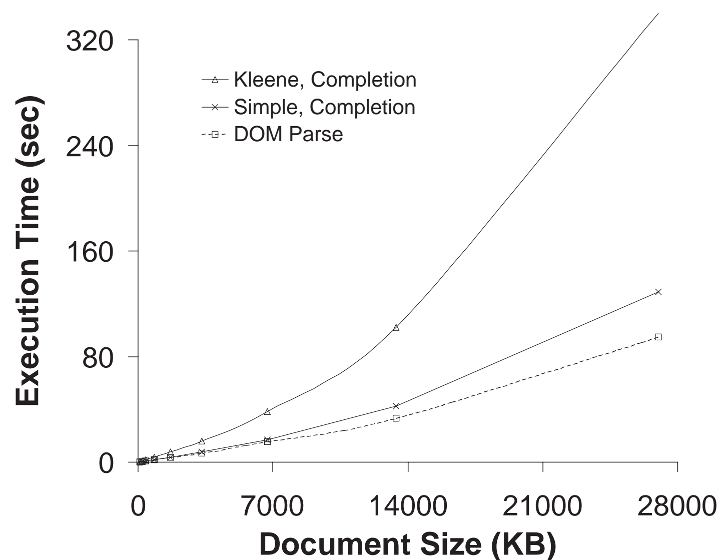


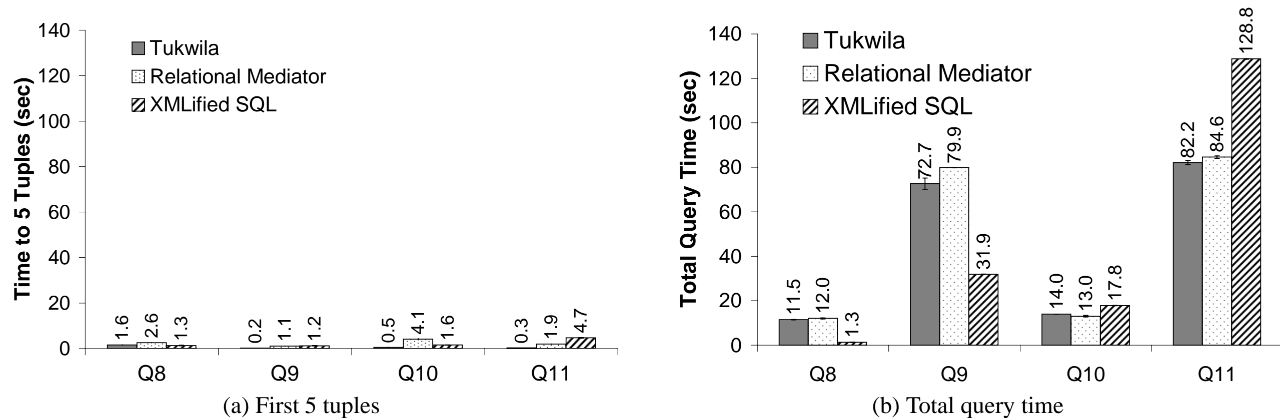
Fig. 15 Scale-up results for query completion time on synthetic data for simple path query, Kleene-* query, and DOM parse. (Time to first 5 tuples was under 2 seconds.)

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 (even given the fast 100Mbps network) 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. We attribute this to the fact that JDBC was a bottleneck in our experiments and the join results were larger than the sum of the combined inputs — as a result, it was more efficient to read the original tables separately and join them within the mediator. Likewise, it was essentially as efficient to read and process the XML version of the data as it was to read the data through JDBC. We conclude that the choice of whether to push an operation into a data source depends greatly on the communication-link costs, even when we are choosing between querying data in its original relational form or converting it to XML first.

While we do not claim that JDBC is the fastest means of exchanging relational data (and we acknowledge that many modern databases provide other mechanisms for exporting XML), we observe that its performance is acceptable for many business and scientific applications. Since Tukwila performs similarly on equivalent queries, we believe that x-scan-based XML data exchange also provides sufficient performance for real-world applications. Moreover, the Tukwila XML-based engine pro-

Table 5 Queries with database-style selection (Q8-Q9) and join (Q10-Q11) operations using relational data mapped into XML.

Nbr.	Class	Input	Query
Q8	Rel. Sel.	5MB	TPC-H Orders for Customer "1234"
Q9	Rel. Sel	31MB	TPC-H LineItems with Quantity $>$ 32
Q10	Rel. Join	5MB x 0.5MB	Join TPC-H Orders for Customer key $<$ "1234" with all Customers
Q11	Rel. Join	31MB x 7MB	Join TPC-H LineItems with Orders



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

Fig. 16 Experimental comparison of relational queries shows that Tukwila performs nearly as well over data mapped into XML as the comparable relational-model integration system. In-SQL execution, included for comparison only, was better for the selection query but not for the joins.

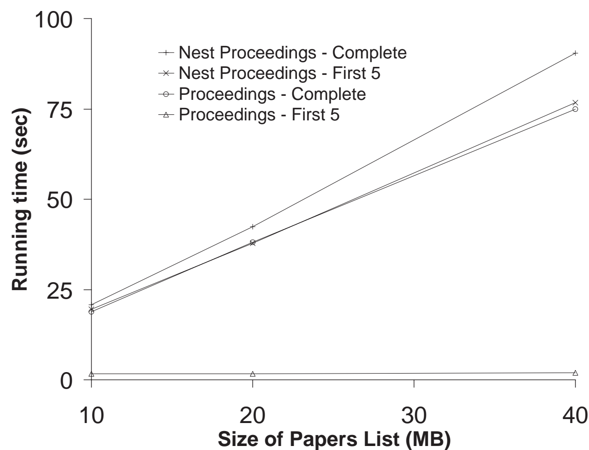


Fig. 17 Comparison of nest and join operations combining DBLP papers and proceedings. Nest requires the (larger) inner relation to be read first, thus it has much longer time-to-first-results and slower overall time than the optimal join.

vides greater interoperability because it can combine relational and non-relational data.

8.2.1 Nesting Data As we observed in Section 4.2.2, the operation of hierarchically nesting XML child elements within a parent element is very similar to a left outer join in relational databases. However, a nesting operation has an important constraint, which is that the elements must appear contiguously, clustered by parent. Clearly, maintaining this grouping incurs some overhead, and we wanted to examine how significant this was.

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 a **nest** operation is used to create a $1 : n$ hierarchical relationship, it must place the *larger* join relation as the inner relation so it can iterate over it for each parent tuple. We can see in Figure 17 that as a result, **nest** performs more slowly than a hash join that has been commuted to the opposite configuration. 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 in interactive applications, where first answers are most important, 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. However, if we output results without preserving order, we must ultimately sort the data to get it into its proper form. We are experimenting with a user

interface in which the final sort operation is performed at the client-side on a periodic basis, which frees the query processor to stream out results in any order and provide faster feedback to the user.

8.3 Supporting Graph-Structured Data

Although most of today's XML queries traverse the document as a tree, there are many potentially interesting uses of XML as a representation for semistructured graphs, encoding edges as both elements and `IDREF`s. Thus, the x-scan operator has a number of features designed for querying graphs. Previous work on `IDREF` traversal has typically been done using the join or **follow** approaches described in Section 7, but we now examine the use of x-scan as an alternative.

8.3.1 X-scan Traversal of `IDREF`*s* In our comparison of strategies for evaluating graph-style references, we suggested that x-scan could be used on moderately sized documents that had low numbers of references. In Figure 18, we see execution times of x-scan across synthetic documents of different sizes. The different lines represent execution times when the ratio of `IDREF`s to elements is 1:8, 1:4, 1:3, and for comparison we include the execution time for a typical tree-traversing query, which does not build the structural index, 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 `IDREF`s, running times are reasonable, especially since initial results are output quickly. With a 1:3 ratio of `IDREF`s to elements, Tukwila 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 55,000 leaf nodes. As the ratio of `IDREF`s gets even higher — not shown in the graph — the XML graph begins to approach full connectivity, and x-scan spends large amounts of time doing repeated evaluations. Clearly, in these situations, the join- or follow-based approach is more appropriate.

8.3.2 Graph Traversal with Limited Memory We also examined in detail the performance characteristics of x-scan, particularly those related to paging data to disk. For simple tree-based queries, memory constraints are typically not an issue — Tukwila needs only to maintain state and subtrees for a limited amount of time, i.e. until all tuples referencing the subtrees have passed through the pipeline. Thus, for example, when we queried the the 159MB Open Directory Project topic hierarchy for all topic aliases, query processing times were approximately 7 minutes 43 seconds whether Tukwila was given 20MB of memory or 250MB. Results were similar for tree-style queries over other data sets such as DBLP. Additional experiments demonstrated that the performance bottleneck was clearly in the areas of network I/O and parsing — saving a locally cached copy of the input XML document to disk (from a separate thread) added no perceptible time overhead to the query.

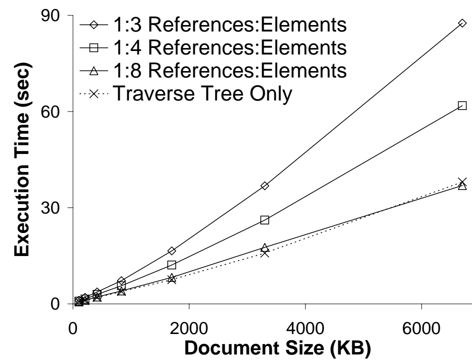


Fig. 18 Scale-up results for Kleene-* graph query on synthetic data, with tree query shown for reference.

Our final experiment, in Figure 19 measures the performance of x-scan graph traversal across large XML data files when the amount of memory available to the Tree Manager and the structural index are constrained. Data sets on the graph include two synthetic data sets of 103MB and 51MB, each with a 1:8 element-to-`IDREF` ratio, and the DBLP conference data set with cross-references from papers to conferences as `IDREF`s[8]. Our experiments do include a data set in which most of the referenced items are relatively clustered (DBLP) and one in which they are randomly distributed throughout the document (the synthetic data). In all cases, the structural index ranged in size from two to three times the data set size. We separately adjusted the size of the index's memory allocation and the Tree Manager's allocation, to see how greatly each affected performance. In general, the variations in memory had less of an impact than one might expect — we attribute this to the fact that the query processor is generally network-bound, and hence can make use of free CPU and disk cycles. Moreover, as expected, the size of the index buffer affects performance more than the size of the Tree Manager. A final observation is that, as expected, the DBLP data set, with a fairly strong locality of references, is basically not impacted by memory, whereas the synthetic data with its randomized reference targets is somewhat more affected.

[8] We also attempted to use the Open Directory data file, but were unable to successfully "clean" the document by removing elements unacceptable to the Xerces parser, while still maintaining `IDREF` link integrity.

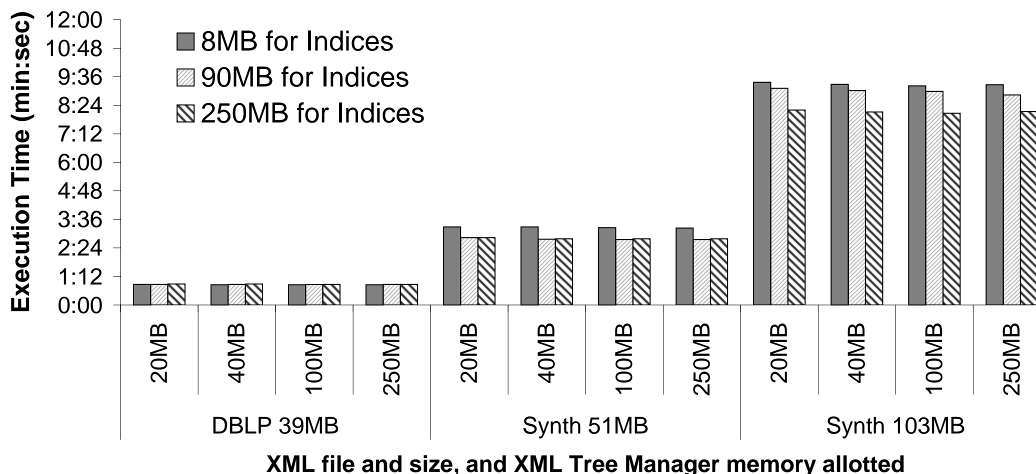


Fig. 19 Query processing times with restrictions on XML tree memory (x-axis) and index buffer memory (bar shades). Index buffer size impacts performance more than tree memory.

9 Related Work

As described earlier, most previous XML query processors have fallen into one of two classes:

- web-oriented processors, including XSLT processors such as XT and Xalan, the Niagara [CDTW00] and MIX [BGL+99, LPV00] data integration systems)
- repository based systems that require all the data to first be loaded into a local store, and then processed, such as those of [FK99b, SGT+99, DFS99, GMW99].

Most web-oriented query processors have shortcomings in terms of scalability and ability to incrementally parse and produce answers.

When we compare Tukwila to repository systems, we note that a repository's particular storage mapping may 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. Frequently, however, 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. However, both of these techniques are typically ill-suited for a network-based query domain with autonomous data sources, unless queries are frequently repeated over the same data; because they invest a great deal of time into mapping, storing, and indexing data before it can be queried, and this often cannot be amortized across multiple queries.

Related to repository-based systems are XML query interfaces over existing databases: SilkRoute [FTS99] and XPERANTO [CFI+00] support creation of XML queries and views over relational systems, and IBM, Oracle, and Microsoft all support some XML export features in their products. These systems are very useful for exporting data into XML to facilitate data integration, but they are clearly not intended to be general-purpose XML processors.

A goal of our architecture is to support fully pipelined execution and leverage sophisticated techniques developed for relational query processing over the network, such as those developed for data integration and distributed databases [KD98, UFA98, UF00]. Our internal execution model bears similarities to object-relational database engines, including the use of references for (potentially out-of-core) large objects — in our case, XML subtrees. This approach bears some similarities to the ADT for structured text described in [BCD+98]. Our physical algebra borrows some of its hierarchical aspects from the nested relational algebra [RKS88].

The key operator responsible for outputting pipelined tuples from an input document in Tukwila is the x-scan 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, x-scan 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 x-scan 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. X-scan also has similarities to the XFilter operator [AF00], developed simultaneously but focusing on filtering XML documents accord-

ing to an XPath expression. XFilter returns a boolean value (match or non-match) rather than a tuple stream, and as a result differs considerably in functionality and implementation. Finally, x-scan 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.

10 Conclusions

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 following abilities:

- the ability to query, combine, and restructure the content of XML documents of arbitrary size,
- the ability to combine data from multiple sources, including data that is the result of dynamically computed queries
- support for a "streaming" or pipelined query processing model that produces results as soon as possible.

This paper describes the architecture of the Tukwila XML data integration system, the first XML processor that satisfies the above requirements. Our key contributions include:

- 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 efficiently on XML;
- two key operators, x-scan and web-join, that map XML data (from both static and dynamically queried sources) into tuples in a streaming fashion;
- and a set of basic operators for combining and restructuring tuples of XML subtrees into new XML content.

We described a set of experiments that demonstrate that our system 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 Tukwila suggests several directions for future research. Clearly, a next step is to develop improved query optimization techniques for the XML context, particularly in the context of data integration — where few statistics will be available. We are in the process of building a new adaptive query processing framework for the next version of Tukwila, which will support multiple strategies for continuous re-optimization of an executing query. Additional important avenues of research include further investigation of processing graph-structured data — in particular, support for features such as Skolem functions to create graph-structured query results — and the performance implications of ordered versus unordered execution.

Acknowledgements The authors would like to thank Phil Bernstein, Daniela Florescu, Hartmut Liefke, David Maier, Rachel Pottinger, Dan Suciu, and the anonymous reviewers for their comments on earlier versions of this paper. This work has greatly benefited from their suggestions.

References

[Aea01] Serge Abiteboul and et al. A dynamic warehouse for XML data of the web. *IEEE Data Engineering Bulletin*, June 2001.

[AF00] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB '00*, 2000.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD '00*, 2000.

[AKJK+02] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Yuqing Wu. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE '00*, 2002.

[BBM+01] Denilson Barbosa, Attila Barta, Alberto Mendelzon, George Mihaila, Flavio Rizzolo, and P. Rodriguez-Gianolli. ToX – the Toronto XML engine. In *International Workshop on Information Integration on the Web*, Rio de Janeiro, 2001.

[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.

[BCF+02] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jerome Simeon, and Mugur Stefanescu. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, 30 April 2002. W3C working draft.

[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 '99*, pages 597–599, 1999.

[BKKM00] Sandeepan Banerjee, Vishu Krishnamurthy, Muralidhar Krishnaprasad, and Ravi Murthy. Oracle8i - the XML enabled data management system. In *ICDE '00*, pages 561–568, 2000.

[CDTW00] Jianjun Chen, David DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD '00*, 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 *ACM SIGMOD WebDB Workshop '00*, 2000.

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

[Col89] Latha S. Colby. A recursive algebra and query optimization for nested relations. In *SIGMOD '89*, pages 273–283, 1989.

[DFF+99] Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. In *Eighth International World Wide Web Conference*, 1999.

[DFS99] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *SIGMOD '99*, pages 431–442, 1999.

[FK99a] Daniela Florescu and Donald Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report 3684, INRIA, March 1999.

[FK99b] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, September 1999.

[FMK00] Daniela Florescu, Ioana Manolescu, and Donald Kossman. Integrating keyword search into XML query processing. In *Ninth International World Wide Web Conference*, May 2000.

[FMN02] Mary Fernandez, Jonathan Marsh, and Marton Nagy. XQuery 1.0 and XPath 2.0 data model. http://www.w3.org/TR/query-datamodel/, 30 April 2002. W3C working draft.

[FMS01a] Mary F. Fernandez, Atsuyuki Morishima, and Dan Suciu. Efficient evaluation of XML middleware queries. In *SIGMOD '01*, May 2001.

[FMS01b] Mary F. Fernandez, Atsuyuki Morishima, and Dan Suciu. Efficient evaluation of XML middleware queries. In *SIGMOD '01*, 2001.

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

[GMOS02] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata and stream indexes. Available from www.cs.washington.edu/homes/suciu/files/paper.ps, February 2002.

[GMW99] Roy Goldman, Jason McHugh, and Jennifer Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *ACM SIGMOD WebDB Workshop '99*, pages 25–30, 1999.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB '97*, pages 436–445, 1997.

[HH99] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD '99*, pages 287–298, 1999.

[HS93] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, 1993.

[HSR91] Tina M. Harvey, Craig W. Schnepf, and Mark A. Roth. The design of the Triton nested relational database system. *SIGMOD Record*, 20(3):62–72, 1991.

[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 '99*, pages 299–310, 1999.

[Ive02] Zachary G. Ives. *Efficient Query Processing for Data Integration*. PhD thesis, University of Washington, August 2002.

[JXT01] Project JXTA: Protocol specification revision 1.1.1. platform.jxta.org/spec/v1.0/JXTAProtocols.pdf, 12 June 2001.

[KD98] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD '98*, pages 106–117, 1998.

[KM90] Alfons Kemper and Guido Moerkotte. Access support in object bases. In *SIGMOD '90*, pages 364–374, 1990.

[KM00] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of XML data. In *ICDE '00*, page 198, 2000.

[KW96] Chung T. Kwok and Daniel S. Weld. Planning to gather information. In *AAAI '96*, pages 32–39, August 1996.

[LPV00] Bertram Ludäscher, Yannis Papakonstantinou, and Pavel Velikhov. Navigation-driven evaluation of virtual mediated views. In *EDBT '00*, pages 150–165, 2000.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB '96*, pages 251–262, 1996.

[MAM01] Amelie Marian, Serge Abiteboul, and Laurent Mignent. Change-centric management of versions. In *VLDB '01*, 2001.

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

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT '99*, pages 277–295, 1999.

[NDM+01] Jeffrey Naughton, David DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayvel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The Niagara Internet query system. *IEEE Data Engineering Bulletin*, June 2001.

[NET01] What are XML web services? www.microsoft.com/net/xmlservices.asp, May 2001.

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

[RS86] Louiqa Raschid and Stanley Y. W. Su. A parallel processing strategy for evaluating recursive queries. In *VLDB '86*, pages 412–419, 1986.

[RSU95] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *PODS '95*, pages 105–112, 1995.

[Rys01] Michael Rys. Bringing the internet to your database: Using SQLServer 2000 and XML to build loosely-coupled systems. In *ICDE '00*, pages 465–472, 2001.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *SIGMOD '79*, pages 23–34, 1979.

[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*, pages 302–304, 1999.

[SKS+01] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene J. Shekita, Catalina Fan, and John Funderburk. Querying XML views of relational data. In *VLDB '01*, pages 261–270, 2001.

[SSB+00] Jayavel Shanmugasundaram, Eugene Shekita, Rimon Barr, Michael Carey, Berthold Reinwald, Bruce Lindsay, and Hamid Pirahesh. Efficiently publishing relational data as XML documents. In *VLDB '00*, 2000.

[SWK+02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *VLDB '02*, 2002.

[Tam] Tamino: Technical description. www.softwareag.com/tamino/details.htm.

[TVB+02] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD '02*, 2002.

[UF00] Tolga Urhan and Michael J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2), June 2000.

[UF01] Tolga Urhan and Michael J. Franklin. Dynamic pipeline scheduling for improving interactive performance of online queries. In *VLDB '01*, September 2001.

[UFA98] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In *SIGMOD '98*, pages 130–141, 1998.

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

[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 Corporation: Platform. www.exceloncorp.com/platform/index.shtml.

[XSL99] XSL Transformations (XSLT), version 1.0. www.w3.org/TR/xslt, 16 November 1999. W3C recommendation.

[ZND+01] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD '01*, 2001.