Bulletin of the Technical Committee on

Data Engineering

June 2001 Vol. 24 No. 2

IEEE Computer Society

Letters

Special Issue on XML Data Management

Conference and Journal Notices

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

The Data Engineering Bulletin web page is http://www.research.microsoft.com/research/db/debull.

Letter from the Editor-in-Chief

TCDE Nominations

I would like to draw your attention to the message below from the TCDE Nominating Committee. The TC will be electing a new TC chair. I would urge you to participate in the electoral process.

The Current Issue

It is an understatement to say that the world is ”moving to XML”. Activity, not only in the research community but in groups throughout are database industry, is at a feverish level. Obviously, XML is important. Industrial groups ”smell” revenue. And in the data engineering community, it is this industrial focus that is so important to the relevance of our field.

XML is rapidly becoming the ”lingua franca” of wire data formats. The database products that make it easy to integrate XML over the wire with their stored data will make the life of application providers both easier and more productive. That will translate into increased sales by the database vendors. The research community is very active in this process, as we were 25 years ago in the emergence of relational database technology. This research will play a large role in what our ”soon to be” XML database industry will look like in the years ahead.

Alon Halevy, our issue editor, has put together a very interesting collection of papers that captures a broad cross-section of the XML activity in data management. These papers range from database integration with the web to data warehouses, from publishing data in XML to XML programming languages, and more. This is an issue that you will want to read carefully, and come back to often. I want to thank Alon for his efforts in bringing this very timely and important issue together.

David Lomet
Microsoft Corporation

Nominations for Chair of TCDE

The Chair of the IEEE Computer Society Technical Committee on Data Engineering (TCDE) is elected for a two-year period. The mandate of the current Chair, Betty Salzberg, expires at the end of this year and the process of electing a Chair for the period 2002-2003 has begun. A Nominating Committee consisting of Paul Larson, Masaru Kitsuregawa and Betty Salzberg has been struck. The Nominating Committee invites nominations for the position of Chair from all members of the TCDE. To submit a nomination, please contact any member of the Nominating Committee before the end of August, 2001.

More information about TCDE can be found at http://www.ccs.neu.edu/groups/IEEE/tcde. Information about TC elections can be found at http://www.computer.org/tab/hnbk/electionprocedures.htm.

Paul Larson, Masaru Kitsuregawa, and Betty Salzberg
TCDE Nominating Committee

Letter from the Special Issue Editor

It has been less than two years since the bulletin had a special issue on XML (September, 1999), but quite a bit has happened. Two years ago we were just beginning to consider the issues of manipulating XML as data. The issue of developing an XML query language was taking center stage, as were initial attempts to store XML data in relational databases. Since then, a working group of the World-Wide Web Consortium has been making rapid progress on developing XQuery, a standard query language for XML. Most relational database vendors already support some form of storage of XML in their products with limited capabilities for querying. In addition, several products support native XML storage (e.g., eXcelon and Tamino) and several companies have been using XML as a platform for integrating data from multiple sources (e.g., Nimble Technology, Enosys Markets).

This issue features several papers that touch on the current topics and debates with respect to XML data management. Naturally, one of the main questions facing the community today is whether XML data management should be added as another feature to relational database systems or whether we should develop new *native* database systems. The proponents of the relational approach point to the success of relational database vendors in incorporating previous non-relational features into their systems and the advantages of exploiting all the infrastructure of existing products. The advocates of native XML management systems argue that XML requires considerably different processing techniques that are impossible to support on top of relational systems, and that existing relational systems are too heavyweight to support the kinds of applications involving XML. In addition to this debate, much of the focus of the research community has been on exploring new types of query services that need to be supported in XML applications. These include processing queries where the data is being integrated from several sources on the WWW, the data is *streaming* from the sources, supporting large numbers of continual queries and subscription queries, and combining features of querying from information retrieval and databases.

The first two papers in the issue consider the relationship of XML data management and relational databases. In the first paper, Rys describes the XML support added to SQL Server 2000. In the second paper, Fernandez et al. describe Silkroute, a system that efficiently supports XML views over relational data. Readers are also referred to the Xperanto Project at IBM Almaden that has many of the same goals.

The next papers describe three of the main systems that explore native XML query processing, but an emphasis on integrating data from multiple sources on the network. Ives et al. describe the Tukwila System, which extends the key techniques from pipelined relational query processors to support XML query processing over networked bound data sources. Naughton et al. describe the Niagara System whose goal is to answer queries over large collections of external XML documents. The paper by Tufte and Maier describes a specific operator in Niagara for efficiently *merging* two streaming XML documents. The paper by the Xyleme group describes the Xyleme Project whose goal is to provide a dynamic warehouse of the XML data on the web and to support query evaluation and efficient update on the warehouse.

Finally, the paper by Kossmann and Florescu steps back from the specific query processing issues and identifies a new impedance mismatch that arises when programmers try to build applications that involve XML, relational databases and a programming language such as Java. The paper proposes an XML query language whose goal is to remove the impedance mismatch.

Alon Y. Halevy
University of Washington

State-of-the-Art XML Support in RDBMS: Microsoft SQL Server's XML Features

Michael Rys, Microsoft Corp. (mrys@microsoft.com)

Abstract

XML is fast becoming the intergalactic data speak alphabet for data and information exchange that hides the heterogeneity among the components of Loosely-coupled, distributed systems and provides the glue that allows the individual components to take part in the loosely integrated system. Since much of this data is currently stored in relational database systems, simplifying the transformation of this data from and to XML in general and from and to the agreed upon exchange schema specifically is an important feature that should improve the productivity of the programmer and the efficiency of this process. This article provides an overview over the features that are needed to provide access via HTTP and XML and presents the approach taken in Microsoft SQL Server.

Keywords*: Loosely-coupled, distributed system architectures, XML, relational database systems*

1 Introduction

Many applications are built today as loosely-coupled, distributed systems where individual components (often called services) are combined together. Since many of these components will be reused for other applications, the architecture needs to be flexible enough to allow individual components to join or leave the heterogeneous conglomerate of services and components and to change their internal design and data models without jeopardizing the whole architecture.

Over the past two years, XML and the vocabularies defined with XML have established themselves as the most prevalent and promising lingua franca of loosely-coupled, distributed systems in the area of business-to-business, business-to-consumer, or generally, any-to-any data interchange and integration. One of the major reasons for the emergence of XML in this space is, that XML is a simple, platform independent, Unicode based syntax that for which simple and efficient parsers are widely available. Another important factor in favor of XML is its ability to not only represent structured data, but to provide a uniform syntax for structured data, semistructured data (data that is sparse or is of heterogeneous types) and marked-up content.

Since the interchanged data may be used independently of the integration context or used in multiple integration contexts, it is crucial that the internal modeling of the data structures is flexible enough to accommodate a changing number of contexts without sacrificing performance. Relational databases have a proven track record in providing the required efficient and flexible management of data in such multiple usage contexts. Local applications already make extensive use of relational systems to manage their data. Thus, in order to provide already existing data to the integration systems and to leverage the flexibility of relational databases, such components are often based on relational systems. However, in order to partake in the integration process as a component, the relational data needs to be translated into XML. In particular, the relational schema needs to be transformed

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

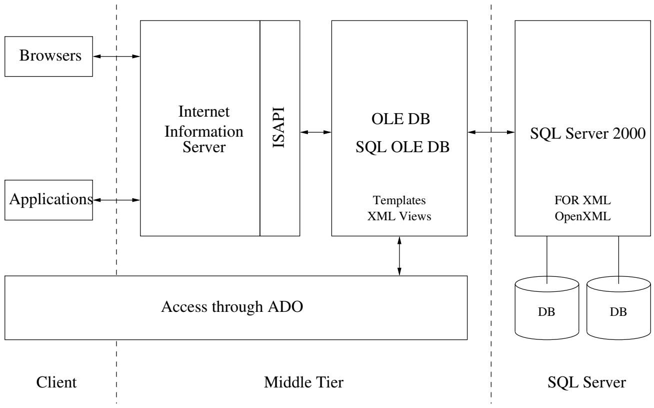


Figure 1: Architectural overview of the XML Access to SQL Server.

as easily and efficiently as possible into the XML language used in the specific application domain and incoming XML data needs to be transformed into relational data in a similar way.

Generally, one can distinguish three major scenarios for producing and storing XML data with (relational) database systems: (i) generating XML from relational data and storing structured XML data in relational format, (ii) providing ways to store and produce semi-structured XML to and from relational data, and (iii) managing arbitrary XML with the database system. Most relational database systems primarily provide support for the first two scenarios because they currently comprise the most common scenarios, support the existing customer base and allow the use of existing relational technology for efficient data management. The third scenario is currently the realm of so called native XML databases, although relational systems may provide some basic technology to support this scenario (such as a CLOB, XML data type, or node table shredders).

This article describes the basic technology to enable a relational database (in particular Microsoft SQL Server) to support these two scenarios. It will describe how to transform relational query results into XML, provide queryable and updateable XML views, and how to elegantly provide rowset abstractions over XML to shred XML into relations. A more detailed description of all features can be found in [7].

2 Architecture of the XML Access to SQL Server

Figure 1 shows a high-level architectural block diagram of SQL Server's XML support. Since different applications have different needs for where to run their business logic, the architecture provides for direct HTTP access when only visualization using XSLT needs to be performed on the mid-tier and the rest of the business logic processing can either be completely pushed to the client or to the database server. For two-tier architectures or where the business logic needs to be performed on the middle-tier, a closer coupling of the business logic to the database access is often used due to performance and programmability reasons. Thus, all access to the XML features is provided via the SQLOLEDB provider for both ADO access as well as HTTP access via the ISAPI extension to the Internet Information Server.

The major mid-tier access methods provided by SQL Server are the template access and the XML view access. Templates are XML documents that provide a parameterized query and update mechanism to the database. Since they hide the actual query (or update) from the user, they provide the level of decoupling that makes building loosely-coupled systems possible. Specially named elements containing queries are processed by the template processor and used to return database data as part of the resulting XML document. Templates can contain either T-SQL statements, updategrams, XPath queries or a combination thereof.

XML views are defined by annotating an XML schema document with the mapping to the relational tables and columns. Hierarchies are mapped from and to the database using a relationship annotation that expresses the outer-join between the parent and the children. This view can then be used to both query it using the XPath 1.0 XML navigation language [3] and to update it using so-called updategrams.

3 Serializing SQL Query Results into XML

People that are familiar with writing SQL select queries want to be able to easily generate XML from their query result. Unfortunately, there are many different ways how such a serialization into XML can be done. SQL Server therefore provides three different modes for the serialization with different levels of complexity and XML authoring capability. All three modes are provided via a new select statement extension called FOR XML. The three modes are: raw, auto, and explicit. The (simplified) syntax of the FOR XML clause is ([] indicates optional, | indicates alternative):

```
FOR XML (raw | auto [, elements] | explicit) [, xmldata]
```

All three modes basically map rows to elements and column values to attributes. The optional directive elements changes the mapping of all column values to subelements in the auto mode (the explicit mode has column-specific control over the mapping, see below). The optional directive xmldata will generate an inline schema using the XML-Data Reduced schema language as part of the result that describes the structure and data types of the XML query result.

All three FOR XML modes share the general design goals that the XML generation should not slow down arbitrary non-FOR XML queries and that the result should be streamed through the serialization process instead of generating the XML in a server-side cache. In order to not impact the database system's relational engine for non-XML queries, the serialization is performed as a post-processing step on the resulting rowset after the query execution is done. As a consequence, information about the instance lineage of the data in case of non-primary key-foreign key joins is not available for the serialization, since one cannot tell if the master data in the join comes from one or multiple rows. The second requirement is to avoid the costly build-ups of large XML fragments on the server. In order to avoid such caching, the serialization rules for hierarchical results in the auto and explicit modes require, that rows containing parent data need to be directly followed by their children and children's children data so that the data can be serialized in a streaming fashion.

3.1 The RAW Mode

The raw mode is the simplest mode. It performs a so-called canonical mapping where any row of the query result is mapped into an element with the name row and any column value that is not null into an attribute value of the attribute with the column name. NULL values are mapped to the absense of the attribute. For example, the query

```
SELECT CustomerID, OrderID
FROM Customers LEFT OUTER JOIN Orders ON Cus-
tomers.CustomerID = Orders.CustomerID
FOR XML raw
```

may return

```
<row CustomerID="ALFKI" OrderID="10643"/><row Cus-
tomerID="ALFKI" OrderID="10692"/>
```

Since the query results do not contain nested rowsets, the raw mode only returns flat XML documents where the hierarchy of the data is lost. It however works with any SQL query and the serialization process is very efficient. On the negative side, such canonical mappings are normally not generating the XML in the semantic format needed for exchanging the data. Thus the additionally required postprocessing step (e.g., using an XSLT stylesheet) adds additional processing overhead which often more than negates the initial performance gains of the serialization.

3.2 The AUTO Mode

Since the instance level lineage is not available to the serialization, the auto mode applies a heuristic on the returned rowset to determine nesting of the data. It basically maps each row to an element while using the table alias as the element name. Nesting is determined by taking schema level lineage information provided by the SQL Server query processor into account. Basically, the left to right appearance of a table alias in the SELECT clause determines the nesting. Columns of aliases that are already placed in the hierarchy are grouped together even if they appear interspersed with columns of other aliases. Computed and constant columns are associated with the deepest hierarchy so far encountered (or with the top level of the first alias). As a consequence, the auto mode cannot provide differently typed sibling elements.

The serialization process streams the XML by looking at each row that arrives from the query processor and opening a new hierarchy level for the level where all the ancestor data is unchanged, previously closing any lower hierarchies of the sibling. The auto mode's hierarchy serialization together with the instance lineage issue mentioned above means that multiple, indistinguishable parents will be merged to one parent and that parents without children and parents with children without properties will be represented as parents with children without properties. As a consequence, children that are not directly following their parent will add a duplicate parent at the place where it reappears in the rowset stream.

For example, the auto mode query

```
SELECT Cust.CustomerID, OrderID as id
FROM Customers Cust LEFT OUTER JOIN Or-
ders ON Cust.CustomerID = Orders.CustomerID
ORDER BY Customers.CustomerID
FOR XML auto
```

may return `<Cust CustomerID="ALFKI"><Orders id="10643"/><Orders id="10692"/></Cust>`.

3.3 The EXPLICIT Mode

The explicit mode allows generating arbitrary XML without any of the auto mode limitations. However, the explicit mode expects that the query be explicitly authored to return the rowset in a specific format. This format, commonly known as a universal table format, provides enough information to generate arbitrary XML. In particular, the explicit mode can generate arbitrary tree structured hierarchies, collapse or hoist hierarchical levels independently of the involved tables, and can generate IDREFS type collection attributes.

The general format and approach is best explained with an example. Explaining every detail of the explicit mode is beyond the scope of this paper. Therefore, the reader is referred to the documentation for the details. The goal is to generate an XML document of the following form:

```
<Customer cid="ALFKI"><name>Alfreds Futterkiste</name><Order oid="O-10643"/>
</Customer>
<Customer cid="BOLID"><name>Bolido Comidas preparadas</name><Order oid="O-
10326"/>
</Customer>
```

Note that while this example still only consists of a simple hierarchy, one Customer column has to be mapped to an attribute and the other one to a subelement, a task that cannot be accomplished by the auto mode. In order to generate XML of this format, the explicit mode expects a universal table of the following format:

Tag	*Parent*	*Customer!1!cid*	*Customer!1!name!element*	*Order!2!oid*
1	0	ALFKI	Alfreds Futterkiste	NULL
2	1	ALFKI	NULL	O-10643
1	0	BOLID	Bolido Comidas preparadas	NULL
2	1	BOLID	NULL	O-10326

Each row corresponds to an element (with the exception of IDREFS where each row is an element of the list). The columns Tag and Parent are used to encode the hierarchy levels for each row (if the parent tag is 0 or

NULL, the tag is the top level). The column names encode the mapping of the hierarchy levels to the element name; in the given example level 1 corresponds to an element of name Customer. The column names also encode the name of the attribute (or subelement) of the values in that column as well as additional information such as whether the value is a subelement or some other information (such as IDREFS, CDATA section, XML annotation etc.).

The serialization process takes each row and determines based on the tag level and the parent tag what level the element is. It uses the information encoded in the column name to only generate the column attributes and subelements for the current level. Thus all other columns can contain NULL. Due to the streaming requirement, children have to immediately follow their parent, thus the key field columns of the parent often contain the key values like in this example because they were used to group the children with their parent element. However the explicit mode only cares about the order and does not care about the parent's key value.

In principle, the explicit mode does not care about the query that generates the universal table format. One could create a temporary table of this format, insert the data and then perform an explicit mode query over the temporary table that guarantees the right grouping of children and parents to generate the XML. This however would most likely not perform well. Thus, the currently best way to generate this format by means of a single query is to issue a selection for each level, union all them together, and use an order by statement to group children under their parents (similar to the sorted union approach presented in [4]). This basically generates a left outer join where each join partner is placed into its own vertical and horizontal partition of the rowset. Thus the query for generating the universal table and therefore the XML in our example would look like:

```
SELECT 1 as Tag, NULL as Parent,
  CustomerID AS "Customer!1!cid", CompanyName AS "Customer!1!name!element",
  NULL AS "Order!2!oid"
FROM Customers
WHERE CustomerID = 'ALFKI' OR CustomerID='BOLID'
UNION ALL
SELECT 2, 1,
       Customers.CustomerID, NULL,
       'O-'+CAST(Orders.OrderID AS varchar(32))
FROM Customers INNER JOIN Orders ON Customers.CustomerID=Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID='BOLID'
ORDER BY "Customer!1!cid"
FOR XML explicit
```

The query processor pushes the order-by operation down to each individual selection and performs a merge union.

4 Providing XML Views over Relational Data

The previous section presented the SQL-centric approach to generating XML. SQL Server also provides a mechanism that allows defining virtual XML views over the relational database which then can be queried and updated with XML-based tools.

4.1 Annotated Schemata

The core mechanism of providing XML views over the relational data is the concept of an annotated schema. Annotated schemata consist of an XML-based schema description of the exposed XML view (using either the XML-Data Reduced or the W3C XML Schema language) and annotations that describe the mapping of the XML schema constructs onto the relational schema constructs. In order to simplify the definition of the annotations, each schema provides a default mapping if no annotation is present. The default mapping maps an attribute or a non-complex subelement (i.e., content type is text only) to a relational column with the same name. All other elements map into rows of a table or view with the same name. Hierarchies are expressed with annotations. A downloadable visual tool called the SQL XML View Mapper provides a simple and intuitive way to specify the mapping, and thus the annotations, graphically. The view can now be queried using an XML query language and updated via an XML-based update language.

The following example shows a simple XML view (XDR-based) that defines a Customer-Order hierarchy over the Customers and Orders table of the relational database. Everything that is in the default namespace belongs to the XDR schema definition; the annotations are associated with the namespace `urn:schemas-microsoft-com:xml-sql`.

```
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:sql="urn:schemas-microsoft-com:xml-sql">
 <ElementType name="Customer" sql:relation="Customers">
   <AttributeType name="ID"/> <attribute type="ID" sql:field="CustomerID"/>
   <element type="Order">
     <sql:relationship key-relation="Customers" key="CustomerID"
                       foreign-relation="Orders" foreign-key="CustomerID"/>
   </element>
 </ElementType>
 <ElementType name="Order" sql:relation="Orders">
   <AttributeType name="OrderID"/> <attribute type="OrderID"/>
 </ElementType>
</Schema>
```

The sql:relationship annotation provides the hierarchy information as a conceptual left-outer join that describes the how the child data relates to the parent data. Additional annotations exists to define XML specific information such as defining ID-prefixes, CDATA sections, and XML overflow.

The annotated schema does not retrieve any data per se, but only defines a virtual view by projecting an XML view on the relational tables. It actually defines two potential views, customers containing orders and just orders, since the schema does not define an explicit root node. The query and the updategram provide the additional information to actually determine which of the two views will be used.

4.2 Querying using XPath

XPath is a tree navigation language defined by a W3C recommendation [3]. XPath is not a full-fledged query language (it does not provide constructive elements such as projection or sub-tree pruning), but serves as a basis for navigating the XML tree structure. Each XPath basically consists of a sequence of location steps that navigate the tree with optional predicates to constrain the navigation paths.

SQL Server uses a subset of XPath to select data from the virtual XML views provided by annotated schemata. The first location step of the XPath determines the view used of the potential views defined by the annotated schema and returns the serialization of the selected nodes and their complete subtree. The currently supported constructs include easily mapable constructs such as the non-order, non-recursive navigation axes, all datatypes, most operators, and variables. Notably not supported in the current release are the id() function, the order and recursive axes such as the descendent axis.

The XPath query together with the annotated schema is translated into a FOR XML explicit query that only returns the XML data that is required by the query. The implementation goes to great length to provide the true XPath semantics such as preserving the node list order imposed by the parent orders when navigating down the tree and the XPath data type coercion rules. The only two places where the implementation differs from the W3C XPath semantics is with respect to the coercion rules of strings with the $<$ and $>$ comparison operations and with respect to node to string conversions in predicates. In the first case, the implementation does not try to coerce to a number but does a string-based comparison on the default collation, which provides support for datetime comparisons. In the second case, XPath mandates a ”first-match” evaluation semantics that cannot be mapped to the relational system. Instead, the implementation performs the more intuitive ”any-match” evaluation.

For example, the XPath `/Customer[@ID='ALFKI']` against the annotated schema above may result in `<Customer ID="ALFKI"><Order OrderID="10643"/><Order OrderID="10692"/></Customer>`.

4.3 Updating using Updategrams

Updategrams provide an intuitive way to perform an instance-based transformation from a before state to an after state. Updategrams operate over either a default XML view implied by its instance data (if no annotated schema is referenced) or over the view defined by the annotated schema and the top-level element of the updategram. The following example, gives a simple example:

```
<root xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
  <updg:sync mapping-schema="nwind.xml" nullvalue="ISNULL">
    <updg:before>
      <Customer CustomerID="LAZYK" CompanyName="ISNULL" Address="12 Orches-
tra Terrace" >
        <Order oid="10482"/>
      </Customer>
    </updg:before>
    <updg:after>
      <Customer CustomerID="LAZYK" CompanyName="Lazy K Store" Ad-
dress="12 Opera Court" >
        <Order oid="10354"/>
      </Customer>
    </updg:after>
  </updg:sync>
</root>
```

Updategrams use their own namespace `urn:schemas-microsoft-com:xml-updategram`. Each updg:sync block defines the boundaries of an update batch that uses optimistic concurrency control to perform the updates transactionally. The before image in updg:before is used both for determining the data to be updated as well as to perform the conflict test. The after image in updg:after gives what has to be changed. If the before state is empty or missing, the after state defines an insert, if the after state is empty or missing, the before state defines what has to be deleted. Otherwise the necessary relational insertions, updates and deletions are inferred from the difference between the before and after image. Several optional features allow the user to deal with identity and aligning elements between the before and after state. The nullvalue attribute indicates that the fields with the specified value needs to be compared or set to NULL respectively.

In the example above, the customer with the given data (including a company name set to NULL) gets a new company name and address. In addition, the relation to the order 10482 is removed and replaced by a new relation to order 10354.

5 Providing Relational Views over XML

In many cases, data will be sent to the database server in the form of an XML message which needs to be integrated with the relational data after optionally performing some business logic over the data inside a stored procedure on the server. This requires programmatic access to the XML data from within a stored procedure. Unfortunately, neither the DOM nor SAX provides a well-suited surface API for dealing with XML data in a relational context. Instead the new API needs to provide a relational view over the XML data, i.e., it needs to allow the SQL programmer to shred an XML message into different relational views.

SQL Server provides a rowset mechanism over XML by means of the OpenXML rowset provider. OpenXML provides two kinds of rowset views over the XML data: the edge table view and the shredded rowset view. The edge table view provides the parent-child hierarchy and all the other relevant information of each node in the XML document in form of a self-referential rowset. The shredded rowset view utilizes an XPath expression (the row pattern) to identify the nodes in the XML document tree that will map to rows and uses a relative XPath expression (the column pattern) for identifying the nodes that provide the values for each column. The OpenXML rowset provider can appear anywhere in a SQL expression where a rowset can appear as a data source. In particular, it can appear in the FROM clause of any selection.

One of the advantages of this rowset-oriented API for XML data is that it leverages the existing relational model for use with XML and provides a mechanism for updating database with data in XML format. Using XML in conjunction with OpenXML enables multi-row updates with a single stored procedure call and multi-table updates by exploiting the XML hierarchy. In addition, it allows the formulation of queries that join existing tables with the provided XML data.

In order to have access to some of the implicit meta information in the tree such as hierarchy and sibling information, a column pattern can also be a so-called meta property of the node selected by the row pattern. Examples of such meta properties are @mp:id that provides the node id (the namespace prefix mp binds to a namespace that is recognized by OpenXML as providing the meta properties), @mp:parentid that provides the node id of the parent node, @mp:prev that provides the node id of the previous sibling, and the special metaproperty @mp:xmltext that allows to deal with unknown open content (the so-called overflow). For more information on these meta properties, please refer to the documentation.

The following presents the syntax of the OpenXML rowset provider ([] denote optional parts, | denotes alternatives):

```
OpenXML(hdoc, RowPattern [, Flag]) [WITH SchemaDeclaration | TableName]
```

The hdoc parameter is a handle to the XML document that has been previously parsed with the built-in stored procedure `sp_xml_preparedocument`. RowPattern is any valid XPath expression that identifies the rows or, in case of the edge table view, the roots of the trees to be returned. The optional Flag parameter allows to designate default attribute- or element-centric column patterns in the shredded rowset view. If the WITH clause is omitted, an edge table view is generated, otherwise the explicitly specified schema declaration or the implicitly through TableName given rowset schema is used to define the exposed structure of the shredded rowset view. A schema declaration has the form `(ColumnName ColumnType [ColPattern], ...)`. The ColumnName provides the name of the column, ColumnType the relational data type exposed by the rowset view, and ColumnPattern the optional column pattern (if no value is given, the default mapping indicated with the flag parameter is applied). Note that XML data types are automatically coerced to the indicated SQL data types. For example, the following T-SQL fragment parses a hierarchical Customer-Order XML document and uses the rowset views to load the customer and order data into their corresponding relational tables.

```
create procedure Load_CustOrd (@xmldoc ntext) as
declare @h int
-- Parse document
exec sp_xml_preparedocument @h output, @xmldoc
-- Load the Customer data
insert into Customers select * from OpenXML(@h, '/load-
doc/Customer') with Customers
-- Load the Order data. Get the customer id from the parent element
insert into Orders(OrderID, CustomerID, OrderDate)
  select * from OpenXML(@h, '/loaddoc/Customer/Order', 2)
    with (oid int, customerid nvarchar(10) '../@CustomerID', OrderDate datetime)
-- Remove the parsed document from the temp space
exec sp_xml_removedocument @h
go
```

6 Conclusion and Future Work

This paper presents the basic technologies required to enable a relational database to become an XML-enabled component. Based on SQL Server's XML support, it gives an overview over the different building blocks such as queryable and updateable XML views, rowset views over XML and XML serialization of relational results. Rowset views over XML and the XML serialization of relational results can be characterized as providing XML support for users feeling comfortable in the context of the relational world, whereas the XML views provides XML-based access to the database for people more familiar with XML. Some of the discussed features are at the time of the publication not part of the shipped version of SQL Server 2000, but are released as a web release that can be downloaded from `http://msdn.microsoft.com/sqlserver`. In particular, the SQL XML View Mapper, Annotated W3C XML schemata, updategrams, bulkload and the nullvalue attribute are part of the web release.

Future work will include support for the upcoming W3C XQuery language [2] against the annotated schema view.

References

[1] XML query language (XQuery) version 1.0; W3C working drafts. http://www.w3.org/XML/Query.html.

[2] J. Clark and S. DeRose. XML path language (XPath) version 1.0; W3C recommendation 16 November 1999. http://www.w3.org/TR/xpath.

[3] M. Rys. Bringing the Internet to your database: Using SQL Server 2000 and XML to build loosely-coupled systems. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 465–472. IEEE Computer Society, 2001.

[4] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Linday, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proceedings of the 26th International Conference on Very Large Databases, 2000, Cairo, Egypt*, 2000.

Publishing Relational Data in XML: the SilkRoute Approach

Mary Fernández
AT&T Labs

Atsuyuki Morishima
Shibaura Institute of Technology

Dan Suciu
University of Washington

Wang-Chiew Tan
University of Pennsylvania

1 Introduction

For exchange on the Internet, relational data needs to be mapped to XML, a process that we call *XML publishing*. The mapping is complex, because the two data models differ significantly. Relational data is flat, normalized into many relations, and its schema is often proprietary. By contrast, XML data is nested, unnormalized, and its schema is public, usually created by agreement between members of a community, after lengthy negotiations. Publishing XML data involves joining tables, selecting and projecting the data that needs to be exported, mapping the relational table and attribute names into XML element and attribute names, creating XML hierarchies, and processing values in an application specific manner.

The XML data is a view over the relational database and, as such, can be *virtual* or *materialized*. In a virtual view, applications consuming XML data do so by applying XML queries to an XML virtual view. Some relational query processing is involved whenever the XML data is accessed. This is the most common scenario, because it guarantees data freshness and has the greatest potential to leverage the processing power of relational engines. In this scenario, one issue is the translation of an XML query of an XML view into SQL; this translation may be complex, because the XML view is itself complex. Another issue is that the automatically generated SQL queries often differ significantly from hand-written queries. In a *materialized* view, a large, unique XML document is created once from the relational data, requiring significant computation. Applications requesting the XML data always get the entire view, unless some external XML engine is available to extract portions of the XML data. This solution has trade-offs similar to those of a data warehouse: applications can access all the data without interfering with the relational engine, but the XML view needs to be refreshed periodically. In this scenario, an issue is the cost of computing the materialized view.

In this paper, we describe SilkRoute, a middle-ware system for publishing XML data from relational databases. In SilkRoute, an XML view is defined using a declarative query language called RXL (Relational to XML Transformation Language). RXL extracts data from the relational database and builds the resulting XML document by adding the appropriate element and attribute *tag* names, nesting elements and attributes, and processing values in an application-specific manner. The view can be either virtual or materialized. In virtual publishing, SilkRoute is in a query-processing mode: it accepts XML-QL queries over the XML view, translates them into SQL queries, sends these to the relational engine, tags the resulting tuple streams, and sends the XML result to the user. This mode of operation is described in detail in [6]. In materialized publishing, SilkRoute first optimizes the (usually complex) RXL query before sending it to the relational engine. There are several strategies

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

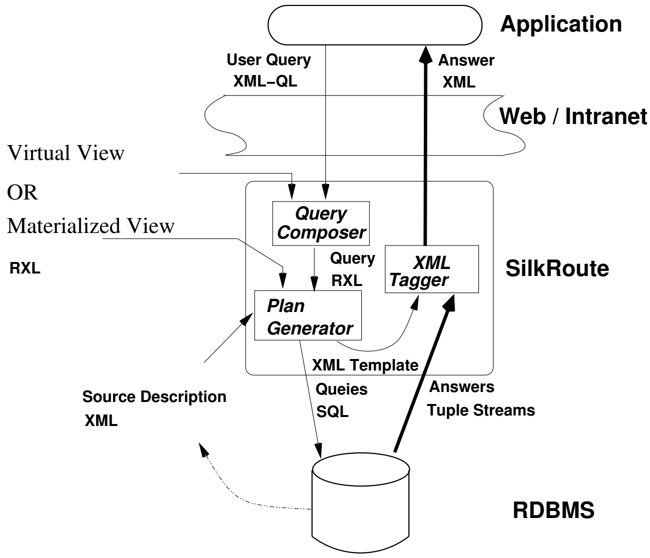


Figure 1: SilkRoute's Architecture.

to decompose a complex RXL query into a collection of SQL queries, and we found that the two simplest ones are significantly sub-optimal. SilkRoute deploys a greedy, heuristic optimization algorithm that decomposes the RXL query into multiple SQL queries. This mode of operation is described in detail in [5].

SilkRoute makes contributions to virtual and materialized XML publishing. In virtual publishing, the main contribution is a query composition algorithm that composes the view-definition query with the user's XML query. This is analogous to view expansion in relational databases, but is complicated by the complexity and hierarchical structure of the view and by the presence of wild-cards in the XML query. Here, we illustrate the algorithm on a simple example and refer the reader to [6] for details.

In materialized publishing, our main contribution is a heuristic algorithm for choosing an optimal execution plan for an XML view. The optimizer's choices range from constructing a single, large SQL query, to constructing many, relatively simple SQL queries both of which produce sorted tuple streams that are merged by an XML tagging module. We found empirically that the optimal plan lies somewhere between these two extremes. The optimizer faces two challenges. First, the size of the search space is, as usual, exponential in the size of the already large view definition. Second, the optimizer has no control over how the resulting SQL queries will be optimized by the relational engine. In SilkRoute, we developed a greedy, heuristic algorithm for finding an optimal query, that bases its decisions on query-cost estimates provided by the relational engine. Although the cost estimates can vary significantly from the real execution times, in our experiments, the greedy algorithm found near-optimal execution plans. We refer the reader to [5] for details.

2 SilkRoute's Architecture

SilkRoute is middle ware between a relational database (RDBMS) and an application accessing that data over the Web. Its architecture is depicted in Figure 1. The database administrator starts by writing an RXL query that defines the XML view of the database. This is called the *view query* and it is typically complex, because it transforms the relational data into a deeply nested XML view. The view can be either virtual or materialized. A

```
Clothing(pid, item, category,
         description, price, cost)

SalePrice(pid, price)

Problems(pid, code, comments)
```

(a)

```
<?xml encoding="US-ASCII"?>
<!ELEMENT supplier (company, product*)>
<!ELEMENT product (name, category, description, retail,
                   sale?, report*)>
<!ATTLIST product ID ID>
<!ELEMENT company (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT retail (#PCDATA)>
<!ELEMENT sale (#PCDATA)>
<!ELEMENT report (#PCDATA)>
<!ATTLIST report code (size|defective|style) #REQUIRED>
```

(b)

Figure 2: Schema of supplier's relational database (a) and the DTD of the published XML view (b).

materialized view is fed directly into the Plan Generator, which generates a set of SQL queries and one XML template. A virtual view is first composed by the Query Composer with a user query (in XML-QL) resulting in another RXL query. A composed XML view is typically much smaller than the complete view definition, therefore plan generation is much simpler for composed views. The SQL queries are sent to the RDBMS server, which returns one sorted tuple stream per SQL query. The XML Tagger merges the tuple streams and produces the XML document, which is returned to the application.

SilkRoute mostly manipulates queries and delegates almost all data processing to the relational database engine. SilkRoute only touches data in the XML tagger, in which it merges multiple tuple streams and generates the result XML document in a single pass.

3 An Example

The Scenario We illustrate how to define an XML view in SilkRoute with a simple example from electronic commerce, in which *suppliers* provide product information to *resellers*. For their mutual benefit, suppliers and resellers have agreed to exchange data in a format that conforms to the DTD in Figure 2 (b). It includes the supplier's name and a list of available products. Each product element includes an item name, a category name, a brief description, a retail price, an optional sale price, and zero or more trouble reports. The contents of a retail or sale element is a currency value. A trouble report includes a code attribute, indicating the class of problem; the report's content is the customer's comments. Most importantly, this DTD is used by suppliers and resellers, and it is a public document.

Consider now a particular supplier whose business data is organized according to the relational schema depicted in Figure 2 (a). The `Clothing` table contains tuples with a product id (the table's key), an item name, category name, item description, price, and cost. The `SalePrice` table contains sale prices and has key field `pid`; the `Problem` table contains trouble codes of products and their reports. This is a third-normal form relational schema, designed for the supplier's particular business needs. The supplier's task is to convert its relational data into a valid XML view conforming to the DTD in Figure 2 (b) and make the XML view available to resellers. In this example, we assume the supplier exports a subset of its inventory, in particular, its stock of winter outer-wear that it wants to sell at a reduced price at the end of the winter season.

The View Definition Figure 3 contains the complete view query for our supplier example in Sec. 3, expressed in RXL. Skolem functions are used to control the creation and grouping of elements: for example, in `<product ID=Prod($c.pid)>` the function `Prod` is a Skolem function ensuring that a distinct `product` element will be created for every distinct value of `$c.pid`. Lines 1, 2, and 27 create the root `<supplier>`

```
 1. construct
 2. <supplier ID=Supp()>
 3.   <company ID=Comp()>"Acme Clothing"</company>
 4.    {
 5.     from  Clothing $c
 6.     where $c.category = "outerwear"
 7.     construct
 8.       <product ID=Prod($c.pid)>
 9.         <name  ID=Name($c.pid,$c.item)>$c.item</name>
10.         <category ID=Cat($c.pid,$c.category)>$c.category</category>
11.         <description ID=Desc($c.pid,$c.description)>$c.description</description>
12.         <retail ID=Retail($c.pid,$c.price)>$c.price</retail>
13.        { from SalePrice $s
14.          where $s.pid = $c.pid
15.          construct
16.          <sale ID=Sale($c.pid,$s.pid,$s.price)>$s.price</sale>
17.        }
18.        { from Problems $p
19.          where $p.pid = $c.pid
20.          construct
21.          <report code=$p.code ID=Prob($c.pid,$p.pid,$p.code,$p.comments)>
22.            $p.comments
23.          </report>
24.        }
25.        </product>
26.    }
27.  </supplier>
```

Figure 3: RXL view query.

element : notice that the Skolem term `Supp()` has no variables, meaning that a single `<supplier>` element is created. The outer-most clause constructs the top-level element `supplier` and its `company` child element. The first nested clause (lines 4–26) contains the query fragment described above, which constructs one `product` element for each "outerwear" item. Within this clause, the nested clause (lines 13–17) expresses a join between the `Clothing` and `SalePrice` tables and constructs a `sale` element with the product's sale price nested within the outer `product` element. The last nested clause (lines 18–24) expresses a join between the `Clothing` and `Problem` tables and constructs one `report` element containing the problem code and customer's comments; the `report` elements are also nested within the outer `product` element. Notice that the Skolem term of `product` guarantee that all `product` elements with the same identifier are grouped together. Usually Skolem terms are inferred automatically, but here we include them explicitly; they are used by the query-composition algorithm.

4 Virtual XML Publishing

In virtual publishing, the view definition query is not executed, but instead SilkRoute accepts XML-QL queries from applications that access the view. Users never see the relational data directly, but only through the XML view. Queries are formulated in XML-QL, a query language for XML [4].

The User Query The reseller can access that data by formulating queries over the XML view. Figure 4 shows an XML-QL query which retrieves all products with sale price less than half of retail price. The `where` clause consists of a pattern (lines 3-10) and a filter (line 11). A pattern's syntax is similar to that of XML data, but also may contain variables, whose names start with `$`. Filters are similar to RXL (and SQL). The meaning of a query is as follows. First, all variables in the `where` clause are bound in all possible ways to the contents of

```
construct
 <results> {
   where <supplier>
           <company>$company</company>
           <product>
              <name>$name</name>
              <retail>$retail</retail>
              <sale>$sale</sale>
           </product>
         </supplier> in "http://acme.com/products.xml",
         $sale < 0.5 * $retail
   construct
         <result ID=Result($company)>
           <supplier>$company</supplier>
           <name>$name</name>
         </result>
 } </results>
```

Figure 4: XML-QL user query.

```
construct
  <results>
  { from  Clothing $c, SalePrice $s
    where $c.category = "outerwear",
          $c.pid = $s.pid,
          $s.price < 0.5 * $c.retail
    construct
      <result ID=Result("Acme Clothing")>
        <supplier ID=Supp("Acme Clothing")</supplier>
        <name ID=Name("Acme Clothing", $c.pid, $c.item)>
                    $c.item
        </name>
      </result>
  }
  </results>
```

Figure 5: Composed RXL query.

elements in the XML document. For each such binding, if the filter condition is satisfied then the `construct` clause constructs an XML value. Grouping is expressed by Skolem terms in the `construct` clause. In this example, the construct clause produces one `result` element for each value of `$company`; each `result` element contains the supplier's name and a list of `name` elements containing the product names. It is important to notice that answer to the user query includes a small fraction of the relational database, i.e., only those products that are heavily discounted.

The Query Composer SilkRoute's query composer takes a user XML-QL query and composes it with the RXL view query resulting in a new RXL query. For lack of space, we only illustrate the query composition for our running example and refer the reader to [6] for details. Given the view query in Fig. 3 and the user query in Fig. 4, the composed query is given in Fig. 5. The composed query combines fragments of the view query and user query. Those fragments from the user query are highlighted. The composed query extracts data from the relational database in the same way as the view query. It also includes the user filter `$s.price < 0.5 * $c.retail` and structures the result as in the user query.

5 Materialized XML Publishing

In materialized XML publishing, we evaluate the view-definition query directly. In general, this is a relatively large query, which returns a large result, therefore choosing an optimal plan is critical. To illustrate alternatives, consider the view definition in Fig. 3. Fig. 6 (a) and (b) illustrate two strategies of computing this query in a relational engine. The first consists of three SQL queries returning three sorted tuple streams; the XML tagger merges the three sorted tuple streams and adds the XML tags. The second strategy is a single SQL query resulting in a single sorted tuple stream with tuples of two different types; tuples are coerced into a common type by padding them with null fields. In this case, the XML tagger reads a single tuple stream and just adds the XML tags.

In general, there are many ways in which an RXL query can be translated into a set of SQL queries, each generating a sorted tuple stream which the XML tagger merges and tags. The choices range from many, simple select-project-join SQL queries like in Fig. 3(a) to one large SQL query involving left outer joins and unions, like in Fig. 3(b). The latter may seem to be the best choice, since the relational query optimizer can always choose to implement it with multiple merge joins, thus mimicking all other strategies we may consider. However, we

```
select c.pid, c.item, c.category,
       c.description, c.price
from Clothing c
where c.category = "outerwear"
order by c.pid

select c.pid, s.price
from Clothing c, SalePrice s
where  c.category = "outerwear" and
       s.pid = c.pid
order by c.pid

select c.pid, p.code, p.comments
from Clothing c, Problems p
where  c.category = "outerwear" and
       p.pid = c.pid
order by c.pid
```

(a)

```
select c.pid, c.item, c.category,
       c.description, c.price,
       Q.price, Q.code, Q.comments
from
       (select c.pid, c.item, c.category,
               c.description, c.price
        from Clothing c
        where c.category = "outerwear")
       left outer join
       ((select s.pid as pid, s.price as price,
               null as code, null as comments
        from SalePrice s)
       union all
       (select p.pid as pid, null as price,
               p.code as code, p.comments as comments
       from Problems p)) as Q
on c.pid = Q.pid
order by c.pid
```

(b)

Figure 6: Two plans for computing the RXL view in Fig. 3.

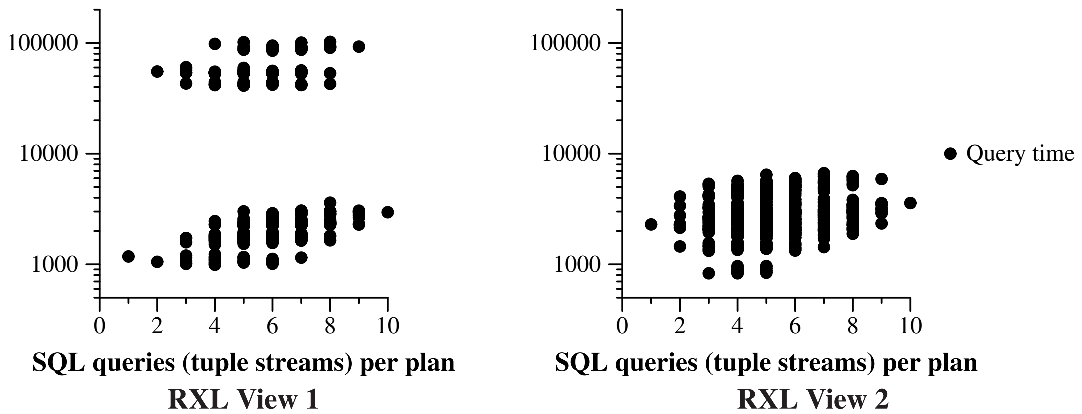


Figure 7: Query times (in msec) for all execution plans (1 MB database)

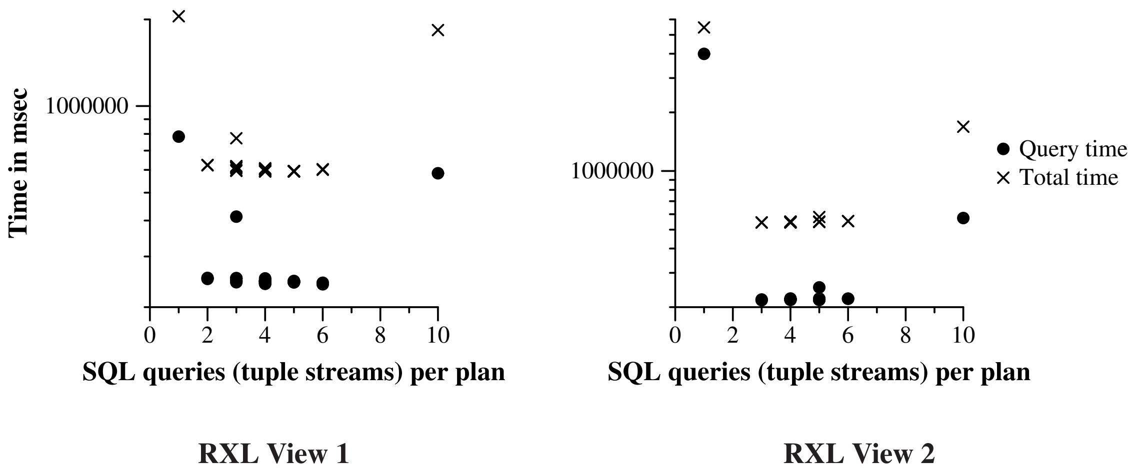


Figure 8: Query times for plans selected by greedy algorithm (100 MB database)

found at least one commercial RDBMS does not optimize effectively large SQL queries containing multiple outer joins. We considered two RXL view queries that create different XML views of the TPC/H benchmark. Each RXL view has $2^9 = 512$ translations into equivalent sets of SQL queries. We report their execution times on a 1 MB instance of the TCP-H database in Fig. 5 (queries that timed out after 5 minutes are not shown). From the graphs, we see that the two extreme queries (with one tuple stream, and with ten tuple streams respectively), were not optimal for either view. Instead, the optimum lies somewhere in the 4-6 tuple streams region.

In SilkRoute, we developed a greedy optimization algorithm that chooses an optimal set of SQL queries for a given RXL view definition. The algorithm bases its decisions on query cost estimations provided by the relational engine. The algorithm can be tuned to return more than one plan, allowing it to be integrated with additional optimization algorithms that optimize specific parameters, such as network traffic or server load. For a one MB instance of the TCP-H database, the greedy algorithm selected the top 32 fastest execution plans. For the 100 MB database, the greedy algorithm selected plans that were substantially faster than the two extreme plans. Fig. 5 reports the query time and total execution time (query + data transfer to SilkRoute) for the selected plans. Details of the greedy algorithm can be found in [5].

6 Discussion

Alternative Approaches We have described what we believe to be the most general approach for exporting relational data into XML. Other approaches are possible, and in some cases, may be more desirable.

Currently, the most widely used Web interfaces to relational databases are HTML forms with CGI scripts. User inputs are translated by a script into SQL queries, and their answers are rendered in HTML. The answers could be generated just as easily in XML. Forms interfaces are appropriate for casual users, but inappropriate for data exchange between applications, because they limit the application to only those queries that are predetermined by the form interface. Aggregate queries, for example, are rarely offered by form interfaces.

In another alternative, the data provider can either pre-compute the materialized view or compute it on demand whenever an application requests it. This is feasible when the XML view is small and the application needs to load the entire XML view in memory, e.g., using the DOM interface. However, pre-computed views are not dynamic, i.e., their data can become stale, and are not acceptable when data freshness is critical.

A third alternative is to use a native XML database engine, which can store XML data and process queries in some XML query language. XML engines will not replace relational databases, but a high-performance XML engine might be appropriate to use in data exchange. For example, one could materialize an XML view using SilkRoute and store the result in an XML engine that supports XQuery, thus avoiding the query composition cost done in SilkRoute. We don't expect, however, XML engines to match in performance commercial SQL engines anytime soon. In addition, this approach suffers from data staleness, and incurs a high space because it duplicate the entire data in XML.

Related Work Shanmugasundaram et al. [1, 8] describe an extension of IBM's DB2 relational engine to support XML publishing, with a focus on efficient computation of materialized views. This work considered a larger class of XML publishing techniques than does SilkRoute, including computing the XML view inside the relational engine. Overall, the in-engine methods perform better than the out-of-engine solutions.

Microsoft SQL Server 2000 allows users to define both virtual XML views and materialized XML views from a relational database [7]. Virtual views are defined in the XDR language. XDR is a specialized language for constructing XML views from a relational database, derived from an XML schema-definition language, and corresponds, in terms of expressive power, to a restricted fragment of RXL. Users can apply XPath [3] queries to the virtual view, which are translated by the system into relational queries. Alternatively, SQL Server 2000 offers a SQL extension called "FOR XML", that allows users to define quite complex XML views, including some that are not expressible in RXL. However, the resulting views are always materialized.

Our virtual view approach is based on the XML-QL query language described in [4]. Currently, a standard query language for XML, XQuery [2], is being defined by the W3C. XQuery is more expressive and complex than XML-QL. An open research problem is whether query composition and decomposition is possible for the complete XQuery language or for only a subset of the language.

References

[1] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. subramanian. XPERANTO: publishing object-relational data as XML. In *Proceedings of WebDB*, Dallas, TX, May 2000.

[2] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: a query language for XML, 2001. available from the W3C, `http://www.w3.org/TR/query`.

[3] J. Clark. XML path language (XPath), 1999. available from the W3C, `http://www.w3.org/TR/xpath`.

[4] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the Eights International World Wide Web Conference (WWW8)*, pages 77–91, Toronto, 1999.

[5] M. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middle-ware queries. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Santa Barbara, 2001.

[6] M. Fernandez, D. Suciu, and W. Tan. SilkRoute: trading between relations and XML. In *Proceedings of the WWW9*, pages 723–746, Amsterdam, 2000.

[7] M. Rys. Bringing the internet to your database: using SQLServer 2000 and XML to build loosely-coupled systems. In *Proceedings of the International Conference on Data Engineering*, pages 465–472, 2001.

[8] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as xml documents. In *Proceedings of VLDB*, pages 65–76, Cairo, Egipt, September 2000.

Integrating Network-Bound XML Data

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

Abstract

Although XML was originally envisioned as a replacement for HTML on the web, to this point it has instead been used primarily as a format for on-demand interchange of data between applications and enterprises. The web is rather sparsely populated with static XML documents, but nearly every data management application today can export XML data. There is great interest in integrating such exported data across applications and administrative boundaries, and as a result, efficient techniques for integrating XML data across local- and wide-area networks are an important research focus.

In this paper, we provide an overview of the Tukwila data integration system, which is based on the first XML query engine designed specifically for processing network-bound XML data sources. In contrast to previous approaches, which must read, parse, and often store XML data before querying it, the Tukwila XML engine can return query results even as the data is streaming into the system. Tukwila features a new system architecture that extends relational query processing techniques, such as pipelining and adaptive query processing, into the XML realm. We compare the focus of the Tukwila project to that of other XML research systems, and then we present our system architecture and novel query operators, such as the x-scan *operator. We conclude with a description of our current research directions in extending XML-based adaptive query processing.*

1 Introduction

The original vision of XML as the data description format that would reshape the web has, to this point, not materialized as expected. HTML documents and dynamic HTML pages predominate, whereas XML documents are rare. Yet behind the scenes, on the Internet and intranets, XML has in fact been widely deployed. Corporations are interchanging data between applications with XML as the common format. Business coalitions and partners are exchanging data and transactions via XML, using standardized DTDs such as those published at OASIS and BizTalk[1]; and Microsoft's .NET "web service" architecture is also based on XML data interchange. Currently, XML has been most successful not as a format for "materialized" or document data, but as a "wire protocol" for exchanging data through virtual XML views.

We are convinced, therefore, that the areas of interoperability (both in terms of schema and data) and data integration are the most significant concerns in XML data management. XML storage and indexing, as well as query processing techniques for local XML repositories, address interesting and relevant problems — yet XML's greatest potential may be as a medium for integrating and exchanging data. Thus we need mechanisms for efficiently processing non-materialized, un-indexed, streaming XML data that is being sent across a network.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*]Supported in part by an IBM Research Fellowship..

[1]Found at www.xml.org and www.biztalk.org.

	Tukwila	Niagara	Xyleme	Lore	XFilter	eXcelon/ Tamino	Silkroute/ XPERANTO
Live Source Data	√			Repository		Repository	Relational
Index of Remote Data		√			√		
Warehoused Data			√				
Query Subscription		√	√	√			
Larger than RAM	√				√	√	√
Native XML	√	√	√	√	√	√	Relational

Table 1: Characterization of XML query processing space and systems

The primary focus of the Tukwila data integration system is to develop new techniques for processing dynamically-generated, streaming XML data. Our emphasis is on developing a query processor that operates on queryable XML data sources, requesting the desired data and efficiently modifying, combining, and restructuring it. This data may potentially be larger than memory, so Tukwila also emphasizes support for out-of-core execution.

This focus on large, "live" queryable sources is in contrast to the other XML query processing projects and systems of which we are aware; see Table 1 for a comparison of system features. Niagara [NDM+01] and Xyleme [Aea01] focus on providing query capabilities for XML documents or data files that are locally indexed or warehoused, respectively; both provide support for "subscription" queries whose results update as the underlying data is modified. Lore [GMW99] is a semistructured data repository with XML extensions, and eXcelon [XLN] and Tamino [Tam] are native XML repositories. Silkroute [FTS99, FMS01] and XPERANTO [CFI+00] provide XML publishing capabilities for relational data. XFilter [AF00] (and the associated DBIS system) is not precisely a query processor, but rather an information dissemination system whose goal is to take a set of XPath queries expressing a user's interests, and to "push" or disseminate all documents matching these queries to the user.

The Tukwila project originated as a relational-model data integration system with a new adaptive architecture to improve query performance [IFF+99]. In our original work, we identified a number of desiderata for a data integration system. Data integration queries are in many cases ad-hoc and interactive (e.g. combining data from several sources to produce a browsable online catalog), so early answers, and thus **pipelining** of results, is desirable. Second, as data is transferred from a source to the integration engine, it is subject to delays and burstiness; hence operators must support **flexible scheduling** so the query processor can process other available tuples while waiting for delayed sources. Finally, since statistics are often incomplete or even nonexistent for remote, dynamically generated data, the query processor should support **runtime re-optimization and re-scheduling** of a query, so it can adjust a query plan as it acquires better knowledge about the data sources. These desiderata are design goals of *adaptive query processing*, and considerable work has recently been done in this area, including query scrambling [UFA98], mid-query re-optimization [KD98], ripple [HH99] and pipelined hash [WA91, IFF+99, UF00] joins, eddies [AH00], dynamic scheduling of pipelines [UF01], and streaming of partial results through blocking operators [STD+00]. (For more details, see the June 2000 issue of the *IEEE Data Engineering Bulletin*.)

In the context of integrating XML data, the data model has changed, but the desiderata remain the same. In fact, XML query languages such as XML-QL and XQuery include operations supported by SQL, such as group-by and join — hence even the same "adaptive" relational techniques should almost directly apply. Yet the XML query model does not operate on tuples in tables — instead, it works 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 input bindings, and the query operators are evaluated against each successive combination. At first glance, this seems quite different from relational-style execution, but a closer examination reveals little difference: if we assign a column in a tuple to each variable, we can view each legal combination of variable assignments as forming a tuple of binding values. Note, however, that this "binding tuple" will likely contain *trees* (or graphs) in its columns, rather than scalar values. Moreover, the content of one column might actually be contained within the tree value of another column.

As a result of this observation, the Tukwila architecture, like a relational data integration system, is based on a tuple-oriented query processing model — but it includes extensions to the tree data type, the ability to incrementally generate binding tuples from an incoming XML stream, and a set of additional XML-specific operations that have no correspondence in a relational system. Our architecture leverages many useful techniques developed for relational data management, while offering the flexibility required to query XML and providing better performance than previous XML query processing approaches.

In the next section, we describe the Tukwila architecture in more detail and present a subset of the experimental results demonstrating Tukwila's impressive performance. We conclude in Section 3 with a discussion of our current research directions in XML and adaptive query processing.

2 The Tukwila System

The Tukwila system supports the majority of features in the XQuery language under development by the World Wide Web Consortium. XQuery includes a tree-structured data model, support for arbitrary recursive functions, conditional queries, and strong typing. Tukwila does not yet attempt to implement the complex sub-typing capabilities of XQuery (which are still under heavy development), and support for recursive functions and conditionals is still under construction (a challenge here is the inherent difficulty of optimizing these queries).

A query is fed into Tukwila's optimizer, which is designed along the lines of Starburst [HFLP89] and Volcano [GM93], with algebraic-level rewrites. The primary difference between Tukwila's optimizer and previous work is in supporting XML-specific operations such as nesting of structure, and in supporting new XQuery features such as arbitrary recursive functions. In our experiments, we have observed that statistics are even harder to obtain for XML data than they are for autonomous relational data sources, and hence it is difficult for any optimizer to generate optimum plans — later in this paper, we present our current focus in attempting to address this problem.

The Tukwila query execution engine features a number of innovations, but preserves several important components from the original relational-model Tukwila engine, including support for multithreaded I/O and operators, plus an event handler for error and exception conditions. In the remainder of this section, we briefly discuss the novel components that provide native XML support; more detail can be found in [IHW01].

2.1 Query Execution Components

The diagram in Figure 1 shows the XML components of the Tukwila system. XML input streams are fed into *x-scan* operators, which match the query's input path expressions against the data and output streams of binding tuples. These binding tuples contain assignments of XML elements (trees) to each of the query's input variables. Since the bound trees may be arbitrarily large and may overlap, binding tuples actually contain *references* rather than the full XML elements; the XML data values are stored separately in an XML Tree Manager that may be paged to disk. The binding tuples are fed into a tree of query operators (described in the next section), which filter, tag, and combine the tuples. Finally, the tagged tuples are passed to an XML Generator, which converts them back into the XML format and returns the result stream to the query initiator.

At first glance it may appear likely that, because of Tukwila's reliance on the Tree Manager, a large XML file could produce "thrashing" in the swap file during query processing, but we have experimental evidence [IHW01] that Tukwila avoids this problem, which we attribute to two factors. First, the system supports "inlining" of scalar values: string, integer, or other "small" data items can be embedded directly in the tuple, avoiding the

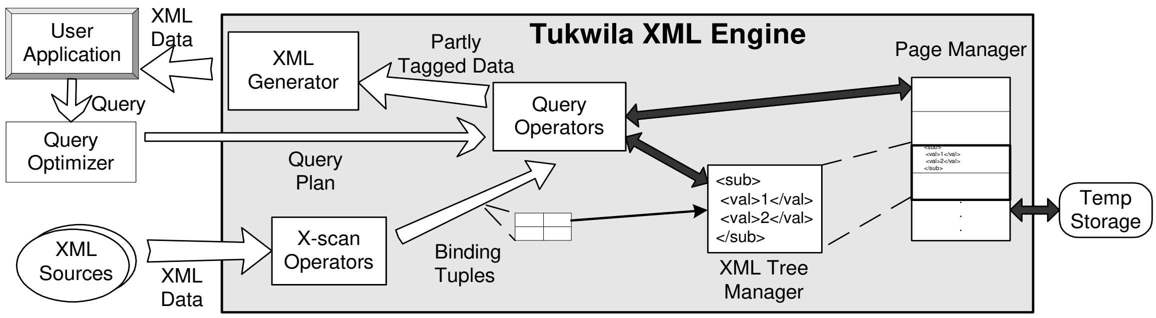


Figure 1: Architecture of the Tukwila query execution engine. After a query plan arrives from the optimizer, data is read from XML sources and converted by x-scan operators into output tuples of subtree bindings. The subtrees are stored within the Tree Manager (backed by a virtual page manager), and 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.

dereferencing operation. Typical query operations in XML-QL and 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 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 is larger in a DOM tree); 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.

2.2 Operations for Processing XML

As was previously suggested, Tukwila includes all of the standard relational database operator implementations, including selection, projection, grouping, sorting, and nested-loops, merge, hash, and double pipelined hash joins. These operators have all been extended to work with data in the Tree Manager, but are otherwise unremarkable. We now provide a brief sketch of the novel XML operators in Tukwila; please see [IHW01] for a more detailed description of each operator.

The novel operator most fundamental to Tukwila's operation, which enables the entire tuple-oriented query processing model, is the *x-scan* operator. XML-QL and XQuery path expressions are related to regular expressions over the XML structure, and the first phase of executing a query consists of matching the desired path expressions against the XML data: this is the operation x-scan performs, while the document is still being read and parsed. X-scan combines and returns the results as a stream of binding tuples. The core of an x-scan operator is a set of dependent finite state machines that match the query's path expressions. Whenever an element open-tag is parsed, this represents a forward-traversal in a state machine; whenever the close-element is encountered, the state machine's previous state is restored. An accept state signifies that a binding to the current XML element should be produced. Finally, separate variable bindings must be combined to produce the appropriate binding tuples.

The x-scan operator is much like an index scan or sequential scan in a relational database — it is given a file and retrieves the desired components. However, in a network-based context, where data from one source might describe a set of inputs to be provided to another query, another operator is very useful: one that takes a series of data values, combines them to produce a new request for data from a different source, and then performs an x-scan operation over the answers to that request. We call this operation *indirect-scan*, and it performs very similarly to a dependent join in a relational database; it retrieves data, sends a request to a dependent source, and

Nbr.	Class	Input	Query
Q1	Extract	0.9MB	Suras in Quran with "The" in title (large trees)
Q2	Extract	1.5MB	Chapters after Chapter 7 in Book of Mormon (medium trees)
Q3	Extract	1.5MB	Sura titles with "Mormon" from Book of Mormon (single result)
Q4	Extract	39MB	DBLP papers that reference conferences (select by existential)
Q5	Extract	9MB	DBLP papers with URLs – across wide area from `nbci.com`
Q6	Integrate	200KB x 9MB	Nest DBLP papers in their conference entries (1 : n nesting)

Table 2: Queries for extraction (Q1-5) and combination (Q6) of XML

then combines the returned results with the current data. However, there are two key differences: (1) indirect-scan can request data from a different web source on each iteration, since it generates a complete request (and not simply an assignment of input parameter values) each time[2]; and (2) indirect-scan must perform an x-scan operation over the dependent data source before joining the results with those from the independent source.

The other novel XML operators in the Tukwila engine relate to creation of XML structural information. There are operators for enclosing data in new XML elements or attributes, and for outputting literal and computed values in the XML stream. We also include operators for nesting child subquery data within a parent query's output (a very common operation in XML query languages) and for separating attributes into hierarchical layers. All of these operators annotate the query processor's binding tuples with structural information that is not part of the data but controls the operation of the XML generator.

2.3 Performance

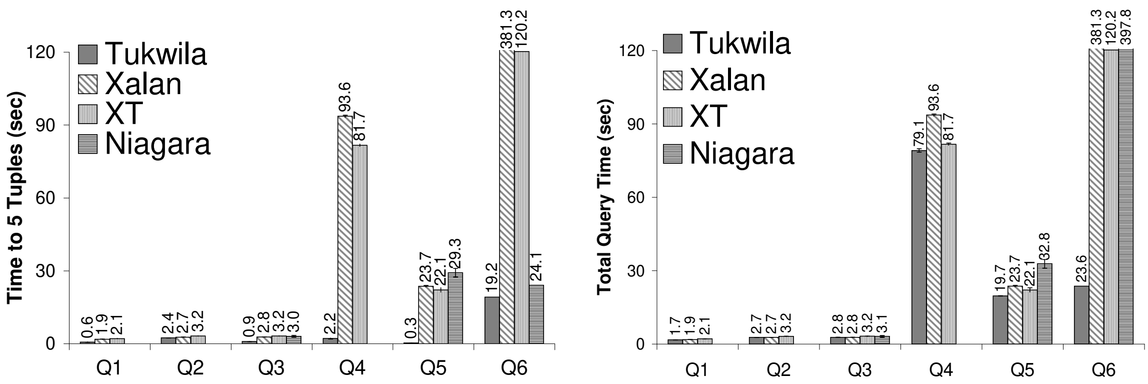
We have performed extensive experimental studies of Tukwila's performance and scalability, which can be found in [ILW00] and [IHW01]. Due to space limitations, in this paper we provide only a brief discussion of our findings, which are that the Tukwila architecture provides superior performance and scales well to the largest XML documents we could find on the Web.

Figure 2 compares the performance of Tukwila with that of the Niagara query processor[3] and with two XSLT query engines, XT and Xalan. We experimented with a range of queries, including selection-oriented document queries and integration queries combining data between multiple sources. The graphs show the running times for the queries of Table 2. The results demonstrate that Tukwila produces initial answers (2a) significantly earlier than the other query processors, largely due to its streaming input model that allows for pipelining. Moreover, Tukwila's overall query completion times (2b) were also faster, particularly if the query combined multiple input documents or restructured them.

Additional experiments have shown that the XSLT engines and the Niagara system do not scale to documents larger than about 25% of available memory, when they fail because the document's memory representation fills available RAM and they do not have mechanisms for overflow to disk. In contrast, the Tukwila engine only showed a small performance loss when querying the largest complex well-formed XML document we could find on the web, at 160MB, when given only 20MB of memory. For these results and others, please see the cited papers.

[2] This allows for greater expressiveness in queries, as one data source can describe a list of alternate sources to be combined in a single query.

[3] We used the version available at www.cs.wisc.edu/niagara during December, 2000.



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

Figure 2: Experimental comparison of XML queries shows that Tukwila has equally good total running time (and better time to first tuples) for simple extraction queries over small documents (Q1-Q3), and first tuples emerge significantly faster for larger documents both on the local area (Q4) and Internet (Q5). Tukwila performance is dramatically better than other systems when combining and restructuring XML data (Q6).

3 Conclusions and Future Directions

The query execution model of the Tukwila data integration system seems very well-suited to our target domain: potentially large, queryable XML data sources. The pipelined execution model and adaptive query processing operators produce good performance even under varying network conditions, and the approach also scales well.

We believe that we have found a good execution model for XML query processing, and now the next major challenge lies in effective *optimization* of XML queries. In our context, this is an extremely challenging problem: data sources may not have statistics, and in fact statistics and cost models for XML are not yet well-developed even for XML repositories. Worse, XML queries may include operations whose costs are difficult to model, such as recursive function calls, conditional queries, and references to external files.

It is our belief that the only real solution to this problem lies in further increasing the adaptive behavior of the query processing system, and in allowing the query processor to frequently re-optimize the query as it executes and gains increasing knowledge of costs and data characteristics. Our original Tukwila system provided mechanisms to re-optimize at query materialization points; later work by the authors of the Telegraph project [AH00] demonstrated that adjustments to a query plan can sometimes be made in the middle of a tuple stream, and their eddy operators performed flow-based tuple routing. We believe that these ideas must be carried even further to support efficient XML query processing: query execution and re-optimization should occur on a nearly continual basis, allowing the system to constantly adjust the query plan to varying conditions, to infer data patterns and possible optimizations, and even to share work between concurrent queries or to distribute a query across multiple nodes — while preserving prior work. It is this problem of integrated, feedback-directed query evaluation upon which we are now focusing.

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. Continuous query optimization. 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.

[FMS01] Mary F. Fernandez, Atsuyuki Morishima, and Dan Suciu. Efficient evaluation of XML middle-ware queries. In *SIGMOD '01*, May 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.

[GM93] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE '93*, pages 209–218, 1993.

[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.

[HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in Starburst. In *SIGMOD '89*, pages 377–388, 1989.

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

[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.

[IHW01] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. An XML query engine for network-bound data. Submitted for publication. Available at data.cs.washington.edu/integration/tukwila/xmlengine.pdf., 2001.

[ILW00] Zachary G. Ives, Alon Y. Levy, and Daniel S. Weld. Efficient evaluation of regular path expressions over streaming XML data. Technical Report UW-CSE-2000-05-02, University of Washington, May 2000.

[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.

[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.

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

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

[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.

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

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

The Niagara Internet Query System

Jeffrey Naughton, David DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, Rushan Chen

Abstract

Recently, there has been a great deal of research into XML query languages to enable the execution of database-style queries over XML files. However, merely being an XML query-processing engine does not render a system suitable for querying the Internet. A useful system must provide mechanisms to (a) find the XML files that are relevant to a given query, and (b) deal with remote data sources that either provide unpredictable data access and transfer rates, or are infinite streams, or both. The Niagara Internet Query System was designed from the bottom-up to provide these mechanisms. In this article we describe the overall Niagara architecture, and how Niagara finds relevant XML documents by using a collaboration between the Niagara XML-QL query processor and the Niagara "text-in-context" XML search engine. The Niagara Internet Query System is public domain software that can be found at http://www-db.cs.wisc.edu/niagara/.

1 Introduction

One of the most exciting opportunities presented by the emergence of XML is the ability to query XML data over the Internet. In our view, a query system for web-accessible Internet data should have the following characteristics. First, the query itself should not have to specify the XML files that should be consulted for its answer. This flexibility is a departure from the way current database systems work; in SQL terminology, it amounts to supporting a "FROM *" construct in the query language, and ideally would allow a user to pose a query, and get an answer if the query is answerable from any combination of XML files anywhere in the Internet. Secondly, a useful query system cannot assume that all the streams of data feeding its operators progress at the same speed, or even that all of the data streams feeding its operators will terminate. Once again, this is a departure from the way conventional DBMS operate. The Niagara Internet Query System is designed to have these characteristics. In this paper we describe how Niagara supports queries that do not explicitly name source XML files; support for network resident and streaming data is described elsewhere [STD+00].

The remainder of this paper is organized as follows. Section 2 gives a brief overview of XML and XML-QL and presents the overall top-level architecture of the Niagara Internet Query System. Section 3 describes the text-in-context Niagara Search Engine and the SEQL language, while Section 4 describes how XML-QL queries are evaluated. We conclude in Section 5.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 XML, XML-QL, and Top-Level Architecture of Niagara

2.1 XML and XML-QL

```
<book>
   <title> Java Programming </title>
   <price> 40 </price>
   <author id = "gosling">
         <name>
              <firstname> James </firstname>
              <lastname> Gosling </lastname>
         </name>
   </author>
</book>
```

Figure 1: An Example XML Document

```
WHERE <book>
              <title> Java Programming </title>
              <author>
                 <lastname> $l </lastname>
              </>
         </> IN *
CONSTRUCT <lastname> $l </lastname>
```

Figure 2: An Example XML-QL Query

Extensible Markup Language (XML) is a hierarchical data format for information representation and exchange in the World Wide Web. An XML document consists of nested element structures, starting with a root element. Element data can be in the form of attributes or sub-elements. Figure 1 shows an XML document that contains information about a book. In this example, there is a book element that has three sub-elements — title, price and author. The author element has an id attribute with value "gosling" and is further nested to provide name information. Much more information about XML and related standards can be found on the W3C Web Site, http://www.w3c.org.

There are many semi-structured query languages that can be used to query XML documents, including XML-QL [DFF+99], Lorel [AQM+97], UnQL [BDH+96] and XQL [R98], and what appears to be the emerging standard, XQuery [CFR+01]. Each of these query languages has a notion of path expressions for navigating the nested structure of XML. In Niagara, we initially chose XML-QL as the query language, although we are currently switching to XQuery. Figure 2 shows an XML-QL query to determine the last name of the author of a book having the title "Java Programming." The query is executed over the XML documents specified in the "IN" construct and the last names thus selected are nested under a <lastname> element. As mentioned in the introduction, one of the design goals of the Niagara query system is to allow the user the flexibility to query the web without having to explicitly specify each individual XML file to be queried. We thus extend XML-QL to support the "IN *" construct, whereby the query is (conceptually) executed over all the XML files present in the World Wide Web.

2.2 Top-Level Architecture of the Niagara Query System

Figure 3 illustrates the main components of the Niagara Internet Query System. Users craft XML-QL queries using a graphical user interface and send them to the query engine for execution. The connection manager in the query engine accepts queries for execution and is also responsible for maintaining sessions with clients. Each query received by the connection manager is parsed and optimized.

A crucial step in the optimization process is reducing the number of XML files to be consulted in order to produce the result of the query (especially in view of the "IN *" construct). Niagara accomplishes this reduction by extracting a search engine query from the XML-QL query. This search engine query is sent for execution to the search engine, which responds with a list of potentially relevant URLs. During execution, data from the relevant input sources is asynchronously fetched from the Internet by the data manager (if it is not already cached), and the results of execution are streamed to the user as they become available. Users can request partial

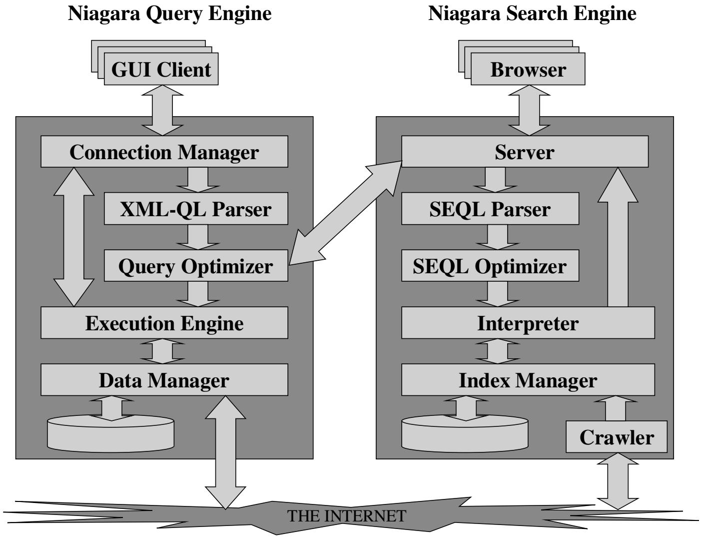


Figure 3: Niagara System Architecture

Figure 4: Query result screen shot.

results at any time during the execution. In this case, the execution engine returns the "result so far" while at the same time processing new input data coming off the Internet.

The Niagara search engine, in addition to its role in answering user XML-QL queries, can also be used as a stand-alone search engine for XML documents over the World Wide Web. The query interface to the search engine, in either case, is SEQL (Search Engine Query Language). SEQL queries are parsed and optimized in the search engine and the search engine interpreter executes the optimized execution plan. The results of the query execution are returned to the query engine or user. The inverted indices used for efficient SEQL execution are built and updated by the index manager. The index manager receives information about new and updated XML files from a crawler.

The GUI is a Java application that can also be run as an applet in a web browser. In addition to providing a simple graphical user interface to generate XML-QL, it is also capable of handling multiple concurrently executing XML-QL or SEQL queries. Both the query engine and search engine are also implemented in Java and are structured as multi-threaded servers. Since the query engine and search engine are individual servers, they run as separate processes (potentially on different machines). The next two sections describe the working of the search engine and the query engine in more detail.

3 The Niagara "Text-in-Context" Search Engine

A novel feature of the Niagara Internet Query System is that users do not need to specify source XML files in their queries. Rather, it is the responsibility of the system itself to examine the query, and from the query to determine the set of XML files that could possibly contribute to an answer to the query. Niagara does this through the use of its search engine, the Niagara "Text-in-Context" Search Engine.

When using an existing Internet search engine, the most common query is "find all the documents that contain these keywords." It is possible to construct more advanced searches based upon such properties as proximity and simple Boolean combinations of keywords. However, it is impossible to query based upon the role of the keyword in a document, because that information is not even available in the document itself. XML changes all this.

As a simple, admittedly contrived example, consider searching for all documents that have information about departures of a ship named "Montreal." One could go to one of the existing search engines and ask for the URLs of all documents that contain the string "Montreal," with predictable results — the query will return thousands of documents, most of which have nothing to do with the ship named "Montreal." Using the Niagara text-in-context search engine, one can instead ask for "all documents that contain departure elements that contain shipname elements that contain the string 'Montreal'." It should now be clear what we mean by "text-in-context" — rather than just searching for words in documents, we search for containment relationships between elements and other elements (e.g., "departure element contains shipname element") and also containment relationships between elements and strings (e.g., "shipname element contains the string 'Montreal'").

The preceding paragraph is overly simplistic — a user looking for departures of ships named Montreal with a traditional search engine would be unlikely to search only on "Montreal," they would be far more likely to search for "Montreal, ship, departure." This will yield a far more focussed search than just giving the keyword "Montreal." It is an open question how this search would fare when compared to the XML structural search. The structural search is more precise, but the value of this precision can only be determined empirically as we gain experience with the engine. Since in our experiments the structural approach is not significantly slower with respect to query evaluation, we have decided to use it.

3.1 SEQL

In this section we describe Search Engine Query Language (SEQL), the language executed by the search engine, briefly describe how the search engine evaluates SEQL, and how the XML-QL engine and the search engine interact. This process is described in more detail in [NDM+00].

SEQL is a simple language designed to specify patterns that can be matched by XML files. The output of a SEQL query is the list of URLs of the files that match the query. An atomic SEQL query is a word or an XML element tag. Such a query returns the URLs of the XML files containing the word or XML element tag, respectively. Complex SEQL queries can be built from the atomic SEQL queries using the binary operators "**contains**," "**containedin**" and "**is**" In addition, SEQL supports a proximity operator and a numeric comparison operator.

SEQL also supports the standard Boolean connectives "**and**," "**or**," and "**except**," which represent the intersection, union and difference of the results of the simple SEQL queries, respectively. Finally, SEQL supports a special construct "**conformsto**", which is used to restrict the result to only the URLs of those XML files that are declared to conform to a given DTD.

3.2 Evaluating SEQL

We now turn our attention to the efficient evaluation of SEQL queries in the Niagara search engine. As mentioned in Section 2.2, the search engine has two logical parts, the crawler and the SEQL execution engine.

The crawler locates XML documents in the web. Since at the time of writing this paper there are relatively few XML files on the web, to evaluate the system we manually built a local collection of XML files, and then used this crawler to crawl the local subtree and "find" these files. The crawler passes URLs of XML files to the search engine to be indexed.

The indices used by the search engine are variants of inverted lists used for information retrieval. There are three categories of inverted lists used by the search engine, which are element lists, word lists and DTD lists. The search engine maintains one element list for every unique XML element name encountered in the crawled XML files. Each entry (or "posting") of the element list has the form (fileId, beginId, endId) where fileId identifies a file containing the XML element, beginId specifies the beginning position of the element's begin tag in that file, and endId specifies the ending position of the element's end tag in the same file.

The search engine also maintains one word list for each unique text word encountered in the crawled XML files. Each posting in a word list for a given word is of the form (fileId, position) where fileId identifies a file containing the word and position indicates the position of the word in that file. Finally, the search engine maintains one DTD list for each unique DTD that the crawled XML files conform to. Each posting in a DTD list for a given DTD is of the form (fileId), where fileId identifies a file that conforms to that DTD. All three types of lists are maintained sorted by fileId to enable the efficient execution of SEQL queries.

4 Generating and Evaluating XML-QL Queries

In this section we describe how users pose queries using the Niagara GUI interface, and how those queries are evaluated by the query engine.

4.1 Extracting SEQL from XML-QL

The Niagara query engine sends SEQL queries to the Niagara search engine to determine the XML files over which to run an XML-QL query. To do so, the XML-QL query engine extracts a SEQL query (or queries) from the original XML-QL query. The XML-QL query engine extracts SEQL during query optimization.

The SEQL extraction process does not extract all possible constraints from the query; rather it uses heuristics to avoid generating SEQL that would be likely to be extremely inefficient to evaluate. The goal of the generated SEQL is to produce a superset of the URLs that need to be consulted to evaluate the XML-QL. It would be optimal if the "superset" were exactly the set actually required, but for efficiency of SEQL evaluation we do not always achieve this goal. Evaluating tradeoffs between the cost of the SEQL query and the precision with which it returns useful URLs is an interesting direction for future work.

4.2 A User Interface for XML Querying

In a classical relational database management query-builder interface, the basic approach is to start with the database schema. Clearly something analogous is needed for querying XML (no user is going to type in XML-QL!), but if a query is being posed over "all the XML files on the Internet" where is the schema?

In our system, we have taken the simple approach that this schema information is derived from document type descriptors (DTDs). Both the XML-QL engine and the text-in-context search engine have graphical user interfaces. To build a query, a user starts by selecting a set of DTDs to work with. After these DTDs have been selected, the GUI displays element names and users can do standard "point and click" or "drag and drop" query building over these DTDs. Once the query has been specified, and the user clicks on "submit query," an XML-QL query is generated (in the case of the XML-QL engine) or a Search Engine Query Language Query is generated (in the case of the text-in-context search engine.)

Note that the resulting query will not run solely over documents conforming to the DTDs selected by the user (unless the user specifies that this is desired by including a "conformsto" clause). Rather, the DTDs are used to generate a candidate set of tags over which to query. Any document that can match the query over these tags will be used in answering the query, whether or not it conforms to the DTD.

Clearly this is only a first step to building a good user interface. What is needed is a higher level mapping from user concepts to XML element vocabularies. We regard this area as important for future research. It is our hope that this sort of mapping will be done at a higher level than the query engine, in a layer that "understands" user-level concepts and can map them to schema information stored as DTDs or XML Schemas.

4.3 A Simple Query

For clarity and brevity of exposition, we present a very simple query. Consider the DTD in Figure 5 that describes XML documents representing movies. The elements are self-explanatory, with the exception of W4F_DOC, which is an element added by the wrapper that converted this data to XML [SA99].

```
<?xml encoding="ISO-8859-1"?>
<!ELEMENT W4F_DOC (Movie)>
<!ELEMENT Movie
(Title,Year,Directed_By,Genres,Cast)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Year (#PCDATA)>
<!ELEMENT Directed_By (Director)*>
<!ELEMENT Director (#PCDATA)>
<!ELEMENT Genres (Genre)*>
<!ELEMENT Genre (#PCDATA)>
<!ELEMENT Cast (Actor)*>
<!ELEMENT Actor
        (FirstName,LastName)>
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
```

Figure 5: Example Movie DTD.

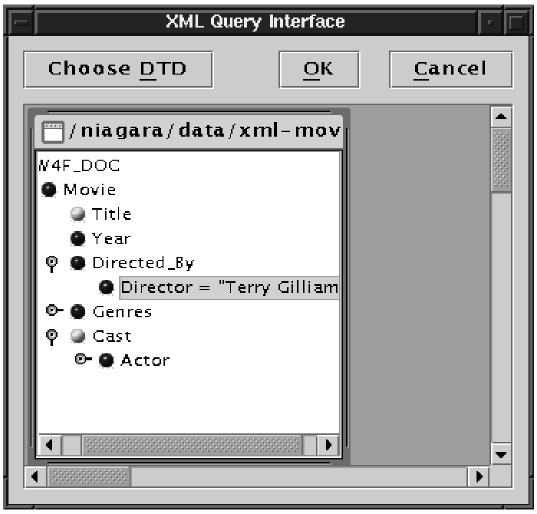


Figure 6: Niagara Query Interface Example.

Figure 6 shows a screen shot of the Niagara GUI specifying the query “retrieve the Movie title and the Cast of movies directed by Terry Gilliam” over the DTD in Figure 5. The XML-QL query that is generated in response to the query specified in the GUI is shown in Figure 7. Figure 8 shows the SEQL query the query engine extracts from this XML-QL query. In response to the SEQL query, the search engine returns three URLs, which have been filtered from over 600 XML documents that conform to the “movies” DTD. These URLs are passed to the query engine, which evaluates the XML-QL query, giving the result shown in Figure 4.

```
WHERE
<W4F_DOC>
 <Movie>
  <Title>$v68</>
  <Directed_By>
   <Director>$v71</>
  </>
  <Cast>$v74</>
 </>
</>
IN "*" conform_to
 "http://www.cs.wisc.edu/niagara/data/
   xml-movies/movies.dtd",
 $v71 = "Terry Gilliam"
CONSTRUCT
<result>
 <Title>$v68</>
 <Cast>$v74</>
</>
```

Figure 7: Generated XML-QL.

```
(W4F_DOC CONTAINS
     (Movie CONTAINS
        ((Directed_By
         CONTAINS (Director
          IS "Terry Gilliam"))
              AND
              (Title AND Cast)
         )
     )
)
conformsto
  "http://www.cs.wisc.edu/niagara/data/
    xml-movies/movies.dtd"
```

Figure 8: SEQL Extracted from Terry Gilliam query.

5 Conclusion

The Niagara Internet Query System is designed to enable users to pose XML queries over the Internet. It differs from traditional database systems in (a) how it decides which files to use as input, (b) how it handles input sources that have unpredictable performance or may be infinite streams or both, and (c) in its support for large numbers of triggers. In this paper we have focussed on the interaction between the search engine and the query engine; other aspects of the system are described elsewhere [CDT+00, STD+00].

The Niagara project is on-going; much more information about the system is available from the project homepage, http://www.cs.wisc.edu/niagara.

Acknowledgements

Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908 and by NSF Awards CDA-9623632 and ITR 0086002.

References

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, “The Lorel Query Language for Semistructured Data,” International Journal on Digital Libraries, 1(1), pp. 68-88, April 1997.

[BDH+96] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, “A Query Language and Optimization Techniques for Unstructured Data,” Proceedings of the 1996 ACM SIGMOD Conference, Montreal, Canada, June 1996.

[CFR+01] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, M. Stefanescu, “XQuery: A Query Language for XML.” W3C Working Draft, available at http://www.w3c.org/TR/xquery/.

[CDT+00] J. Chen, D. J. DeWitt, F. Tian, Y. Wang, “NiagaraCQ: A Scalable Continuous Query System for Internet Databases”, Proceedings of the 2000 ACM SIGMOD Conference, Dallas, TX, May 2000.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, “XML-QL: A Query Language for XML”, Proceedings of the Eighth International World Wide Web Conference, Toronto, May 1999.

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

[R98] Jonathan Robie, ”The Design of XQL”, Texcel Research, http://www.texcel.no/whitepapers/xql-design.html, November 1998.

[SA99] Sahuguet, Arnaud and Fabien Azavant. Web Ecology, “Recycling HTML pages as XML documents using W4F”, Proceedings of the 1999 WebDB Workshop, June 1999.

[STD+00] J. Shanmugasundaram, K. Tufte, D. DeWitt, D. Maier, J. Naughton, “Architecting a Network Query Engine for Producing Partial Results”, Lecture Notes in Computer Science, Vol. 1997, Springer-Verlag Publishers, 2001. A short version of this paper appeared in the proceedings of WebDB 2000 and is also available from http://www.cs.wisc.edu/ niagara/papers/partialResultsPerformance.pdf.

Aggregation and Accumulation of XML Data*

Kristin Tufte[1,2], David Maier[2]
[1]University of Wisconsin-Madison, [2]Oregon Graduate Institute
tufte, maier@cse.ogi.edu

Abstract

XML is rapidly becoming a standard for data exchange on the Internet. While XML's document management roots have led many to focus on querying and processing of large documents, we believe much XML data will be in the form of streams. One can envision streams of XML data flowing throughout the Internet: a stream of stock quotes or minute-by-minute updates on positions of a fleet of vehicles - one XML fragment per vehicle report. In this paper, we propose a new operation, Merge, which provides the capability to create aggregates over streams of data and the ability to take XML documents from different inputs and piece them together to create a new XML document. The Merge operation effectively handles highly-nested, semi-structured data and was designed to be used in an environment where there are long-running queries and stream-based data sources. We describe a flexible mechanism, called a Merge Template, which we have developed to specify how to merge two XML documents.

1 Introduction

XML is fundamentally changing the nature of data on the Internet. A prominent feature of this change is the appearance of data streams. Data no longer lives in local files that can be read and re-read at the discretion of the DBMS. In fact, some XML data exists only in streams on the Internet which must be processed as they arrive. Examples of this kind of data include stock quotes and B2B and B2C messages. In this paper, we present a new operation, Merge, which provides a step towards effectively processing XML data streams.

The Merge operation combines two XML documents into a result document. One application of Merge is accumulating and aggregating data from an XML stream. Consider a stream of stock quotes from the NYSE expressed in XML; each quote contains a ticker symbol and trade price. Merge can be used to create and maintain an XML document containing today's current, high, low and average prices for all stocks traded on the NYSE. This evolving document is called an Accumulator and it can be queried like any XML document stored in a DBMS. Figure 1 shows a graphical representation of such an Accumulator. Figure 2 shows a stock quote for IBM that has just arrived on the NYSE stock ticker (stream) and Figure 3 shows the result when this quote is Merged into the Accumulator. Note the functions that the Merge operation performs. The first time a ticker symbol (i.e. IBM) appears in the quote stream, Merge inserts a Stock element for IBM into the Accumulator.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908 and by NSF ITR award IIS0086002.

This new element contains IBM's ticker symbol and elements for IBM's low, high, average and current stock prices, the values of which are all set to the initial trade price. As additional trades occur on IBM stock, the updates arrive as small XML documents and are merged into the existing element for IBM: The current stock price is replaced with the trade price and the aggregate values low, high and average are updated appropriately.

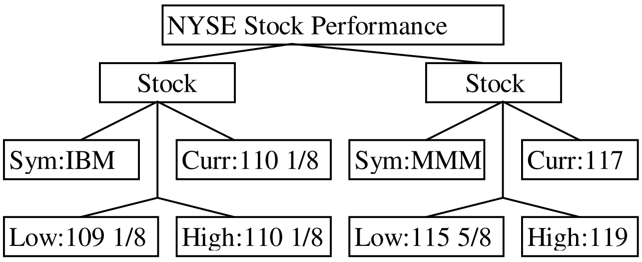


Figure 1: NYSE Stock Quote Accumulator

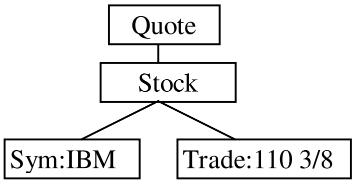


Figure 2: Stock Quote for IBM

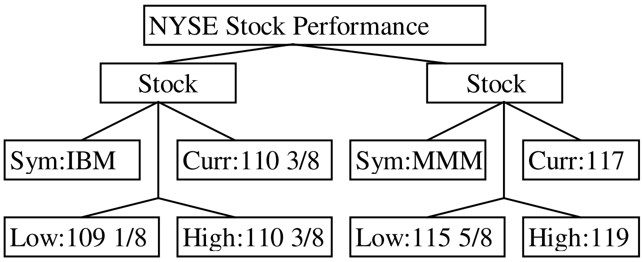


Figure 3: Accumulator after Merge of Stock Quote

A new mechanism, called a Merge Template, specifies what actions should occur when combining two XML documents. Merge Templates are very flexible and provide for operations including content update, aggregation and sub-tree replacement. A Merge Template also incorporates a simple form of equality predicates, called Match Templates, to allow a user to specify when two elements represent the same entity.

In addition to aggregating streaming XML, Merge is useful for replication, cache maintenance, and incremental query processing. Merge is specifically designed to work on irregular and incomplete data and can function without a DTD. A single Merge Template can handle a range of input structures. Finally, a merge-based accumulate operator has been implemented in Version 1.0 of the Niagara Query Engine [8]. (See the Niagara paper in this volume for an overview of the system.)

The rest of this paper is organized as follows. Section 2 describes the Merge operation and Merge and Match Templates, Section 3 discusses related work, and Section 4 concludes and discusses future work.

2 The Merge Operation and Merge Templates

Formally, the Merge operation merges two XML documents into a single result document. In this section, we use a simple example to explain the operation of Merge and to illustrate the structure and function of Merge Templates. In short, a Merge Template specifies the structure of the merge result, predicates to apply and operations to perform, and can itself be represented in XML. For simplicity, we ignore XML attributes in this section; extending Merge Templates to handle attributes is straightforward.

2.1 Merge Templates

Consider an XML document that consists of information about classes offered in the Computer Science Department at the University of Wisconsin. Figure 4 shows a graphical representation of such a document. Each box in the figure represents an XML element and the tag names and content (PCDATA) of the elements are written as TagName:Content. Figure 5 shows a diagram of a small XML document representing the addition of a student to a class. The Merge operator can be used to merge the Add Record (Figure 5) into the Class List (Figure 4) to produce the document shown in Figure 6. The Merge process is driven by a Merge Template.

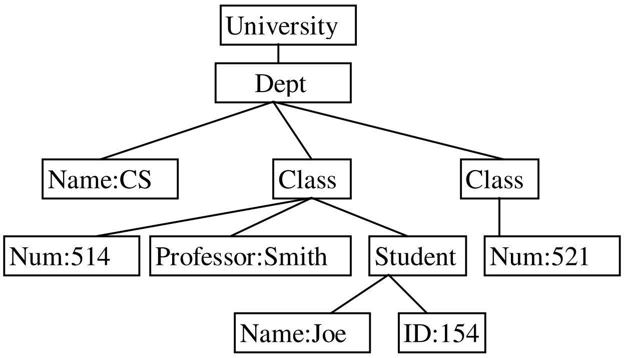


Figure 4: Class List

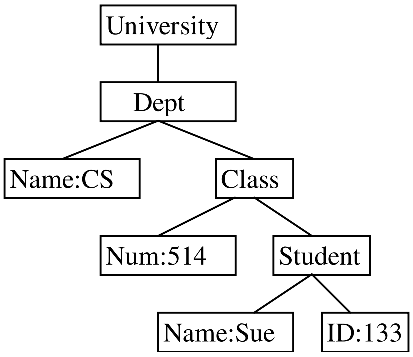


Figure 5: Add Record

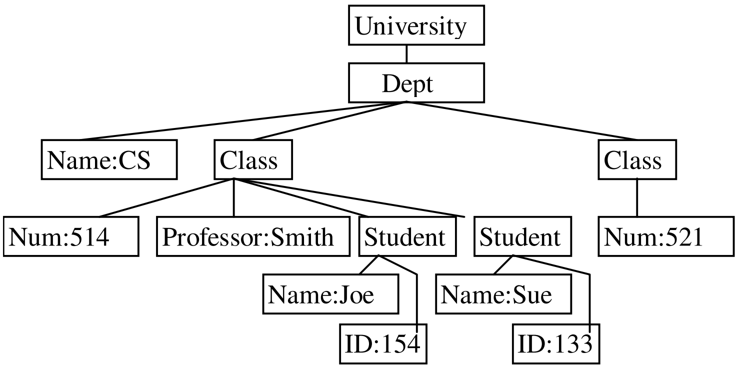


Figure 6: Merged Document

Merge Templates are a general mechanism for specifying how to merge two XML documents together. Figure 7 shows a logical representation of the Merge Template for the Class List example; Figure 8 shows the Merge Template in XML. Conceptually, a Merge Template is a tree of Element Merge Templates. Each Element Merge Template (EMT) specifies how to produce a (possibly empty) set of elements for the result document. For example, the Merge Template for the Class List example contains a Class EMT that specifies how to create the Class elements in the result from the Class elements in the input documents. Thus, each EMT is logically associated with a set of elements in the result document. In Figure 7, the name of each EMT is the name of the result element(s) to be produced using that EMT. In general, this is not a requirement.

Each EMT consists of three parts: an optional Match Template, a required Content Combine Function and a possibly empty set of sub-EMTs. The Match Template is a predicate that indicates when two elements should be deemed equivalent. The Content Combine Function specifies how to create the content of the result element(s) associated with an EMT. The set of sub-EMTs specifies how to produce the result elements' children. Consider the EMT for Class elements. The Match Template, represented by "Match on Value of Num," indicates that that two Class elements match if they have the same number. The Content Combine Function, represented as "Empty," indicates that Class elements in the result should not have any content. Finally, the Class EMT contains a sub-EMT for each type of sub-element (Num, Professor, Student) that a Class may have. The structure of the Merge Template parallels the structure of the result document of the merge. Notice how the "tree" of gray EMTs in Figure 7 is similar to the structure of the Merged Document in Figure 6.

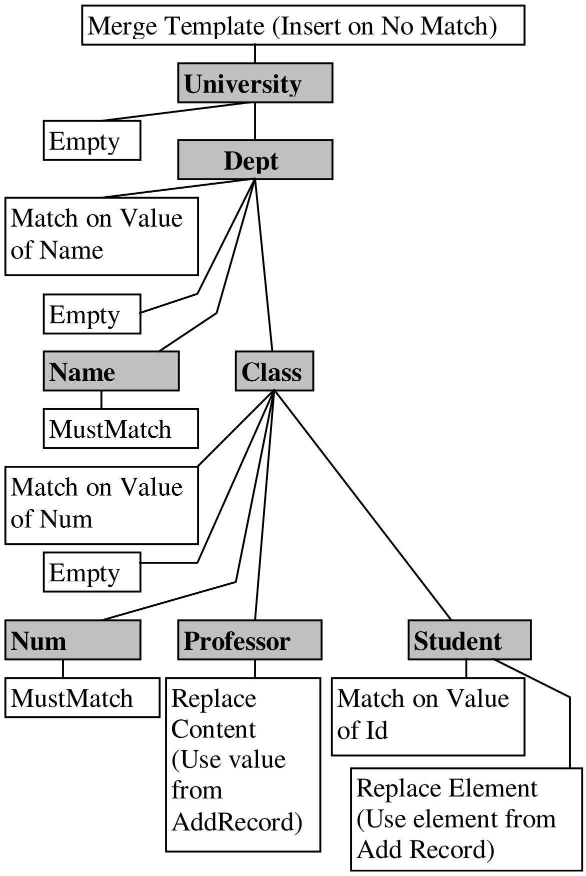


Figure 7: Logical Representation of Merge Template

```
<?xml version="1.0"?>
<!DOCTYPE DocMergeTemplate
                          MergeTemplate.dtd>
<DocMergeTemplate MergeType="outer">
  <EltMergeTemplate, Name = "University">
   <Empty/>

     <EltMergeTemplate, Name = "Dept">
     <MatchTemplate>...</>
     <Empty/>

        <EltMergeTemplate, Name = "Name">
          <MustMatch/> </>

        <EltMergeTemplate, Name = "Class">
        <MatchTemplate>...</>
        <Empty/>

          <EltMergeTemplate, Name = "Num">
          <MustMatch/> </>

          <EltMergeTemplate, Name = "Professor">
          <ShallowContent Function =  "replace"
                        DominantSide ="right" /> </>

          <EltMergeTemplate, Name = "Student">
          <DeepReplace DominantSide="right"/> </>
</> </> </>
```

Figure 8: Class List Merge Template in XML

A Merge Template incorporates a form of equality predicates, called Match Templates. Figure 9 shows the Match Template for the Class element. In general, a Match Template contains a list of paths. Each path can reference an element or sub-element value or attribute value. Two elements match if the values referenced by all paths are equal. Merge allows matching on values found through traversal of IDREFs, but does not support merging through IDREFs. We next discuss the Merge operation.

```
<MatchTemplate>
   <MatchNode>
      <Path> Num </>
      <Content ValueType = "String"/>
   </>
</>
```

Figure 9: Match Template

2.2 Merge Operation

The Merge operation works by traversing the Merge Template and the input documents (Class List and Add Record in our example) in parallel and combining the input documents on an element-by-element basis. For each EMT in the Merge Template, a possibly empty set of elements is produced for the result document.

The process begins by using the root EMT in the Merge Template to produce the content of the root element of the result document, by combining the content of the root elements of the input documents. The sub-elements of the result root element are produced using the sub-EMTs of the root EMT. For each EMT encountered, the Merge operation locates two sets of elements - one from each input document (the elements in each set are siblings). These sets are joined based on the Match Template in the EMT. The join is an inner, outer, left-outer or right-outer join as is specified by the Merge Type of the Merge Template and produces a relation containing pairs of elements. We currently require that this be a one-to-one relation, with the exception of the nulls generated by the outer joins. For each pair in the relation, an element is produced for the result document. The content of the result element is produced by combining the content of the two elements in the pair based on the Content Combine Function in the EMT. The possible combine functions include replace content, various aggregates and replace element. Merge Templates support a limited version of typing so mathematical functions such as aggregates can be calculated properly. The sub-elements for the result element are produced by scanning the sub-EMTs of the current EMT and processing them as just described. Very loosely, Merge is a series of nested joins - with potentially one join for each ”type” of element in the result document. This proliferation of joins may seem excessive; however, a join is executed only when an EMT produces a set of elements for the result.

Merge is designed to handle a range of structures in its inputs, so it can be used with streams of varied structure and content. For example, we have discussed the Class List Merge Template (Figure 7) in the context of adding a student to a class, but it is not specific to that function. A single instance of a Merge operator using the Class List Merge Template could merge documents representing any of the following events into a Class List: the addition of a student to a class, a change of professor for a class, or the addition of a new class. Note that for each type of input document, different actions must occur. For a professor change, the appropriate Class element must be updated and for a class addition, a new Class element must be inserted into the Class List. This type of functionality is difficult to effect in a relational system, where the type and order of operations must be decided in advance. The flexibility of the Merge operation is possible in part because the Class List Merge Template fully specifies how to merge all of these updates into the Class List document and in part because the Merge operation does not require the input data to exactly match the structure of the Merge Template.

3 Related Work

The Niagara [8] and Lore [6] projects have built database systems designed for querying XML data. The Tukwila [7] project has developed an adaptive query processing system and is currently focusing on processing XML streams. Our work provides a new operation, Merge, which is not currently available in any of these systems. YAT uses XML for data integration [3]. Merge differs from data integration in that we focus solely on aggregating data over XML streams and do not address issues such mediation and query reformulation. Some XML processing systems [3, 8] flatten the data and then process it using relational-style operators, constructing the XML result once the processing is complete. In contrast, Merge works on XML data in a tree-based fashion without flattening it first. Flattening may not be tractable when the data is highly irregular.

The Accumulator described in the Introduction, which maintained current, high, low and average prices for a set of stocks, can be seen as a view over the incoming stream of stock quote data. As data (stock quotes) arrive on the stream the ”view” (Accumulator) is updated. In this way, our work is similar to work on Materialized View Updates. View updates have been studied extensively in the context of relational database systems [4]. Research has been done on view updates for semi-structured data; however, much of that work supposes that the updates are identified by OIDs [1, 12]. Updates to XML documents may be XML documents, Updategrams [11], or update operations as proposed by Tatarninov *et al.* [10]. In any case, OIDs are unlikely to be available.

Buneman *et al.* have proposed two operators: Deep Union and Deep Update, which are similar in nature to Merge [2]. Deep Union and Deep Update are used to combine and update edge-labeled trees that are similar, but not identical to, XML trees. The WHAX view update system uses Deep Union as a mechanism for updating views [5]. Deep Union and Deep Update resemble the Union and Join operators of the Verso Algebra [9], which uses a data model based on Nested Relations. These data models require that elements (tree nodes in the models used by Deep Union and WHAX and tuples in the Verso model) have keys. Neither work has the concept of an explicit external Merge specification such as our Merge Template. In these systems, the functions that occur when two documents are unioned or joined is solely defined by the operator definition and the structure of the document. The difference is similar to that between natural join and traditional explicit join.

4 Conclusion and Future Work

We have described a new operation, Merge, that pieces together two XML documents in a flexible, general fashion. Repeated application of Merge can be used to accumulate and aggregate data from an incoming stream. Two key features of Merge are that its actions depend directly on the structure of the input data and that it can easily and effectively handle irregular and incomplete input. Documents are not required to have DTDs to be merged and IDREFs are supported. A Merge-based Accumulator has been implemented in the Niagara Internet Query System and performance tests are in process. In addition, we are in the process of creating a formal specification for the Merge operation based on lattice theory.

In the future, we expect to add support for deletion and ordered documents to Merge. Currently, we do not have a mechanism for updating the Accumulator to reflect, for example, that a student has dropped a class. In addition, the concept of order needs to be integrated into the Merge definition so that Merge can support XML documents that convey semantic meaning by the order of their elements.

References

[1] S. Abiteboul, J. McHugh, M. Rys,V. Vassalos, J. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. *Proceedings of the 1998 VLDB Conference*, New York, NY, August 1998.

[2] P. Buneman, A. Deutsch, and W.C. Tan. A deterministic model for semi-structured data. *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel, Jan. 1999.

[3] V. Christophides, S. Cluet, and J. Simeon. On Wrapping Query Languages and Efficient XML Integration. In *Proceedings of the 2000 ACM SIGMOD Conference*, Dallas, TX, May 2000.

[4] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):3-18, June 1995.

[5] H. Liefke and S. B. Davidson. View Maintenance for Hierarchical Semistructured Data. *2nd International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2000)*, London-Greenwich, United Kingdom, September 2000.

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

[7] Z. Ives, A. Levy, D. Weld, D. Florescu, M. Friedman. Adaptive Query Processing for Internet Applications. *IEEE Data Engineering Bulletin*, 23(2):19-26, June 2000.

[8] J. Naughton, D. DeWitt, D. Maier, *et al*. The Niagara Internet Query System. http://www.cs.wisc.edu/niagara.

[9] M. Scholl, et. al. VERSO: A Database Machine Based on Nested Relations. *Nested Relations and Complex Objects, Papers from the Workshop "Theory and Applications of Nested Relations and Complex Objects"*, Darmstadt, Germany, April 1987. Lecture Notes in Computer Science, Vol. 361, Springer 1989.

[10] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *Proceedings of the 2001 ACM-SIGMOD Conference*, Santa Barbara, CA, May 2001.

[11] http://msdn.microsoft.com/xml/general/updategrams.asp

[12] Y. Zhuge, H. Garcia-Molina. Graph Structured Views and Their Incremental Maintenance. In *Proceedings of the 14th IEEE Conference on Data Engineering (ICDE '98)*, Orlando, February 1998.

A dynamic warehouse for XML Data of the Web

Lucie Xyleme*

Abstract

Xyleme is a dynamic warehouse for XML data of the Web supporting query evaluation, change control and data integration. We briefly present our motivations, the general architecture and some aspects of Xyleme. The project we describe here was completed at the end of 2000. A prototype has been implemented. This prototype is now being turned into a product by a start-up company also called Xyleme [14].

Xyleme: a complex tissue of wood cells, functions in conduction and storage ...

1 Introduction et motivation

The current development of the Web and the generalization of XML technology [13] provides a major opportunity which can radically change the face of the Web. We have developed a prototype of a *dynamic warehouse for XML data of the Web*, namely Xyleme. In the present paper, we briefly present our motivations, Xyleme's general architecture and its main aspects.

The Web is huge and keeps growing at a healthy pace. Most of the data is unstructured, consisting of text (essentially HTML) and images. Some is structured, mostly stored in relational databases. All this data constitutes the largest body of information accessible to any individual in the history of humanity. The lack of structure and the distribution of data seriously limit access to the information. To provide functionalities beyond full-text searching ala Alta Vista or Google, specific applications are being built that require heavy investments. A major evolution is occurring that will dramatically simplify the task of developing such applications:

XML is coming!

XML is a standard to exchange semistructured data [1] over the Web. It is widely adopted. It is clear that progressively more and more Web data will be in XML form and that DTDs, or XML Schemas will be available to describe it, see, e.g., Biztalk and Oasis. Communities (scientific, business, others) are defining their own DTDs to provide for standard representations of data in specific areas.

Given this, we decided to study and build a *dynamic Warehouse for massive volume of XML data* from the Web. The number of accessible XML pages is marginal for the moment, less that 1% of the number of HTML

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Lucie Xyleme is a nickname for a large group of people who worked on the project: S. Abiteboul, V. Aguilera, S. Ailleret, B. Amann, F. Arambarri, S. Cluet, G. Cobena, G. Corona, G. Ferran, A. Galland, M. Hascoet, C-C. Kanne, B. Koechlin, D. Le Niniven, A. Marian, L. Mignet, G. Moerkotte, B. Nguyen, M. Preda, M-C. Rousset, M. Sebag, J-P. Sirot, P. Veltri, D. Vodislav, F. Watez and T. Westmann.

pages. However, we believe that this number will grow very rapidly soon. Thus we want a warehouse capable of storing huge volumes of pages and a major issue is *scalability*. The problems we address are typical warehousing problems such as change control or data integration. They acquire in the Web context a truly distinct flavor. More precisely, the research directions we studied and that we briefly discuss here are as follows:

- *efficient storage* for huge quantities of XML data (hundreds of millions of pages).
- *query processing* with indexing at the element level for such a heavy load of pages.
- *data acquisition strategies* to build the repository and keep it up-to-date.
- *change control* with services such as query subscription.
- *semantic data integration* to free users from having to deal with many specific DTDs when expressing queries.

These goals are technically very challenging. To handle such a volume of data (terabytes) and workload, we rely on parallelism. More precisely, we are distributing data and processing on a local network of Linux PCs.

Certain functionalities of Xyleme are standard and are or will soon be found in many systems. For instance, from a query viewpoint, search engines will certainly take advantage of XML structure to provide (like Xyleme) attribute-equal-value search and some will certainly support XML query languages. Other features such as change monitoring are already provided (to some extent) by systems, e.g., MindIt [10]. What is specific to Xyleme? Perhaps, the main distinguishing feature is that *Xyleme is based on warehousing*. The advantages of such an approach may best be illustrated by a couple of examples. First, consider query processing. Certain queries requiring joins over pages distributed over the Web are simply not conceivable in the current Internet context, for performance reasons. They become feasible with a warehousing approach. To see another example, consider monitoring. It is trivial to notify users when some pages of interest change. Now, if the pages are warehoused, one can support more precise alerts, e.g., when some particular elements change. More examples of new services supported by Xyleme will be encountered in the paper.

The Xyleme Project functioned as an open, loosely coupled network of researchers. Serge Abiteboul, Sophie Cluet (INRIA researchers) and F. Bancilhon (CEO of Ariso) initiated the project. Xyleme was rapidly joined by researchers from the database groups from INRIA-Rocquencourt (http://www-rocq.inria.fr/verso/), Mannheim U. (http://pi3.informatik.uni-mannheim.de/ mitarbeiter.html), the CNAM-Paris (http://sikkim.cnam.fr/), the IASI Team of University of Paris-Orsay (http://- www.lri.fr/iasi/introduction.fr.html), and a few individuals from other places. The project we describe here was completed at the end of 2000. A prototype has been implemented. This prototype is now being turned into a product by a start-up company also called Xyleme [14].

The paper is a short survey. We provide references to some longer reports describing specific aspects of the work. References for related works may be found there. The paper is organized as follows. Section 2 introduces the architecture. The repository is discussed in Section 3. Section 4 deals with query processing and indexing. Data acquisition is considered in Section 5 and change control in Section 6. The topic of Section 7 is data integration.

2 The Architecture

In this section, we present the architecture of the system.

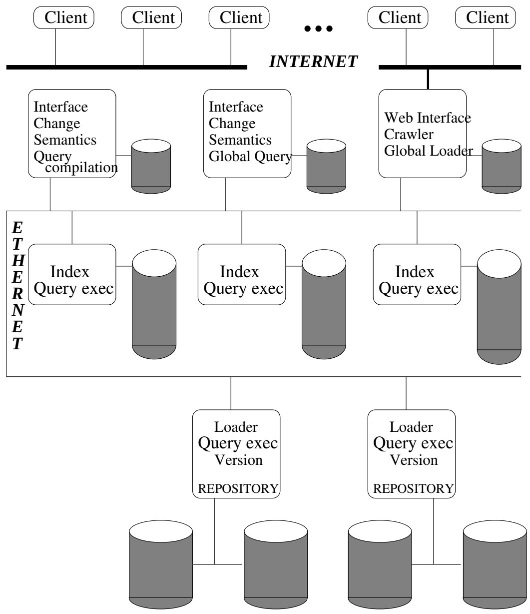


Figure 1: Architecture

The system is functionally organized in four levels: (i) physical level (the Natix repository tailored for tree-data and an index manager); (ii) logical level (data acquisition and query processing); (iii) application level (change management and semantic data integration); (iv) interface level (interface to the web and interface with Xyleme clients).

As already mentioned, the system runs on a network of Linux PCs. All modules are written in C++ and communicate using Corba. The Xyleme clients run on standard Web browsers using Java applets. As shown in Figure 1, we (logically) distinguish between several kinds of machines. This aspect will be ignored here.

3 The Repository

In this section, we discuss the repository. More details may be found in [6].

Xyleme requires the use of an efficient, update-able storage of XML data. This is an excellent match for Natix developed at the U. of Mannheim. Typically, two approaches have been used for storing such data: (i) store XML pages as byte streams or (ii) store the data/DOM tree in a conventional DBMS. The first presents the disadvantages that it privileges a certain granularity for data access and requires parsing to obtain the structure. The second artificially creates lots of tuples/objects for even medium-size documents and this together with the effect of poor clustering and the additional layers of a DBMS tend to slow down processing. Natix uses a *hybrid approach*. Data are stored as trees until a certain depth where byte streams are used. Furthermore, some minimum structure is used in the "flat" part to facilitate access with flexible levels of granularity, Also, Natix offers very flexible means of choosing the border between the object and flat worlds.

Natix is built, in a standard manner, on top of a record manager with fixed-size pages and segments (sequences of pages). Standard functionalities for record management (e.g., buffering) or fast access (e.g., indexes)

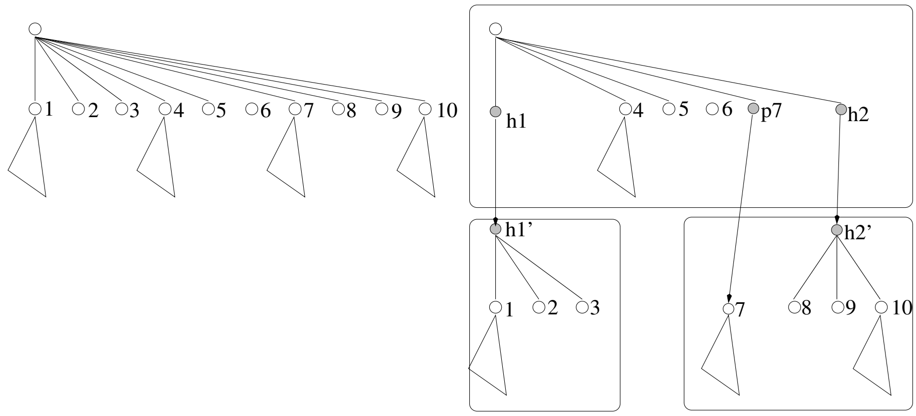


Figure 2: Distributing nodes on records

are provided by Natix. One interesting aspect of the system is the splitting of XML trees into pages. An example is shown in Figure 2. Observe the use of “proxy” nodes (p-nodes) and “aggregate” nodes (h-nodes). One can control the behavior of the split by specifying at the DTD level, which splits are not allowed, are forced, or are left to the responsibility of the system.

The performances of Xyleme heavily depend on the efficiency of the repository. We believe that, by taking advantage of the particularities of the data, Natix does offer appropriate performances.

4 Query Processing

In this section, we consider processing of static queries. Change queries are studied in Section 6. A more detailed presentation of this work can be found in [2].

The evaluation of structured queries over loosely structured data such as XML has been extensively studied during the last ten years. The techniques differ slightly depending on the system that is used to store the data, object-oriented, dedicated or relational. However, none of the existing approaches scale to the Web. We are considering here billions of documents (Google recently indexed more than one billion HTML documents) and millions of queries per day. So, query processing acquires in Xyleme a truly distinct flavor.

As is often the case, we represent queries using an algebra. The novelty of our approach lies in (i) a distribution pattern that scales to the Web (see Figure 1) (ii) the efficient implementation of a complex algebraic operator, named *PatternScan*, that captures so-called tree queries, i.e., queries that filter collections of documents according to some tree pattern and extract information from the selected documents. We capture the added expressive power provided by database-like XML query languages using standard operators (e.g., *Join*, *Map*, etc.).

The *Patternscan* operation is implemented using an index mechanism, named *XyIndex*, that is an extension of the full text index (FTI) technology. Whereas a standard FTI returns the documents in which a word occurs, a *XyIndex* adds annotations to position each occurrence of a word within a document relatively to the other words. Also, a *XyIndex* respects an order relation that allows to use efficient and pipelinable merge algorithms to combine the various sets of words occurrences corresponding to a tree query.

As will be explained in Section 7, Xyleme partitions documents into clusters of documents, each corresponding to a domain of interest (e.g., *tourism*, *finance*, etc.). Also, Xyleme provides *a view mechanism*, that enables the user to query a single structure summarizing a cluster rather than the many heterogeneous DTDs it contains.

A view in a relational system builds one relation by combining information coming from others. Similarly, a view in Xyleme combines several clusters into a virtual one. However, whereas the tuples of a relation share a common type, a cluster is a collection of highly heterogeneous documents. Thus, a view cannot be defined by one relatively simple join query between two or three collections but rather by the union of many queries over different sub-clusters (Section 7 explains how views are (semi-)automatically generated by the system). As a matter of fact, there is no *a priori* limit to the size of a view definition in Xyleme since it depends on the number of DTDs (that may be implicit in documents that are only well-formed) that one can find on the Web. Obviously, techniques that are used to maintain and query relational views have to be seriously re-considered to fit Xyleme's needs.

The view information in Xyleme is decomposed into two parts corresponding to the standard query processing steps: compilation and execution. The compilation part of a view is replicated on each upper-level machine (see Figure 1) and is mainly used to understand which index and repository machines are concerned by a query [4]. The size of this information is usually small, depending on the number of domains and the size of the view schema. The execution part of a view is distributed over the index machines (see Figure 1). Each stores only the view information that concerns its indexed cluster (or sub-cluster). This data is used to translate tree-queries just before they are evaluated by XyIndexes and is order of magnitude smaller that the indexes.

5 Data Acquisition

In this section, we consider the issue of data acquisition. More details may be found in [9].

We *crawl* the Web in search of XML data. We also need to refresh pages to keep the repository up to date. We implemented a crawler that can read millions of pages per day. Several crawlers can be used simultaneously to share the acquisition work. Note that we store only XML pages. For HTML, we read them to discover new links. A critical issue is the strategy to decide which document to read/refresh next. First, a Xyleme administrator is able to specify some general acquisition policies, e.g., whether it is more urgent to explore new branches of the Web or to refresh the documents we already have. The system then cycles around all pages it knows of (already read or not) deciding for each page whether this page has to be refreshed/read during this particular cycle. The decision for each page is based on the *minimization of a global cost function under some constraint*. The constraint is the average number of pages that Xyleme is willing to read per time period. The cost function is defined as the dissatisfaction of users being presented stale data. More precisely, it is based on criteria such as:

- *Subscription and Publication:* A customer may specify a desired refresh frequency for some particular documents. Also, some query subscription may request the regular refreshing of some pages. Finally, the owner of a document may let Xyleme know when a change occurs and request a refresh of the warehouse for this document.
- *Temporal information* such as last-time-read or change rate: These allow to estimate the number of updates that were missed for a particular page.
- *Page importance*: We want to read in priority documents that, we believe, are important and refresh them more often than others. To estimate page importance, we use the structure of the Web and the intuition that important pages carry their importance to pages they point to. This leads to a fixpoint computation.

Note that each page that is known by Xyleme is eventually read/refreshed (no eternal starvation) unless there is an explicit decision by the administrator to leave a portion of the Web out, e.g., pages below a certain importance threshold. Finally, observe that page importance can also be used to rank pages in query results as done, e.g., in Google.

6 Change Control

In this section, we discuss change control. Users are often not only interested in the current values of documents but also in their evolution. They want to see changes as information that can be consulted or queried. Indeed, change detection and notification is becoming an important part of Web services.

The first aspect of change control we considered is a subscription mechanism [11]. At the time we load or refresh a document, we detect whether it verifies patterns specified by some subscriptions. Examples of changes that can be monitored include, for instance, a modification of a particular document, the discovery of new documents with a given DTD. We also support monitoring conditions at the element level, e.g., monitoring of the newly discovered documents with an element tagged *address* containing the word "Marseille". The monitoring system we developed allows to monitor, on a single PC, the loading of millions of documents per day, with millions of subscriptions.

The second aspect of change control we considered is a versioning mechanism [8]. For each page, Xyleme gets snapshots of the page. It is possible to version some pages. To represent versions, we opted for a *change-centric* approach as opposed to a data-centric one that is more natural for database versioning. We use a representation based on *deltas* in the style of Hull's deltas. Given two consecutive versions of a page, we compute the delta using a very efficient *diff* algorithm we developed, tailored to our needs [5]. As often done in physical logs, we use *completed deltas* that also contain additional information so the deltas can be reversed and composed. We also assign persistent identifiers that we call Xyleme IDs (XIDs) to the elements of the documents. These identifiers are essential to represent and control changes. One of the roles of the *diff* is to assign XIDs to elements in consecutive versions of a document. An example of our logical representation of versions is shown in Figure 3. In the figure, XIDs are shown in the XML tree although in reality, they do not exist inside the document but only in an auxiliary structure, the XID-Map. Note that a delta is also an XML document and thus, that it can be queried like any XML document.

We use deltas for versioning documents. We believe they can also be very useful to version results of continuous queries, i.e., queries that are evaluated regularly as in the Conquer [7] or Niagara [3] projects. Deltas may be used for other services as well; see [8].

7 Semantic Data Integration

In this section, we address the issue of providing some form of semantic data integration in Xyleme. More details may be found in [12].

Queries are precise in Xyleme because, as opposed to keywords searches, they are formulated using the structure of the documents. In some areas, people are defining standard documents types or DTDs, but most companies publishing in XML often have their own. Thus, a modest question may involve hundreds of different types. Since, we cannot expect users to know them all, Xyleme provides *a view mechanism*, that enables the user to query a single structure, that summarizes a domain of interest, instead of many heterogeneous schemata.

Views can be defined by some database administrator. However, this would be a tedious and never ending task. Also, metadata languages such as RDF may be used by the designer of the DTD or by domain experts to provide extra knowledge of a particular DTD. We believe that this will become common in the long range.

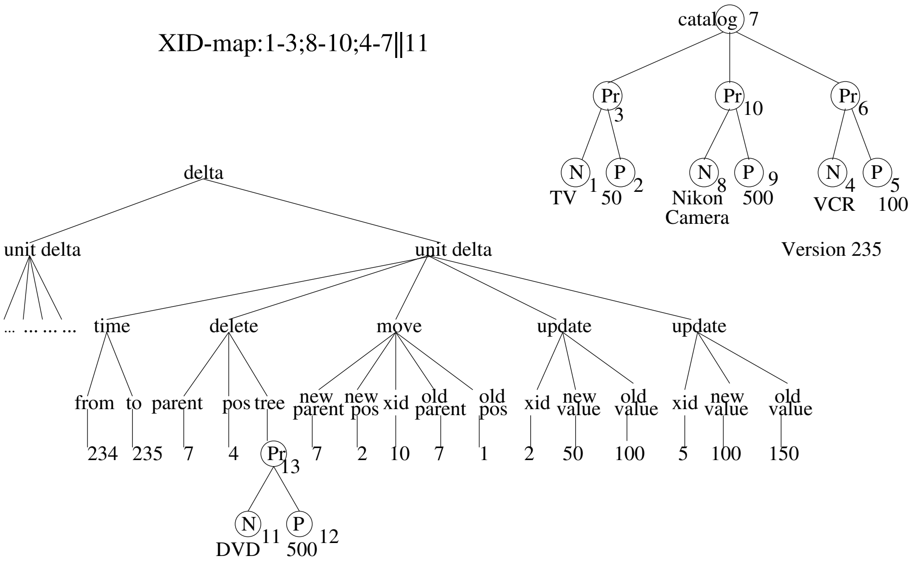


Figure 3: Storage of a versioned document

However, for now, the field is too young, standards are missing and such knowledge is not available. Thus, we studied how to automatically extract it using natural language and machine learning techniques.

The first task is to classify DTDs into domains (e.g., *tourism*, *finance*) based on a statistical analysis of the similarities between the words found in different DTDs and/or documents. Similarity is defined based on ontologies or thesauri such as Wordnet.

Once an *abstract DTD* has been defined to structure a particular domain, the next task is to generate the semantic connections between elements in the abstract DTD and elements in concrete ones. Each element is identified by a path from the root of its DTD (e.g., *hotel/address/city*). The problem is then to map paths to paths. Obviously, all tags along a path (i) are not always words and (ii) do not have the same importance. Concerning (i), we use standard natural language techniques (e.g., to recognize abbreviations or composite words) and a thesaurus. As for (ii), we mainly rely on human interaction. Notably, the abstract DTD designer can indicate to the mappings generator that some tag is not particularly meaningful in a path, or, on the contrary essential. For instance, if we consider *hotel/address/city*, we can imagine that $hotel$ is meaningful but $address$ not that much. In this fashion, we will be able to map *auberge/town* to *hotel/address/city*, but not *company/address/city*.

Conclusion Working in the Xyleme Project was a fascinating experience. Many problems were encountered that still require to be further investigated. The work is of course continuing in the various research groups and in the Xyleme start-up.

Acknowledgments: We want to thank those who discussed many aspects of this work with us or those, in particular within INRIA's management, who supported us.

References

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web.* Morgan Kaufmann, California, 2000.

[2] Vincent Aguilera, Sophie Cluet, Pierangelo Veltri, and Fanny Watez. Querying XML documents in xyleme. *ACM SIGIR Workshop on XML and information retrieval*, 2000.

[3] Jianjun Chen, David DeWitt, Fend Tian, and Yuan Wang. Niagaracq: A scalable continous query system for the internet databases. *ACM SIGMOD*, page 379, 2000.

[4] Sophie Cluet, Pierangelo Veltri, and Dan Vodislav. Views in a large scale xml repository. In *VLDB'01*, 2001.

[5] Grgory Cobna, Serge Abiteboul, and Amlie Marian. Detecting changes in XML documents. Technical report, Verso Group, INRIA-Rocquencourt, 2001. `http://www-rocq.inria.fr/verso/`.

[6] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of XML data. Technical Report 8/99, University of Mannheim, 1999. `http://pi3.informatik.uni-mannheim.de/`.

[7] Ling Liu, Calton Pu, Wei Tang, and Wei Han. Conquer: A continual query system for update monitoring in the www. *International Journal of Computer Systems, Science and Engineering*, 2000.

[8] Amlie Marian, Serge Abiteboul, Grgory Cobna, and Laurent Mignet. Change-centric management of versions in an XML warehouse, September 2001. VLDB'01.

[9] Laurent Mignet, Mihai Preda, Serge Abiteboul, Sbastien Ailleret, Bernd Amann, and Amlie Marian. Acquiring XML pages for a webhouse, October 2000. BDA'00.

[10] Mind-it web page. http://mindit.netmind.com/.

[11] Benjamin Nguyen, Serge Abiteboul, Gregory Cobena, and Mihai Preda. Monitoring XML data on the web. In *ACM Sigmod*, 2001.

[12] C. Reynaud, J.P. Sirot, and D. Vodislav. Semantic integration of xml heterogeneous data sources. In *International Database Engineering and Applications Symposium, IDEAS*, 2001.

[13] World Wide Web Consortium pages. http://www.w3.org/.

[14] Xyleme Home Page. http://www.xyleme.com/.

An XML Programming Language for Web Service Specification and Composition

Daniela Florescu
Propel
danaf@propel.com

Donald Kossmann
TU Munich
kossmann@in.tum.de

1 Introduction

XML is the lingua franca for data exchange on the Internet. Among its many possible uses, XML is ideal for publishing documents on Web sites, for storing catalogs in electronic market places, and for exchanging data between business processes. Even though some data sources will probably continue to use relational and object-relational database systems as a primary form of storage, we expect that most data sources will eventually provide XML access for their published data.

Software vendors and standard bodies, like the W3C consortium, have been very active in providing tools (XML parsers) and standardized languages (XSLT, XPointer, XPath, XQuery, etc.) for XML. So far, however, no imperative programming language has been proposed that is specifically tailored for building XML applications and Web services.

As of today most Web services are built using classic programming languages, such as Java and Visual Basic, and some kind of SQL-based RDBMS (Oracle, DB2), a mixture of paradigms inherently implying a fair number of logically irrelevant but costly and error prone intermediate manipulations. An XML Web application built on such technologies will have to deal with difficulties such as:

1. XML-Java mismatch: XML data must be converted into Java (or any other language of the sorts) internal representation as objects before it can be processed by the Java program. Likewise, Java objects must be converted back into XML data at the end of processing.
2. Java-Database mismatch: Java objects must be marshalled back and forth through JDBC-like interfaces to access and update the RDBMS, the infamous "database impedance mismatch" that triggered the development of object databases technology in the recent past[CM84].

While most people seem to have resigned themselves to using interfaces like ODBC and JDBC, the additional XML-Java "stuttering step" [Lam] might have a good chance to finally turn people's minds. It is not unusual that eighty percent or more of Web applications code is indeed due to data marshalling. Moreover, a great deal of the plumbing and hand optimizations are craftly hard-coded into the Java part of current applications. As an example, data cacheing in proxy objects, the ubiquitous astute way of reducing round-trips

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

to the database system, is typically implemented at the application-level, making it very hard to extend these applications and guarantee their correct semantics whenever the inevitable DB schema evolution comes along.

Language implementors and database manufacturers are scrambling themselves to increase their products with “XML extensions” and to introduce automatic treatment for those chores whom programmers are currently dealing with manually. Indeed there are significant efforts both on the Java side, from database and third-party vendors in these directions (see Section 4.) However, we believe that the type systems of XML, Java, and relational database systems are simply too different and ultimately incompatible for productively building large scale applications that span accross the three different paradigmes.

The alternative that we are pursuing with this paper is to introduce XL (short for XML Language), a new high-level programming language for Web services. Five foundamental principles underlie the design of this new language:

1. XL should support a unique data model and type system: the standard XML one [Que].
2. XL should be expressive enough to describe the logic of most Web services. For most applications it should be possible to specify a Web service entirely in XL, without any need for Java or SQL.
3. XL should not just be complete with respect to Web service specification, but also comfortable to use. Hence, it should provide special constructs for important Web services programming patterns (e.g., logging and retry of actions).
4. with the help of XL, programmers should concentrate on the logic of their application and not on implementation issues as performance aspects, data formats, or specific Internet and database protocols. Such aspects have to be dealt with in an automatic way and not via hand-coded and hard-coded solutions.
5. XL must be compliant with all W3C standards and it must gracefully co-exist with the current (Java-based) Web services and infrastructure.

In this paper we will outline the requirements for such an XML programming language and present a sketch of an initial design. Our design is based on familiar concepts of imperative programming languages as well as on those of parallel programming [Hoa85] and workflow management [Moh00b]. The long-term goal of this paper is to stimulate a discussion about the foundamental concepts and syntax of XL, and eventually to submit a proposal to the W3C with the help of as many practitioners and researchers as possible. Towards the end of this paper we will also indicate possible directions for future research to provide efficient implementations.

2 Requirements for an XML Programming Language

In this section, we will describe a list of specific requirements for XL. Some of the requirements are derived from the global architecture where a Web service specified using this language should be integrated.

1. **Compliance with the W3C standards.** XL must be fully compliant with the XML W3C standards such as XML Schema [Sch], XQuery [Que], XPath [XPa], XSLT [XSL] or XML Protocol [Prob].
2. **Business processes, Web conversations.** The language must provide constructs to implement business processes and, more generaly, it must support *conversations* between two or more Web services.
3. **Service composition.** XL should allow the construction of high-level services out of the composition of more basic services. It should also be possible to compose new services out of services not necessarely written in XL. It should be possible, for instance, to integrate Web services written in Java in a transparent and seamless way. The web services participating to a *conversation* have to be loosely coupled (i.e. changes in the implementation of one such service should not affect the other services that invoke or are being invoked by it).

4. **Message-based programming.** A web service implemented in XL should communicate with other web services via messages. Services are invoked via messages and results are also returned via messages.

5. **Location independent invocation.** In general web services should be uniquely identified using URIs. The invocation of a Web service should use the respective URI and be independent from the location where its code is stored or executed.

6. **Platform independence, code mobility.** The language we propose must be platform independent. It must be possible to run programs virtually on any machine that runs an interpreter for the language — independent of the operating system or the database system used.

7. **Unique XML-based data model and type system** The data manipulated should be modeled by the standard XML data model and type system [Que]. No other data models and type systems should be allowed.

8. **Optional strong typing.** Types for data components (e.g., variables) are optional. However, if a variable is associated with a type, strong typing must be enforced (i.e., type checking can be done at compile-time). Special constructs must be provided such that programmers can enforce properties of components dynamically if no specific type is statically associated with a component.

9. **Logical/physical data independence.** Programmers should be aware only of the logical structure of the XML data (i.e. the XML Schema) and they need not be aware of the physical representation of the data (e.g. DOM trees, SAX events, XML files or database tuples).

10. **High-level programming.** The language we propose must be high level and use declarative constructs whenever possible. The language must also support high-level exception handling and other special constructs to easily implement more complex services like logging, data lineage, time-triggered actions, etc.

11. **Imperative programming.** The language must provide standard imperative constructs like sequencing and iteration. However, we believe that given the particular nature of the applications we are targeting such imperative constructs will be seldom used.

12. **Transactions.** The language must provide constructs allowing programmers to specify sequences of actions to be executed in an isolated and atomic way; i.e., transaction.

13. **Universal naming for each component.** Each component (program, conversation, message, transaction, exception) must have a URI for data lineage and information traceability.

14. **Authentication, authorization and Security.** It must be possible to implement discretionary access control and, thus, restrict the use of a service.

15. **Automatic optimization.** The language design should enable and encourage automatic run-time optimizations and should discourage or even disallow low level hard-coded optimizations.

16. **Protocol transparency.** Accesses to a database and invocation of remote web services must be transparent. The protocols used (e.g., JDBC or HTTP) must be hidden in the implementation of the language.

3 Basic Concepts of XL

In the following, we will describe the concepts of an initial design of XL. These concepts provide a core functionality — more functionality (and syntactic sugar) are probably necessary to achieve wide acceptance. The semantics of certain concepts (in particular, transactions) definitively need further discussion and review — in this paper we will only discuss basic options for what we think are the most critical concepts.

XL Programs

Each XL program represents a Web service. In order to compose Web services, XL programs can send messages to other XL programs or any other Web service (e.g., written in Java) that understands SOAP messages (see the subsection on "Service Invocation" below). An XL program defines implicitly two variables: *$input* and *$output*. The variable *$input* is bound with the XML message sent to the XL program at the time the program is invoked. *$input* can also be bound by an XML Form [oWF], if the Web service is invoked by an interactive (human) interface. The variable *$output* contains the results and is automatically sent back as an XML message to the caller or an agent of the caller. Furthermore, XL programs can throw exceptions which are also sent back as XML messages to the caller or its agent.

An XL program is composed of two parts: a header and a body. The body is described by a statement (different kinds of statements and their possible combinators are detailed in the following subsections.) The header contains meta-data of the service. In particular, the header contains (optional) XML schema types for the *$input* and *$output* of the XL program and for exceptions. Declaration of such types is optional; in other words, the *$input*, *$output*, and exceptions can be untyped just like any other XML document can be untyped. In addition, the header of an XL program defines the globally unique URI of the XL program; XL programs are invoked using their URI. Furthermore, other meta-data that is necessary in order to implement transactions (see the subsection on transactions below), for security (e.g., the public key of the Web service for encryption), versions of the XL program, and for automatic optimization of XL programs (e.g., side effects of the program) can be specified in the header of an XL program. A more complete description of such metadata will be given and continuously updated in a forthcoming technical report [FK01].

The proposed syntax for the header of an XL program is as follows (optional parts are represented in brackets, keywords are represented in capital letters):

```
PROGRAM uri [typeOfInput] -> [typeOfOutput]
  [THROWS  typeOfException1]
  [meta data]
```

Basic Statements

In this section we introduce some of the basic statements that can be used in the body of an XL program. (Statements used for specific services will be discussed in the following subsections.)

Assignments, Local Variables, and Expressions The simplest statement is the assignment of a local variable. The syntax is as follows:

```
  LET [type] variable := expression
```

Variables need not be declared before being used. However, the (XML schema) type of a variable can optionally be set as part of the first assignment to this variable. The scope of a variable is the whole XL program. Expressions can be any expression defined by the W3C XQuery proposal [Que]. XQuery is very powerful and a great deal of the power of XL is gained by simply leveraging the expressive power of XQuery. XQuery expressions include normal XML elements, function calls, arithmetic expressions (i.e., use of built-in operators), XPath, and more complex queries similar in nature with the SELECT-FROM-WHERE constructs of SQL. As in XPath and XQuery, we denote variables with a $ sign.

As an example for a simple assignment, consider the following statement:

```
   LET \$harry :=  <person>
                      <name>Harry Potter</name>
                      <hobby>Quidditch</hobby>
                      <hobby>Broomsticks</hobby>
                   </person>
```

As mentioned earlier, typing is optional, but it is strictly enforced if it is used.

Update Statements Unfortunately, XQuery does not yet provide expressions to manipulate XML data. There are plans to extend XQuery in this respect and once a recommendation has been released by the W3C, XL is going to adopt the syntax and semantics of these expressions. In the meantime, we will use the following statements to manipulate XML data: (1) *insert* to add elements; (2) *delete* to delete elements; (3) *replace* to adjust elements; (4) *rename* to rename element tags. For instance, the following statement gives Harry an age:

```
INSERT <age>14</age> INTO \$harry
```

Service Invocation

Probably the most relevant atomic statements in XL are those used for invoking other Web services; i.e., send a message to another Web service. Often, the other Web service will be another XL program, but in effect it can be any service that responds to SOAP messages [Proa] or, more specifically, to the XML Protocol which is currently under development by W3C working group [Prob]. In the following, we will define Web services as any service that has a URI and that is able to listen and respond to messages according to the (emerging) XML protocol standard. Web services are invoked independently of the specific way they are implemented.

We propose two ways to invoke a Web service as part of an XL program: synchronous and asynchronous. The syntax of a synchronous call is as follows:

```
expression --> uri  [--> variable]
```

The semantics is rather straightforward. A message with the value of *expression* is sent to the service identified by *uri*. The execution is halted until the called service finishes it execution and returns the entire result (also wrapped in an SOAP message). If a *variable* is given as part of the call, then the body of the message returned by the called service is copied into this variable. The exact semantics of synchronous calls in the presence of transactions, will be discussed in a later subsection on transactions.

The syntax of an asynchronous call is as follows:

```
expression ==> uri [==> uriOfHandler]
```

In this case the execution of the XL program will not block and the program will immediately continue executing the next statement after the message to the called service is sent. If the output of the called service (errors or any other reply message) needs to be processed, then the URI of the Web service in charge with this processing can be given as part of the call. The exact semantics of asynchronous calls in the presence of transactions will be discussed in a later subsection on transactions.

Currently, XL does not explicitly support business conversations. In order to implement business conversations, every step of a conversation needs to be implemented by a separate XL program and there need to be special XL programs that keep track of the context of the conversation and invoke the XL programs that carry out the individual steps. Alternatively, contextual information about the current conversation can be carried out in the XML messages themselves. Furthermore, there is no explicit support for multicasts yet. In order to send a message to multiple services, each service must be individually invoked by a separate message. We plan to extend XL along these lines as part of future work.

Statement Combinators

Obviously, the body of an XL program can contain more than one atomic statement. There are several ways to combine statements in order to generate other statements, as explained below. In the following “statement1 and “statement2 can refer to any atomic statement as the ones described in the previous sections or the any combination of statements.

1. **Sequence.** The typical way to combine statements is by using the “;” symbol, like in C++ or Java. Thus, the following means that “statement1” is executed before “statement2.”

   ```
   statement1; statement2
   ```

2. **Failure.** If “statement1 fails, execute “statement2.

```
statement1  ?  statement2
```

3. **Choice.** Execute either “statement1 or “statement2, but not both. Which one is executed is nondeterministic.

```
statement1  |  statement2
```

4. **Parallel execution.** Execute “statement1 and “statement2 in parallel. In other words, the order in which the individual statements are carried out is not specified.

```
statement1  ||  statement2
```

5. **Dataflow.** If there are data dependencies between “statement1 and “statement2 (e.g., “statement1 binds a variable that is used in “statement2), then execute the statement that depends on the other statement last. If there are no dependencies, then execute “statement1 and “statement2 in any order. If there is a cyclic dependency, then this combination of statements is illegal.

```
statement1  &  statement2
```

Access to Persistent Data

Similarly to Web services, any kind of persistent data that can be accessed by an XL program must have an associated URI. Examples for persistent data are tables of a relational database (viewed as XML), an XML document stored in the file system, a Web page provided by a Web server. In order to support access to persistent data, XL provides an *alias* statement that binds a local variable name to the persistent data referenced by a given URI. The syntax of this statement is as follows:

```
ALIAS  uri  AS  variable
```

If such a variable is involved in the rest of the program in an expression, then the persistent data will be queried. If the variable is on the left side of a LET statement, then a new value will be assigned to the persistent data (the old value of the persistent data will be lost). Furthermore, persistent data can be manipulated using the update statements described before (or the equivalent constructs once XQuery has been extended in this direction). For instance, if `HTTP://personDB.com/address` is the URI identifying the XML view of a table in a relational database containing contact information of wizzards, then the following sequence of statements will insert a new record into this database (note that wizzards use owls to send messages):

```
ALIAS  HTTP://personDB.com/addresses AS $contact;
INSERT <person> <name>Harry Potter</name>
                <owl>Hedwig</owl>
       <person> INTO $coordinates
```

Of course such updates will only work if the source of the persistent data (e.g., a Web server or a relational database) supports updates. If not, then the execution of such an INSERT statement will fail; i.e., an exception will be raised. Similarly, the execution of LET, DELETE, RENAME, and REPLACE statements can fail if the data source does not support updates. Web pages, for instance, will typically not be updatable, but, of course, Web pages can easily be queried.

An important question is how persistent data are created. To answer this question, XL provides special web services (part of a built-in Web services library) that can be invoked to create persistent data. Such a service takes the initial value or the expected Schema description of the persistent data as input XML message and returns the URI of the newly generated persistent data. Obviously, several such web services may exist, depending for example on the particular requested implementation for the persistent data (e.g. relational databases, XML files).

Our way to create and manage persistent data is very similar in spirit with the approaches proposed for persistent programming languages [AM95]. However, the novel Web services context imposes the reconsideration of a certain number of issues. For example, unlike most traditional persistent programming languages, we do not recommend for XL the concept of persistence by reachability. In other words, the newly created data persists even if all references to it are deleted, as we believe that automatic garbage collection is both not feasible and not desirable on the Internet.

Further Statements

As we previously mentioned, The XL language also contains additional statements necessary to implement basic services. We briefly list them here; the full syntax and semantics is described in detail in the upcoming technical report [FK01]. Those statements are: (1) exception handling (e.g. *raise, try-catch*); (2) retry the execution of failed statements for a certain period of time or for a certain number of times; (3) periodically invoke certain services; (4) (temporarily) halting the execution of a program for a certain time; (5) logging actions and the state of variables; (6) loops and conditions (e.g., *for, repeat, while, switch* statements); (7) assertions and invariants in order to dynamically check properties (e.g., types).

Transactions

One of the most important requirements for the composition of Web services is to support transactions; i.e., a sequence of statements that are carried out in an atomic and isolated way. In order to specify transactions in XL we propose the following syntax:

```
ATOMIC   statement   ENDATOMIC
```

Here, *statement* can be a sequence of statements or any other combination of statements. For each transaction a URI is allocated. The URI is passed along when a Web service is invoked as part of a transaction.

There has been a great deal of work since the seventies on distributed transactions and models for distributed transactions (see, e.g., [Moh00a]). All this work is relevant. In our initial design, we propose to enforce the full ACID paradigm for transactions. Enforcing an ACID paradigm involves the implementation of a two-phase commit protocol [BN96].

To implement a two-phase commit protocol, additional meta data for a Web service must be available. If a Web service A is invoked by an XL program B as part of a trasaction, the URIs of services that *prepare, prepareToCommit, commit,* and *compensate* the Web service A must exist. The purpose of those services is to take the URI of a transaction as input and carry out necessary actions in order to implement isolation and two-phase commit. If they do not exist, then the Web service A cannot be invoked as part of the transaction, and a compile time or at run-time exception is raised.

Likewise, transactional services must be supplied if persistent data is read or modified as part of a transaction. Obviously, special attention must be paid if a service is invoked asynchronously as part of a transaction; details of our solution are given in the upcoming technical report [FK01].

Finally, an important question concerns the semantics of nested *ATOMIC* statements. If an XL program A calls as part of a transaction T_1 an XL program B that itself defines a transaction T_2, then the *atomic - endatomic* statements of B are ignored; that is the statements of T_2 are carried out as part of the transaction T_1. In other words, nested *ATOMIC* statements are automatically flattened. An alternative would be to implement the original nested transaction model.

As mentioned at the beginning of this subsection, this is only an initial design. Ultimately, more flexibility will probably be needed. In some scenarios, for instance, two-phase commit is simply not feasible or not necessary. Also, other transaction models [Moh00a], specially designed for the composite, distributed Web services might be more useful. It is also conceivable that the W3C working group on XML protocol will recommend such a transaction model. Our purpose here is to give a starting point for further discussion and to provide a placeholder in our design.

Security

Currently, XL provides no constructs to authenticate the addressee of a message and to check whether the addressee is authorized to invoke the service. However, authentication and authorization can be implemented and integrated as specific Web services. For instance, an XL program that requires authentification would be invoked by sending it a signature as part of the *$input* message (e.g., in a special `<signature>` element). That XL program would then invoke a trusted Web service (provided by, e.g., VeriSign) in order to check the signature. In a future version of XL, we plan to provide special statements (such as those for logging) in order to help the developers of critical Web services that require authentification and authorization. For instance, such requirements could be declared in the header of an XL programs and the programmer could be relieved from the details of calling a service that checks signatures.

In order to protect messages from being trapped by third-parties, any implementation of XL will use standard encription mechanisms as proposed by the W3C XML Encryption working group [Enc]. Therefore, encryption will be automatic and XL programmers need not worry about the security of their messages.

4 Related Work

The development and composition of Web services (or e-services) is currently a very active area in both industry and academic research. Very good resources that address various aspects of this area are the recent W3C workshop on Web services [Con] and the latest issue of the IEEE Data Engineering Bulleting on e-services [deb01].

The main purpose of our work was to provide a clean basis for a new XML programming language for rapid developing of Web services. Such a language will obviously not be built from scratch but using the knowledge and technology advancements accumulated in the last 30 years.

In the industry, there have been a number of concrete proposals for new languages and frameworks related but not identical to our programming language proposal. Compaq has developed the WebL language [Web]; HP has developed the eFlow and eSpeak systems [CIJ$^+$00, eSp], IBM is working on a language called Web Service Flow Language [WSF], and Microsoft has recently released their BizTalk Server 2000 [Biz]. There is also a Business Process Model Initiative with the goal to implement cross-organization processes and workflows on the Internet [BPM]. The state of the art in the Java world is to support XML via so-called servlets that translate (XML and HTML) requests into Java classes and back [JAK]. At the time of this writing, Sun Microsystems just introduced the JXTA project on peer-to-peer computing to support distributed computing on the Internet [JXT]. Finally, the notion of a service composition is based on a solid theoretical background consisting on the calculus developed first by Hoare[Hoa85] and more recently by Cardelli [CD99].

However, none of those languages and frameworks are totally consistent with the current W3C standards, and we believe that this is a mandatory condition for the success of such a programming language. There a number of (de-facto) standards which are currently under development and which are important requisites in order to enable the open composition of Web services; in particular SOAP [Proa], XML protocol [Prob], UDDI [UDD], and WSDL [CCMW].

5 Summary

In this paper we sketched the requirements and presented an initial design of an XML programming language whoose purpose is to render the implementation and the composition of new and existing Web services as easy as possible.

Developers should not worry about details of Internet protocols, database systems, and the infrastructure. Developers should also not worry about hand-tuning their applications or about marshalling particular data formats, but instead, they should concentrate on the application logic.

Our short-term goal is to foster discussions that will eventually come to a consensus for a complete design of such a language. Open questions involve the semantics of transactions, support for comfortable conversations, and security aspects (e.g., authorization and encryption). However, given the current great interest on Web services in industry and in academic research projects, we expect discussions to open very quickly and to be carried into W3C working groups. A more complete design of XL is given in a technical report [FK01]. This report is constantly updated and it contains more elaborated examples that demonstrate the power of XL.

In the long run, the implementation of XL and of a global infrastructure for Web service composition will involve a large number of new research opportunities; e.g., code mobility, automatic caching and optimization.

References

[AM95] M. P. Atkinson and R. Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3), 1995.

[Biz] BizTalk.org. Biztalk initiative. http://www.biztalk.org/home/default.asp.

[BN96] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan-Kaufmann Publishers, 1996.

[BPM] BPMI.org. Business management initiative. http://www.bpmi.org/index.esp.

[CCMW] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[CD99] L. Cardelli and R. Davies. Service combinators for Web computing. In *IEEE Transactions on Software Engineering (TSE)*, 1999.

[CIJ+00] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan. eFlow: a platform for developing and managing composite e-services. Technical report, Hewlett Packard Software Technology Laboratory, 2000.

[CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 316–325. ACM, 1984.

[Con] W3C Consortium. Workshop on Web services. http://www.w3.org/2001/01/WSWS.

[deb01] Special issue on Infrastructure for Advanced E-services. *Data Engineering Bulletin*, 24(1), March 2001.

[Enc] XML Encryption. http://www.w3.org/encryption/2001/.

[eSp] eSpeak. The universal language of e-services. http://www.e-speak.hp.com/.

[FK01] D. Florescu and D. Kossmann. An XML Programming Language for Web Service Specification and Composion. Technical report, TU Munich, June 2001. To appear at http://www3.in.tum.de/public/mitarbeiter/kossmann.html

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[JAK] JAKARTA. The JAKARTA project. http://jakarta.apache.org/.

[JXT] JXTA. Project JXTA. http://www.jxta.org/.

[Lam] L. Lamport. The temporal logic of actions. Technical report, Digital Systems Research Center.

[Moh00a] C. Mohan. Transaction processing and distributed computing in the internet age. Technical report, 2000.

[Moh00b] C. Mohan. Workflow management in the internet age. Technical report, 2000.

[oWF] XForms: The Next Generation of Web Forms. http://www.w3.org/markup/forms/.

[Proa] Simple Object Access Protocol. http://www.w3.org/tr/soap/.

[Prob] XML Protocol. http://www.w3.org/2000/xp/.

[Que] XML Query. http://www.w3.org/xml/query.

[Sch] XML Schema. http://www.w3.org/xml/schema.

[UDD] UDDI.org. Universal description, discovery and integration of businesses for the web. http://www.uddi.org/.

[Web] WebL. Compaq's web language. http://www.research.compaq.com/SRC/WebL/.

[WSF] WSFL. Web services flow language. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[XPa] XML Path Language XPath. http://www.w3.org/tr/xpath.

[XSL] Extensible Stylesheet Language XSLT. http://www.w3.org/style/xsl/.

Call for Participation

International Conference on Scientific and Statistical Database Management

July 18-20, 2001

George W. Johnson Center, Fairfax, Campus
George Mason University

Sponsored by ***George Mason University***,
In Cooperation With:

- ***ACM SIGMOD,***
- ***IEEE TC on Data Engineering,***
- ***VLDB Endowment.***

The SSDBM 2001 Organizing Committee invites you to participate in the technical and social programs associated with scientific and statistical database management. The conference is single-track to encourage discussions and interactions.

Highlights of the conference include:

Keynote Address:"A Scaleable Infrastructure for Browsing, Processing, and Fusing Distributed Heterogeneous Earth Science Data," by Professor Joseph JáJá, *University of Maryland, College Park*

Invited Tutorial:"Clustering Algorithms and Validity Measures," M. Halkidi, Y. Batistakis, and M. Vazirgiannis, *Athens University of Economics and Business*

Panel Session: "The Science Data Highway - Where Are We Today?

Technical Session titles include: Statistical Databases; Index Structures for SSDBM; Query Processing, Optimization and Constraints; Clustering Techniques; System Architectures and Methods; Performance of Spatio-Temporal Data Structures; Data Mining and OLAP; and Posters and Demonstrations.

The Advance Program may be viewed at:
http://www.ssdbm.gmu.edu/advance_program.htm

We are planning a dinner cruise on the Potomac River with views of Washington, D.C.

Please visit our web site at: http://www.ssdbm.gmu.edu/ for details.
On the web site you can register for the conference via a secure web page, and obtain conference hotel information.

Should you require additional assistance, please contact Ms. Rickee Mahoney, GMU Events Management, 703-993-2122, or by e-mail at rhanks@gmu.edu

Advance Registration closes June 15, 2001!

We look forward to seeing you at SSDBM 2001 on George Mason University's Fairfax Campus.

Larry Kerschberg	Menas Kafatos
Program Chair	General Chair

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398