Dynamic XML Documents with Distribution and Replication *

Serge Abiteboul Angela Bonifati Grégory Cobéna Ioana Manolescu Tova Milo*

INRIA Futurs

FirstName.LastName@inria.fr

ABSTRACT

The advent of XML as a universal exchange format and of Web services as a basis for distributed computing, has fostered the apparition of a new class of documents: *dynamic XML documents*. These are XML documents where some of the data is given explicitly while other parts are given only intentionally by means of embedded calls to web services, that can be called to generate the required information. By the sole presence of Web services, dynamic documents already include inherently some form of distributed computation. A higher level of distribution that also allows (fragments of) dynamic documents to be distributed and/or replicated over several sites is highly desirable in today's Web architecture, and in fact is also relevant for regular (non dynamic) documents.

The goal of this paper is to study new issues raised by the distribution and replication of dynamic XML data. Our study has originated in the context of the Active XML system [1, 3, 22] but the results are applicable to many other systems supporting dynamic XML data. Starting from a data model and a query language, we describe a complete framework for distributed and replicated dynamic XML documents. We provide a comprehensive cost model for query evaluation and show how it applies to user queries and service calls. Finally, we describe an algorithm that, for a given peer, chooses data and services that the peer should replicate to improve the efficiency of maintaining and querying its dynamic data.

1. INTRODUCTION

XML has become a standard format for data exchange and new standards have emerged for querying XML data, such as XPath [31] and XQuery [31]. In the spirit of including code in HTML documents (e.g. Sun JSP [15], PHP [27]), one is led naturally to the idea of embedding active fragments in XML documents. Indeed the idea of interpreting or running XML documents is already promoted by technologies such as Macromedia MX or Apache Jelly. This work was developed in the context of the Active XML system and language [3, 1, 22]. The system is centered around Active XML documents, which are XML documents where parts of the content is explicit XML data whereas other parts are generated by calls to Web services (typically performing queries or updates). In the present paper, we are only concerned with certain aspects of Active XML, common to many other systems, like the Macromedia MX and Apache Jelly mentioned above. We will thus use the term *dynamic XML documents* to denote XML documents where parts of the content is materialized XML data present in the document, whereas other parts are generated by calls to programs (typically code for database queries, business logic, or inclusion of graphical plug-ins), when the content of the full document is needed.

We are concerned here in particular with dynamic XML documents where the dynamic part is provided by Web services. Recent standards for *Web services* such as SOAP [31] and WSDL [31] normalize the way programs can be invoked over the Web, and become the standard means of publishing and accessing dynamic, up-to-date sources of information. The increasing acceptance of these standards naturally leads to including into XML documents calls to Web services to capture the non-static parts of the documents. This approach is followed, for example, in the .Net[8] framework, or in DreamWeaver, that recently changed its interface from a Java-based model of "components" to a web-service based one [20]. The aforementioned projects promote dynamic XML documents primarily for presentation purposes. Pushing the idea further, the Active XML system views more generally dynamic XML as an essential tool for distributed data management, and in particular, for distributed data integration[1, 3]. Note that the publication of data via Web services is already a reality (see for instance, the Web service interfaces to Google, to yellow-pages like UDDI, or to geographical data) which further motivates the use of Web services.

The documents we consider are dynamic XML documents, which may be distributed and / or (partially) replicated. In a large-scale data and service integration context, performance, scalability and availability of documents are essential issues. These issues have been addressed by a large body of research on replication and distribution for relational databases [29], client-server datbases [9], and LDAP directories [17]. With rare exceptions such as [7], this domain remains largely unexplored for XML. To some extent, the problem is not specific to dynamic XML documents. Whether dynamic or static, an XML document may be (i) *distributed* in several parts located at different peers, while maintaining the logical unity of the separated pieces, and (ii) partially or entirely *replicated* on different peers. In the case of static XML, one may argue that a distributed XML document is in essence not very different than a distributed LDAP directory. However, the embedding of calls to Web services in a document leads to a number of new interesting aspects of distribution, particular to dynamic documents, that we

*This project is partially supported by EU IST project DBGlobe (IST 2001-32645)

*On sabbatical from Tel-Aviv University

SIGMOD 2003, June 9-12, 2003, San Diego, CA.

study in this paper:
(1) Accessing remote services: Such a document provides the means to access remote services. This feature is already provided by platforms supporting embedded scripts in HTML/XML documents, e.g., JSP, ASP.Net.
(2) Replicating data fragments with embedded service calls: a call included in a replicated fragment may be activated from the replica's site, following a rather different communication path.
(3) Replicating service definitions: A special form of replication may be achieved by replicating not only data, but also service definitions. This is in the spirit of "code-shipping".

When replicating a service definition to a site in order to execute it locally, it may be interesting to replicate as well the data that the service call uses, so that the service is invoked on local data, with less data transfer costs. If the service is *declaratively* specified (e.g. an XML query language) it is possible to infer from the service definition which data is needed, and replicate it accordingly. The issue is then, given a query, to compute a tight superset of the data it needs, avoiding to ship and replicate more than necessary.

We can now state more precisely the context of the paper and its contributions. We are concerned with dynamic XML documents (XML documents including calls to Web services) that are possibly distributed over several sites, with portions of them possibly replicated. In this context, we make the following contributions:
(1) Model. We introduce a simple model for replicating and distributing XML documents over several sites. The model may be used for standard or dynamic documents. In general, users querying distributed/replicated data prefer to ignore data location and expect the system to locate data for them. But it is sometimes desirable to specify which replicas of a given fragment to use (e.g., the one in the local cache, or the most recent one). We thus provide a simple syntax for *location-aware* queries in XPath and XQuery.
(2) Query evaluation and optimization. In the presence of replicas and distribution, many evaluation strategies are possible for a given query, depending on the choice of the replica to use, and of the sites performing each elementary computation. Typically, several peers will collaborate to evaluate a query; each involved peer will have to make choices in order to improve its *observable performance*, based on a cost metric specific to this peer. We provide a complete framework for XML query processing, in the presence of distribution and replication.
(3) Tailored replication. To improve its observable performance, a peer may be willing to replicate some data, possibly including service calls, and even service definitions, as explained above. Such replication is subject to natural constraints (e.g., storage space). We provide a *replication algorithm* that, given a set of peers with replication and distribution, and a specific peer, recommends some replication steps that are guaranteed to improve the peer's observable performance. This algorithm is specially tailored to suit replication of dynamic XML documents.

XML and Web services are predicted a very promising future, and XML documents with embedded calls to web services are already found in several existing products (see the examples above). In the line of previous works on distribution and replication of relational or LDAP data, distribution and replication of XML documents, whether dynamic or regular, is becoming an essential component of distributed data management. This motivates our work. Although the techniques we introduce are set in the context of a very precise model, we believe that they are pertinent to a very large number of applications dealing with the management of distributed XML data with replication and the access to such resources via Web services.

The rest of the paper is organized as follows. Section 2 introduces, through an example, the data model for distributed and replicated dynamic XML documents, and the specific query constructs needed in this context. Section 3 presents the cost model that we use to determine the queries cost. Then, Section 4 shows how queries are evaluated at runtime, whereas Section 5 provides a dynamic replication algorithm aiming at reducing the global costs observed at a given peer. Note that the main difference between query optimization and data replication is that the first one is a short-term investment, whereas the second one is a long-term investment that will benefit to many query executions. After considering related work in Section 6, we conclude.

2. DATA MODEL AND QUERY LANGUAGE

We start by describing our data model for dynamic documents and highlight the particular issues that we study in the paper. Then we consider the particular query constructs required in this context. Due to space limitations, the presentation is mostly informal and based on a running example that will be used throughout the paper.

We consider here dynamic XML documents that are XML documents possibly containing calls to functions provided by web services. When calls included in a dynamic document are executed, the latter is enriched by the corresponding results. In some sense, such a document may be seen as a (partially) materialized view, integrating plain XML data and intentional data obtained from function calls. The function calls inside the documents typically have input parameters (which are passed to the service being called). They also contain (in form of attributes) indication on when the calls should be activated (e.g. hourly, daily, only when the data is needed) and for how long the returned data is considered valid (e.g. until the call is invoked again and fresher data is available, forever - namely all the returned data is accumulated, etc.).
Dynamic XML Documents As in the standard XML data model, a dynamic XML document may be viewed as a labeled tree. The tree nodes represent the XML elements/attributes (with the labels denoting the elements/attributes name/value) and the edges representing the component-of relationship among document elements. A particularity here is that some elements, called *function* elements, have a particular interpretation and represent calls to web service functions. We will see their exact structure in the sequel. Since we are mostly interested here in *dynamic* XML documents, unless stated otherwise whenever we say further *a document*, we mean a dynamic XML one.
Web Services We will consider two kinds of web services. (1) Regular SOAP-based web services, which are essentially viewed as black boxes. Via their standard WSDL specification [31], one can find the type of their input and output, but essentially nothing is known about their actual internal implementation. (2) Declarative web services whose implementation is known and described in terms of XQuery [31] queries on top of dynamic XML documents. To the outside world, declarative services look just like the regular ones: They are wrapped and exposed to the world via a WSDL specification. But, as we shall see, the extra knowledge about their internal structure allows to better optimize their usage.
Peers A *peer* in our context offers some web services and contains some dynamic XML documents which may include calls to services provided by the same or other peers. To support data distribution and replication among peers, (as will be explained further), all the document elements in a given peer are assumed to have unique identifiers, different from those of other peers. This can be achieved, for instance, by prefixing the identifiers by the peer name. This is to simplify the presentation since in reality only some particular elements that are replicated need to be named.

Distribution Since dynamic documents may contain calls to services on other peers, some minimal form of distributed computation is inherently part of this basic model. A higher level of *data* distribution can be achieved by allowing a *document* to be distributed over several peers. In terms of the tree data model mentioned above, this means that document nodes may now have *external children edges* pointing to children nodes on other peers, and analogously, an *external parent edge* if the parent of the node is on another peer. We will provide further a syntax for representing such external edges.

Note that there is a fundamental difference between distributing document fragments among several peers via external edges, and connecting two nodes in remote documents by an XLink[31]. No restriction is imposed on the nodes that can be connected by XLink; in particular, it is possible that by following XLink edges, one encounter a cycle. In contrast, a distributed XML document is always a *document*. Namely if the distinction between different peers is ignored and the whole world is seen as one unique peer, the document becomes just a regular (dynamic) XML document with standard tree structure.

Replication of data and services To support data replication, we allow the same document fragment to exist in several peers. More precisely, recall that document nodes have identifiers. All children of the same node with the same ID are considered replicas of a single node. Analogously, a node may register more that one parent node, in which case, these nodes are considered replicas of the parent node. The general graph structure may no longer be a tree but a DAG. However, we require that it is still *representing* a document. We will come back to this aspect at the end of the section.

Observe that, in general, replicas of a given node in distinct peers may look different, either because only part of the node's data was replicated (e.g. only a subset of its children) or due to local updates that have not yet been propagated. Much research has been devoted to the issue of change control for replicas and conflict resolution in the context of replicated data. Many of these techniques can be used here, for instance [18]. Since this is not the focus of this paper, we simply assume in the sequel that, when needed, one of these algorithms is employed. Nevertheless, we will explain below how, by a simple extension to XQuery, the user can get the means to explicitly specify the location of data, when some particular copy (e.g. the "master" copy, or the most recent one) is preferred over the others.

Finally, a special form of replication can be achieved by replicating not only document fragments, but also *web services*. This can be used in particular for *declarative web services* whose definition is given in terms of XQuery queries. When the data used by the service (or some part of it) is replicated to another peer, the service may be replicated as well, with its definition adjusted such that, when operating on the replicated data, it returns the same answer as it would have return if evaluated on the original data (assuming the two copies are in sync.) We omit here the formal definition of this notion of replication and will instead illustrate next the main points via a running example.

2.1 Example

In the example, we start by considering a simple dynamic XML document, without distribution and replication. Then we show how some of its data fragments (and services) can be replicated or distributed, and discuss advantages of such a replication or distribution.

Consider a ski portal providing information about the various states and their ski resorts. The portal contains the (dynamic) XML document given in Figure 1. W.r.t the standard XML data model[31],

```
<document name="SkiPortal">
  <state> <state_name> Colorado </state_name>
    <resorts>
      <resort ID="AspResort"> <name> Aspen </name>
        <snow_cond ID="AspSC"> good
          <fun peer="UnisysWeather" fun="SnowConditions"
               frequency="every round hour"
               validity="last">
               <params><resort> Aspen </resort></params>
          </fun>
        </snow_cond>
        <hotels ID="AspHotels"> <hotel>...</hotel>
        </hotels>
      </resort> <resort> ... </resort> ...
  </resorts>
  </state> <state> ... </state> ...
</document>
```

Figure 1: A dynamic XML document of the Ski Portal.

```
function OperativeSkiResorts($state)
implementation:XQuery
for $x in document("SkiPortal")/state[state_name=$state]
        /resorts/resort[snow_cond/value()="good"]
return $x

function HotelsInfo($state,$resort)
implementation:XQuery
for $x in document("SkiPortal")/state[state_name=$state]
        /resorts/resort[name=$resort]/hotels/hotel
return $x
```

Figure 2: The web services of the Ski Portal.

dynamic documents contain, besides regular data elements and attributes, some special function elements denoted by the tag *fun*. As can be seen, the document provides for each state, the state name and information about its ski resorts, including in particular the resort name, the current ski conditions, and the listing of hotels. Some of the data is given explicitly (i.e., the states and resort names and hotel list) while some is obtained dynamically via function calls. For instance the snow condition in each resort is obtained by querying *Unisys Weather*. This data is refreshed once an hour. To support data distribution and replication, all elements are assumed to have unique identifiers. For brevity we detail explicitly in the example below only some of these ids (e.g. in the `resort`, `snow_cond`, and `hotels` elements) while the rest are assumed implicitly and omitted from the text.

To simplify the presentation we first assume below that the document is not updated by any other means, namely the changes in it are all due to functions invocation. We will remove this restriction afterwards and consider consequences.

The dynamic part of the data is obtained by calling functions provided by web services. As already mentioned, we consider in this paper two kinds of web services. First, there are opaque ones such as the `SnowConditions` function. It gets a `resort` element as input, returns a string (e.g.,: "good", "bad") as output, and no information is available regarding its the actual implementation. We also consider declarative web services, whose internal specification is known and given as an XQuery [31] query on top of dynamic XML documents. For instance, the ski portal above may provide a web service with the two functions specified in Figure 2. The first retrieves, given a state name, all the "sky-able" resorts in that state, namely the resorts with good ski conditions. The second, retrieves the hotels of a given state and resort names.

To the users, declarative services look just like the opaque ones. They are wrapped and exposed to the world via a WSDL specification. However, the fact that the internal specification, and the data being used is known, will prove to be very beneficial for optimization purposes, and in particular for deciding which data needs to be distributed or replicated and where.

To continue with the above example, consider a state ski center, say of Colorado, that assists visitors in selecting good ski resorts in Colorado and helps them find hotels near these resorts. The ski center uses frequently the two service functions provided by the ski portal, namely OperativeSkiResorts("Colorado"), and HotelsInfo("Colorado",$resort), where $resort is the name of some operating resort (returned by the first function) that the visitors are interested in.

There are a number of possibilities for running these requests. If the two functions were *opaque* and the resort knows nothing about their internal implementation, there are essentially two possibilities: (1) Call the ski portal *each time* a service is needed and have the portal compute the answer and return it, or (2) *cache* the returned result and use it for some time, trading communication cost for data accuracy.

For declarative services when their specification is exported, a few other possibilities open up.

Query Frequency First, we have more precise information regarding the validity of the cached data: by analyzing the OperativeSkiResorts query, we can see that its answer may change only every hour - when the SnowConditions functions is invoked. Hence, to give fully accurate answers to its visitors, the ski center needs to invoke the function every hour, and cache data in between. Naturally, the communication could be further reduced if some form of "update propagation" mechanism were available, allowing to inform the ski center on the changes in the portal data, and thus save the full call re-invocation.

Replicating relevant data and services But even if such an update propagation mechanism is not available, the communication cost can still be reduced by providing the ski center sufficient information to compute the correct answers by itself. Assume that the Colorado ski center computer is capable of (1) storing dynamic XML documents, (2) invoking the web service calls embedded in them, and (3) processing XQuery queries. Rather than just caching the current query result, one could then decide to replicate (and maintain) in the ski center computer all the relevant data, and provide a local version of the service queries. To continue with the above example, it suffices to store at the ski center a portion of the ski portal document, with the local variants of the OperativeSkiResorts and HotelsInfo services, as depicted in Figure 3.

The communication is now reduced to querying Unisys Weather once an hour for all resorts in Colorado, rather than re-shipping every hour, for all operative resorts, the full resort information. On the other hand, more storage and processing resources are needed.

Observe the approach we take to distributing data. In the example, the dynamic parts of the copied data are all computed using *UnisysWeather* (an opaque service here). In general, the copied data may also contain calls to non-opaque declarative services. In this case, the process described above may be repeated for those services as well, i.e., we may replicate locally the data they use as well as the service code. This results in transforming recursively these service calls into local calls. The main difficulty in the context of turning external calls into local ones is (i) in deciding which service should be made local, and (ii) determining how to update the service code, which data should be replicated locally, and which other services should then be made local also (and so on recursively). These computations are based on the peer capabilities and the tradeoff between the resources (typically communication vs. storage and computation).

The main contributions of this paper are the definition of an appropriate cost model and the design of optimization algorithms to guide this choice.

```
<document name="ColoradoSkiCenter">
  <resort ID="AspResort"> <res_name> Aspen </res_name>
    <snow_cond> good
      <fun peer="UnisysWeather" fun="SnowConditions"
        frequency="every round hour"
        validity="last">
        <params><resort> Aspen </resort></params>
      </fun>
    </snow_cond>
    <hotels ID="AspHotels"> <hotel>...</hotel>...
    </hotels>
  </resort> <resort> ... </resort> ...
</document>

function OperativeSkiResorts("Colorado")
implementation:XQuery
for $x in document("ColoradoSkiCenter")/
        resort[snowCondition.value()="good"]
return $x

function HotelsInfo("Colorado",$resort)
implementation:XQuery
for $x in document("ColoradoSkiCenter")
        /resort[res_name=$resort]/hotels/hotel return $x
```

Figure 3: The Colorado dynamic document and services

```
<document name="ColoradoSkiCenter">
  <resort ID="AspResort"> <res_name> Aspen </res_name>
    <snow_cond> good
      <fun peer="UnisysWeather" fun="SnowConditions"
        frequency="every round hour"
        validity="last">
        <params><resort> Aspen </resort></params>
      </fun>
    </snow_cond>
  <hotels ID="AspHotels">
    <externalURL> http://www.ski.com/SkiPortal
    </externalURL>
  </hotels>
  </resort> <resort> ... </resort> ...
</document>
```

Figure 4: The Colorado document with external edges

Partial replication In the example above, the full subtree rooted at the `state` element was copied at the local ski center. There are cases where one would prefer not to replicate so much data, e.g., when storage space is limited and hotel listings for the resorts only rarely used. In such case, it suffices to replicate just the resort names and their ski conditions, without the hotels data, and just provide access to this data through the ski portal, when needed. This can be achieved using *external edges*, as illustrated in Figure 4.

The externalURL sub-element of the hotels element, together with the ID indicate where the data of this element may be found. From a semantic view point, the actual physical location of the data is irrelevant, i.e., the "value" of the document is as if the data was physically there. The external edge is simply viewed as an intentional description of this missing data and gives the means to obtain it if needed.

Observe that while the above external edge points to "real" XML data, one can, in the same way, point to dynamic external data. For instance, if *Unisys Weather* charges money for its services, one may prefer to leave the dynamic snow_cond element on the ski portal and just point to it via an external edge:

```
<snow_cond ID="AspSC">
  <externalURL> http://www.ski.com/SkiPortal
  </externalURL>
</snow_cond>
```

The use of external edges brings new challenges to query pro-

```
<document name="SkiPortal">...
  <resort ID="AspResort">
    <snow_cond ID="AspSC">
      <LRUlanretxe> http://www.HS.com/ColoradoSkiCenter
      </LRUlanretxe> ...
    </snow_cond>
    <hotels ID="AspHotels">
      <LRUlanretxe> http://www.HS.com/ColoradoSkiCenter
      </LRUlanretxe> ...
    </hotels>
  </resort> ...
  </document>
```

Figure 5: Inverse external edges

cessing. The query processor should follow external edges as if they were locals, and the query computation should continue at the remote peer. For instance, consider the Colorado HotelsInfo service from Figure 3. The query starts at the root of the ColoradoSkiCenter document. When reaching the hotels item, the external edge is traversed. The remaining part of the query (i.e. the path expression hotels/hotel) is evaluated at the *Ski Portal* peer and the resulting hotel items are sent back to the ski center computer. The query is thus "split" into two parts, each being executed on a different peer, with the communication between the peers consisting of query shipping, in one direction, and result shipping, in the other way.

The cost model and query optimization algorithms, that we present in the sequel, account for such query splitting, for both the static and the dynamic parts of the data..

In the same way, if several external replicas exists, one may record them all (or some preferred subset) by listing the relevant URLs. The query processor may then choose any one of these replicas. In all replicas are in sync, this does not matter semantically, i.e., an equivalent result is obtained no matter which replica is used. But it certainly may effect the performance. When the replicas are not synchronized, we may want to give the user the possibility to specify explicitly which copy is preferred. We will consider these issues in more details further.

Remark: The external edges illustrated above allow a node (e.g. resort node at the Colorado document) to point to its external children (the hotels and snow_cond elements of the ski portal document). Similarly, a node may have an external parent. Such external *inverse* edge can be recorded as depicted in Figure 5 via the LRUlanretxe sub-element (the inverse of " externalURL"). Note that now, a query that traverses the tree upwards and crosses one of these elements has a choice whether to continue the computation with a local parent resort element (if it exists), or cross the external edge to an external one.

We assume that if a document points to external data the other side also records the inverse edge. Note however that one may still have replicas that do not point to each other. The intuition is that when part of a document replicated to a distant peer, the owner may or may not be in interested in keeping the connection between the distant copies (e.g. in the style of caching). But when such a connection is desired, we assume it to be symmetric for both peers.

Updates and "master/slave" policy So far we assumed that the only changes in the documents are due to function invocations. When updates are allowed things get naturally more complex. Maintaining consistency over replicated objects is a well-known difficult issue [18]. A typical solution, which is quite acceptable in P2P environments, is to have each object owned by a single master who is in charge of maintaining the various copies in sync. If the various copies are the children of a single element, then this element is naturally the candidate for being in charge of synchronization, i.e., for being the master of these replicas. Other policies may be preferred

```
<document name="SkiPortal">
  <state> <state_name> Colorado </state_name>
    <hotels ID="AspHotels" status="stale">
      <externalURL status="master">
        http://www.HS.com/ColoradoSkiCenter
      </externalURL>
      <hotel>...</hotel>...
    </hotels>
  </state> ...
</document>
```

Figure 6: Master and stale replicas

in some cases.

For instance, going back to our example, assume that the Colorado ski center is in charge of updating the hotel information for the state. One option is to put the state's hotels information at the Colorado peer and simply have the ski portal point to it via external edge. If the ski portal users are willing to settle for less accurate information in return to faster response, the portal could keep a (possibly stale) replica of the data and only refresh it periodically. Note that the portal may also decide to keep both a (possibly stale) copy of the local data and an external edge to the master copy, as illustrated in Figure 6. Depending on particular user needs, one or the other will be chosen.

Remark (consistency of documents with distribution): Consider a dynamic document in our framework. It may now spread over several peers and have replicated fragments. We insist that the document be consistent. More precisely, we insist that its *collapsed version*, i.e., the graph obtained by merging all nodes that are determined to be replicas (children of two nodes with the same ID or two parents of the same node) should represent a regular tree-structured dynamic XML document. In the present paper, we will simply assume that this is the case, i.e., that all our distributed documents are consistent. It is easy to design a distributed algorithm to determine whether this is the case but running such an algorithm may be costly. This cost may be greatly reduced by forbidding a single node to have two distinct parent replicas. Now, in a typical application, a distributed document evolves over time from (a) updates at the various peers, and from (b) adding or removing data replication among peers. Some operations requiring extensive global checks of consistency should be avoided as much as possible. To guarantee that changes yield a consistent document (without actually having to perform expensive verification), updates should be restricted to a *safe* set of operations. For instance, external edges should be added only as part of a global operation that guarantees the bi-directionality of links.

2.2 Queries

In the usual XML context, the evaluation of an XQuery query involves traversing the document tree and selecting nodes based on the path expressions and filters appearing in the query. With distributed/replicated documents, things become more complex. Each element encountered in the evaluation of a path expression, on a given peer p, may contain some data (residing on that peer), and may also point (via external edges) to some replicas (on different peers). The question is which of the element versions should be used. Some of the possibilities are the following.

(1) One can choose to ignore all the external edges and consider only the data residing within the given peer p. For instance if communication cost is very high and the local information is known to be sufficient.

(2) At the other extreme, one may want to use the element's local data as well as follow *all* the given external edges to its replicas, in order to get the maximal available information. This seems a

```
for $x in document("SkiPortal")/state[state_name="Colorado"]
        /resorts/resort
replicate $x with resort_name//*
                  snow_cond//*
                  hotels as external link
at peer "http://www.HS.com/ColoradoSkiCenter"
```

Figure 7: A replication query.

common choice for P2P architecture. In Gnutella [11] for instance, the queries are forwarded to all known neighbor peers, and so on recursively until a time-to-live expires.

(3) An intermediate choice is to (a) choose some arbitrary copy (for instance the cheapest one, based on some cost metrics), or perhaps (b) consider the element's local data when available, and follow an external edge (e.g. the cheapest one) only when such local information is not available.

(4) One may also be interested in following a particular edge (for instance (a) "leading to a replica on some specific peer p_i", or (b) "a *master* replica, if such exist among the given edges").

(5) Finally, one may want to give a preference list, for instance "use the master version if such exists among the pointed replicas; otherwise use the local data if available; and if not use the cheapest external one."

While often users would like to let the system choose which version to use, based on some predefined policy/cost scheme, it is convenient to have the means to describe such policies declaratively, as well as override them when needed. To support this, we use XQuery$_{dr}$, a natural extension of XQuery, that allows to account for distribution and replication. We detail bellow how the *for/let* clause, the *where* clause, and the *return* clause of XQuery are extended. Next we consider a cost model for XQuery$_{dr}$ queries.

The for/let clause is similar to the one of XQuery except that path expressions (and their subexpression) can now be annotated by *peer-qualifiers* (to be defined below). For instance rather than using a path of the form $p = path_1/path_2/\ldots/path_n$, where $path_i$, $i = 1\ldots n$, are standard XPath expressions, we can use $p' = \{path_1\}@P_1/\{path_2\}@P_2/\ldots/\{path_n\}@P_n$, where the P_i's are peer-qualifiers. The intuitive semantics is that, for each element encountered in the evaluation of $path_i$, the only replicas being considered for the evaluation of the remaining path are those residing on peers that satisfy the corresponding peer-qualifier P_i.

The peer-qualifiers can in general be any boolean function over elements/external edges; we use a shorthand notation for some common qualifiers. In particular, we use local, all, any, localORany, p$_i$, master, and masterORlocalORany to denote the selection criteria described in items **1**, **2**, **3(a)** and **3(b)**, **4(a)** and **4(b)**, and **5** above. When no peer-qualifier is used for the path expression the default interpretation is any. For instance, the path expression

{document("ColoradoSkiCenter")
/resort[resort_name="Aspen"]/hotels/hotel}@local,

when evaluated on the Colorado document of Figure 4, returns the empty answer since the hotels element on that Colorado peer contains no local data. On the other hand the two path expressions

{document("ColoradoSkiCenter")
/resort[resort_name="Aspen"]/hotels/hotel}@localORany

and

{document("ColoradoSkiCenter")
/resort[resort_name="Aspen"]}@p$_1$
/{hotels}@p$_2${hotel}@local

where p_1 and p_2 are the Colorado and ski portal peers, resp., (with documents as in Figures 3 and 1), both return the set of hotel elements at p_2. Finally, the query

{document("SkiPortal")
/state[state_name="Colorado"]/resorts
/resort[resort_name="Aspen"]/hotels
/hotel}@masterORlocalORany,

when evaluated on the ski portal document of Figure 6, returns the master copies of the hotel elements, from the Colorado peer.

Note that, if all the data replicas are in sync, all the peer qualifiers considered above, besides local, can be used interchangeably: they all return (replicas of) the same result; in fact the same result that would be obtained if the path expression where to be evaluated on the "collapsed" version of the data.

The where clause in XQuery can compare the nodes assigned to the query variables, e.g. where $x = $y. For regular XML documents the semantics is clear: the equality is satisfied when $x and $y are assigned the same document node. In the presence of replication one can distinguish two cases. (1) $x and $y contain replicas of the same node (namely have the same node id but may reside on distinct peers), and (2) $x and $y contain the same node replica (namely same id and same peer). In XQuery$_{dr}$ we use $=$ to denote to the former type of equality and $==$ for the later.

The return clause. As in standard XQuery, the document fragment associated with each variable assignment in XQuery$_{dr}$ consist of the subtree rooted at the assigned node, on the peer where the node resides, (plus the external edges embedded in this data, if such exist.) The semantics of the return clause is standard, with one notable exception: all elements in the result are assigned fresh new ids. This is compliant with the usual XQuery semantics where the answer forms a *new* document.

Remark (replication) We conclude this section with a remark regarding data replication. XQuery$_{dr}$ queries, as described above, return new documents. A similar syntax can be also used for *replicating* existing data. An analogy to the return clause, XQuery$_{dr}$ introduces a special replicate clause which operates like the return except that the element ids in the result are preserved.

A replicate $var at peer p$_i$ clause copies into peer p_i the subtrees rooted at the nodes assigned to $\$var$ (fusing nodes with common IDs). A more selective replication, copying only parts of these subtrees, can be performed by adding to the replicate clause a with statement that describes, using a path expression, the specific data to be copied and whether the remaining non copied one should be pointed to via external links or not.

For instance, Figure 7 replicates, to the Colorado peer, all the resort elements in the Colorado state, yielding the document depicted in Figure 4. For each resort, the nodes residing along the paths resort_name//, snow_cond//, and hotels are replicated as well. Namely, the resort_name and *snow_cond* subelements are replicated with all their rooted subtrees, while the hotels element is replicated alone, without its rooted subtree. For the last one, an external link pointing to this external subtree is added instead.

The syntax and semantics of our with clause is similar to the one introduced in [14] for the definition of views over semi structured data. Indeed, replicas can be considered as materialized views of the external data. For space reasons we omit the full syntax here.

3. COST MODEL

A set of peers, each containing some data and providing some web services (opaque or XQuery-based ones), is called a *configuration*. For a given configuration, the system *workload* consists of the service calls invoked by the dynamic documents in the configuration, as well as of queries/web service requests posed by users at the various peers. The goal of this section is to establish a model for evaluating the costs entailed by such a workload.

In a peer-to-peer environment it is unrealistic to hope to optimize

```
for $x in document("ColoradoSkiCenter")/resort
        [resort_name=$resort]/hotels/@ID
return $x

for $y in document("SkiPortal")//hotels[@ID=$x]/hotel
return $y
```

Figure 8: Intra-peer sub queries at the Colorado peer (top), and USA Ski Portal (bottom) peer

the overall system performance. Indeed, our cost model intends to reflect the *observable performance* of a given peer: the costs and performance metrics perceived by, and important for, this particular peer. This observable performance is influenced by some *objective* parameters (e.g. size of data transfers, from/to a given peer, incurred by the execution), and some *subjective* parameters (e.g. the relative impact of communication, space, and computation costs on the overall cost afforded by the peer.) This two types of parameters will be incorporated in our cost formulas.

Before presenting the cost model, let us first explain two of the main ideas guiding its particular design.

Unifying user queries and services. As explained above, the workload in a given configuration consists of (1) the invocation of web services entailed by the dynamic documents, and (2) queries and web services requested by the user. The frequency of the former is specified in the dynamic documents while that of the latter is determined by the user needs. But, ignoring this minor difference, they can all be thought of as *requests* with some associated *frequency*. With this view, user queries and XQuery-based services are naturally unified. Opaque web-services can be seen as "black-box" queries whose actual code is unknown. *We thus model a given workload as a set of queries (having known or unknown code), each invoked at some given peer, and having some associated frequency.*

Decomposing queries on peers. Consider a query Q invoked at some peer P_1. The processing of Q may involve some local data of P_1 as well as some remote data pointed by external links and residing on another peer P_2. Q may need to consult this data (and, if it points to additional peers, their data as well, recursively). The processing of Q can thus be viewed as decomposed into several "intra-peer" sub-queries: Each such sub-query is evaluated in a particular peer, consulting only the peer's local data, and communicates with other peers in order to forward some finer sub-queries or send/receive data or computation results. *Consequently, rather than defining the cost of a given workload directly, for the original user/service queries, we will look at their decomposed version and define the cost via these smaller, intra-peer queries.*

For instance, consider the Colorado document in Figure 4 and the HotelsInfo query in Figure 3. A possible way to evaluate the query is to decompose its processing into two parts. The first one queries the Colorado document, using the query depicted at the top of Figure 8; it retrieves the id of the remote hotels and asks the USA Ski Portal to evaluate the rest of the query, starting from this id. The USA Ski Portal then evaluates the query depicted at the bottom of Figure 8 and returns back the answer.

For "black box" queries (representing opaque web services) we cannot know if, or how, they are decomposed, and will thus consider them as being evaluated locally at the peer that provides the service. For regular queries, we defer the presentation of the decomposition algorithm to Section 4; we will assume, in the rest of this section that we are given that decomposed workload, and provide a cost model for such workloads. Our cost model will nevertheless capture the overall cost of the original workload. For that, it uses various elementary parameters such as the computation power consumed by each intra-peer query at its given peer, the size of the data transfered between various intra-peer queries etc.

How the value of these parameters is obtained for each decomposed query will be shown in Section 3.3 and Section 4.2. For now, we just assume that they are given, and explain below their particular role in the cost model.

Remark: To conclude the discussion, recall that service calls may have parameters. To fit into the above picture, each of the parameters can be viewed as another query whose output is sent to the service query before its evaluation.

We are now ready to define the cost model. We first describe in Section 3.1 the various (subjective and objective) parameters used by the cost model. Then Section 3.2 presents the cost formulas that aggregate these elementary cost components. Finally, Section 3.3 describes the statistic and cost parameters available at every peer; however, running a query on documents of several peers requires some distributed knowledge, which has to be gathered by the peers through a collaborative process. It turns out that this process is similar to the process of query evaluation itself, and therefore we present them both in Section 4.

We will use below the following notations. We denote by $M[r, c]$ a matrix M of size $r \times c$, and by $M_{i,j}$, where $1 \leq i \leq r, 1 \leq j \leq c$, its component at line i and column j. M_j denotes its j^{th} column. Similarly, $V[r]$ denotes a vector V of r components, and V_i denotes its i^{th} component. A (decomposed) workload $\mathcal{W}$ with n intra-peer queries is represented by an n-ary vector $\mathcal{W}[n]$, each component $\mathcal{W}_i$ being an intra-peer query. The peer of each query is recorded in a matrix $L[n, s]$, s being the number of peers, where $L_{i,j} = 1$ if W_i runs at peer P_j, and $= 0$ otherwise. Note that each row of the matrix contains only one 1 value.

3.1 Cost Model Ingredients

The following *objective cost parameters* are needed to characterize a given workload $\mathcal{W}$:

Frequency vector F. To each workload query $\mathcal{W}_i$, a frequency $freq(W_i)$ is associated, describing how many times the query is asked/the service call is invoked, on average, during a day. A vector $F[n] = [freq(\mathcal{W}_1), \ldots, freq(\mathcal{W}_n)]$ records these frequencies.
Data dependency matrix δ. Recall from above that producer consumer dependencies may exist among the decomposed sub-queries / service calls, where one takes as input data output by the other. These dependencies are described in data dependency matrix $\delta[n, n]$, where $\delta_{i,j}$ is a real number between 0 and 1, specifying which fraction of the output of $\mathcal{W}_i$ is used as input to $\mathcal{W}_j$.
Output vector O. We use a vector $O[n]$ detailing the actual size (in Kb) of the output of each workload query $\mathcal{W}_i$. Normally, one also needs to consider the size of the input received by each W_i. By definition of O and δ, the input vector is given by $\delta * O$.
Computational and space costs. Each query W_i requires some CPU effort and some intermediate storage space on the peer on which it runs. These are described in two vectors $Comp[n]$ and $Space[n]$, where $Comp_i$ and $Space_i$ detail, resp., the number of CPU cycles and space consumption (in kilobytes) of W_i.

The following are the *subjective parameters (or weights)* influencing the observable performance for a given set of s peers.

Communicating cost weight. Two vectors $BW^{IN}[s]$ and $BW^{OUT}[s]$ describe the relative importance, for each peer, of the volume of the received, resp. sent data.
Space cost weight. We use a vector $sp[s]$ to record the relative

importance of the consumed storage space at a given peer.
Computing power cost weight. A vector $cp[s]$ details the relative importance of computing power consumed on each peer.

The components of all vectors above are numbers between 0 and 1, which allow *weighting* the relative costs of communicating (resp. storage and power) among different peers. The higher are these numbers, the more expensive is for the peer to afford the communication, storage or computing tasks (measured in Kb, Kb and CPU cycles respectively).

3.2 Cost Formulas

Using the previous objective and subjective parameters, we are now able to compute the global costs incurred by the workload.

The formulas for calculating the data used by a given workload on a set of peers are the following:

$$M_{i,j} = \delta_{i,j} * O_j * min(F_i, F_j) \quad (1)$$
$$D = {}^{T}L * M * L \quad (2)$$

In the above formulas, (1) defines the inter-query data transfer matrix M, where $M_{i,j}$ is the volume of data transferred from one query $\mathcal{W}_i$ to another query $\mathcal{W}_j$. (2) defines the inter-peer data transfer matrix, where $D_{i,j}$ represents the volume of data transferred from peer P_i to peer P_j due to *all* queries in $\mathcal{W}$.

Remark In a peer-to-peer context, it may be possible that not all parameters in the formulas above are known. It may happen that the size of data transfer between two sites is unknown or unpredictable. Our hypothesis is that each peer has some cost knowledge about its "neighborhood" (e.g. its LAN), and, as a consequence, is always eager to delegate work rather to peers that it "knows" (as opposed to unknown peers with unknown performances).

At this point, we can define the computation, communication and storage costs incurred by the workload. These are represented by the four vectors $\mathcal{C}^{GlobComp}[s]$, $\mathcal{C}^{GlobReceiv}[s]$, $\mathcal{C}^{GlobSend}[s]$, and $\mathcal{C}^{GlobSpace}[s]$. The j^{th} components of each vector describes the observable cost of computation, received data, sent data, and space, resp., of peer j. Weights are used to adapt the result to the cost investment of each peer. The vectors are computed as follows.

$$\mathcal{C}_j^{GlobComp} = \{Comp * L\}_j * cp_j \quad (3)$$
$$\mathcal{C}^{GlobReceiv}[s] = D * BW_{IN} \quad (4)$$
$$\mathcal{C}^{GlobSend}[s] = {}^{T}BW_{OUT} * D \quad (5)$$
$$\mathcal{C}_j^{GlobSpace} = \{Space * L\}_j * sp_j \quad (6)$$

3.3 Local Cost and Statistic Information

Each peer has information about the following cost parameters. The *frequency* of queries can be easily inferred by execution traces, while the frequency of service calls is encoded in the dynamic documents. The *computational cost* vector is obtained for each query by measuring the CPU effort required to execute it. A slightly more difficult task is to measure the output data size and the space cost vector (which takes into account the sizes intermediate results). To that purpose, each peer provides estimates of the *cardinality* of a given path expression over the documents of that peer. The simplest way to implement such estimates is to store for each possible path in the document the number of nodes to be found under that path. Still, such statistics may be quite imprecise, since they do not account for the skew in the parent-child distribution. For constructing and storing more elaborated XML data statistics we rely on previous work described in [2] or [10].

Finally, in addition to data cardinalities, each peer has to provide also some measures on the average *element size*, in Kb, as well as regular *data statistics*, regarding the value nodes (leaves of the XML tree).

Besides of the above information which is rather standard for XML query processing, we introduce here a new summary information, specific to our context of distribution and replication regarding *exit points*. *Exist points* are nodes which have outgoing external links, i.e. those connected to remote data nodes. They correspond to: (1) a node e_2 if it has an external outgoing edge to the replicate (on some other peer) of one of its children e_1; and (2) a node e_1 if it has an ascending incoming edge from a replica (on some other peer) of its parent node e_2.

As we will see later, this information can be used for decomposing a query among several peers.

4. DISTRIBUTED EVALUATION

This section presents our query evaluation strategy over a fixed configuration of replicated and distributed dynamic XML documents. Section 4.1 explains briefly the principles of our evaluation strategy, in particular, our query decomposition strategy, and shows its innovative aspects with respect to the state of the art in distributed query processing. We then move to explaining our cost information gathering and query processing strategy. For ease of explanation, we first focus on a simple class of *linear XPath queries* only. For such queries, Section 4.2 presents collaborative gathering of cost information, while Section 4.3 discusses their evaluation. Finally, Section 4.4 shows how to evaluate in general XQuery$_{dr}$ queries.

4.1 Outline of Query Evaluation

Consider a peer P which has to execute a simple path expression query Q. It may happen that some data required by Q is missing from P, because of distribution; also, data needed by Q can be found at more than one peer, due to replication. P adopts the heuristic of executing as much of Q as possible, say Q_{local}, obtaining an intermediate result, and delegates one or several further subqueries Q_{next} to one or several other peers P_{next}. Each P_{next} will receive the intermediate results and continue processing, by applying the same method: attempt to evaluate all Q_{next} and, if all data is not available, delegate further etc.

Data shipping vs. query shipping This novel approach combines data shipping and query shipping, and is specific to our problem: answering queries over distributed data *without knowledge of the data distribution*. This information is missing due to the autonomy of peers in a large-scale system; therefore, traditional distributed query decomposition methods for distributed DBMSs [29] and wrapper-mediator systems [19] do no longer apply. Among the latter, our approach is most similar to Garlic's [19], where wrappers decide how much of the query sent by the mediator they solve. However, in Garlic, the mediator has global information about data location, and all wrappers report directly to it, which is no longer the case in our context. Control over execution is distributed also in [25], where peers may tweak an existing query plan; in contrast, in our setting, no plan can be computed in advance. In the PeerDB peer-to-peer system [23], a query asked on peer P is propagated as such from P to other peers, and the answers are all unified at P. This strategy does not apply for us, as data distribution and replication lead to query decomposition.

Communication pattern In our query processing strategy, at each step, the sub-query Q_{next} includes the address of the peer P on

```
    Algorithm what-if
    Input: linear path expression query Q, current peer P
    Output: query decomposition and cost information info
1   decompose Q = Q_next(Q_local), such that Q_local is the
2        longest prefix of Q that can be evaluated at P
3        by analyzing the exit points at P_c
4   info ← <record peer=P>
5                <decompose local=Q_local next=Q_next/>
6                <local cost=cost(Q_local) fanout=N_{Q_local}
7                      size=size(Q_local)/>
8           </record>
9   if Q_next ≠ ∅
10  then cand ← the set of peers known to P to possess some
11       data needed by Q_next, nextCost ← ∞
12       foreach P_cand ∈ cand
13            info_cand ← what-if(Q_next, P_cand)
14            if P.subjCost(info_cand) < nextCost
15            then nextCost ← P.subjCost(info_cand)
16                 P_next ← P_cand
17                 add <bw>P→P_next</bw > as child of info
18                 add info_cand as child of info
19  return info
```

Figure 9: What-if cost analysis.

which Q was originally asked, so that the result is returned directly to P, since it requires less communications (also done in PeerDB [23]). The drawback, with respect to confidentiality and security, is that all peers get to know who initiated the query; we felt the performance gains outweigh this disadvantage.

4.2 Collaborative Information Gathering

Linear path queries In this section, we are only concerned with linear path queries. These are extended XPath queries as described in section 2.2, and further simplified. First, without loss of generality, we assume they only go downwards, i.e., no ../ step [24]. Second, linear queries can only have interspersed predicates of the form $[text() = c]$, applying simple selections on the values of an element. For example, $hotel/name[text() ='' Star'']$ is a linear query, while $hotels[hotel/name/text() ='' Star'']$ is not. Following XPath [31] semantics, respected also in XQuery$_{dr}$, any XPath query can be decomposed into a number of linear correlated queries. For example, the non-linear query Q_{nl} above can be decomposed into: $Q_{l1} = hotels$, $Q_{l2}(\$x) = \$x/hotel/name$ $[text() ='' Star'']$, such that $Q_{nl} = \{x | x \in Q_{l1}, Q_{l2}(x) \neq \emptyset\}$.

With respect to our query processing strategy, linear queries can always be decomposed between P and a single P_{next}. To decide on P_{next}, the local statistic and cost estimates at P (Section 3.3) are not enough. P needs to learn, first, the peers which may contribute to evaluating Q, and second, query cost information regarding these peers. We now explain how this information is gathered.

Simplifying assumption : Q location-unaware For ease of explanation, we first assume that Q specifies no preferred peer for evaluation, which in XQuery$_{dr}$ we denote as $\{Q\}@any$.

Candidates for the role of P_{next} are those peers (i) having (at least some of) the data needed by Q_{next}, and (ii) such that P knows they have this data. It turns out that the set of such candidate peers are exactly those to which lead external edges from P. Among these, P makes his choice aiming at an *observable optimization*, meaning the chosen P_{next} is the one minimizing the costs of Q which are relevant to P[1]. This cost is determined by several components, a priori unknown to P: (i) the statistics of data at P_{next}, (ii) the cost parameters (CPU, bandwidth etc.) specific to P_{next}, and, (iii) the exit points of Q_{next}, which influence the decomposition of Q_{next} that P_{next} may apply at its turn.

To obtain such estimates, P holds an *what if* analysis, shown in Figure 9. P gathers, in an XML-structured *info* record, cost information from several peers, in a bottom-up manner. At lines 4-8, *info* is initialized with the objective statistic and cost information gathered at the local decomposition step, performed by P. For example, the cost element encapsulates all CPU and space costs associated to running Q_{local} at P, denoted $Comp$ and $Space$ in Section 3.1. Similarly, the size element encodes the output size of query Q_{local}, corresponding to the O vector in Section 3.1. Also, the decompose element by itself gives information about workload decomposition, and implicitly encodes the data dependency between Q_{local} and Q_{next}, modelled by the matrix δ in Section 3.1.

If Q_{next} is not empty, P asks what-if questions on all candidate peers P_{cand} (line 12). If P_{cand} is empty, the query fails (omitted for brevity in Figure 9), since P does not know where to find the data it needs. Otherwise, P selects as P_{next} the peer which optimizes *P's own subjective (observable) global cost.* To evaluate this cost, P applies its own weights on the objective cost information returned in $info_{cand}$ (line 14). The information returned by the most promising peer is inserted into P's own information record, together with the bandwidth required to ship the result of Q_{local} to P_{next}. The final *info* contains thus a global view of: the best way Q may be decomposed from the point of view of P's observable performance, and all objective associated costs.

Remark (optimality) Q is decomposed by several peers, each of which returns the optimal decomposition from its own viewpoint. This does not lead to the absolute globally optimal (for P) decomposition of Q. But this decomposition is not achievable for several reasons: universal data location information cannot be found on each peer; and all peers are not willing to please P, at any incurred cost for them. They collaborate in query processing, but have the freedom of choosing how to do this. As a simple extension, a peer may also refuse to evaluate Q_{next} if its minimal subjective cost is too high; in this case, the peer returns <null> as a record, to signal it will not work. Thus, our algorithm achieves the most consensual decomposition of Q from the perspective of all collaborating peers.

General case: Q location aware If (fragments of) Q specify restrictions on the data location, using some XQuery$_{dr}$ keyword other than any, the algorithm in Figure 9 is modified as follows. The choice of the $cand$ set is restricted to those peers that fullfill the conditions in Figure 9, while also satisfy the location restriction.

Finding master copy nodes Following our data model and query decomposition strategy, if Q specifies as location $@master$, and P has no link to the master, Q may fail; thus, it is possible that from P, the master document is forever lost. Since normally users don't expect, say, the master copy of USASkiCenter to disappear, we take two new measures. First, *every document name is an URI*, identifying the peer on which it originated; either the document root will always be there, or the peer will know where it has moved (in the style of HTTP redirection). Second, we restrict our replication and distribution model so that *external links always connect master nodes in both directions.* These measures ensure that master copies of a document always exist, are connected, and can be found starting from the root.

The decomposition algorithm in Figure 9 involves many message exchanges; we note that (i) these messages are small and the benefits of making a good choice are likely to outweigh this overhead, (ii) the decomposition cost is reduced if it is run when the

[1] Remember from Section 3.1 that a peer's observable performance may include, with different weights, computing costs associated to other peers.

bandwidth is cheaper (e.g. at night for a mobile phone). Furthermore, the obtained cost information could be *cached* for some time, avoiding running the what-if analysis every time Q is evaluated.

Variant: cost-ignorant query decomposition However, if P wishes to avoid such an analysis, but has no cost information, instead of the loop at lines 12-18, it may choose P_{next} just by minimizing the transfer costs from P to P_{next} (potentially, a bad choice).

4.3 Evaluating Linear XPath Queries

A linear XPath query Q (as defined at the beginning of Section 4.2), asked on peer P, can be evaluated in two ways.

First, if P runs a what-if cost analysis (or had done so previously and cached the result), then the returned *info* contains all the information for decomposing Q: ignoring its <local> and <bw> descendent elements, *info* encodes in fact a distributed query plan for Q. To evaluate Q, P and all other peers will (i) evaluate their Q_{local} as identified by *info*, and (ii) ship the result to their P_{next}, together with the indication that the final result should arrive at P. The last peer involved in evaluation returns the result to P.

Alternatively, if no *info* is available, then the evaluation of Q can be explained as a variation of the algorithm in Figure 9. At lines 1-3, Q_{local} is determined by actually *evaluating* as much as possible from Q at P; no <record> is constructed. At lines 13-15, instead of running what-if on candidate peers, P directly chooses a P_{next} (e.g. by optimizing its subjective cost for data transfer from P to P_{next}), and asks P_{next} to evaluate Q_{next} and make sure the result will be transmitted back to the original peer P.

Note that while we use top-down XPath semantics to explain query decomposition, linear query fragments do not need to be evaluated this way. Instead, as soon as the definition of Q_{local} is understood by looking at the exit points, Q_{local} can be delegated to whatever storage the peer uses for its evaluation; efficient methods for evaluating XPath have been proposed, e.g., in [12].

4.4 Evaluating XQuery$_{dr}$

The evaluation of complex XQuery$_{dr}$ queries builds upon the algorithm above for evaluating linear path queries. The key point here is that any XQuery$_{dr}$ query (indeed, any XQuery one) can be syntactically rewritten (while preserving its meaning) so that it only contains linear path expressions, by introducing new variables, as we did in Section 4.2; rules for such XQuery simplification of XQuery are described in [21] and [31]. Among the resulting linear path queries, there may exist dependencies, as for example the one identified by the intermediary variable $\$x$ in Section 4.2. These dependencies entail a partial order on the linear path queries.

On top of the linear path queries, the original query imposes a set of joins, perhaps outerjoins [28], and projections. We adopt the simple strategy of running all such operators at the peer P where the query was asked. The reasons are the following: (i) the increased difficulty of estimating precise join costs at distant peers, (ii) the potential blow-up in the search space of join orderings, if we distribute them over different peers. Furthermore, such join distribution is impossible if the query decomposition is not known before actually evaluating Q (variant at the end of Section 4.2).

A simple case when distributing joins is easy and beneficial is the following. If query decomposition is performed before evaluating it, and if two linear path queries are delegated at some point to the same peer P_x, then P may push to P_x also any join predicate over the results of these linear path queries. This is subject to P_x's agreement to do the join, and the subjective interest of P in pushing the join (e.g., if the join produces too much data, P may prefer to perform it by itself). Projections can be pushed in a similar manner.

An XQuery$_{dr}$ query is thus evaluated as follows. First, simplify it as described above. Second, evaluate as described in Section 4.3 the resulting linear path queries, in the topological order dictated by their dependencies. Finally, order and apply the remaining operators at P.

5. REPLICATING DATA AND SERVICES

For a given configuration and workload, every peer measures its *observable performance*, as previously explained. In order to improve its observable performance, the peer may want to change the configuration; due to peer autonomy, the peer can only modify his own set of data and services. We present below an algorithm that, given a configuration and a specific peer in the configuration, recommends some replication steps that are guaranteed to improve the peer's observable performance.

Before presenting the algorithm, let us consider what are the possible replication scenarios that peer P may consider, when attempting to improve the observable performance of a given query.

Accessing remote information (do not replicate) When not all the data needed for the query evaluation resides on P, it may need to consult remote data, for instance via external links, as in Section 4. This entails some communication costs.
If the query frequency is high and storage cost at the given peer is low, P may prefer to replicate the relevant data and use a local version rather than the remote one.
Replicating data fragments with or without service calls When replicating data, P may take the replicated fragment including the service calls embedded in it; thus P will call the service itself. Alternatively, P may leave (some of) the calls to be executed at the remote peer, and just refer to the data they return via external links (as illustrated in the running example of Section 2).

This second scenario may be cost effective. For example, if the service provider charges some fee from the caller, leaving the call on the remote peer spares P from this fee; or, if the call is invoked more frequently than the query that utilizes its data, its output is transmitted to P at the frequency of the query rather than that of the call invocation, hence entails less communication.
Replicating service definitions When the data is replicated together with its embedded calls, we may want to also replicate, for non opaque services, the code of the called services as well as the data that they use. (see for instance Figure 3 for the services *OperativeSkiResorts* and *HotelsInfo*). This allows the services to be executed locally and further reduce the communication cost.

For the first two cases, namely replication of data and service calls, the decision can be made based on the cost model described in Section 3, comparing P's observable performance for the two scenarios - replication vs. remote access. Things are more complex when service definitions are replicated. One has to decide (i) if and how to modify the service code to best fit the needs of P, (ii) which data the code uses, and how much of it to replicate, and (iii) recursively, for which service calls appearing in this replicated data, the code (and the data that it uses) should be also replicated.

We devote the rest of this section to this problem and explain how points (i)-(iii) above can be solved. Note that the algorithm that we propose makes its decisions by "simulating" candidate configurations (computing their potential cost by using the *what-if algorithm* of Figure 9). The actual replication is done only after all these costs are compared, and the replication policy that will best improve the *observable performance* of the given peer is chosen. This relates to the fact that, especially when working in a decentralized P2P network, a replication decision may not be beneficial to all the peers at the same time.

The Replication Algorithm As explained above, when replicating an XQuery service to a given peer P, we need to identify the (minimal) data that the service needs in order to run at P, replicate it, and reformulate the service code so that it correctly exploits the local data (this is a restricted case of XQuery rewriting to exploit materialized views).

A naive strategy would be to replicate the whole documents that the service implementation touches; in this case the modification to the replicated service query code is minimal (only the document names need to be changed to refer to the local replicated documents). However, this may not be feasible for storage limitation, copyright or security reasons.

At the other extreme, we may completely evaluate the service query at the original peer, copy the query results to the new peer, and transform the replicated service into an identity query, that just reads the result (as in many proposals of *local query cache*). The disadvantage here is that this cached data needs to be refreshed whenever the query result changes. (See Section 2.)

More refined alternatives consist of only *partially* evaluating the service query at the original peer, replicating these partial results at the new peer, and modifying the code of the replicated service to perform the remaining part of the query, on the partial results. If the replicated data is "self-maintainable", as in our Colorado peer example in Figure 3, this is the preferable solution.

This last approach is the one taken by our algorithm, depicted in Figure 10. The algorithm takes as input the service implementation Q, the current configuration (data and services) at the peer for which we try to improve observable performance by replication, and produces: a new configuration for that peer. Note that the modified service implementation Q', which uses the local replicas instead of the original documents referred by Q, is also part of the new configuration of replicated data (the output by the algorithm).

For simplicity we assume that Q is unnested (by applying the syntactic rules described in [21]). Q may include, perhaps in its for, where or return clauses, several path expressions. For each such path expression pe, over document *doc*, our algorithm attempts to identify what is the minimal part from *doc* that needs to be replicated to the new peer so that pe can be locally evaluated.

Let *data traversed by pe* denote the subtree of *doc* including all the nodes touched by the evaluation of pe in the step-by-step manner described in the XPath specification [31]. The principle of our algorithm is the following.

(1) If the data traversed by pe contains no service call, then we evaluate pe on *doc* and replicate at the new peer exactly the nodes in the result. In this case, our replication strategy simply consists of *query result caching*.

(2) If, on the contrary, data traversed by pe contains some service calls, and furthermore, these service calls produce data that may change the result of evaluating pe, then a different strategy is needed, since caching would bring on the new peer an outdated result. Rather, the new peer should take the smallest replica that is big enough to *evolve by itself* on it by means of service call activations. On this replica, the copy of Q (including pe) will yield the same results as if it ran over the original *doc*.

In both cases, the data to replicate may enclose other service calls, for which another replication decision must be taken. The algorithm examines recursively (lines 14-16) all the possible replication decisions that concern the service calls included to the data previously committed for replication. For each of these service calls, also, the other two possible ways of replication are possible. In particular, one may want to replace the service call with an external link, or solely copy the service call without the service implementation. For brevity, we do not show the other two choices in the algorithm; they should be included in the recursive loops (line 14, 23). Our algorithm leaves to the service replica on the new peer the task of combining the path expressions results into the complete query result.

	Algorithm **repDecision** Input: configuration $conf$, service implementation Q Output: configuration $conf'$
1	$conf' \leftarrow conf$, $repData \leftarrow \emptyset$
2	foreach path expression pe over doc in Q
3	pe is of the form $l_1[c_1]/l_2[c_2]/\dots/l_k$
4	*// evaluate pe by top-down navigation in doc:*
5	foreach step j in the evaluation of pe, $j = 1, 2, \dots, k$
6	$Q' \leftarrow ../l_{j+1}/l_{j+2}/\dots/l_k$
7	if exists $\{sc \mid sc$ child of a node in the current node list,
8	sc is a call to a service sv, whose output type
9	may contain a path $l_{j+1}/\dots/l_k\}$
10	then $repData \leftarrow$ the set of subtrees
11	rooted at the current node list
12	$conf' \leftarrow conf \cup repData \cup Q'$
13	if $cost(conf') < cost(conf)$
14	then foreach sv' call of service in $repData$
15	$conf' \leftarrow$repDecision($conf'$, $def(sv')$)
16	endfor
17	break *// stop here evaluation of pe*
18	else nop;
19	else nop;
20	endfor *// the evaluation of pe is over*
21	$repData \leftarrow$ the result of pe on doc
22	$conf' \leftarrow conf \cup repData$
23	foreach sv' call of service in $repData$
24	$conf' \leftarrow$repDecision($conf'$, $def(sv')$)
25	endfor
26	endfor
27	return $conf'$

Figure 10: Service replication algorithm

The algorithm takes into account the service arguments as follows: if they are values, they are used in the evaluation of the replica as filters of the path expressions (line 3); if they are dynamic documents that contain function calls, the algorithm is recursively invoked on these function calls (line 14). Service arguments are not represented explicitly in the algorithm for brevity.

Going back to the running example of Section 2, assume we want to replicate the services OperativeSkiResorts("Colorado") and HotelsInfo("Colorado",$resort)), depicted in Figure 2, to the Colorado peer. Running the above algorithm copies to the Colorado peer the data and service replicas depicted in Figure 3.

The actual document replication is done using the replicate clause of our XQuery$_{dr}$ language, as described in Section 2.2. Note that the algorithm may decide to include in the new configuration some data, which has already entirely or partially been copied in a previous running. In such a case, ID-based fusion, as in [26], is used to unify common elements.

For the time being, we do not take into account possible updates to the documents except for the updates enforced by services. Updates can be viewed as special kinds of queries with side effects, and expressed in XQuery-style query language [30]. Given this, and the declarative nature of our XQuery$_{dr}$ replication language, we believe that update propagation in the style of [14] can be employed in our context as well. This is left for future research.

6. RELATED WORK

Distribution of data have been extensively studied in the past, for relational databases [6, 5], where the query language can be aware or not of horizontal and vertical table fragmentation and of different locations of table fragments. For the Web, data replication is a key to performance and scalability [4]. To the best of our knowledge, replication of *dynamic documents* has never been studied before.

Peer-to-peer systems have gained popularity for file sharing applications [11], and recently captured the attention of the database community [32]. However, only recent work (e.g. PeerDB [23]) builds a query processing mechanism on top of a generic P2P system. As explained in section 5, our replication and distribution scenario requires a different query evaluation strategy than PeerDB's. Furthermore, we propose a dynamic replication mechanism to improve each peer's performance, which is not done in PeerDB. Hybrid system between query-shipping and data shipping has also been studied before in the context of client-server databases [9].

[17] addresses query results caching for directory servers in LDAP. Given a sequence of queries, they present algorithms to select beneficial query templates whose results are put in the cache. ACE-XQ[7] and [13] study semantic caching targeted to XQuery, which may apply in a generic distributed context. The problem of caching query results is also addressed in PeerOLAP [16] for OLAP queries. As in [16], one of our goal is to save bandwidth costs. In our proposal, replications is used to cache not only data, but also query (and service call) relevant data and results. Moreover, we consider the replication of services definitions (i.e. query definitions). To do so, we provide (in Section 5) an algorithm that proposes cost-based changes of the configuration of the dynamic XML documents.

XInclude [31] is a syntax for including external XML fragments in XML documents, similar in principle to the one presented in Section 2. Going further, we address peer-to-peer query processing over distributed documents, and replication of dynamic XML documents. Moreover, as opposite to XInclude which stores external edges at parents only, we allow them to be stored at both peers.

7. CONCLUSION AND FUTURE WORK

In this paper, we considered the new issues raised by the distribution and replication of dynamic XML documents. We have presented a data model, close to the standard XML model, and suitable extensions for XQuery. Based on these, we designed a cost-model for a peer-to-peer context, and used it for efficient collaborative query processing. Furthermore, we proposed a new query decomposition strategy, for querying XML data in the presence of distribution and replication.

The ideas presented in this paper were developed in the context of the Active XML peer to peer system [1, 3, 22], and are currently being implemented and integrated into the system. Active XML already supports dynamic (Active) XML documents and remote invocation of service calls. We are extending it with external links and with distributed query processing capabilities. Among the perspectives of our work, we plan to experiment with our architecture and cost model on mobile peers, in the framework of an industry and research project that we are currently involved in.

8. REFERENCES

[1] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-Peer Data and Web Services Integration (demo). Proc. of VLDB, 2002.

[2] A. Aboulnaga, A. Alameldeen, and J. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *The VLDB Journal*, pages 591–600, 2001.

[3] Active xml. `http://www-rocq.inria.fr/verso/Gemo/Projects/axml/`.

[4] The Akamai webpage. `http://www.akamai.com/`.

[5] Peter M. G. Apers. Data allocation in distributed database systems. *TODS*, 13(3):263–304, 1988.

[6] S. Ceri and G. Pelagatti. *Distributed Databases - Principles and Systems*. McGraw-Hill Inc., 1984.

[7] L. Chen and E. A. Rudensteiner. ACE-XQ: A CachE-aware XQuery Answering System. In *Proc. of WebDB*, 2002.

[8] Microsoft .Net. `http://www.microsoft.com/net/`.

[9] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. In *SIGMOD*, 1996.

[10] J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Simeon. StatiX: Making XML count. In *SIGMOD*, 2002.

[11] Gnutella homepage. `http://www.gnutella.com/`.

[12] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. of VLDB*, 2002.

[13] V. Hristidis and M. Petropoulos. Semantic caching of XML databases. In *Proc. of WebDB*, 2002.

[14] J. Mc Hugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. Technical report, Stanford University Database Group, Feb 1997.

[15] Sun's JavaServer Pages. `http://java.sun.com/products/jsp/`.

[16] P. Kalnis, W.S. Ng, B. C. Ooi, D. Papadias, and K.L. Tan. An Adaptive Peer-to-Peer Network for Distributed Caching of OLAP Results. In *Proc. of ACM SIGMOD*, 2002.

[17] O. Kapitskaia, R. Ng, and D. Srivastava. Evolution and revolutions in LDAP directory caches. In *EDBT*, 2000.

[18] A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas, 2001.

[19] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4), 2000.

[20] Macromedia Dreamweaver. `http://www.macromedia.com/`.

[21] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries over heterogeneous data sources. In *Proc. of VLDB*, 2001.

[22] Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, and Fred Dang Ngoc. Excahnging intensional xml ducuments. In *Proc. of ACM SIGMOD*, 2003.

[23] Wee Siong Ng, Beng Chin Ooi, Kian Lee Tan, and AoYing Zhou. PeerDB: A P2P-based system for distributed data sharing. In *Proc. of ICDE*, 2003.

[24] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: looking forward. Int'l Workshop on XML data management, 2002.

[25] V. Papadimos and D. Maier. Mutant query plans. In *OOPSLA*, 2001.

[26] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In *Proc. of VLDB*, 1996.

[27] PHP. `http://www.php.net/`.

[28] J. Shanmugasundaram, E. Shekita, and R. Barr. Efficiently publishing XML views of relational databases. In *Proc. of VLDB*, 2000.

[29] T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems, 2nd Edition*. Prentice-Hall, 1999.

[30] I. Tatarinov, Z. Ives, A. Levy, and D. Weld. Updating XML. In *Proc. of ACM SIGMOD*, 2001.

[31] The World Wide Web Consortium (W3C). `http://www.w3.org/`.

[32] B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. In *VLDB*, pages 561–570, 2001.