A Web-Services Architecture for Efficient XML Data Exchange

Sihem Amer-Yahia
AT&T Labs–Research
180 Park Ave, Florham Park, NJ 07932
sihem@research.att.com

Yannis Kotidis
AT&T Labs–Research
180 Park Ave, Florham Park, NJ 07932
kotidis@research.att.com

Abstract

Business applications often exchange large amounts of enterprise data stored in legacy systems. The advent of XML as a standard specification format has improved applications interoperability. However, optimizing the performance of XML data exchange, in particular, when data volumes are large, is still in its infancy. Quite often, the target system has to undo some of the work the source did to assemble documents in order to map XML elements into its own data structures. This publish&map process is both resource and time consuming.

In this paper, we develop a middle-tier Web services architecture to optimize the exchange of large XML data volumes. The key idea is to allow systems to negotiate the data exchange process using an extension to WSDL. The source (target) can specify document fragments that it is willing to produce (consume). Given these fragmentations, the middle-ware instruments the data exchange process between the two systems to minimize the number of necessary operations and optimize the distributed processing between the source and the target systems. We show that our new exchange paradigm outperforms publish&map and enables more flexible scenarios without necessitating substantial modifications to the underlying systems.

1. Introduction

Large organizations use a plethora of systems to support their daily operations. Depending on the application, a system may act as a *data broker* by disseminating information that is consumed by the receiving applications. For instance, in a telecom provider like AT&T, a *sales* and *ordering* system provides an interface to extract data on customer orders. This data is used to drive a *provisioning* process that implements changes to the physical network in order to support the line features requested by customers. Finally, this data, along with usage information generated from the operation centers, is consumed by a *biller* to setup customer accounts in order to collect revenue. In such real-world applications, the data that is exchanged can reach very large volumes. As an example, usage data from the telephony network easily exceeds 60GB per day. In this paper, we focus on the optimization of data exchange between two applications when the amount of data is large.

In order to collaborate, applications implement pair-wise agreements that define the format of the data to exchange. To this end, XML is most commonly used. Web services are typical examples that use XML as the grammar for describing services on the network as a collection of systems capable of exchanging data and messages. The specification of a Web Service Description Language (WSDL) document hides the details involved in data communication by focusing on the format in which that data is being produced and consumed and the services that are provided at each endpoint [12]. However, optimizing the performance of exchanging large data volumes has not attracted a lot of attention. Quite often, the target system has to undo some of the work the source did to assemble documents in order to map XML elements into its own data structures. This process is both resource and time consuming.

In a typical data exchange scenario, that we will refer to as publish&map, XML documents are built at a source application and shipped to be consumed at a target one. The process of publishing an XML document from stored data translates often to costly *combine* operations (through joins in case of relational stores) that piece document fragments together. Quite often, some of these fragments are stored similarly at the source and at the target systems, in which cases combining such fragments at the source is unnecessary because the target system will *split* them again into its internal structures. Furthermore, in publish&map, XML documents are *built at the source* and *consumed at the target*, imposing a strict processing distribution that does not explore the capabilities of the underlying systems.

1.1. Motivating Example

We sketch a typical exchange scenario between a *sales* and *ordering* system in which data is stored in a relational database conforming to schema S, and a *provisioning* system that stores data in a LDAP directory that conforms to schema T. A LDAP instance is a tree. The LDAP data model [7] provides the notion of a class defined by a name and a set of attributes. A class has at least two attributes: DN that stands for distinguished name and corresponds to the Dewey [7] identifier of a node in the tree instance; and objectclass (omitted here) that contains the name of the class to which a node in the tree belongs.

Schema S

```
CUSTOMER[C_CUSTKEY,C_NAME]
ORDER[O_ORDERKEY,O_CUSTKEY]
SERVICE[S_SERVICEKEY,S_NAME,S_ORDERKEY]
LINE_FEATURE[L_LINEKEY,L_FEATUREKEY,L_TELNO,
           F_FEATUREID,L_ORDERKEY]
SWITCH[S_SWITCHKEY,S_ID,S_LINEKEY]
```

Schema T

```
CUSTOMER_T OBJECT-CLASS
  MUST CONTAIN DN,C_NAME
  TYPE C_NAME STRING
ORDER_SERVICE_T OBJECT-CLASS
  MUST CONTAIN DN,S_NAME
  TYPE S_NAME STRING
LINE_SWITCH_T OBJECT-CLASS
  MUST CONTAIN DN,L_TELNO,S_SWITCHID
  TYPE L_TELNO STRING
  TYPE S_SWITCHID STRING
FEATURE_T OBJECT-CLASS
  MUST CONTAIN DN,F_FEATUREID
  TYPE F_FEATUREID STRING
```

S and T (sketched above) contain information on customers that order phone service. A customer might have several phone lines each of which has a telephone number, a switch that the line is connected to and a list of features (for example, caller ID). S and T agree on exchanging data using the WSDL specification of Figure 1. This specification defines a single service, CustomerInfoService, providing information on orders that are grouped per customer and information on phone services (local, long-distance etc.) grouped per order. This service is deployed using the SOAP 1.1 protocol [11] over HTTP. When requested, it returns a set of XML documents that conform to the XML Schema specified in the WSDL definition. The service can take one or several arguments that will be used to subset the data. Since they are not required in further discussions, we omit message, port and binding elements in the WSDL specification and refer the reader to [12] for examples of complete definitions.

When T requests CustomerInfoService, S executes it by building one XML document for each customer and sending those documents to T. Intuitively, this process can be improved as follows. Building XML data from the relational data stored in S requires *combining* document fragments together. This operation translates into relational joins which are the most expensive operations when building XML documents from relational data (see [5, 6] and our experiments in Section 5). If we knew that the XML fragment containing Customer elements is stored similarly at S and T (which is the case in our example), S could ship the customers fragment to T without combining it with the other document fragments. In addition, in T, orders and services are stored together in the class ORDER_SERVICE_T. This class can be populated by creating a fragment that *combines* data from the two relations ORDER and SERVICE without further combining this data with the rest of the document. Finally, in order to populate the class FEATURE_T, we could *split* and project LINE_FEATURE in S into a fragment containing features and ship it to T.

```
<?xml version="1.0"?>
<definitions name="CustomerInfo"
          targetNamespace="http://customers.wsdl"
 <types>
   <schema targetNamespace="http://customers.xsd"
           xmlns="http://www.w3.org/XMLSchema">
     <element name="Customer">
       <sequence>
         <element name="CustName"
                  type="string"/>
         <element name="Order"
                  maxOccurs=unbounded/>
           <sequence>
             <element name="Service"/>
               <element name="ServiceName"
                        type="string"/>
             <element name="Line"
                      maxOccurs=unbounded>
               <sequence>
                 <element name="TelNo"
                          type="string"/>
                 <element name="Switch">
                   <element name="SwitchID"
                            type="string"/>
                 </element>
                 <element name="Feature"
                          maxOccurs=unbounded/>
                   <element name="FeatureID"
                            type="string"/>
                 </element>
               </sequence>
             </element>
           </sequence>
         </element>
       </sequence>
     </element>
   </schema>
 </types>
 <service name="CustomerInfoService">
   <documentation>Provides customer
                  information</documentation>
     <port name="CustomerInfoPort"
                 binding="tns:CustomerInfoBinding">
       <soap:address location="http://customerinfo"/>
     </port>
 </service>
</definitions>
```

Figure 1. WSDL specification for Customer Data Exchange

Consequently, if data could be sent *fragmented* from S to T, unnecessary computations would be avoided. How-

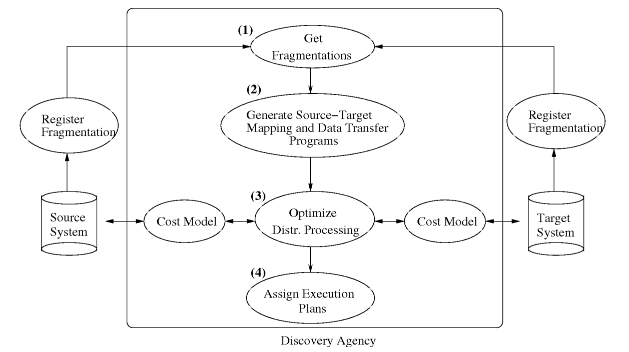

Figure 2. End-to-End XML Data Exchange

ever, in general, internal data representations (schemas, indices, etc.) are not exposed in a Web services architecture. Negotiations have to be established using a higher level interface such as WSDL. Thus, WSDL needs to be extended with a notion of *fragmentation* of the initial XML Schema. As an example, `S` could provide a fragmentation into five XML Schemas, each of which corresponds to a relation in `S`. The interpretation of a fragmentation at a source is that it is willing to produce fragments of documents that conform to that fragmentation. The interpretation of a fragmentation at a target is that it is best for it to consume XML fragments that conform to that fragmentation. Specifying a fragmentation does not correspond to revealing systems internals. In general, systems participating in a data exchange might have different performance reasons to provide a fragmentation of the agreed upon XML Schema. For example, a system might already have indexing mechanisms that are more efficient to build if data is consumed in a particular fragmentation. Systems *should not* have to specify a fragmentation. The initial XML Schema would be used by default if no fragmentation is provided as in publish&map.

Having the ability to specify a fragmentation does not mean that the source and target systems are not willing to do some work to exchange data. In fact, fragmentations generate a whole new set of optimization opportunities. For example, `S` could define a fragmentation in which it produces `Order` and `Service` elements in two separate fragments and `T` could specify that it expects to consume a fragment in which `Order` and `Service` are combined in which case, the two source fragments might be combined at either systems. Furthermore, `S` and `T` could have specified the same fragmentation, in which case, `S` will do all the work to generate the corresponding fragments which are shipped to `T`.

This new level of abstraction does not require substantial modifications to the underlying systems because the least we can expect from `S` (resp. `T`) is to produce (resp. consume) documents that conform to the agreed upon XML Schema and therefore, to a fragmentation of it. Furthermore, the exchange process does not violate the principle behind Web services: manipulate XML data without revealing internal data structures and how data is produced and consumed. The lowest granularity of a fragment is a single element in the XML Schema. However, a fragment could correspond to the result of a service call. For instance, `S` could provide a fragment that defines a service, `TotalMRCService`, standing for the total monthly recurring charges for all lines ordered by a customer, without revealing how this fragment is computed.

1.2. Contributions

We summarize our contributions as follows:

- We exploit a Web services architecture to enable the exchange of document fragments thereby optimizing the exchange process. Our new architecture does not necessitate substantial modifications to the underlying systems and complies with the platform independence principle provided by the use of XML as the exchange format.
- We define high level operations that operate on XML fragments and are used to express the exchange of XML documents between two systems without revealing their internals.
- We present experiments that study the costs of exchanging XML data between two relational back-ends. To the best of our knowledge, this is the first attempt to chart detailed performance in an end-to-end realistic XML data exchange. We compare our solution with publish&map in which publishing is optimized using the techniques introduced in [6]. Our first finding is that, in most cases, our solutions compare favorably to the time to do XML publishing only! Furthermore, when compared to publish&map, our techniques are up to six times faster in data processing and they provide up to 43% reduction in the overall data exchange time.

Section 2 describes our Web services architecture. Section 3 defines the data model and primitive operations. Section 4 describes the cost model, the plan generation and the cost-based distributed processing algorithms. Experiments are presented in Section 5. Related work is given in Section 6 and concluding remarks in Section 7.

2. Data Exchange Architecture

In a typical Web services architecture [9], a *service provider* has a service that is made available to other systems to use. The provider creates a WSDL service description that defines the service interface, that is, the operations of the service and the input and output messages for each operation. The provider publishes its WSDL service description to a *discovery agency*. *Service requesters* find services via discovery agencies and use the WSDL description to interact with the corresponding service provider. A typical exchange scenario that results from a Web service call induces the following steps: (i) execute the service at the provider and produce relevant XML documents from source data and, (ii) ship the produced documents to the requester that consumes them. We use this architecture as the basis for exchanging large data volumes and extend it with the ability to register fragmentations and with optimization capabilities.

Figure 2 extends the basic architecture as follows. First, the source and target systems independently specify their respective fragmentations using an extension to WSDL and register it at a discovery agency (Step **(1)**). Discovery agencies are repositories of WSDL specifications which may be mapped to Universal Description Discovery and Integration (UDDI) [9] for publishing and discovery of existing services. The discovery agency generates a mapping between the two fragmentations and a data transfer program that combines and splits source fragments to generate target ones (Step **(2)**). The decision of where to perform an operation depends on how much it costs at each system. As in publish&map, we expect the service endpoints to be able to split fragments in order to store them. They may not however have the ability, or the intention, to combine fragments. This is captured in our cost model. Distributed processing is achieved by probing the source and target systems, which implement an interface to provide the cost of each primitive operation (Step **(3)**). In step **(4)**, the discovery agency assigns operations to the source and the target that generate and execute code on their internal data structures.

The discovery agency acts as a middle-ware that does not know about the internal data structures used by the source and target systems. All it sees is the fragmentations defined by each system and a cost interface. The way each fragment is actually produced or consumed by a system is hidden by the WSDL interface. Therefore, the discovery agency needs only high level operations to transform fragments. The next section defines the data model and these operations.

3. Data Model and Primitive Operations

3.1. Data Model

We view XML Schemas as trees.

Definition 3.1 (Fragment) *A fragment of a XML Schema is any subtree of that schema. In addition, the root of the fragment is assigned two attributes:* `ID` *and* `PARENT`.

The `ID` attribute of a fragment uniquely identifies fragments. Its `PARENT` attribute contains the `ID` value of its parent fragment. For example, the XML Schema fragment `Order_Service` is defined below from the XML Schema in Section 1.1.

```
<fragment name="Order_Service">
  <element name="Order">
    <attribute name="ID" type="string"/>
    <attribute name="PARENT" type="string"/>
    <element name="Service">
      <element name="ServiceName" type="string"/>
    </element>
  </element>
</fragment>
```

Definition 3.2 (Fragment Instance) *A fragment instance of a XML Schema fragment is any XML document that conforms to the XML Schema fragment.*

Definition 3.3 (XML Schema Fragmentation) *A XML Schema fragmentation is a set of fragments of that XML Schema.*

Definition 3.4 (Validity) *A fragmentation $\mathcal{F}$ is valid with respect to an XML Schema iff, (i) each element in the XML Schema is defined only once in $\mathcal{F}$ and, (ii) If $\mathcal{F}$ contains more than one fragments, for any fragment f_1 in $\mathcal{F}$, there exists another fragment f_2 in $\mathcal{F}$ such that f_1 is a parent of f_2 or f_1 is a child of f_2.*

Validity defines non-redundant and complete fragmentations. The example below is a valid fragmentation of the XML Schema defined in Section 1.1 that `T` might provide. We refer to it as the `T`-fragmentation.

```
<fragmentation name="T-fragmentation">
  <fragment name="Customer.xsd">
    <element name="Customer">
      <attribute name="ID" type="string"/>
      <attribute name="PARENT" type="string"/>
      <element name="CustName" type="string"/>
    </element>
  </fragment>
  <fragment name="Order_Service.xsd">
    <element name="Order">
      <attribute name="ID" type="string"/>
      <attribute name="PARENT" type="string"/>
      <element name="Service">
        <element name="ServiceName"
                 type="string"/>
      </element>
    </element>
  </fragment>
  <fragment name="Line_Switch.xsd">
    <element name="Line">
      <attribute name="ID" type="string"/>
      <attribute name="PARENT" type="string"/>
      <sequence>
        <element name="TelNo" type="string"/>
        <element name="Switch">
          <element name="SwitchID"
                   type="string"/>
        </element>
      </sequence>
    </element>
  </fragment>
  <fragment name="Feature.xsd">
    <element name="Feature">
      <attribute name="ID" type="string"/>
      <attribute name="PARENT" type="string"/>
      <element name="FeatureID" type="string"/>
    </element>
  </fragment>
</fragmentation>
```

Definition 3.5 (Mapping) *A mapping is defined by* (XMLSchema,S,T,M)*, where* XMLSchema *is the initial XML Schema,* S *and* T *are valid fragmentations of* XMLSchema *and* M *is a function from* T *to the powerset of* S *that associates each fragment in* T *with one or multiple source fragments.*

3.2. Primitive Operations

We now introduce our primitive operations on fragments.

Definition 3.6 (Scan) Scan(f) *reads an input fragment* f *and returns it as output. It also computes the* ID *and* PARENT *attributes of each fragment instance.*

Since each system implements it own Scan, this operation might correspond to a set of lower-level operations that are not revealed to the middle-ware.

Definition 3.7 (Combine) Combine(f_1, f_2) *modifies the input fragment* f_1 *by combining its child fragment* f_2 *with it.* Combine *removes the* ID *and* PARENT *attributes of* f_2.

Combine is used to construct document structure by "inlining" a child fragment with its parent fragment. In the initial XML Schema, a child fragment could be either unique or repeated (*) and the order of children elements matters. This information is recovered from the initial XML Schema when combining fragments. Given the T-fragmentation defined in Section 3.1, Combine(Customer,Order_Service) results in a new fragment Customer_Order_Service:

```
<fragment name="Customer_Order_Service.xsd">
  <element name="Customer">
    <attribute name="ID" type="string"/>
    <attribute name="PARENT" type="string"/>
    <sequence>
      <element name="CustName" type="string"/>
      <element name="Order" maxOccurs=unbounded/>
        <element name="Service">
          <element name="ServiceName" type="string"/>
        </element>
      </element>
    </sequence>
  </element>
</fragment>
```

Definition 3.8 (Split) Split($f, f_1, \ldots, f_n$) *splits a fragment* f *into the set of disjoint fragments* $\{f_1, \ldots, f_n\}$.

Split resembles projection. In addition, it introduces distinct ID and PARENT attributes in each projected fragment to represent the parent/child relationships dictated by the XML Schema. As an example, the T-fragmentation is obtained by performing three splits on the XML Schema.

Definition 3.9 (Write) Write(f) *stores a fragment.*

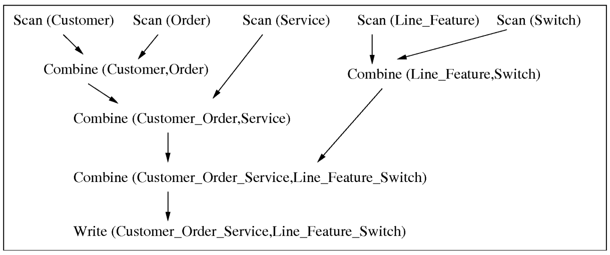
Scan (Customer)
Scan (Order)
Scan (Service)
Scan (Line_Feature)
Scan (Switch)
Combine (Customer,Order)
Combine (Line_Feature,Switch)
Combine (Customer_Order,Service)
Combine (Customer_Order_Service,Line_Feature_Switch)
Write (Customer_Order_Service,Line_Feature_Switch)

Figure 3. Publishing from the S-fragmentation to XML Schema

This operation depends on the system on which it is executed. As an example, if we want to publish an XML document from the source system S defined in Section 1.1, Write would write its input to a file which is then published. If the application is to load documents into a relational database, Write would correspond to SQL LOAD statements.

We do not use a relational or XML algebra because (i) their granularity is too fine and would reveal the underlying storage structures and, (ii) source and target systems could be any persistent store, relational, object, LDAP directories or file systems. Our operations focus on the most expensive parts of a data exchange process: combining and splitting fragments. If the Web service takes arguments as input, we assume the source system will filter the data accordingly and provide us with the relevant pieces. For example, the service CustomerInfoService defined in Section 1.1 could take an argument that specifies customers location based on their state. In this case, the ordering application will provide us with customers that reside in that state. In Section 6, we briefly discuss extensions to this framework to support more complex operations on fragments.

3.3. Data Transfer

Definition 3.10 (Data Transfer) *A data transfer, associated to a mapping* (XMLSchema,S,T,M)*, is a program that defines instances of fragments in* T *from instances of fragments in* S*. We express this program as a set of expressions (composition of primitive operations), one for each fragment in* T *and the fragments in* S *that are associated to it with* M*. Expressions that define fragments in* T *from fragments in* S *may not be disjoint in which case, a single expression might define multiple fragments in* T.

Figures 3 and 4 show two possible data transfer programs between a S-fragmentation and a T-fragmentation. A data transfer program can express transforming data from one format to another. Thus, it can be used to represent publishing data into XML documents, loading XML documents into a database or sending data from a system to another.

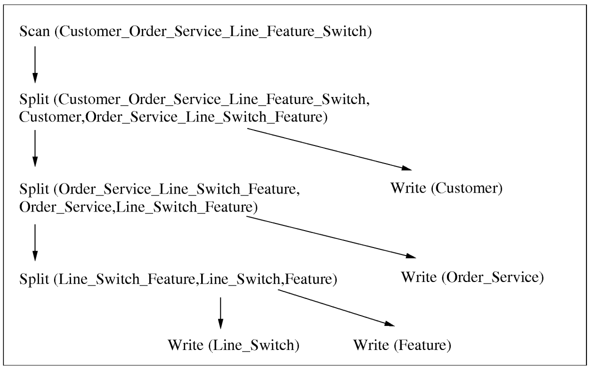
Scan (Customer_Order_Service_Line_Feature_Switch)
Split (Customer_Order_Service_Line_Feature_Switch, Customer,Order_Service_Line_Switch_Feature)
Write (Customer)
Split (Order_Service_Line_Switch_Feature, Order_Service,Line_Switch_Feature)
Write (Order_Service)
Split (Line_Switch_Feature,Line_Switch,Feature)
Write (Line_Switch)
Write (Feature)

Figure 4. Loading from XML Schema to the `T`-fragmentation

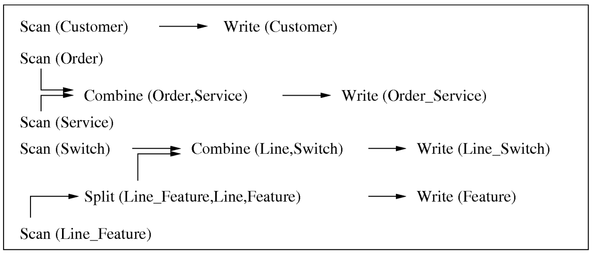

Figure 5. Data Transfer from `S` to `T` fragmentations

A data transfer program is a directed acyclic graph (DAG) whose nodes are primitive operations and whose edges describe data flow between those operations. For instance, in the DAG in Figure 3, `Combine(Customer,Order)` consumes the outputs of operations `Scan(Customer)` and `Scan(Order)`. In this Figure, relational data stored in `S` is published as XML documents conforming to the XML Schema given in Section 1.1. Input fragments are combined to reconstruct the tree structure. In the loading example (Figure 4), the fragment conforming to the XML Schema defined in Section 1.1, is scanned and then split into multiple fragments in the `T`-fragmentation.

Figure 5 shows a possible data exchange program between a `S`-fragmentation and a `T`-fragmentation. We expect this scenario to beat publish&map (See Section 5) because several combines (that build full documents) are avoided.

4. Optimization

There is often more than one program that can be used to express a data transfer for a given mapping. For example, a program that is equivalent to the one in Figure 3 may first combine `Order` with `Service` and then combine the resulting fragment `Order_Service` with `Customer`.

Once fragmentations are registered at the middle-ware, we explore different rewriting strategies for data transfer (Sections 4.2 and 4.3). The resulting programs may differ substantially in their execution times based on the cost of each operation. In what follows, we present the cost model that we implemented in our experiments of Section 5.

4.1. Cost Model

The cost of a program is the execution cost of the operations at the nodes of the DAG added to the cost of shipping data from the source to the target. Each node in the DAG has an annotation `S` or `T`, indicating that the operation is to be performed at the source or at the target.

Let `OP` $\in$ {`Scan`, `Combine`, `Split`, `Write`} be a node in the DAG. Function *comp_cost(OP,location)* returns the cost of executing the operation *OP* at the system specified by *location* (`S` or `T`). We assume that this function is given to us or that reliable estimates can be obtained from the individual systems. For instance, in [6] the middle-ware probes underlying systems for collecting estimates of the execution cost of a given query.

We can easily encode additional restrictions in this setup. For instance, in a publishing scenario, the target system might not have the capability to implement a `Combine` (a *dumb client*). In such cases, we assume the computation cost of these operations to be infinite at the target.

The second component of the execution cost is the cost of shipping fragments from one system to another. When fragments are shipped in XML format there are only small differences in the size of the data. In case fragments are shipped in the form of sorted feeds such as in [5, 6], different plans may have different communication costs because of the presence of NULL values and repeated elements due to inlining. Furthermore, when the Web service has additional arguments that subset the data, the selectivity of the combines affects the amount of data being shipped.

A *cross-edge* in the input DAG, is an edge whose end-nodes are executed at different systems. It indicates data shipping, since the corresponding output fragment has to be sent from one system to another. We only consider one-way data shipping, that is, from the source to the target. A cross-edge in this scenario is an edge between an operation executed at the source, whose output is consumed by an operation executed at the target. Let *e=(OP1,OP2)* be an edge in our DAG; function *comm_cost(e)* returns some positive value if *e* is a cross-edge, zero otherwise. In our implementation, we use:

$$comm\_cost(e) = \begin{cases} size(OP1.out) & \text{if } e \text{ is cross-edge} \\ 0 & \text{otherwise} \end{cases}$$

where *OP1.out* is the output fragment of *OP1* that is consumed by *OP2* and *size()* is a function that returns the size

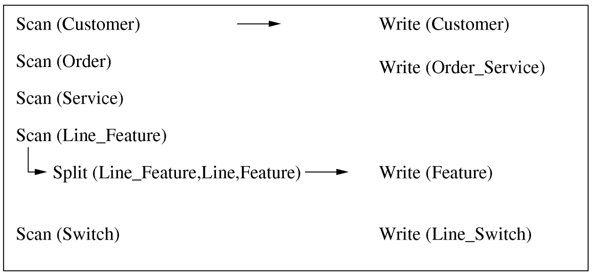

Figure 6. Intermediate Graph G_1

of a fragment.

The execution cost of a program is the sum of the computation and communication costs, weighted accordingly (w_{comp} and w_{com} are weights that might be used to balance computation and communication costs), $G = \{N, E\}$:

$$\begin{aligned} cost(G) = \;& w_{comp} * \textstyle\sum_{OP \in N} comp\_cost(OP) + \\ & w_{com} * \textstyle\sum_{e \in E} comm\_cost(e) \end{aligned} \qquad (1)$$

4.2. Optimal Program Generation

The generation of a data exchange program starts with instantiating a DAG G_0 as follows:

- Create node `Scan`(f) for each fragment f in `S`.
- Create node `Write`(f) for each fragment f in `T`.
- Create a cross-edge between a `Scan` and a `Write` if they both operate on the same fragment.

The next step augments G_0 by adding `Splits` when needed. For instance, `Line_Feature` in the `S`-fragmentation is split to produce `Line` (used to populate `Line_Switch` in the `T`-fragmentation) and `Feature` (used to populate `Feature`). If the output of a split corresponds to a target fragment, for instance `Feature` in this example, we also add an edge between the `Split` and `Write`. This process results in a new DAG G_1, shown in Figure 6.

If every `Write` node in G_1 has an incoming edge, then the final data transfer program is $G = G_1$. If this is not true (like in Figure 6), a final step combines intermediate fragments to generate the input of each *dangling* `Write` node. In Figure 6, fragments `Line` (the result of the `Split`) and `Switch` will be combined using a `Combine(Line,Switch)` and input to `Write(Line_Switch)`. Similarly, `Order` will be combined with `Service` to produce `Order_Service`.

In the general case, a series of pair-wise `Combines` might be needed to produce a composite fragment for a `Write`. This problem roughly relates to the problem of join-ordering in relational query optimization [8]. Each possible combine order results in a different graph instance G.

Algorithm 1 Cost_Based_Optim

```
Require: G
for each $OP$==Write $\in G$ do
    set $OP$.location=T
end for
set OpenProblems={$G$}
while OpenProblems.size() $>$ 0 do
    $G'$=OpenProblems.pop()
    for each $OP \in G'$ with $OP$.location={} do
        set OP.location to S
        {list nodes in all paths from OP to a Write}
        assign all downstream nodes (from OP) to T
        {list nodes in all paths from a Scan to OP}
        assign all upstream nodes (to OP) to S
        if no unassigned operation left in $G'$ then
            compute cost($G'$), keep best seen so far
        else
            add $G'$ to OpenProblems
        end if
    end for
end while
return best program seen
```

A significant difference with relational join order is that we must adhere to the XML Schema tree structure when we combine fragments (Definition 3.7). For example, `Line` and `Customer` have no parent-child relationship and cannot be combined which results in specific join conditions (primary-key/foreign-key joins). This property reduces the size of the search space considerably.

The next optimization step is to decide where to execute each primitive operation of the program (at the source or at the target), so that the overall cost (processing and communication) is minimized. Obvious choices are to place all `Scans` at the source and all `Writes` at the target. The remaining operations, namely `Splits` and `Combines` need more consideration.

Given a graph instance G, we assign each node to the source or the target using the exhaustive `Cost_Based_Optim` algorithm (Algorithm 1). We assume the input graph G has no assigned operation.[1] As an example, if `Combine(Line,Switch)` in Figure 5 is assigned to the source, then all upstream operations (i.e. `Scan(Switch)`, `Split(Line_Feature,Line,Feature)`, `Scan(Line_Feature)`) will be assigned to the source, while `Write(Line_Switch)` will be assigned to the target and its incoming edge will be a cross-edge denoting data shipping between the two systems.

Each program G (generated by different combine ordering) is given as input to the `Cost_Based_Optim` algorithm. The best program is the least expensive one (determined by formula 1) among the ones returned by the cost-based distributed processing algorithm.

[1] Some details are omitted here, for instance the algorithm as presented may enumerate the same program multiple times; we resolve this by marking nodes with the id of the target fragment that the down-stream path leads to.

4.3. Greedy Program Generation

Searching exhaustively all possible programs based on different orderings of combines and different processing distribution alternatives is computationally prohibitive for complex programs. In our tests, we saw that optimal program generation takes too long for XML Schemas with more than 40 nodes. For such cases, we propose a single algorithm that chooses combine ordering and distributed processing greedily.

The greedy algorithm works are follows. Given graph G_1 (i.e. before adding combines), we add combines one by one using the least expensive one first. For estimating its cost, this heuristic assumes the operation is executed at `S`. The result is a single program G to be considered for distributed processing.

The greedy algorithm further expedites the search when doing distributed processing using the following heuristic. For each unassigned operation in the plan, we probe both the source and the target to estimate its cost. The operation OP with the largest absolute difference of the two estimates is the one that will be most affected by a wrong placement. Thus, our heuristic is to fix OP to its location of preference. If this location is the source, all upstream operations in a path from a `Scan` to OP are assigned to the source. If instead, OP is assigned to the target, all downstream operations from OP to a `Write` are placed on the target. Finally, if no difference in the two costs is observed, we make the edge between two unassigned operations $(OP1, OP2) \in G$ a cross edge, in particular the one with the minimum communication cost, i.e., the one that outputs the smallest fragment size ($OP1.out$) and update the location of the upstream/downstream operations. The intuition is that we want to avoid shipping large document fragments. The greedy algorithm terminates when all operations in G have been assigned a location.

5. Experiments

We present an evaluation of our data exchange architecture on a real data transfer between two remote relational systems and demonstrate that our optimizations result in substantial benefits in terms of overall exchange times. We also present experiments using a simulator we developed for exploring data exchange scenarios under a variety of source and target system configutarions.

We used the XMark XML data generator[2] and a subset of the XMark DTD, shown in Figure 7. Leaf nodes are omitted. We created two different relational schemas. Each schema is seen as a fragmentation registered by a system. In both schemas, document structure is captured through foreign keys. The first fragmentation, denoted *MF* (for *Most-Fragmented*), contains a separate fragment for each element in the DTD. The second fragmentation, denoted *LF* (for *Least-Fragmented*), inlines fragments that have an one-to-one relation with their parent. It contains the following three fragments:

1. `SITE_REGIONS_AFRICA_ASIA_AUSTRALIA_EUROPE_NAMERICA_SAMERICA_CATEGORIES_CATGRAPH_PEOPLE_OPENAUCTIONS_CLOSEDAUCTIONS`
2. `ITEM_LOCATION_QUANTITY_INAME_PAYMENT_IDESCRIPTION_SHIPPING_MAILBOX`
3. `CATEGORY_CNAME_CDESCRIPTION`

```
<!-- DTD for subset of auction database -->
<!ELEMENT site            (regions,categories,
                           catgraph,people,
                           openauctions,
                           closedauctions)>
<!ELEMENT categories      (category+)>
<!ELEMENT category        (cname,cdescription)>
<!ATTLIST category        id ID #REQUIRED>
<!ELEMENT cdescription    (id ID)>
<!ELEMENT catgraph        (id ID)>
<!ELEMENT regions         (africa,asia,
                           australia,europe,
                           namerica,samerica)>
<!ELEMENT africa          (item*)>
<!ELEMENT asia            (item*)>
<!ELEMENT australia       (item*)>
<!ELEMENT namerica        (item*)>
<!ELEMENT samerica        (item*)>
<!ELEMENT europe          (item*)>
<!ELEMENT item            (location,quantity,
                           iname,payment,
                           idescription,
                           shipping,mailbox)>
<!ATTLIST item            id ID #REQUIRED
                          featured CDATA #IMPLIED>
<!ELEMENT idescription    (id ID)>
<!ELEMENT mailbox         (id ID)>
<!ELEMENT people          (id ID)>
<!ELEMENT openauctions    (id ID)>
<!ELEMENT closedauctions  (id ID)>
```

Figure 7. DTD used in Data Exchange

Given *MF* and *LF*, we have four potential data exchange scenarios: transfer from *MF* to *MF*, denoted $MF \rightarrow MF$, and similarly $MF \rightarrow LF$, $LF \rightarrow MF$ and $LF \rightarrow LF$.

For these experiments, the source and the target systems were similarly equipped Pentium III 500MHz PCs with 128MB of memory and a 10GB IDE drive running Linux. We installed and used MySQL3.23.49 on both machines. The two machines were physically located in different states in the US and were connected through the Internet.

For the results reported here, we tested transferring a single document of varying size (2.5MB, 12.5MB and 25MB) from one system to the other. Before each run the PCs were rebooted to avoid having the data buffered in memory. The target database was initially empty.

[2] http://monetdb.cwi.nl/xml

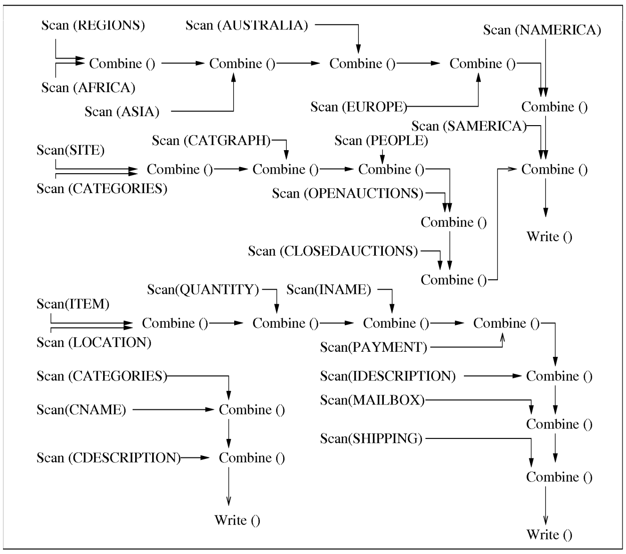

Figure 8. $MF \rightarrow LF$ **Data Transfer**

5.1. Data Exchange using Publish&Map

Publish&Map is obtained by publishing the full XML document at the source and transferring it to the target system where it is stored into relations. Recent research however [6], indicates that naive publishing of the XML document at the source is inefficient. Assuming the data is stored at the source using a relational schema that corresponds to MF, one may simply write a SQL query to obtain a sorted feed (i.e., a fragment) for each element in the DTD. These fragments are then merged and tagged in order to build the XML document. The other extreme alternative is to create the document through a single complex SQL query.

Between these two choices there is a large spectrum of queries that can be used for publishing. For a fair comparison with our system, we explored various ways to do publishing, as described in [6], and picked the set of queries that minimize the overall processing and communication times for publishing the document.

For storing a document in a relational schema (i.e., shredding), we implemented the SAX C API for expat-1.95.1 (*http://www.jclark.com/xml/expat.html*). We used a stack to maintain paths when parsing and discarded the content of the stack as soon as tuples were flushed to files. Shredding time corresponds to the time to parse the document, manipulate the stack and generate ASCII files for a particular fragmentation.

The following steps are required by publish&map for an end-to-end transmission: (1) Execute queries at the source for publishing the document, (2) Tag query results, (3) Ship XML document to target, (4) Parse and Shred document at target, (5) Load shredded pieces to target database and, (6) Update indexes at the target.

Document Size:	2.5MB	12.5MB	25MB
$MF \rightarrow MF$	5.37	25.21	50.42
$MF \rightarrow LF$	6.67	32.89	66.06
$LF \rightarrow MF$	4.21	20.64	41.77
$LF \rightarrow LF$	1.25	14.11	28.55

Table 1. Times (secs) to execute queries (Step 1) in Optimized Data Exchange

Document Size:	2.5MB	12.5MB	25MB
$MF \rightarrow MF$	7.16+7.85	39.76+42.52	87.32+85.83
$MF \rightarrow LF$	7.16+4.66	39.76+41.65	87.32+81.44
$LF \rightarrow MF$	3.13+7.85	6.80+42.52	31.36+85.83
$LF \rightarrow LF$	3.13+4.66	6.80+41.65	31.36+81.44

Table 2. Times (secs) for Publish (first value/Step 1)&Map (second value/Step 4)

5.2. Optimizing Data Exchange

The data exchange architecture of Figure 2 allows the source and target systems to register their fragmentations that are used to generate an optimized data exchange program thus avoiding the need for publishing the whole document. In the case where the source and target fragmentations are the same ($MF \rightarrow MF$ and $LF \rightarrow LF$) the program simply transfers the corresponding fragment instances from one system to the other. In this setup, the program is a series of `Scan(f)`→ `Write(f)` operations. This observation offers an opportunity for parallelism in the execution that we did not pursue here. All pieces of the programs were executed sequentially in all of our experiments.

The exchange when the source and target fragmentations are different are more interesting. Figure 8 shows the program for transferring a XML document in the $MF \rightarrow LF$ setup. It contains one expression per target fragment. The program for $LF \rightarrow MF$ is a mirrored image where each group of `Combine` operators is replaced with a `Split`.

The following steps are required in the optimized data exchange for an end-to-end transmission: (1) Execute queries (parts of the program) assigned to the source, (2) Ship query results to the target, (3) Execute queries (parts of the program) assigned to the target (if needed), (4) Load data to target database and, (5) Update indexes at the target.

Document Size:	2.5MB	12.5MB	25MB
Optimized Data Exchange (Target is MF)	17.85	65.02	131.45
Optimized Data Exchange (Target is LF)	14.96	52.82	101.75
Publish&Map	22.98	81.37	158.65

Table 3. Communication Times (secs)

Document Size:	2.5MB	12.5MB	25MB
MF	3+8.20	29.12+40.32	49.74+121.57
LF	1.06+2.36	10.20+11.62	24.79+33.50

Table 4. Times (secs) to load target db (first value) and create indices (second value)

5.3. Comparison of Times

We ran the Cost_Based_Optim algorithm in the $MF \rightarrow LF$ setup. The output of the algorithm suggested to run the whole data exchange program, except the Writes, at the source (source and target machines are similar), therefore, for data exchange, there were no queries executed at the target (no Step 3). Loading and indexing the target database take the same time in both publish&map and the optimized data exchange since the underlying data is the same. Thus, in this setup, the differences between the two exchange strategies are between steps (1), (3), (4) in publish&map and steps (1), (2) in the optimized data exchange.

Table 1 shows the execution times of the optimized data exchange programs (Step 1) for the four exchange scenarios between *MF* and *LF*. Programs where data is shipped from LF require fewer combines and are, thus, faster. Table 2 shows the corresponding times for publishing (using an optimized set of queries as in [6]) and shredding the document in publish&map (Steps 1,4). The cost of shredding the XML document is significant. In particular, when the source is LF (bottom two rows), it shadows the cost of publishing. In most cases, running the optimized data exchange program compares favorably to the cost of publishing only. We point out here that we tried several optimizations for speeding up shredding but such a difference cannot be erased by simple optimization tricks. Shredding the document requires first parsing it. For that we used a public domain parser (Expat). The times for parsing (that are included in the reported times) were 0.87, 9.08 and 15.14 secs for the 2.5MB, 12.5MB and 25MB documents respectively.

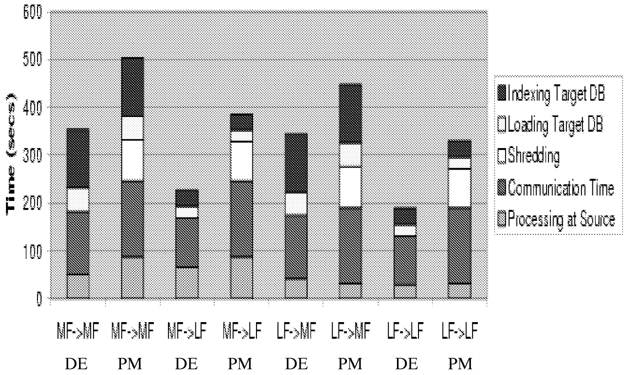

Figure 9. Times for end-to-end transfer

Table 3 shows the times for sending the data through TCP connections over the Internet. In the optimized data exchange, the communicated fragments depend only on the fragmentation on the target, since all combines are done at the source.

Table 4 shows the times for loading and indexing the data at the target database (which are the same for both publish&map and optimized data exchange) and depend only on the target fragmentation and the size of the document.

Figure 9 combines all our measurements for transferring the 25MB document in the $MF \rightarrow MF$, $MF \rightarrow LF$, $LF \rightarrow MF$ and $LF \rightarrow LF$ setups. DE indicates optimized data exchange and PM, publish&map. The optimized data exchange architecture provides saving between 23% and 43% in the overall execution depending on the case. The main reason of this gain is the reduction of the number of operations. For the $LF \rightarrow LF$ case, the optimized data exchange is faster in all accounts. In fact, if we ignore loading and indexing of the target database (which are the same between DE and PM), the reduction in total execution is about 53%.

5.4. Simulation Study

In this section, we present multiple experiments using a simulator that we developed for testing various data exchange configurations. All experiments presented here were run on a 1GHz Pentium-III PC with 256MB of memory running Linux 7.2. All of our algorithms have been implemented on top of this simulator, using the same code-base, thus providing a fair platform for timing the algorithms.

5.4.1. Data Exchange between Systems with Different Processing Power. The purpose of these experiments is to see how the data exchange programs benefit from cost-based distributed processing when the source and target sys-

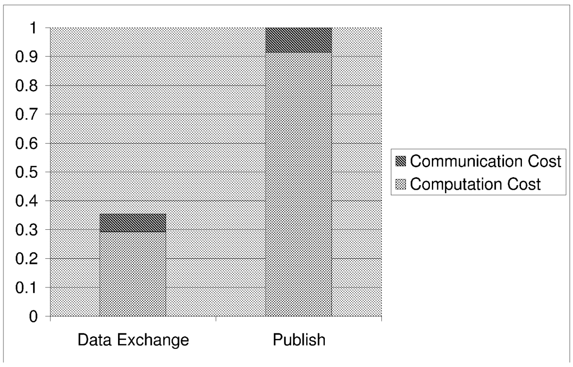

Figure 10. Optimized Data Exchange versus Publishing, similar source and target systems

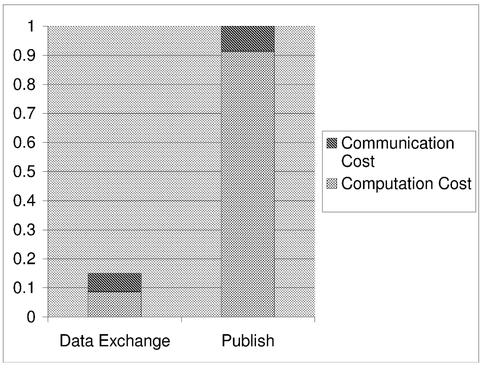

Figure 11. Optimized Data Exchange versus Publishing for fast (x10) target

tems vary significantly in terms of processing power. In what follows the cost of shredding (in publish&map) is omitted.

We first used a configuration in which the server and the target systems are equally fast. The DTD was a balanced tree with 3 levels and fan-our 4. The source and the target contained different complete sets of 11 randomly selected fragments.

Figure 10 shows the relative cost of the optimized data exchange program over the cost of publishing. For publishing, unlike the real measurements in the previous section, we used a single query for producing the document and we did not try optimizing this part. Again, we do not account for tagging. Overall, data exchange compared to publishing only, results in about 65% reduction in the estimated cost of the transfer. This relates to the 53% reduction observed in the real experiment.

One benefit of the data exchange program is that it balances much better than publish&map processing between the two systems. This may not be very important when the source machine is faster (i.e., when we want to do all combines there anyway). If the target system happens to be significantly faster, or if the source is loaded, the distributed processing algorithm may decide to place some of the combines at the target. To demonstrate this, we repeated the same experiment using a target system that was 10 times faster than the source. This resulted in the relative costs shown in Figure 11. The optimized data exchange program provides saving of 85% because it takes advantage of the very fast client and places all combines there.

5.4.2. Evaluation of the Greedy Algorithm. In order to evaluate the quality of the solutions selected by the greedy algorithm for program creation and distributed processing, we compare them to (i) the optimal program produced by an exhaustive search in the space of all solutions and (ii) the worst program that we see in the search space of algorithm `Cost_Based_Optim` (in terms of estimated cost). The comparison against the worst-case program is necessary to assess the opportunity for optimization that both the optimal and the greedy algorithms may build upon.

For these tests, we used a DTD of height 2 with fanout 5, resulting in a tree with 31 nodes. We varied the relative speed of the source over the target as follows: 5/1, 2/1, 1/1, 1/2 and 1/5. That is, we tested a source being five times faster, twice faster, equally fast, twice slower and five times slower than the target system respectively.

Relative speed (source/target)	Ratio Worst/Optimal	Ratio Greedy/Optimal
5/1	1.9354	1.0077
2/1	1.3120	1.0045
1/1	1.0786	1.0095
1/2	1.2269	1.0024
1/5	1.8725	1.0127

Table 5. Ratios of cost of greedy and worst-case programs over the cost of optimal one

For each source and target setup, we used ten different DTDs and documents, each with randomly selected fragmentations at the source and the target. For each document, we computed the costs of the optimal, and worst-case data exchange programs. Table 5 shows the average ratio of these programs over the optimal one. For these documents, optimizing the data exchange program offers a window of opportunity up to 94%, that is the worst-case program has almost twice the cost of the optimal one.

Notice that this window is larger when there are significant differences among the relative speeds of the two systems e.g., when one system is five times faster than the other. When the two systems are equally fast the worst program is near optimal (7.8% difference)! This is explained as follows. In these tests, we assumed a fast interconnect network, so computation cost was the major factor (see also Figures 10 and 11). When the source is as fast as the target, distributed processing (placement of the operations) is not that important and any performance difference arises from optimizing the order of the combines.

Overall, the greedy algorithm produces highly optimized programs, practically as good as the optimal ones. We note here that finding a solution using the greedy algorithm takes a few milliseconds (even on larger DTDs) while algorithm `Cost_Based_Optim` is substantially slower; on the average it took 80.9secs, for each run, in these experiments.

6. Related Work

WSDL is becoming increasingly popular in advertising information sources on the Web. In a recent publication [10], the authors argue that the current WSDL paradigm is inadequate for declaring the large, or even infinite, number of parameterized queries supported by many powerful data sources with structurally rich information. In turn, they propose a WSDL extension called Query Set Specification Language (QSSL) for the concise description of parameterized queries. Our data exchange paradigm involves fewer parameters. Interestingly, QSSL uses a notion of fragmentation of the XML schema but is driven by the input parameters supported by the Web service and concentrates on query rewriting, while our notion of fragmentation is related to the underlying storage and indexing decisions. In [1], the authors present their work in the context of the Active XML project. Web services are described by functions embedded in XML documents. The authors focus on devising a strategy to distribute and replicate Active XML documents in a P2P architecture.

Publishing relational data in XML [3, 4, 5, 6] is a special case of the data exchange problem. In [6], the authors emphasize the need for optimization when publishing relational data as XML documents. Their framework is akin to publish&map where publishing is optimized.

This work could also be seen as a special case of distributed cost-based optimization of relational queries [8, 13]. However, the fact that we manipulate fragments, and thus, limit the complexity of join optimization, reduces the complexity of the problem. A more detailed analysis of complexities could be carried to illustrate this.

7. Conclusions

We presented a novel approach to XML data exchange of large documents based on a middle-tier Web services architecture. The source and target systems can, optionally, specify fragmentations that are then used by the middle-ware to derive an exchange program that combines and splits document fragments as needed. The middle-ware then decides where to execute each operation using a cost model based on the capabilities of the two systems involved in the exchange. Our experiments show that this new data exchange approach is very promising. In the future, we would like to explore solutions to derive the best fragmentation for a system based on its internal indices and data structures.

8. Acknowledgments

We would like to thank Jérôme Siméon, Mary Fernàndez and the anonymous reviewers for their valuable comments.

References

[1] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, T. Milo. *Dynamic XML documents with distribution and replication.* SIGMOD 2003.

[2] S. Amer-Yahia, M. Fernández. *Techniques for Storing XML.* Tutorial. ICDE 2002.

[3] M. Benedikt, C.Y. Chan, W. Fan, R. Rastogi, S. Zheng, A. Zhou. *DTD-Directed Publishing with Attribute Translation Grammars.* VLDB 2002.

[4] P. Bohannon, H. Korth, P.P.S. Narayan, S. Ganguly, P. Shenoy. *Optimizing View Queries in ROLEX to Support Navigable Tree Results.* VLDB 2002.

[5] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, S. N. Subramanian. *XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents.* VLDB 2000.

[6] M. Fernandez, A. Morishima, D. Suciu. *Efficient Evaluation of XML Middle-ware Queries.* SIGMOD 2001.

[7] T. Howes, M. Smith, G. S. Good. Understanding and deploying LDAP Directory Services. Macmillan Technical Publishing, Indianapolis, Indiana, 1999.

[8] Y. Ioannidis, Y. Kang. *Randomized algorithms for Optimizing Large Join Queries.* SIGMOD 1990.

[9] C. Ferris, J. Farrell. *What Are Web Services?* Communications of the ACM 46(6), June 2003.

[10] M. Petropoulos, A. Deutsch, Y. Papakonstantinou. *The Query Set Specification Language (QSSL)* . WebDB 2003.

[11] Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/soap

[12] Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl

[13] M. .T. Ozsü, P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall 1991.