Evaluating XQuery in a full-XML Mediation architecture

Tuyet-Tram Dang-Ngoc* — **Georges Gardarin****

PRiSM Laboratory
University of Versailles,
45, avenue des Etats-Unis
75035 Versailles CEDEX
France
**Tuyet-Tram.Dang-Ngoc@prism.uvsq.fr*, ***Georges.Gardarin@prism.uvsq.fr*

ABSTRACT: *XML has emerged as the leading language for representing and exchanging data not only on the Web, but also in general in the enterprise. XQuery is emerging as the standard query language for XML. Thus, tools are required to mediate between XML queries and heterogeneous data sources to integrate data in XML. This paper presents the XMedia mediator, a unique tool for integrating and querying disparate heterogeneous information as unified XML views. It describes the mediator architecture and focuses on the unique distributed query processing technology implemented in this component. Query evaluation is based on an original XML algebra simply extending classical operators to process tuples of tree elements. Further, we present a set of performance evaluation on a relational benchmark, which leads to discuss possible performance enhancements.*

RÉSUMÉ : *XML s'est imposé comme le méta-langage permettant de représenter et d'échanger des données non seulement sur le web mais aussi de façon générale en entreprise. XQuery s'impose comme le langage de requête standard pour XML. En conséquence, des outils sont nécessaires pour interroger des sources de données hétérogènes avec XQuery, et ainsi intégrer des données hétérogènes en temps réel sur demande. Cet article présente le médiateur XMedia, un outil permettant d'intégrer et d'interroger des informations hétérogènes distribuées sous la forme de vues XML unifiées. Il décrit l'architecture du médiateur et se concentre sur la technique d'analyse de requêtes distribuées qui a été implémentée dans ce composant. L'évaluation de requête est basée sur une algèbre XML étendant simplement les opérateurs classiques de l'algèbre relationnelle à des traitements de tuples d'éléments arborescents. Enfin, nous parlerons d'extensions que nous étudierons qui permettrons d'améliorer les performances du médiateur.*

KEYWORDS: *Mediation Architecture, XML Algebra, XQuery Evaluation*

MOTS-CLEFS : *Architecture de médiation, Algèbre XML, Evaluation XQuery*

1. Introduction

In recent years, there have been many research projects focusing on heterogeneous information integration. Typical information integration systems have adopted a wrapper-mediator architecture (Wiederhold, 1993). In this architecture, mediators provide a uniform user interface to query integrated views of heterogeneous information sources. Wrappers provide local views of data sources in a global data model. The local views can be queried in a limited way according to wrapper capabilities. Although the local as view (LAV) approach has been considered in some systems (Levy *et al*., 1996) (Manolescu *et al*., 2001), most systems follow the global as views (GAV) approach, in which the integrated views are designed in terms of the local views of sources. Well-known research projects and prototypes based on this architecture include Garlic (Haas *et al*., 1997), Tsimmis (Chawathe *et al.*, 1994), IRO-DB (Fankhauser *et al.*, 1998) and Yat (Cluet *et al.*, 1998). While in the 90's most studies were based on using the object model as data integration model, the focus has come to XML as global model at the beginning of the new century.

The advantages of XML as an exchange model, (i.e., it is rich, clear, extensible and secure), makes it the best candidate for supporting the integrated data model. In addition, using XML views for local data sources hides the local specificities of each system. Furthermore, the richness of the XML schema model simplifies wrapper mappings. Also, the emergence of XQuery as a powerful universal query language for XML makes it possible to query XML global and local views in a uniform way based on a standard interface. Thus, these advantages explain that several research projects have emerged to query in a uniform way heterogeneous data sources based on XML as exchange model, see for example (Christophides *et al*., 2000)(Manolscu *et al*., 2001)(Shanmugasundaram *et al*., 2001).

e-XMLMedia is providing one of the first products based on XML to integrate heterogeneous data sources, namely the e-XML mediator (see www.e-xmlmedia.fr). It is the result of a technology transfer from the university of Versailles (PRiSM Laboratory). This mediator with the associated wrappers provides the required functionalities to query in a uniform way heterogeneous data sources. It is a sophisticated component composed of several packages in charge of decomposing queries into mono-source sub-queries, efficiently shipping local sub-queries to data sources, getting results in XML through a SAX interface, processing and assembling them. Queries as well as sub-queries are expressed in XQuery. In addition, capabilities are associated to wrapper so that the mediator sends only supported queries to wrappers. In summary, the mediator uses XML to represent disparate data in a common format and create a unified view of that data. Using advanced distributed query processing technology, the mediator provides an application with the services it needs to integrate on demand heterogeneous information.

This paper describes a version of the mediator called XMedia. This version differs from the industrial version in some ways, notably it is based on an original algebra for XML processing called the XAlgebra. The contributions of this paper are three-fold. First we describe the modular system architecture of the XMedia Mediator. Second, we describe the query processing algorithm, which is based on query transformations and the algebra operating on tuples of XML trees. A critical result is that the mediator is capable of processing most queries in pipeline on XML event flows. Third, we report on a benchmark of the architecture showing the weaknesses and strengths of the main system components, thus leading to new ideas for query optimization. Some of them should be integrated in a future version of XMedia.

The rest of this paper is organized as follows. The next section focuses on the middleware objectives and architecture. Section 3 describes the XAlgebra, a simple extension of relational algebra to process XML forests. Then, we give an overview of the unique query processing technology embedded in the XMedia mediator through a query example. Section 5 reports on some performance measurement based on the TPC/R benchmark adapted to XML. In section 6, we discuss possible extensions of the query processing engine. We conclude by summarizing the contributions and discussing future developments.

2. System Overview and Architecture

2.1 Integrating and Querying XML Views

XMedia mediator is a data integration middleware managing XML views of heterogeneous data sources. It follows the global as view approach. Global views are defined by administrators through Queries referencing local collections of XML documents. They are queried by users through a Java API extending JDBC to XQuery, called XML/DBC. Data sources can be of various types, including relational databases, XML files, XML databases, legacy applications, etc. Specific wrappers delivering metadata through introspection and providing at least a subset of XQuery on exported collections encapsulate them. Ideally, a wrapper can provide mapping functionalities as XML views to achieve local mappings of data and metadata at the source.

The mediator aims at supporting fully XML standards, including XML schema, XQuery, DOM and SAX interfaces. XML schemas are used intensively for metadata representation. In particular, schemas describe wrapped data sources and views at any layer. XQueries are type-checked through schemas. We support currently most XQuery use-cases. Finally, we internally process XML as SAX event flows for efficiency reasons. Indeed, DOM is in general too costly to instantiate XML documents during processing. However, the user can if required get DOM

trees as results and we sometimes use DOM inside the mediator to keep XML documents for latter processing.

Queries are decomposed in optimal mono-source sub-queries and global query plans expressed in a specific algebra (the XAlgebra), extending the relational algebra to process trees. Queries are optimized in a simple but efficient way. Simple heuristics are supported in the current version, while cost-based query optimization could be introduced in the future. Heuristics include the XML counter-part of classical relational detachment of selections and semi-join transformations. Several algorithms are implemented for processing XAlgebra operators.

To discover relevant sites for a query and decompose it, metadata are maintained describing the sources. When a wrapper is registered to a mediator, metadata describing the source are sent to the mediator through a configuration file. This file contains an XML document containing a schema for each collection exposed by the source wrapper. If the schema of a collection is not known, a schema by default is generated, which describes the path set of the collection; it is a form of dataguide (Goldman *et al.*, 1997). Metadata schemas are kept in the mediator memory and indexed by source, namespace, collection and path for fast access during query processing.

2.2 A Recursive Dataflow-based Architecture

The mediator architecture is represented in Figure 1. The XML/DBC API is the only interface with external components. Thus, notice that the mediator ships requests to wrappers through XML/DBC and thus get results through it. This makes possible for a mediator to see another mediator as a wrapper. Furthermore, results are supplied in XML/DBC through SAX readers. Thus, flows of events are transferred between mediators and wrappers, avoiding the overhead generated by the allocation of intermediate memory structures. The recursive and data flow-based architecture is interesting for applications that can perform data integration at multiple stages without much performance degradation.

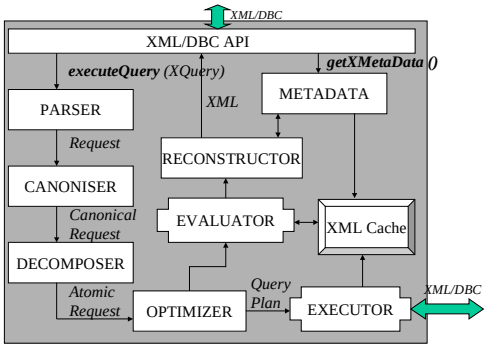


Figure 1. *Overview of the mediator architecture*

The major sub-components are the XQuery parser, the metadata manager, the query evaluator, the query decomposer, and the result reconstructor. All components are briefly described below.

Parser: The parser parses the query and generates the query structure if the query is syntactically and type correct. Otherwise, it returns a documented error.

Canoniser: The canoniser first normalizes the query and generates a query in normal form. Normalization applies the transformation rules described in (Manolscu *et al.*, 2001). For example, let clauses are treated as temporary variable definitions and eliminated. Expressions of the form FLWR(FLWR) are unnested when possible. Second, the canoniser transforms normalized queries in simple queries plus a reconstruction operator. A simple query is a query in which all return expressions are simple path expressions. The reconstruction operator is a sequence of element constructors whose tags and data are either constants or come from simple path expressions.

Decomposer: The decomposer decomposes each simple query in atomic queries, i.e., query involving only one global collection. It also generates a join tree (possibly empty) to keep track of the dependency between the atomic queries. Nesting and unnesting operators may also be generated to restructure intermediate results. Moreover, the decomposer identifies from the metadata the relevant data sources and the collection localization. Based on this information, it translates the atomic queries on a global collection in a union of queries on local collections. In particular, it translates global paths with regular expressions in local paths replacing jokers by the possible paths extracted from the metadata. Finally, it creates a first execution plan for the query.

Optimizer: The execution plan is composed of operators of the XAlgebra. The role of the optimizer is to transform and annotate it to get the best possible plan. Simple optimizations of the query plan are performed in the current version, but more complex ones are planned based on a cost model. For example, the optimizer groups the operators that refer the same source in a single query for shipping once. It also orders the global operators according to query heuristics and selects the best processing method (parallel, sequence or pipeline) for global operators. It should also choose the best algorithm for each algebra operator.

Executor: The executor is in charge of shipping the sub-queries to the wrappers using XML/DBC and collecting the results in cache memory. In general, results are not fully instantiated in main memory but SAX events are produced and directly processed by the evaluator when possible. We represent each ordered collection of XML tree shipped from a wrapper as an XTuple, i.e., a tuple of references to forest of XML trees instantiated in cache.

Evaluator: Based on the query plan, the evaluator evaluates the remaining global query and applies the algebraic operators in main memory. The XAlgebra operators are able to perform XPath-based projection, restriction, product, join,

nesting, sorting, union, intersection and difference of ordered collections of XTuples. For each operator, we implement one or more specific algorithms. For example, several global join algorithms are possible. The evaluator may work with intermediate collections fully stored in main memory, but can also work on a SAX flow of events, thus implementing pipelining and hash joins. Dependent join algorithms requesting XTuple to one source and querying the other based on the results are also possible.

Reconstructor: It applies the reconstruction operator to the intermediate results represented as XTuples and generates the query answer. In other words, it nests and tags the data so as to construct the final result. Finally it built the SAX event flow to deliver the results to the user.

Metadata manager: This package manages the schemas of all registered sources. Further, for each source, it maintains the collection names with the associated queryable path set. The path set is a kind of dataguide giving an overview of all paths instantiated in the source. If a path is missing, it will not be queried. The path set has to be given by the wrapper when registering the source (on command XDescribe).

3. Physical Algebra

As mentioned above, XQuery requests are translated in a physical algebra simple enough to be amenable to optimization and implementation. Several algebras have been recently proposed (Christophides *et al.*, 2000)(Jagadish *et al.*, 2001) (Fernandez *et al.*, 2000)(Galanis *et al.*, 2001) for XML. Our goal is to be as close as possible to some extended relational algebra (Zaniolo, 1985), but to be able to manipulate trees and ordered collections of trees. We now introduce our extended relational data model and its associated algebra for processing XML collections.

3.1 Data model

A relation is classically a subset of the Cartesian product of a list of domains. With simple relations, domains are simple set of values; with object relations, domains can be set of objects or values. We introduce XRelation, that can be considered as a special case of object relations, domains being XML trees. Classically, an XML tree is a set of labeled ordered rooted trees. In addition, cross-tree hyperlinks can be supported as special edges.

With XRelation, domains are XML trees of given path set. Attributes are XPath referencing nodes in the XML trees (see Figure 2). Each attribute can be multi-valued, i.e., refers several sub-trees. XRelation are ordered collections of XTuples. Thus, each XTuple is composed of XPath named attributes, values of which reference subtrees in the collection of trees. As a result, the schema of an XRelation

is of type R(XPath+, [Path+]), where XPath's are defining the attributes and Path's compose the path set of the XML trees.

Figure 2 shows an example of an XRelation composed of four XTuples. The schema of the XRelation is Example (person/fname, person/address, person/address/street, person, book/title, book/author/lname, book/date [person/fname, person/lname, person/address, person/address/street, person/address/town, book/title, book/author, book/author/lname, book/date]). An XTuple refers to nodes and can be perceived as an index of XML trees. Processing through references computed once is much more efficient than processing the trees through direct navigation.

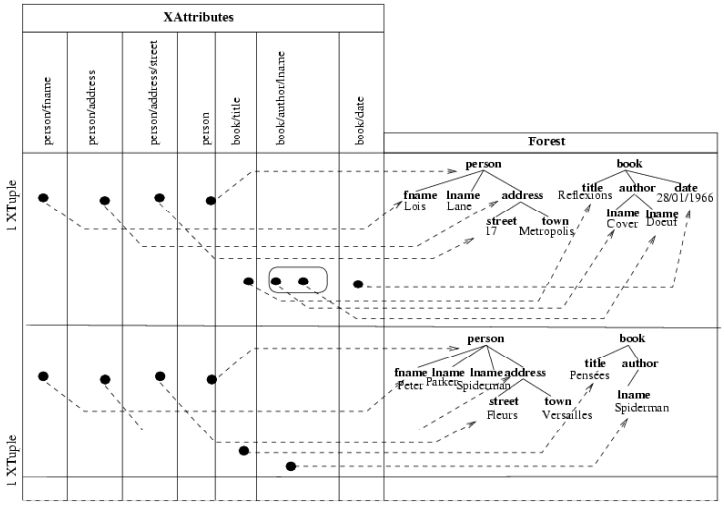


Figure 2. Example of an XRelation

3.2 XAlgebra Operators

The XAlgebra includes both relational operations to process the tables of references and navigation in the XML trees. The algebra is a physical algebra in the sense that algebraic expressions are used to process XML flows and that algorithms are directly implementing them.

XML documents are sent to the mediator in the form of event flows (based on SAX). XTuples are created "on the fly" when XML documents of known schemas are received from the wrappers. Non-blocking operators work in pipeline on the event flows. Blocking operators require the full instantiation of an input flow in cache memory. Non-blocking N-ary operators works in general in parallel on the input flows.

All operators of the XAlgebra receive a collection of XTuples as input and return a collection of XTuples as output. In general, we modify directly the XRelation in memory. Operators also have specific parameters; we only give the some logical ones in the sequel.

The evaluation process of each operator is composed of two steps: a preparation step and an execution one. The preparation step analyzes the input XRelation(s) and the parameters associated to the operator to determine what will be the exact operation to do when the XTuples will flow in. For example, for an operation that requires merging trees, the preparation step determines to which reference node the new sub-tree will have to be linked and which paths will be in common. Thus, the execution step is efficient, as the major part of processing has already been done.

Xsource: XSource is the starting operation to process an XML data source. XSource takes a particular XRelation of schema (Root, [P1, … Pn]) representing a data source as input and generates an XRelation of given schema (a, b, c,… [P1,… Pn]), where a, b, c, … are XPaths over P1, …, Pn. In practice, XSource ships a query to a data source and returns the result as an XRelation. For that, it parses the SAX flow result "on the fly" and generates the collection of XTuples by constructing the trees and identifying the nodes that must be referenced in the references part of the XTuples. XSource preserves the order of the source documents. It is a non-blocking operator, which can construct XTuples as soon as the SAX reader has began to send events.

XRestrict: The XRestrict operator filters each XTuple of an XRelation on a predicate logical expression, each elementary predicate comparing an attribute to constants or checking range constraints over an attribute. If the condition is true, the XTuple is kept. Otherwise, it is removed with its associated XML trees. XRestrict is order-preserving and non-blocking.

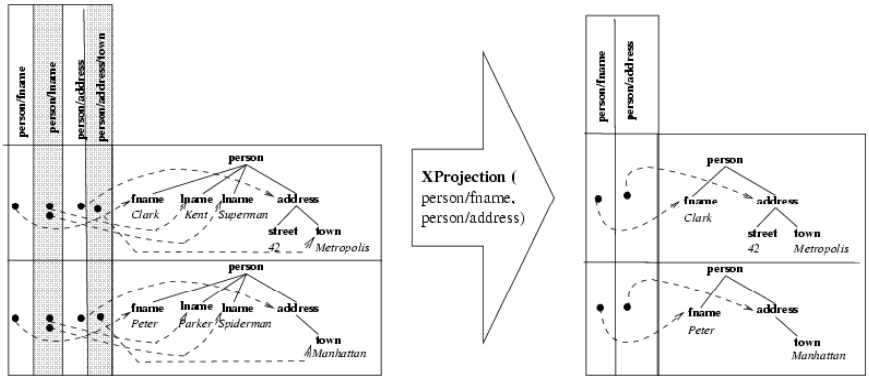


Figure 3. An example of XProject operation

XProject: XProject generalizes the classical projection to XRelations. It takes an XRelation as input and returns an XRelation with only the selected XAttributes in table; non-referenced sub-trees are also removed. In practice, it processes the reference part of the XTuples to determine if the XAttribute must be kept. If not, it deletes the reference and suppresses non-referenced paths in the tree part. Figure 3 illustrates an example of XProject. As well as XSource, XProject is order-preserving and non-blocking.

XSort, XNest, Xunnest: XSort is a simple operator sorting an XRelation on a given list of XPath, by ascending or descending order.

XNest applies a grouping operator to an XRelation. It groups XTuples that have the same values on a set of attributes (i.e., XPath's) by merging their common subtrees and inserting the non common branches in a unique composed tree. Multi-valued references are in general created. This is a costly operator that first applies an XSort and then a merging of trees with similar path set.

XUnnest is the reverse of XNest: it decomposes multi-valued sub-trees of similar path set in the XRelation by replicating common sub-trees in the reference part and creating as mono-valued trees as needed in the tree part.

Xproduct, Xjoin: The XProduct operation takes two collections as input and computes the Cartesian product of them. Moreover, the trees of each XTuple are merged if their path set overlaps from the roots. In general, the Cartesian product can be pipelined but order is then non-preserved. It is possible to preserve the order of one input relation using a nested-loop algorithm, but then the operator is blocking. In general, these parameters depend on the implementation algorithm as well as for relational algebra.

XJoin is the generalization of a relational join. It is an XProduct combined with an XRestrict. The XJoin is a core operator of the physical algebra. Several algorithms have been implemented for the XJoin including nested loops, sort-merge and "query one source with the other". While the nested loop can be pipelined, others cannot. Only non-pipelined nested loops are order-preserving, but sort-merge can produce an interesting order.

Xaggregate: As with extended relational algebra, the purpose of aggregation is to apply a MIN, MAX, COUNT, AVG or SUM function to a collection of values. The collection is simply given by an XPath attribute of the XRelation. Except with COUNT that counts directly the number of references, the functions apply to the values referred by the attributes, which have to be correctly typed (numeric with classical functions). XAggregate is a blocking operation non order-preserving.

XReconstruct: Reconstruction is in general the final operation in an algebraic tree to publish the final SAX event flow as result. It takes as input parameters an XRelation and an XML document in which values are replaced by attributes of the XRelation (i.e., XPaths). The effect is to produce one result instance per XTuple. The operation is order-preserving and non-blocking.

XUnion, XDifference, XIntersection: They are classical set operations applied to set of XTuples.

4. XQuery Processing Example

As introduced in the architecture section, the construction of an execution plan follows the following steps: (1) Normalization and canonization (2) Atomization and join extraction (3) Source identification (4) Execution plan creation (5) Execution plan optimization.

We are now going to illustrate these steps with a simple example. For our experiments, we adapted the TPC-R benchmark to a scenario suitable for a federated and semi-structured system. We basically grouped some tables together to obtain hierarchical data. Figure 4 describes the schema and distribution of data extracted from the TPC-R benchmark.

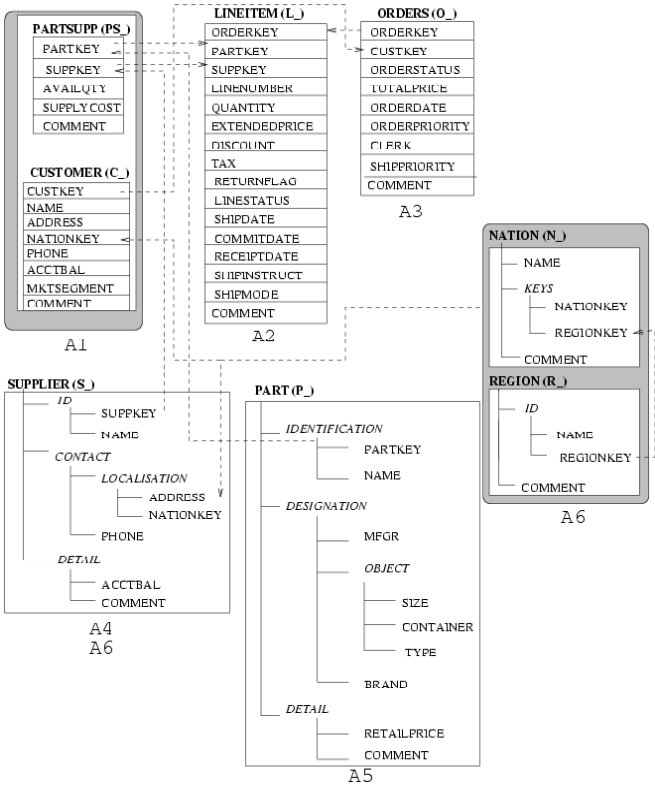


Figure 4. Schema and distribution of data

Relational tables PARTSUPP and CUSTOMER, LINEITEM, ORDERS are managed by wrappers A1, A2 and A3 on top of a relational DBMS. Tree-structured XML collections SUPPLIER, PART, NATION and REGION are stored in an XML DBMS. SUPPLIER is partitioned on wrappers A4 and A6 while PART is managed by wrapper A5. NATION and REGION are managed by wrapper A6.

To illustrate the query processing steps described above, we assume the following query: “Display for each nation having iron in comment, the list of suppliers (name and phone) located there with nested partsup (partkey and supplycost) having an available quantity greater than 45.” The formal query can be written in XQuery as follows:

```
for $n in Collection("*")/nation
where contains ($n/comment, "iron")
return
  <nation>
    <name>{$n/name}</name>
    <suppliers>
      for $s in Collection("*")/supplier, $ps in Collection("*")/partsupp
      where $s/id/suppkey = $ps/suppkey and $ps/availqty > 45
          and $s//nationkey = $n/nationkey
      return
          <supplier>$s/name</supplier>
          <phone>$s/contact/phone</phone>
          <partsupp>
              <partkey>{$ps/partkey}</partkey>
              <supplycost>{$ps/supplycost}</supplycost>
          </partsupp>
    </suppliers>
  </nation>
```

As there is not any LET clause in the example, the query is directly unnested. Applying rules similar to that defined in (Manolescu *et al*., 2001), we unnest sub-queries from the request and purge them of all reconstruction tagging. We call them *elementary queries*. Then, the canonization phase generates the reconstruction query (XReconstruct operator) plus the elementary query. The reconstruction query is simply the returned document with XPath expressions in place of constants.

Canonized request
Elementary query 1
let t1 **::= for** $n **in** Collection("*")/nation **where contains** ($n/comment, "iron") **return** ($n/nationkey, $n/name)
Elementary query 2

```
let t2 := for $t in $t1; $s in Collection("*")/supplier, $ps in Collection("*")/partsupp
where $ps/availqty > 45
return($s/contact/localisation/nationkey,$s/id/suppkey,$s/name,$s/contact/phone,
($ps/suppkey , $ps/partkey, $ps/supplycost))
```

Reconstruction query

```
<nation>
   <name>{$t1/name}</name>
   <suppliers>
      <supplier>{$s/name}</supplier>
      <phone>{$s/contact/phone}</phone>
      <partsupp>
         <partkey>{$ps/partkey}</partkey>
         <supplycost>{$ps/supplycost}</supplycost>
      </partsupp>
   </suppliers>
</nation>
```

Next, the atomization step extracts from the elementary query the maximum sub-queries for each logical collection with associated restrictions and other unary operators as sort or aggregate. It also generates the final join conditions possibly followed by aggregate, sort and a final nest operator to get the resulting XTuple's correctly nested for reconstructing the final XML documents. In our simple case with only restrictions and joins, we obtain three atomic queries and two joins followed by a nest. They can be expressed as follows in XQuery-like syntax.

Decomposed Request

- Atomic Request t1

```
let t1 ::= for $n in Collection("*")/nation
where contains($n/comment, "iron")
return ($n/nationkey, $n/name)
```

- Atomic Request t2

```
let t2 := for $s in Collection ("*")/supplier
return($s/contact/localisation/nationkey,$s/id/suppkey,$s/name, $s/contact/phone)
```

- Atomic Request t3

```
let t3 := for $ps in Collection ("*")/partsupp
where $ps/availqty > 45
return ($ps/suppkey , $ps/partkey, $ps/supplycost)
```

- Global Request

```
for $n in t1, $s in t2, $ps in t3
where $s/id/suppkey = $ps/suppkey  and $s/ /nationkey = $n/nationkey
return($n/name,($s/name,$s/contact/phone, ($ps/partkey, $ps/supplycost)))
```

The request is then analyzed further to identify the data sources that may contribute to the result. The metadata describing each registered source are used both to determine source relevance and to complete XPath's with jokers. Notice that

a source can handle several collections and that a collection can be found on several sources. For atomic queries t1, t2 and t3, we obtain:

Atomic request	Relevant path sets	sources
t1	Collection("NATION")/nation/comment Collection("NATION")/nation/nationkey Collection("NATION")/nation/name	A6
t2	Collection("SUPPLIER")/ supplier/contact/localisation/nationkey Collection ("SUPPLIER")/supplier/id/suppkey Collection("SUPPLIER")/ supplier/id/name Collection("SUPPLIER")/ supplier/contact/phone	A4, A6
t3	Collection("PARTSUPP")/availqty Collection("PARTSUPP")/suppkey Collection("PARTSUPP")/partkey Collection("PARTSUPP")/supplycost	A1

The execution plan can now be constructed in terms of XAlgebra operator. For each atomic request, an XSource operator is created. Its role is to ship the request to the wrapper and get the result under the form of XTuple's. The global request is used to compose the join tree and the nest operator. Finally, the XReconstruct operator is added to generate the correct XML result. The proposed execution plan for the example request is represented in Figure 5. Of course, this one should be further optimized.

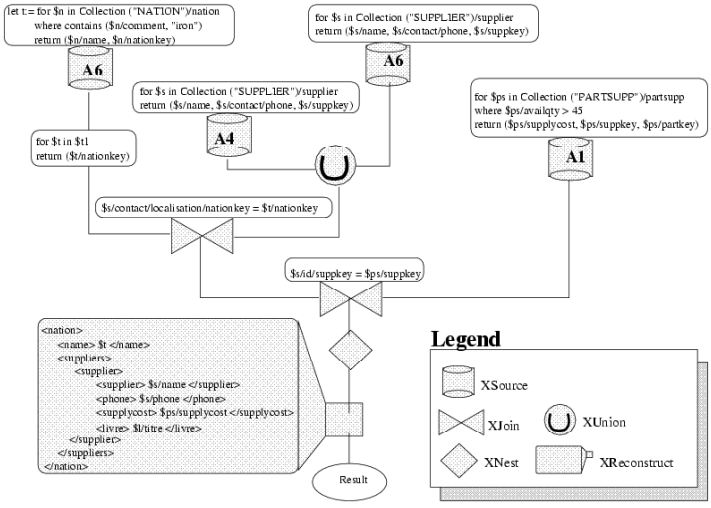


Figure 5. Proposed execution plan for the request

The algebraic tree can be optimized using traditional rules of the nested relational algebra: perform restriction at first, push project and nest operator up the

tree, order joins, select the best algorithm for each operator. This last optimization requires either user hints or a cost model. We shall discuss that further in the sequel.

5. Performance Measurement

To further understand the system bottlenecks and determine useful optimizations, we experiment with a beta version of the industrial system. In this section, we describe some results of our experiments that try to capture the overhead induced by each component of the architecture.

5.1 Evaluated Architecture

We use client-server architecture with two servers. The client processor is a Celeron 600 MHz with 64 Mb RAM, while the servers are both Pentium 4 1.6 GHz with 256 Mb RAM. The network is 10 Mbits/second. All systems are running Linux. 2.4

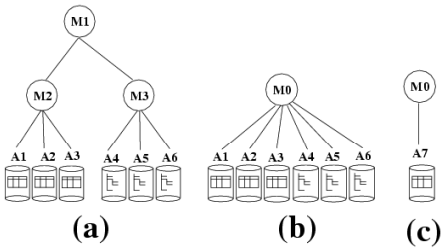


Figure 6. The compared mediation architecture

To compare various architectures, we use different arrangements of mediators and wrappers, as shown in Figure 6. M0, M1, M2, M3 and M4 are mediators. They are all run on the client computer. A1, A2, A3 are wrappers on top of a relational databases. A4, A5, A6 are wrappers on top of a semi-structured database (this is indeed the e-XML Repository of e-XMLMedia). A7 is a wrapper on top of a relational database that contains exactly all the data of A1, A2 and A3. M1 is connected to mediators while all others are connected to wrappers. This is possible as mediators and wrappers have the same interfaces.

5.3 Time per step

The processing of a request follows the steps below:

1. Request parsing that transforms the XQuery request into an internal form.
2. Algebra tree construction that normalizes, canonizes, atomizes the request and finally constructs the algebraic tree.

3. Execution initialization establishing the connection to the wrappers and getting the first XTuple.
4. Local execution of the request on the wrapper including sending the request to the wrapper, getting the result by XML/DBC in the SAX format and transforming the SAX flow in XTuple.
5. Global execution of the request and reconstruction, i.e., processing the XTuples through the algebraic tree to return the result.

Steps 1, 2 and 3 compose the initialization phase of request processing.

The time spent for the initialization phase, for steps 4, 5, and for complete processing are depicted on Figure 7. The initialization step is almost insignificant with regards to other times. The total time is still approximately the double of the wrapper time. The evaluation on the wrapper consists of: (1) Transforming the request into SQL (2) Executing the request on the database (Oracle) (3) Getting the tuples and changing them into XML document.

As the results are measured with a hot database, the tuples are in cache and SQL requests are executed in main memory. This confirms that the dominant time is XML construction and shipping.

Figure 7. Execution time for each step

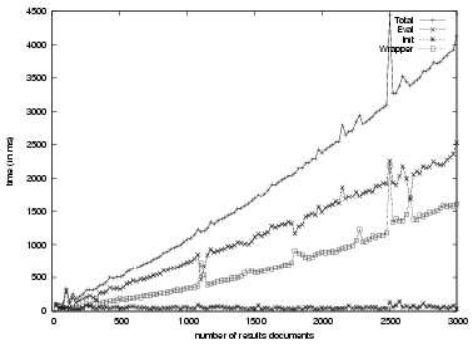
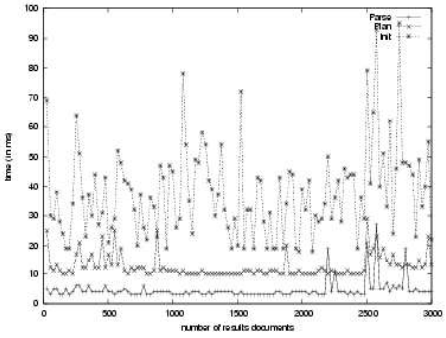


Figure 8. Execution time of init steps



In Figure 8, we detail the initialization time between time to parse the query, time to generate the execution plan and time to get the first result. All these times are small. Request parsing is very short (< 10 ms). Generating the execution plan takes a little more time (< 15 ms). Getting the first result requires a little more time, showing again that exchange time is dominant.

5.2 Mediation Cost Overhead

To evaluate the costs of the mediator overhead, we consider a set of queries of the modified TPC/R benchmark introduced above. The following simple query

```
for $O in collection("ORDERS")
where $O/orderkey < N
return <result> <O>$O/comment</O></result>
```

is executed successively on mediator M0, mediator M4 and wrapper A3. N varies from 1 to 3000 to get different result size. In this way, we can compare the overhead cost of a mediator on another mediator, and of a mediator on a wrapper. To compare with a direct access to wrapper A3, all orders are managed by wrapper A3. Figure 9 hows the execution time depending on the number of resulting documents for each type of execution.

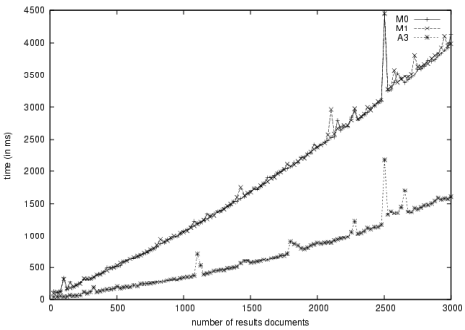


Figure 9. Execution time on M0, M1, A3

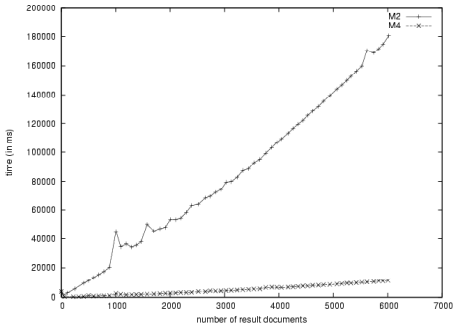


Figure 10. Execution time on M2 and M4

The Figure 9 shows that the execution time of the query on M1 accessing M2 then A3 differs by less than 10% from the execution time of M0 accessing directly A3. This demonstrates the value of our recursive architecture and in general the small overhead induced by the mediator for simple queries. Notice that running the query directly on the wrapper takes approximately half time. This is due to the time required for transferring and converting the data in XML.

5.4 Intersite join

We now submit a set of requests that perform a join between two tables. The request is as follows:

```
for $L in collection("LINEITEM"), $O in collection("ORDERS")
where $O/orderkey = $L/orderkey and $L/orderkey < N
return
<result>
    <lcom>$L/comment</lcom>
    <ocom>$O/comment</ocom>
</result>
```

As previously, N varies from 1 to 3000 to change selectivity. We first evaluate the request on the mediator M4 and then on the mediator M2. In the first case, the join is executed by the data source (Oracle) in the server memory, in the second case, the join is performed on the mediator and XML tuples are transferred on the network. Again, the result (see Figure 10) shows that the transfer time is dominant. It also shows that intersite joins are costing operations that should be pushed to the wrapper when possible.

6. Possible Improvements

The results of the experiments, some being reported above, demonstrate the high communication cost to exchange XML documents between wrappers and mediators. Thus, this is the first point to improve. We propose several improvements that should reduce this cost drastically.

6.1 XML Compression and Bulk Transfers

Transferring XML documents between wrappers and mediators appears to be costly. Each XTuple is encoded in an XML message and sent over the network. The XML message is then parsed on the client and transformed internally in an XTuple descriptor and XML trees as event flows. Thus, the number of messages is important and the processing time is high. One may argue that our network is slow (10 M bits), but this is not sufficient to explain the results.

To save in number of messages, we could use bulk transfer, and send several messages in one block. The number of messages per block should be tuned such that the pipeline on the client continues to proceed smoothly. Nevertheless, this does not save parsing and unparsing of lengthy messages. This is somehow inherent to XML and may degrade performances forever.

One solution is to use a compressed format for transferring XTuples. Schemas of XTuples are known both by the client and the server under the form of a list of paths. The types of values (leaves of XML trees) are also known through XML schemas. Thus, an obvious compression mechanism consists in sending an XTuple as a sequence of path identifiers (16 bits is sufficient) followed by the leaf value encoded according to its type. Parsing will then be an obvious task. However, we may loose the purity of XML and the generality of the communication mechanism.

Although it is a bit contrary to XML principles, we believe that a compression device saving parsing time is crucial.

6.2 Operator Algorithms

The benchmarked version of the mediator uses a simple join algorithm (optimized nested loops). It is obvious that other algorithms should be considered for joins notably, but for other operators as well (e.g., nest is quite complex). Implementing dependent joins, i.e., join by reading an XRelation and querying the other with the read value, could be helpful to save in number of messages in case of small answers. Merge join and hash join could also be useful. Thus, we are currently integrating a library of algorithms for each XAlgebra operator. The problem is then how to select the best plan. A possible answer is to develop a cost model.

6.3 Cost Model

The classical solution for choosing the best execution plan is to compare plan costs using a cost model. We propose a cost model somehow inspired from DISCO (Tomasic *et al.*, 1996). The mediator has a generic cost model derived from a relational cost model extended with tree manipulation. Then each wrapper can export specifics statistics and formulas to the mediator. The generic cost model is generally used with the exported statistics (to evaluate cardinalities), but specific formulas exported by a wrapper can override generic formulas. This approach gives a framework to compute the global cost of a query plan integrating local information on sources.

To communicate their cost model to the mediator, a wrapper uses a cost model language. In an XML environment, the cost language has to be defined in XML. As formulas and statistics definitions use a lot of mathematics notations, we based our cost language on MathML. MathML is a specification of the W3C for coding in XML the representation or the structure of a mathematical object. Only the semantic information about a mathematical object is interesting for our purpose. The advantages of using MathML for describing cost formulas are three-fold: it is full XML, it supports general formulas, and calculation software can be used to compute formulas.

Parameters used for evaluation of a cost model are statistics relative to the system (system statistics) and statistics relative to the data (data statistics). For semi-structured data, some other system parameters should be defined, such as comparison between two typed values, comparison between two trees, navigation in a tree (pointer chasing). Data statistics depends on data and collections of data contained in the source. Classical data statistics used are: cardinality of a collection, distribution of an attribute in a collection, minimum and maximum values taken by an attribute. For semi-structured data, one must add some parameters such as average depth and width of trees in a collection. Such information could be derived from XML schemas.

A mediation cost model depends on its system parameters and its data parameters. One or more formulas are defined in order to calculate the evaluation cost of a request in this system (large granularity) or a predicate in a particular operator (thin granularity). Formulas for the thinner granularity are specifics to the sources and can be expressed with specific parameters. Formulas for the larger granularity consist of cardinality, total cost and execution cost.

In summary, developing a complete generic cost model with overloading per wrapper is possible in an XML mediator. Cost formulas can be exchanged in XML. A cost model is required to select the best execution plans, based on estimators of communication costs and processing costs.

6.4 Wrapper Capabilities

In the described version of the mediator, source capabilities are taken into account by classes. We support three classes of sources: XQuery source, SQL source, XML file. Basically we push XQuery queries to our XQuery source, basic SQL to the SQL sources, and just selection to files wrapped by a filter. This is nice but insufficient for distinguishing detailed functionalities of sources. To go further and take into account detailed functionalities of sources at the mediator level, a precise description of source capabilities is required. This can be done globally for a source by sending an XML file associated to the metadata detailing what XML operator is allowed globally on all collections or specifically on one collection, the specific prevailing.

6.5 Semantic Cache

Another way to save messaging is implementing a semantic cache at the mediator level. XTuples answering a given query run by the mediator could be kept in cache. XML format will not be appropriate as too large; we would rather use the compressed format introduced above. Thus a table of queries ordered by execution time with associated results should be kept in cache and used to answer new queries. Of course, update on source data will not be taken into account. Thus, semantic caching is only possible for certain collections of XML documents not updated frequently. It is very valuable in the case of slow sources, e.g., Web sources.

With semantic caching, a new request should be first checked against the cache to determine if it can answer the request or a part of it. If yes, the request is split in two parts (one part can be null): a local request that can be answered by the cache and a source request that must be answered by the distant sources. The two results have to be correctly assembled. This can be done by comparing the algebraic trees in canonical form of the request with the one of each cached request. If one computes a subset of the other, the cache can be used to process part of the request. The request algebraic tree has to be pruned to replace the common part by a call to the XRelation in the cache. Using an XML semantic cache for XQuery is a complex subjects that has to be further worked out.

7. Conclusion

We have presented the XMedia system for querying integrated views of heterogeneous data. A first version of the system has been developed at the university at the end of the 90's, and then transferred to the industry from 2000 to 2002 where it was completely redesigned. Currently, a new research project is planned to develop an improved mediator, which should take into account lessons from the past. The second version is commercialized and has several ongoing applications and planned ones, notably on tourism data, health data, and chemistry data.

The version described in this paper has unique features. XQueries are compiled in execution plans expressed in an extended relational algebra capable of processing in pipeline XML trees. Query processing is clearly divided in steps. We isolated the query rewrite step from the decomposition step that generates algebraic trees processing localized data sources. Localization of collections is performed using metadata under the form of XML schemas. The optimization step requires a cost model to be fully efficient. Hints have been introduced in the industrial version.

Performance measurement demonstrates the validity of the approach but the cost of transferring XML files from wrappers to mediators appears to be excessive. Several possible improvements that should be partly implemented in XMedia have been suggested. We would like also to develop a more efficient X-machine to process XAlgebra expressions on XML flows.

References

Chawathe S., Garcia-Molina H., Hammer J., Ireland K., Papakonstantinou Y., Ullman J., and Widom J.: *"The TSIMMIS Project: Integration of Heterogeneous Information Sources"*, IPSJ Conf, Tokyo, Japan, October 1994, p. 7-18.

Christophides V., Cluet S., Siméon J.: *"On Wrapping Query Languages and Efficient XML Integration"*, ACM SIGMOD 2000, Dallas, USA, p.141-152.

Cluet S., Delobel C., Siméon J., Smaga K.: *"Your Mediators Need Data Conversion"*, ACM SIGMOD Intl. Conf. on Management of Data, Seattle, USA, 1998, p.177-188.

Fankhauser P., Gardarin G., Lopez M., Muñoz J., Tomasic A.: *"Experiences in Federated Databases: From IRO-DB to MIRO-Web"*, 24rd Intl Conf VLDB, New York City, USA, 1998, p.655-658.

Fernandez M., Simeon J., Wadler P.. *"An Algebra for XML Query"*, In Foundations of Software Technology and Theoretical Computer Science, New Delhi, December 2000.

Galanis L., Viglas E., DeWitt D.J., Naughton J.F., Maier D. *"Following the Paths of XML: an Algebraic Framework for XML Query Evaluation"*, Technical Report, 2001

Goldman R, Widom J. "'DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases", Proc 7th VLDB, Athens, 1997.

Haas L., Kossman D., Wimmers E., Yang J., *"Optimizing Queries across Diverse Data Sources"*, 23rd Intl Conf VLDB, Athens, Greece, 1997.

Jagadish H.V., Lakshmanan L.V.S., Srivastava D., Thompson K *"TAX: A Tree Algebra for XML"*, Proc Of DBPL Conf., Roma Italy, Sept 2001.

Levy A., Rajaraman A., Ordille J. *"Querying Heterogeneous Information Sources Using Source Descriptions"*, Intl. Conf. on VLDB, Bombay, 1996.

Manolescu I., Florescu D., Kossmann D.: *"Answering XML Queries over Heterogeneous Data Sources"*, 27th Intl Conf VLDB, Roma, Italy, 2001, p.241-250.

Shanmugasundaram J., Kiernan J., Shekita E., Fan C., Funderburk J.: *"Querying XML Views of Relational Data"*, Proc. Of the 27th Intl Conf VLDB, Roma, Italy,. 2001, p.261-270.

Tomasic A., Raschid L., Valduriez P. *"Scaling Heterogeneous Databases and the Design of DISCO"*, Intl Conf. on Distributed Computing Systems, Hong Kong, 1996.

Wiederhold G.: "Intelligent Integration of Information", ACM SIGMOD Conf. On Management of data, Washington D.C., USA, May 1993, p. 434-437.

Zaniolo C. *"The Representation and Deductive Retrieval of Complex Objects"*, Proc of the 11th Intl Conf VLDB, Stockholm, Aug., 1985.