Virtual Partitioning ad-hoc Queries over Distributed XML Databases

Carla Rodrigues[1], Vanessa Braganholo[2], Marta Mattoso[1]

[1] COPPE/Federal University of Rio de Janeiro, Brazil
{carlarod, marta}@cos.ufrj.br
[2] Fluminense Federal University, Brazil
vanessa@ic.uff.br

Abstract. XML query processing on large repositories suffers from performance issues. Despite many efficient indexing techniques, oftentimes only physical XML data fragmentation techniques can improve query processing performance. In such approaches, the database is physically partitioned based on the attributes and selection criteria used in the most frequent queries of the system. Distributed query processing can then take advantage of pruning irrelevant fragments and processing the relevant ones in parallel. However, in many applications, such as Decision Support Systems, input queries are ad-hoc. In such cases, there is no frequent attribute access and physical partitioning is not an option. In relational settings, virtual partitioning has been successfully used to improve performance in such scenario with parallel query processing. Inspired by those solutions, in this work we adapt the relational virtual partitioning technique to the XML model, and apply it to improve the performance of ad-hoc analytical XML queries. Our experimental analysis shows the effectiveness of our approach.

Categories and Subject Descriptors: H. Information Systems [**H.2 Database Management**]: Systems—*Distributed databases*

Keywords: distributed query processing, virtual partitioning, XML

1. INTRODUCTION

Processing queries over XML data has become more efficient since the development of native XML Database Management System (XDBMS). However, the growth of XML databases in size brings us to a challenging problem in terms of response time for complex analytical queries. In general, processing queries over large databases implies limited performance due to poor memory use and I/O. The ever-increasing volume of data, especially XML, is directly related to the performance issues of query processing.

A lot of work on optimizing XML query performance through distributed XML databases has been done over the last few years. These efforts may be divided into two categories: database integration [Wiederhold 1992] and parallel query processing [Kossmann 2000]. Database integration involves integrating data residing in different repositories in order to provide users with a single view of XML data. It requires using complex rules which express the procedures we should adopt to integrate XML data from multiple sources. In contrast, parallel query processing keeps focus on the problem of improving query performance through distribution. Our work is based on parallel query processing and our techniques have been entirely developed with focus on query performance. As far as we are concerned, most of the parallel processing work propose distributed query processing methods by using physical partitioning techniques over XML data [Kido et al. 2006; Waldvogel et al. 2008; Kling et al. 2010; Moreira et al. 2011]. In this approach, XML documents are partitioned and the generated

Authors would like to thank CNPq and FAPERJ for partially supporting this research work.

partitions are distributed to multiple computational nodes. A control node, named mediator, is adopted in several works [Figueiredo et al. 2010; Kurita et al. 2007; Moreira et al. 2011] to control distributed query execution. This node is responsible for receiving the user query, decomposing the input query into subqueries, controlling the distribution of these subqueries to remote nodes and collecting partial results in order to compose the final query result.

The main disadvantage of these solutions is that the query workload needs to be known in advance to design the distributed partitioning schema. In the distributed design, the DBA analyzes attributes and selection predicates in the most frequent queries to decide how to fragment XML documents. In the last decade, XML data warehouses have been exploited in decision-support applications [Mahboubi and Darmont 2009; Golfarelli et al. 2001; Boussaid et al. 2006; Park et al. 2005; Pokorný 2002; Vela et al. 2010]. Considering that a Decision Support System (DSS) is characterized by ad-hoc analytical queries, physical partitioning design techniques can be an inappropriate solution, because we do not know the frequent queries which will be submitted by the user in advance. In the relational DSS benchmark, the TPC-H[1], no physical partitioning techniques can be applied in order to reflect the unpredictability of ad-hoc queries, meaning that it is not possible to elect frequent attributes.

Taking these DSS characteristics into account, we consider the virtual partitioning (VP) technique [Akal et al. 2002] as an attractive solution. So far, this technique has been applied only in relational databases. VP consists in a dynamic partitioning of the database. This flexibility is very attractive to ad-hoc queries, since the partitioning can be decided at query execution time. VP builds subqueries (to represent virtual partitions) by adding new selection predicates into the user query. These new selection predicates will force partitioning according to a virtual partitioning attribute of the tables involved in the specific query to be executed. This virtual partitioning attribute, in general, is the primary key of the chosen table.

The work proposed by Akal et al. (2002) and Paes et al. (2008) have successfully used virtual partitioning in a relational database cluster. A middleware was developed to establish communication between the client applications and a replicated relational DBMS. The full replication approach was adopted, so each node stores the complete database and executes its own replica of the DBMS. These proposals aim to improve query performance through parallel query processing, once each virtual partition is processed in parallel by a different node in the cluster machine.

In the XML model, Khatchadourian, Consens and Siméon's (2011) have recently proposed a solution which presents similarities with VP. In their work, the XML queries are processed in a cluster which implements the MapReduce model [Dean and Ghemawat 2008]. The input data is partitioned into a set of splits which can be processed in parallel by different nodes. They describe the ChuQL language which is a MapReduce extension to XQuery[2]. The main disadvantage of this solution refers to the fact that the users must translate their original XML queries to the ChuQL language, once the methods (map/reduce) invocation depends on this language. Besides, the user is also responsible for specifying the number of partitions and the partitioning function.

Considering the need for solutions to increase performance of analytical queries as well as the good results obtained in relational databases using VP, in this article we propose to apply VP in XML databases. Adapting VP to the XML model is a challenging problem, since the generation of virtual partitions cannot be done by using primary keys (they may not exist in the XML documents). Another challenge, not found in the relational model, is that elements and attributes in XML documents are often assigned to string values, which complicates the definition of the intervals that characterize the virtual partitions. Yet another challenge refers to the types of input data a query can use. In the relational model, SQL queries refer to tables. In XQuery, a query can use specific documents (by using *doc()* clauses) or collections (by using *collection()* clauses). This difference impacts in the cardinality

[1] `http://www.tpc.org`
[2] `http://www.w3.org/TR/xquery/`

of elements the query will deal with, and thus impacts in the partitioning algorithm.

Our work uses an existing methodology for XQuery query processing over distributed XML databases [Figueiredo et al. 2010]. We have adapted this methodology to support ad-hoc queries by using the virtual partition technique. The main contribution of this article consists of providing techniques for applying virtual partitioning over distributed XML databases in order to support ad-hoc analytical XML queries. We describe how the input query can be decomposed into subqueries through query rewriting. The effectiveness of our proposal was evaluated by the execution of an XML query workload in a cluster. The results show that our approach is effective and can reach speed up of up to 22 times in high cost queries when compared to centralized environments.

This article is structured as follows. Section 2 presents concepts related to inter- and intra-query parallelisms and virtual partitioning. We discuss existing work on distributed query processing in Section 3. We consider solutions applied to both relational and XML databases, especially those which use the virtual partitioning technique. Our proposal is presented in Section 4, and its experimental evaluation on Section 5. Finally, conclusions and future work are presented in Section 6.

2. INTER-QUERY, INTRA-QUERY PARALLELISM AND VIRTUAL PARTITIONING

Parallelism can be employed during query processing according to two approaches, among others: inter-query and intra-query parallelism. Inter-query parallelism [Mattoso 2009a] consists in simultaneously processing different low-cost queries in distinct nodes. Intra-query parallelism [Mattoso 2009a] is obtained when multiple nodes process the same query at the same time. Generally, intra-query parallelism is applied to scenarios where users submit high-cost read intensive queries, which are decomposed into subqueries that are processed by different nodes.

The virtual partitioning technique implements intra-query parallelism through query rewriting [Mattoso 2009b]. It rewrites the original query by adding range predicates to it, which generates as many subqueries as the number of nodes available. For example, let Q be a query over the *insurance* table, which contains information about customers and their car insurances. This query returns the names and ages of all customers whose vehicles suffered some kind of damage.

```
Q: SELECT name, age FROM insurance WHERE damaged_vehicle = 'yes'
```

Suppose *insurance_ id* is the primary key of *insurance* and that it is used as the virtual partitioning attribute. A generic subquery on a virtual partitioning is built by adding selection predicates "AND *insurance_ id* $\geq v_1$ AND *insurance_ id* $< v_2$" to query Q, where $[v_1, v_2[$ indicates different ranges of *insurance_ id* in the *insurance* table. Thus, each generated subquery can be processed by a different node in a database cluster which processes a different portion of the query input data.

VP has obtained performance gains during query processing when compared to the results obtained in centralized environments [Akal et al. 2002]. The typical virtual partitioning approach, which we will call "Simple Virtual Partitioning - SVP" [Akal et al. 2002] considers uniform data distribution. Data distribution is measured by counting the occurrence frequencies of each assigned value to an attribute on a table. In relational databases, a set of values can be assigned to each attribute of a table, according to the type defined to this attribute at table creation time. An attribute X is said to follow a uniform distribution if the assigned values occur with similar frequencies, i.e., the values of X are uniformly distributed over the table. Besides, data distribution is related to two concepts: selectivity and density. An attribute is said to have high selectivity and low density if a query with a selection predicate over this attribute returns a low number of tuples. Otherwise, if the query returns a high number of tuples, it is said to have low selectivity and high density. Foreign keys, for example, have low selectivity and high density, whereas primary keys have high selectivity and low density, once its values are unique and this limits the quantity of tuples which satisfies the selection criteria to one.

Considering these concepts, SVP may perform poorly if the values of the virtual partitioning attribute do not follow a uniform distribution, which is called data skew [Walton et al. 1991]. The problem is that, in most real scenarios, there is a non-uniform data distribution. This is because insert and delete operations are frequently executed over tables. Thus, even considering primary keys we cannot guarantee the values are uniformly distributed over the table, because some values may simply not exist. In an unbalanced scenario, query performance will be affected if nodes have different query workloads to process, once the number of tuples related to different equal-sized intervals may vary. In addition, performance gains are limited to the size of the intervals. In some relational DBMS, indexes are ignored when the number of tuples to be accessed is high. So, if SVP generates subqueries characterized by large intervals, the tuples can be sequentially processed, and this results in poor performance.

As an alternative solution, Lima et al. (2004) proposed an extension of the Simple Virtual Partitioning, in order to provide dynamic load balancing and assure the use of indexes. This new approach is named "Adaptive Virtual Partitioning - AVP" and has several implemented variations [Lima et al. 2009; Paes et al. 2008; Furtado et al. 2005]. AVP includes two additional phases: adjustment of the size of the intervals and dynamic load balancing.

Our proposal is based on simple virtual partitioning as proposed in [Akal et al. 2002]. Our idea is first to solve the challenges involved in adapting SVP to XML and evaluate the effectiveness of using VP over XML data in heavy query workloads. To the best of our knowledge, no VP approach has been applied to partition XML documents. Since AVP uses the principles of SVP, in this article, we propose an approach to perform distributed XML query processing by using SVP. As future work, we intend to investigate the AVP approach.

3. RELATED WORK

Parallelism has been successfully employed to efficiently process queries over database clusters. In general, it has been applied as a low-cost solution to improve performance of heavy-weight queries, which access huge amounts of data. Most studies adopt physical partitioning techniques, virtual partitioning or a combination of both strategies. In the following sections we discuss some of the existing proposals in this area.

3.1 Virtual partitioning

Virtual partitioning techniques have been successfully used in several scenarios. As an example, [Sousa et al. 2008] apply a VP technique, similar to AVP from [Lima et al. 2004] on genome databases. Considering relational databases, in this section we give a brief description of three solutions based on virtual partitioning: PowerDB [Rohm et al. 2000; Akal et al. 2002]; Pargres [Paes et al. 2008]; and Smaqss [Furtado et al. 2005].

PowerDB [Rohm et al. 2000; Akal et al. 2002] is a middleware which provides communication between client applications and a relational DBMS in database clusters. PowerDB implements intra-query parallelism by using SVP and a full database replication approach [Akal et al. 2002]. PowerDB has also been applied to a data warehouse environment [Rohm et al. 2000] by means of a hybrid partitioning technique. In this technique, fact tables are partitioned and its partitions are stored with no replication. Dimension tables are fully replicated on all nodes. This approach takes advantage of the storage capacity of nodes, once full replication may be infeasible when the database is large.

Pargres [Paes et al. 2008] is an open-source middleware between client applications and a relational DBMS that exploits inter-query and intra-query parallelism. Pargres implements the adaptive virtual partitioning and it verifies the need of combining both kinds of parallelism when intercepts SQL queries. In addition, Pargres supports data updates by using a scheduling mechanism that controls

the order in which operations are executed. The scheduler also blocks all operations in the database when an update operation is started. Pargres has been employed in a database cluster and has presented very good performance results during analytical query processing.

Smaqss [Furtado et al. 2005] is a parallel solution which combines physical partitioning techniques to adaptive virtual partitioning. This proposal has been employed in a data warehouse scenario. On Smaqss, physical partitioning techniques are applied over fact tables and the generated fragments are partially replicated. On the other hand, dimension tables are fully replicated in all nodes. In addition, Smaqss takes advantage of dynamic load balancing provided by AVP, once virtual partitioning is applied over the input queries.

In respect to the XML model, the most similar solution to VP is proposed in [Khatchadourian et al. 2011]. It provides parallel query processing by using the map/reduce model through the ChuQL language. The map and reduce operations are decomposed into parallel tasks which are processed by different nodes in a cluster. On the map phase, each element of the input XML data is mapped in order to generate a set of intermediate key/value pairs which are grouped together according to the key. Next, on the reduce phase, groups of key/value pairs are processed, usually, by using an aggregation operation. Both phases are processed in parallel, since the work is decomposed into independent tasks, and each node processes different parts of the input data. This solution has been successfully employed to deal with the problem of processing huge amounts of data through parallel processing and data distribution. However, the mandatory usage of the ChuQL language is a drawback, since users cannot express their queries in other XML query languages.

Virtual partitioning has been an attractive solution to improve query performance through parallel processing. However, to the best of our knowledge, there is no work in literature that proposes VP, neither the simple nor the adaptive approach, over a distributed XML database.

3.2 Physical partitioning in XML databases

In respect to XML databases, there are many solutions based on distributed query processing which use physical partitioning techniques. Some studies adopt an architecture based on a central node [Figueiredo et al. 2010; Kurita et al. 2007; Moreira et al. 2011], also called mediator, which is responsible for: (i) receiving the main query, (ii) generating the subqueries, (iii) reducing and removing fragments which do not contribute to results, (iv) sending the subqueries to distributed XDBMS on the remote sites and (v) consolidating the partial results.

In Figueiredo's et al. (2010) proposal, physical partitioning techniques are applied over XML documents, considering two different kinds of databases: a single large document (SD) and a large collection of multiple documents (MD). XML fragments follow the definition by Andrade et al. (2006). It presents a clear definition of XML fragmentation by distinguishing between horizontal, vertical and hybrid fragmentation. Horizontal fragmentation consists in applying a selection operation over a homogeneous collection of XML documents. It groups the XML documents which satisfy a given selection predicate. Vertical fragmentation is obtained by applying a projection operation over the data structure in order to group elements which are frequently accessed together in queries. Finally, hybrid fragmentation combines vertical and horizontal fragmentations, in which the order of the operations (selection and projection) depends on the fragmentation design. Note that horizontal fragmentation may be employed only in MD databases.

Figueiredo et al. (2010) work provides means to verify the correctness of the XML fragments and decompose queries properly, once it uses reconstruction and correctness fragmentation rules [Andrade et al. 2006]. The fragments are stored in the distributed environment with no replication. A middleware has been developed to support communication between the client applications and the distributed XDBMS in order to provide distributed query processing. The mediator node receives the main query, expressed in XQuery, generates the subqueries, eliminates the irrelevant fragments and

distributes the subqueries to the appropriate nodes. This is done according to a catalog file which contains information about the fragments and its localization.

Mediator architectures are also adopted in other works. In Moreira et al. (2011) proposal, both partial and total replication approaches are employed. A dynamic data relocation strategy is proposed by Kurita et al. (2007). In their work, XML fragments are exchanged between the nodes with the longest and the shortest query processing time in order to provide load balancing.

Physical partitioning presents improvements in query performance when compared to query processing in centralized environments. A set of fragmentation algorithms based on workload-awareness is proposed in [Kling et al. 2010]. Given a set of queries, the most appropriate fragmentation design is defined and the XML fragments are generated. Their work focuses on horizontal and vertical fragmentation, and deals with the problem of pruning and localization in distributed XML environments. Their model attempts to access the lowest possible number of fragments in order to improve performance. However, this technique does not support ad-hoc queries, once input queries must be known in advance, and are an input to the proposed algorithm.

As we can see, the employment of physical partitioning techniques over XML databases is still subject of much research. The response time and the throughput depend on the data distribution between the XML fragments and the number of fragments to be accessed. Besides, query performance may be affected by costs with communication, transfer of partial results and consolidation of results. In scenarios where queries are not known in advance, all of these problems become a threat to the system's performance. In the next section, we present our approach to adapt SVP to the XML model, aiming at mitigating these problems in ad-doc scenarios.

4. VIRTUAL PARTITIONING OVER XML DATABASES

Our work proposes a technique to employ simple virtual partitioning (SVP) over XML databases. We use the methodology for XQuery query processing over distributed XML databases presented in [Figueiredo et al. 2010]. This includes parsing and validating queries, generating partitions, processing queries in distributed XDBMS and consolidating results by using a control node. Our proposal involves the decomposition of a given input query into as many subqueries as the number of desired fragments. Then, each generated subquery can be processed by a different node on a database cluster. All nodes process subqueries, however, only one node among them is selected as the control node, which is responsible for consolidating the results.

In general, in relational databases, the virtual partitioning attribute is the primary key of the tables that are involved in the query. In most cases, the primary key is assigned to integer values which makes query rewriting easier, once it favors the definition of the intervals for the virtual partitions. However, even in relational databases, primary keys may be defined as attributes of *varchar* type, for example, which is an arbitrary string of alphanumeric characters. In such cases, it is difficult to define equal-sized intervals considering groups of characters. In XML model, this restriction is more severe, once keys are often assigned to string values. In XML databases, a key can be defined in XML Schema[3] in order to identify which element of the document should be unique. However, this is not mandatory, so we cannot assure that the user have defined keys for every XML document that is stored in database.

Adapting the virtual partitioning technique to the XML model is a challenging problem. The choice of the virtual partitioning attribute and the definition of the intervals are more difficult in XML model when compared to the relational model. As an example, consider the document *insurance.xml* shown in Fig. 1.

[3] `http://www.w3.org/TR/xmlschema-0/`

```
<insurance>                                          <vehicles>
 <customers>                                          <vehicle>
 <customer>                                             <customerCpf>50375061827</customerCpf>
   <CPF>50375061827</CPF>                               <VIN>9BD15802764839341</VIN>
   <name>Bruno Fernandez</name>                         <model>Uno Mille</model>
   <age>27</age>                                        <percent_of_damage>0.00</percent_of_damage>
   <damaged_vehicle>No</damaged_vehicle>               </vehicle>
 </customer>                                          <vehicle>
 <customer>                                             <customerCpf>94258266418</customerCpf>
   <CPF>94258266418</CPF>                               <VIN>9BWCA05Y61T213405</VIN>
   <name>Vanessa Ribeiro</name>                         <model>Gol</model>
   <age>42</age>                                        <percent_of_damage>20.00</percent_of_damage>
   <damaged_vehicle>Yes</damaged_vehicle>             </vehicle>
 </customer>                                          ...
 ...                                                 </vehicles>
</customers>                                         </insurance>
```

Fig. 1. XML document which contains information about insurances

```
<results> {
   for $ctm in doc('insurance.xml')//customer
   where $ctm/damaged_vehicle = "Yes"
   return <injured_customer>
              {$ctm/name} {$ctm/age}
          </injured_customer> }
</results>
```

Fig. 2. Input query expressed in XQuery language

```
<results> {
   for $ctm in doc('insurance.xml')//customer
   where $ctm/damaged_vehicle = "Yes"
   and (starts-with($ctm/CPF, '0')
        or starts-with($ctm/CPF, '1') )
   return <injured_customer>
              {$ctm/name} {$ctm/age}
          </injured_customer> }
</results>
```

```
<results> {
   for $ctm in doc('insurance.xml')//customer
   where $ctm/damaged_vehicle = "Yes"
   and (starts-with($ctm/CPF, '8')
        or starts-with($ctm/CPF, '9') )
   return <injured_customer>
              {$ctm/name} {$ctm/age}
          </injured_customer> }
</results>
```

Fig. 3. Subqueries for the [0-1] (left) and [8-9] (right) intervals

The document *insurance.xml* contains information about customers and their insurances. Assume that each customer may have more than one insured vehicle and each vehicle keeps a reference to its owner. Suppose the elements *CPF* and *VIN* (*Vehicle Identification Number*), also known as chassis number, are assigned to string values and are defined as keys. Consider the input query shown in Fig. 2 which returns the names and ages of all customers whose vehicles have suffered some damage.

The first step to apply virtual partitioning is to define the virtual partitioning attribute. Considering the same approach adopted in relational databases, we may choose one of the defined keys for the given XML document. Once the element *customer* is involved in the query and it is uniquely identified by the key *CPF*, we have chosen *CPF* as the virtual partitioning attribute. The next step is to decompose the input query into as many subqueries as the number of desired fragments. For this example, assume that the user will require five nodes in a database cluster. It means the input query must be decomposed into five subqueries, i.e., five virtual partitions. There are ten possible values to the first digit of each CPF, which may vary from zero to nine. So, the virtual partitioning divides the ten possible values by five. It generates the following intervals: (i) [0-1] CPF that starts with value 0 or 1, (ii) [2-3] CPF that starts with value 2 or 3, (iii) [4-5] CPF that starts with value 4 or 5, (iv) [6-7] CPF which starts with value 6 or 7 and (v) [8-9] CPF which starts with value 8 or 9. Fig. 3 shows the subqueries corresponding to the first (left) and the fifth (right) intervals.

After applying virtual partitioning, the subqueries are processed by different nodes. Although the generated subqueries are defined by intervals of equal size, this approach may cause a severe load imbalance. This is due to the fact that the data distribution over the element CPF is unknown. So, the document may contain many customers whose CPF start with values [0-1], [4-5] and [6-7], but few customers whose CPF start with values [2-3] and [8-9]. So, three nodes would have a very high query workload, whereas the other two nodes would have a low query workload, which leads to a non-efficient query processing. Both in the relational model and the XML model, performance depends on data distribution and we cannot guarantee the virtual partitioning attribute follows a uniform distribution. This problem is more severe in cases where the partitioning attribute is of type string.

Another problem refers to the definition of the intervals when the quantity of values which can be assigned to each digit of element CPF are lower than the required nodes. As an example, assume that sixteen nodes have been required instead of five. This makes the definition of the intervals more difficult, once we can no longer analyze only the first digit of the value for the element CPF. We must analyze the second digit to define the intervals. In this case, we could define the following intervals: [00-14] CPF whose two first digits are between the values 00 and 14, [15-27] CPF whose two first digits are between the values 15 and 27, and so on, until we generate the sixteen partitions. However, in such cases, determining a rule to define the values of the intervals is infeasible. Without a well-defined algorithm, the virtual partitions cannot be generated.

As we have seen, the absence of keys, the definitions of keys of type string and a non-uniform data distribution related to the partitioning attribute complicate the definition of the intervals. Taking into account these challenges, we propose a solution that does not depend on the data distribution and elements with unique values. It can be achieved by using the XPath *position()* function[4] which returns the position of the current element. Our work uses this function to create the selection predicates and generate the subqueries properly. The first step consists in defining the partitioning attribute. The algorithm analyzes the document schema in order to obtain the complete paths of the XPaths expressed in f*or/let* statements in the input query. According to our example, the original XPath is *//customer* whose complete path is *insurance/customers/customer*. Our selection criteria is based on 1:N relationships between XML nodes. The algorithm loops through each node on the complete XPath towards the child nodes from the root node. As soon as it finds a node X which occurs N times in the document and whose parent node Y occurs only once, it selects X as the partitioning attribute. This is due to the fact that the value of the *position()* function is restarted at each parent node found in the XPath. For this reason, the partitioning attribute must have a parent node with cardinality value equal to one. In our example, the node represented by *customer* is selected as the partitioning attribute, since it occurs many times under its parent *customers*.

Finally, the virtual partitions can be generated by adding the selection predicates "[*position()* $\geq v_1$ *and position()* $< v_2$]". The parameters $[v_1, v_2[$ correspond to disjoint ranges of values which indicate the positions of the element *customer* which is the virtual partitioning attribute. Thus, each virtual partition is defined by a filter that is added in the *for* statement of the XQuery: "*for $ctm in doc('insurance.xml')//customer[position()* $\geq v_1$ *and position()* $< v_2$*]*". Suppose the document *insurance.xml* have 200,000 customers, and five nodes have been required. Based on this, we generate the following intervals: [1-40001[, [40001-80001[, [80001-120001[, [120001-160001[and [160001-200001[. Fig. 4 shows the subqueries which represent the first (left) and the second (right) intervals for the example query. Note that we use the *position()* function as a filter of an XPath[5] expression.

Virtual partitioning is more complex when applied to queries which involve join operations. If SVP is applied to each document accessed on the query, the number of fragments will be different from the required cluster nodes. Let f be the number of partitions to be generated and j the number of join operations on the input query. When SVP is applied to each document referenced on a join,

[4] `http://www.w3.org/TR/xpath-functions/`
[5] `http://www.w3.org/TR/xpath/`

```
<results>
  {for $ctm in doc('insurance.xml')//customer
   [position() >= 1 and position() < 40001]
   where $ctm/damaged_vehicle = "Yes"
   return
      <injured_customer>
         {$ctm/name} {$ctm/age}
      </injured_customer> }
</results>
```

```
<results>
  {for $ctm in doc('insurance.xml')//customer
   [position() >= 40001 and position() < 80001]
   where $ctm/damaged_vehicle = "Yes"
   return
      <injured_customer>
         {$ctm/name} {$ctm/age}
      </injured_customer> }
</results>
```

Fig. 4. Subqueries for the [1-40001[(left) and [40001-80001[(right) intervals

```
<results> {for $ctm in doc('insurance.xml')//customer
           for $vhc in doc('insurance.xml')//vehicle
           where $vhc/customerCpf = $ctm/CPF and $ctm/damaged_vehicle = "Yes"
           return <injured_customer> {$ctm/CPF} {$vhc/percent_of_damage} </injured_customer> }
</results>
```

Fig. 5. Input query with a join operation

```
<results> {
  for $ctm in doc('insurance.xml')//customer
[position >= 1 and position < 100001]
  for $vhc in doc('insurance.xml')//vehicle
[position >= 1 and position < 125001]
  where $vhc/customerCpf = $ctm/CPF
  and $ctm/damaged_vehicle = "Yes"
  return <injured_customer>
           {$ctm/CPF} {$vhc/percent_of_damage}
         </injured_customer> }
</results>
```

```
<results> {
  for $ctm in doc('insurance.xml')//customer
[position >= 100001 and position < 200001]
  for $vhc in doc('insurance.xml')//vehicle
[position >= 125001 and position < 250001]
  where $vhc/customerCpf = $ctm/CPF
  and $ctm/damaged_vehicle = "Yes"
  return <injured_customer>
           {$ctm/CPF} {$vhc/percent_of_damage}
         </injured_customer> }
</results>
```

Fig. 6. First and fourth partitions after applying SVP over a query with a join operation

the total number of virtual fragments will be *(f * (j+1))* which is greater than *f*, the desired number of partitions. This is due to the fact that all combinations between all fragments are executed. For example, consider the document *insurance.xml* previously presented. Let Q be a query which selects the customers whose vehicles have suffered some damage. It must return the customer CPF and the percentage of damage of the vehicle. Consider the input query shown on Fig. 5.

Suppose the XML document have 200,000 customers and 250,000 vehicles. Assume that the input query must be decomposed into two subqueries. In this example, *customer* and *vehicle* are the virtual partitioning attributes. If SVP is applied to the partitioning attribute *customer*, it generates two disjoint intervals: [1, 100001[and [100001, 200001[. Next, SVP is applied again over the partitions generated by the first attribute. Now, it analyzes the partitioning attribute *vehicle* and generates the following intervals: [1, 125001[and [125001, 250001[. At the end, SVP generates four partitions (two partitions*(one join operation + 1)). Fig. 6 shows the first (left) and fourth (right) partitions.

As a solution, only one partitioning attribute is selected, in order to reach the desired number of fragments. In our example, each element *customer* may be associated to N elements *vehicle*. Note that *customer* is equivalent to a primary key which is referenced by *vehicle*, which in turn is equivalent to a foreign key. So, *customer* is the primary element, whereas *vehicle* is the secondary element. In such cases, the algorithm to select the partitioning attribute uses only one of the *for* statements expressed in the input query. This *for* statement must contain the primary element. In our example, the element *customer* is selected as the partitioning attribute. If there is more than one primary element, i.e., more than one join involved in the query, the partitioning attribute is the one with the shortest cardinality.

The strategy previously described to generate the virtual partitions can be directly applied over

```
<results> { for $ctm in collection('insurances')//customer
   where $ctm/damaged_vehicle = "Yes"
   return <injured_customer>
              {$ctm/name} {$ctm/age}
          </injured_customer> }
</results>
```

Fig. 7. Input query over a collection

```
<result> {
   let $ctm1 := doc("Ins1.xml", "insurances")//customer
   let $ctm2 := doc("Ins2.xml", "insurances")//customer
   let $ctm3 := doc("Ins3.xml", "insurances")//customer
   let $ctm4 := doc("Ins4.xml", "insurances")//customer
   for $ctm in ($ctm1 | $ctm2 | $ctm3 | $ctm4)[position() >= 376 and position() < 751]
   return
      <injured_customer>
         {$ctm/name} {$ctm/age}
      </injured_customer> }
</result>
```

Fig. 8. Virtual partitioning for the interval [376, 751[

input queries which involve documents, but not collections. This is due to the fact that the proposed technique could cause load imbalance if collections are involved. For example, consider the input query shown on Fig. 7 which involves a collection instead of a document.

Suppose the collection *insurances* contains four documents similar to the document described on Fig. 1. The element *customer* is the virtual partitioning attribute and the data distribution over this attribute is as follows: *Ins1.xml* has 200 customers; *Ins2.xml* has 100 customers; *Ins3.xml* has 150 customers; and *Ins4.xml* has 300 customers. Assume that SVP must generate only two partitions. So, it divides the total number of customers by two in order to define the values of the intervals. Once there are 750 customers in total, one node processes the elements which position varies in the interval [1, 376[and the other node processes the elements which position varies in the interval [376, 751[. Thus, one of the nodes processes all the elements *customer* of the four documents, whereas the other node does not process any element, once there is no element *customer* on positions higher than 300 in the individual documents. So, the subquery corresponding to the interval [376, 751[does not return any element. It is due to the fact that the documents inside the collection are processed one by one. Note that this leads to a severe load imbalance.

To solve this problem, the input query is always checked before applying the SVP. If the input query involves a collection, the algorithm is divided into three steps: (i) it defines a new XML query which expresses the union of the documents that exist in the collection, (ii) it selects the partitioning attribute and (iii) it creates the virtual partitions over this new XML query. Considering our example, it first accesses the database to verify which XML documents belong to the collections involved. For each of these documents, it generates a *let* statement which is added to the original query. After, it replaces the *collection()* clause on the *for* statement by using the union operation to unite all returned documents. Next, the partitioning attribute is selected according to the previously mentioned method. Finally, the SVP techniques are applied over the new query which avoids load imbalance, since the union of the documents contains 750 customers in total. This solution guarantees that both nodes process an equal number of elements. Fig. 8 shows the second virtual partition generated by our algorithm, which covers the interval [376, 751[.

As we have seen, our solution includes a set of steps to apply SVP to the XML model. These steps cover the reading and validation of the input query; the query reformulation in cases of MD databases; the selection of the partitioning attribute; the generation of selection predicates by using the XPath *position()* function over the partitioning attribute; the parallel processing of the virtual partitions;

```
1:  procedure query_rewriting(input_XQuery, number_of_partitions)
2:  number_of_joins <- get_number_of_joins(input_XQuery);
3:  input_q <- input_XQuery;
4:  if number_of_joins = 0 then
5:    XPath <- get_XPath_in_For_Let_Statement(input_XQuery);
6:  else
7:    join_expression <- get_join_expression(input_q);
8:    input_q <- remove_join_expression(input_q);
9:    primary_element <- get_primary_element(join_expression);
10:   previous_cardinality <- get_cardinality(primary_element);
11:   XPath_left_side <- get_XPath_left_side_join (join_expression);
12:   XPath_right_side <- get_XPath_right_side_join(join_expression);
13:   if XPath_left_side contains the primary_element then
14:     XPath <- get_XPath_For(XPath_left_side, input_XQuery);
15:   else
16:     XPath <-get_XPath_For(XPath_right_side, input_XQuery);
17:   end if
18:   loop
19:     if input_q contains join_expressions then
20:       join_expression <- get_join_expression(input_q);
21:       input_q <- remove_join_expression(input_q);
22:       primary_element <- get_primary_element(join_expression);
23:       current_cardinality <- get_cardinality(primary_element);
24:       XPath_left_side <- get_XPath_left_side_join(join_expression);
25:       XPath_right_side <-get_XPath_right_side_join(join_expression);
26:       if current_cardinality < previous_cardinality then
27:         if XPath_left_side contains the primary_element then
28:           XPath <- get_XPath_For(XPath_left_side, input_XQuery);
29:         else
30:           XPath <-get_XPath_For(XPath_right_side, input_XQuery);
31:         end if
32:       end if
33:     else exit loop;
34:     end if
35:   end loop
36: end if
37:   if XPath is an incomplete path then
38:     complete_path <- get_complete_path(XPath);
39:   else
40:     complete_path <- XPath;
41:   end if
42:   parent_node <- get_root_node(complete_path);
43:   complete_path <- remove_root_node(complete_path);
44:   parent_cardinality <- 1;
45:   partitioning_attribute <- undefined;
46:   cardinality_selected <- 0;
47:   loop
48:   if complete_path contains nodes then
49:     current_node <- get_root_node(complete_path);
50:     complete_path <- remove_root_node(complete_path);
51:     current_cardinality <- get_cardinality(current_node);
52:     if current_cardinality > 1 then
53:       if parent_cardinality = 1 then
54:         partitioning_attribute <- current_node;
55:         cardinality_selected <- current_cardinality;
56:         exit loop;
57:       end if
58:     else
59:       parent_cardinality <- current_cardinality;
60:       parent_node <- current_node;
61:     end if
62:   else exit loop;
63:   end if
64:   end loop
65:   if partitioning_attribute is not undefined then
66:    intervals <- generate_intervals(partitioning_attribute,
cardinality_selected, number_of_partitions);
67:     add_intervals_to_input_query(intervals, input_XQuery);
68:   else
69:    notify_user_SVP_cannot_be_applied();
70:   end if
71: end procedure
```

Fig. 9. Procedure for selecting the partitioning attribute and generating the subqueries

and the consolidation of the final result. The last two steps are performed according to [Figueiredo et al. 2010].

Our solution allows us to apply virtual partitioning over XML databases by using any virtual partitioning attribute, once our approach does not depend on the values assigned to the chosen attribute. The query rewriting algorithm is shown in Fig. 9.

5. EXPERIMENTAL EVALUATION

The experimental environment is a SGI Altix ICE 8200 cluster with 64 CPUs 2.66GHz Quad Core Intel Xeon 5355 with 1GB of RAM memory per core, running Linux. All the computers are diskless computers, i.e., they do not have a hard disk and programs and data are retrieved from the network. The cluster has a shared storage area that can be accessed by any computer at any time. We have implemented our technique using the native XML database system Sedna [Fomichev et al. 2006]. Since the key point of SVP is to divide the workload to several nodes, each running the original query over a different part of the data, we need each core to access a different database instance to guarantee the subqueries will be run in parallel. In our experiments, we have used 32 nodes of the cluster, and we have installed 32 instances of Sedna on the shared storage area in order to provide parallel access to XML data. Each node accesses a different one. Our experiments were performed over SD XML databases. For this, we have stored large XML documents in the XDBMS. The documents were provided by the benchmark XMark[6] and the DBLP database[7].

[6] http://www.xml-benchmark.org/

[7] http://www.informatik.uni-trier.de/~ley/db/

Table I. Definition of the queries used in experiments

Query	# Join	Crd.Join	Agg.	Ord.	Crd. VPA	Size of InDoc	Size of OutDoc
Q1	0	0	No	Yes	212273	127000 Kb	22340 Kb
Q2	0	0	No	No	212273	127000 Kb	48424,11 Kb
Q3	1	25500000	No	Yes	25500	111130 Kb	39345,81 Kb
Q4	0	0	Yes	No	212273	127000 Kb	27500 Kb
Q5	1	51000000	Yes	No	25500	111130 Kb	99,12 Kb
Q6	1	25500000	No	No	25500	111130 Kb	80,26 Kb
Q7	0	0	No	Yes	111609	127000 Kb	23,77 Kb
Q8	0	0	Yes	No	25500	111130 Kb	2870 Kb
Q9	0	0	No	No	25500	111130 Kb	7200 Kb
Q10	0	0	Yes	Yes	25500	111130 Kb	1090 Kb
Q11	1	5362500	No	No	9750	111130 Kb	3,42 Kb
Q12	0	0	Yes	No	12000	111130 Kb	15450 Kb

We have defined a set of XML queries in order to evaluate its performance. The set of queries contains queries without join operations, queries which involve join operations, queries which perform aggregation and queries which use the *order by* clause. We have analyzed twelve queries by varying the number of required partitions. It means each query has been decomposed into two, four, eight, sixteen and thirty two subqueries.

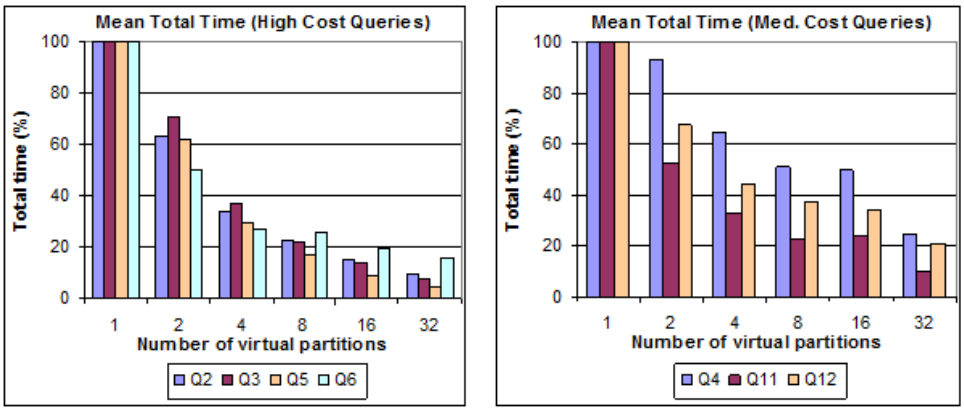
The analysis of results was done by comparing the performance gains obtained in each experiment. The queries we used are described in Table I. Its columns indicate, respectively: (i) the query identifier, (ii) the number of join operations involved, (iii) the cardinality of the join operation, (iv) the existence of aggregation, (v) the use of the *order by*, (vi) the cardinality of the virtual partitioning attribute (VPA), (vii) the size of the processed document (InDoc - Input Document) and (viii) the size of the document with the final result (OutDoc - Output Document), specified in Kilobytes (Kb).

We have separated the queries into three groups: high cost, medium cost and low cost. Queries Q2, Q3, Q5 and Q6 (marked in dark gray in Table I) belong to the first group: they are the most costly ones. Queries Q4, Q11 and Q12 have medium cost (they are marked in light gray in Table I), whereas queries Q1, Q7, Q8, Q9 and Q10 are low cost queries.

We have performed eleven executions for each query and have used the average of its total time execution, discarding the first execution, to analyze the performance. Each one of the twelve queries was processed without applying SVP in order to collect its sequential execution time and get a baseline for measurements. These sequential execution times are represented as 100% in order to facilitate the observation of performance gains. Fig. 10 shows the performance of the high (a) and medium (b) cost queries.

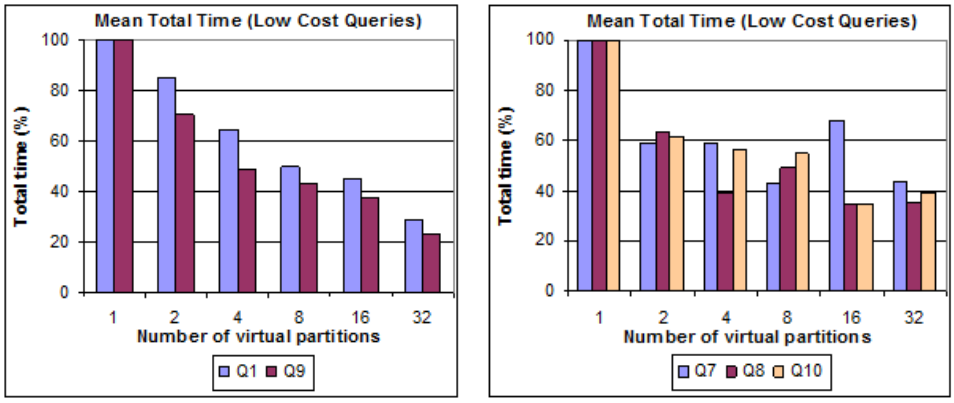
Fig. 10 shows a decrease in the mean total time, especially on the queries Q3 and Q5 which access huge amounts of data, once they involve join operations. Besides, these two queries retrieve results which are uniformly distributed in the XML document. The best performance was obtained by Q5 whose execution time with 32 virtual partitions is 22 times faster than the sequential one. Besides, query Q5 generates a final document of small size. This reduces impacts in performance due to the low costs with the consolidation of results. Queries Q2 and Q6 have also reached good speed up, once the cardinality of the partitioning attribute in Q2 is very high and Q6 involves a join operation. When SVP is applied, the query workload is distributed and processed in parallel, which improves performance. However, the performance gains obtained in executing query Q6 is lower than that reported for the previous two queries. This is due to the fact that the queries Q2 and Q6 are less costly than the queries Q3 and Q5.

With regard to the medium cost queries (Q4, Q11 and Q12) we have also observed a decrease in the mean total time. Query Q11 has presented the best performance among the three queries. This is due to the fact that the size of its final results is smaller if compared to the final results for Q4



(a) Performance of the queries Q2, Q3, Q5 and Q6 (b) Performance of the queries Q4, Q11 and Q12

Fig. 10. Statistical results of the high and the medium cost queries



(a) Performance of the queries Q1 and Q9 (b) Performance of the queries Q7, Q8 and Q10

Fig. 11. Statistical results of the low cost queries

and Q12. So, the performance of Q4 and Q12 are affected by communication costs, transfer of partial results and consolidation of results. Besides, query Q4 suffers more impact since it has an aggregation operation.

As expected, queries which access huge amounts of data and generate smaller final results show the highest performance gains. Note that the four most costly queries (Q2, Q3, Q5 and Q6) have reached the best performances. Besides, the queries Q2, Q3 and Q5 do not have selection predicates over the values of any element or attribute in the XML documents. It means that the results retrieved on these queries are uniformly distributed. On the other hand, medium cost queries which generate large final results suffer some impacts, since costs with data transmission seems to have influence on query performance. This is reasonable since the control node needs to access many instances of the XDBMS to retrieve the partial results to generate the final result. Thus, the greater amount of data to consolidate the results, the greater the impact on performance.

In contrast with the previous results, some of the twelve queries have not obtained good performance in all the experiments, since the decrease in the mean total time was not continuous. In such cases, we have observed a combination of consecutive decreases and increases in execution times, which varies with the number of virtual partitions. Fig. 11 shows the decrease in total time of low cost queries.

The worst performance has been observed in queries Q7, Q8 and Q10, since the decrease in total time stops at eight and sixteen virtual partitions. At these points, the total time reaches approximately

Table II. Mean total time (ms)

Query	1 Vpt.	2 Vpt.	4 Vpt.	8 Vpt.	16 Vpt.	32 Vpt.
Q1	9427,30	8040,00	6086,70	4708,70	4281,50	2689,20
Q2	104157,70	65859,10	35541,30	23281,40	15800,40	10187,00
Q3	370972,50	263112,80	136695,30	80305,70	53235,90	28299,60
Q4	11743,60	10908,40	7598,50	5986,00	5802,00	2899,60
Q5	419864,40	259151,40	125096,50	72767,80	37443,00	18514,20
Q6	169319,50	85602,20	46202,30	43560,80	32851,50	27404,00
Q7	4341,70	2532,40	2549,60	2643,10	2628,90	1958,20
Q8	6434,60	4088,60	2539,80	3163,90	2224,60	2265,40
Q9	9351,10	6571,70	4515,20	4056,40	3493,20	2181,30
Q10	4936,10	3039,20	2782,20	2705,90	1723,00	1943,20
Q11	23481,20	12251,20	7659,10	5333,20	5638,60	2265,40
Q12	14078,10	9466,10	6235,40	5235,70	4756,30	2951,90

40% of the sequential time. Note that these queries are not costly queries, once they do not involve join operations. Thus, these queries suffer impact from the parallelization strategy, since costs with communication are greater than the performance gains when a high number of nodes are involved. The performance of queries Q1 and Q9 is a few better than the queries Q7, Q8 and Q10, once they are more costly than the others. Queries Q8 and Q10 have reached its best total time when decomposed into only sixteen virtual partitions. On the other hand, the queries Q1 and Q9 show performance improvements until 32 virtual partitions.

Note that the input documents have 127000 Kb and 111130 Kb, but the queries Q7, Q8 and Q10 retrieves only 23,77 Kb, 2870 Kb and 1090 Kb, respectively. This indicates that the amount of data in input documents which satisfy the selection criteria of these queries is very small. Thus, decomposing queries like those into a high number of fragments may not be the best strategy. Table II shows in detail the mean total time obtained in each query, according to the number of virtual partitions (Vpt).

These initial results show that SVP is an appropriate solution to improve performance of heavy-weight queries in XML model. However, queries which demand the access of small amounts of data may suffer impact on performance, due to costs with communication between the nodes involved and costs with data transmission. In such cases, the overhead may be greater than the benefits provided by parallel query processing. Although some queries have not reached a continuous decrease in the mean total time, all the analyzed queries have shown better performance with SVP technique than in the scenario which simulates the centralized environment. Even the query that had the worst performance executed two times faster using 32 virtual partitions when compared to the centralized environment.

We have also observed that queries whose results are not uniformly distributed over the XML document suffer impact on performance, since some of the nodes involved process a huge amount of data, whereas other nodes process none or a small amount of data. The selection predicates of some queries restrict the results to some part of the XML document and causes load imbalance, once some virtual partitions do not retrieve results, because the data on it do not satisfy the selection criteria. The adaptive approach [Lima et al. 2004] seems to be a suitable strategy to deal with this problem. We intend to investigate this in the XML scenario as future work.

6. CONCLUSIONS

This article has shown a solution to improve the performance of ad-hoc analytical XML queries. In opposite to existing solutions, which use physical partitioning techniques, we have proposed a solution which does not depend on the input queries, once we do not know the queries which will be submitted by the user in advance. Our proposal is based on virtual partitioning techniques which implement intra-query parallelism through query rewriting. We have adapted this technique to XML model, since it has been applied only in relational databases.

Our solution builds upon Simple Virtual Partitioning [Akal et al. 2002]. In our approach, we have defined specific rules to provide appropriate query rewriting of XML queries. We rewrite the queries by using the position of the elements in the documents. This allows us to define the intervals which characterize the subqueries properly. This work was developed to support XQuery queries over MD databases and SD databases. In order to support MD databases, we first define a new XML query which expresses the union of the documents that exist in the collection. Then, this new XML query is used as the main query on the SVP algorithm.

Our work uses the methodology for XQuery query processing over distributed XML databases presented in [Figueiredo et al. 2010]. We propose the use of a control node which is responsible for consolidating the results. We have implemented our proposal by using the Sedna native XML database [Fomichev et al. 2006]. It was developed in Java and uses the MPJExpress[8] library which provides methods to support parallel and distributed communication.

In order to evaluate the effectiveness of our proposal, we have performed several experiments in a real cluster. We have installed many instances of Sedna XDBMS on the shared stored area. Thus, each node accesses one instance to process its own subquery and to store its partial result. Our experiments have shown that our solution can achieve performance improvements of up 90% when compared to the centralized environment. Queries which involve the processing of huge amounts of data, as in join operations, have presented mean total time up to 22 times faster with SVP technique when compared to the centralized environment.

In our experiments, some queries did not present a continuous decrease in the mean total time. It is due to the fact that the data which satisfy the selection predicates of the queries are not uniformly distributed in the XML document. In such cases, some nodes process subqueries which return a huge amount of data, whereas other nodes process subqueries which do not return results or which return a small amount of data. For queries like those the performance may vary according to the number of partitions, since the results do not follow a uniform distribution and it causes impact on performance.

Although some queries have not provided good results due to load imbalance, we have shown that even with the Simple Virtual Partitioning technique we can achieve significant improvements in performance. Our experiments show that SVP is a suitable strategy to optimize the performance of ad-hoc analytical XML queries, since our technique does not depend on workload awareness. Further optimizations can be performed in order to avoid performance loss due to load imbalance. We intend to investigate the adaptive approach proposed in [Lima et al. 2004] to minimize this problem. The employment of the AVP into our work is the subject of ongoing research.

REFERENCES

Akal, F., Bohm, K., and Schek, H.-J. OLAP query evaluation in a database cluster: a performance study on Intra-Query parallelism. In *East-European Conference on Advances in Databases and Information Systems (ADBIS)*, Y. Manolopoulos and P. Navrat (Eds.). Lecture Notes in Computer Science, vol. 2435. Springer, pp. 218–231, 2002.

Andrade, A., Ruberg, G., Baião, F., Braganholo, V., and Mattoso, M. Efficiently processing XML queries over fragmented repositories with PartiX. In *International Workshop on Database Technologies for Handling XML Information on the Web (DATAX)*. Lecture Notes in Computer Science, vol. 4254. Springer, pp. 150–163, 2006.

Boussaid, O., Messaoud, R. B., Choquet, R., and Anthoard, S. X-warehousing: An xml-based approach for warehousing complex data. In *East-European Conference on Advances in Databases and Information Systems (ADBIS)*, Y. Manolopoulos, J. Pokorný, and T. K. Sellis (Eds.). Lecture Notes in Computer Science, vol. 4152. Springer, pp. 39–54, 2006.

Dean, J. and Ghemawat, S. MapReduce: simplified data processing on large clusters. *Communications of the ACM* 51 (1): 107–113, Jan., 2008.

Figueiredo, G., Braganholo, V., and Mattoso, M. Processing queries over distributed XML databases. *Journal of Information and Data Management* 1 (3): 455–470, 2010.

[8] http://mpjexpress.org/docs/guides/windowsguide.pdf

Fomichev, A., Grinev, M., and Kuznetsov, S. Sedna: A native XML DBMS. In *SOFSEM 2006: Theory and Practice of Computer Science*, J. Wiedermann, G. Tel, J. Pokorný, M. Bieliková, and J. Stuller (Eds.). Lecture Notes in Computer Science, vol. 3831. Springer, pp. 272–281, 2006.

Furtado, C., Lima, A., Pacitti, E., Valduriez, P., and Mattoso, M. Physical and virtual partitioning in OLAP database clusters. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Washington, DC, USA, pp. 143–150, 2005.

Golfarelli, M., Rizzi, S., and Vrdoljak, B. Data warehouse design from xml sources. In *ACM International Workshop on Data Warehousing and OLAP*. New York, NY, USA, pp. 40–47, 2001.

Khatchadourian, S., Consens, M. P., and Siméon, J. Having a ChuQL at XML on the Cloud. In *Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*. Santiago, Chile, pp. 1–12, 2011.

Kido, K., Amagasa, T., and Kitagawa, H. Processing XPath queries in PC-Clusters using XML data partitioning. In *International Conference on Data Engineering Workshops (ICDEW)*. Atlanta, GA, USA, pp. 114–114, 2006.

Kling, P., Özsu, M. T., and Daudjee, K. Generating efficient execution plans for vertically partitioned XML databases. *PVLDB* 4 (1): 1–11, October, 2010.

Kossmann, D. The state of the art in distributed query processing. *ACM Computing Surveys* 32 (4): 422–469, Dec., 2000.

Kurita, H., Hatano, K., Miyazaki, J., and Uemura, S. Efficient query processing for large XML data in distributed environments. In *International Conference on Advanced Information Networking and Applications (AINA)*. Niagara Falls, Canada, pp. 317–322, 2007.

Lima, A., Mattoso, M., and Valduriez, P. Adaptive virtual partitioning for OLAP query processing in a database cluster. In *Simpósio Brasileiro de Banco de Dados (SBBD)*. Brasília, Brazil, pp. 92–105, 2004.

Lima, A. A., Furtado, C., Valduriez, P., and Mattoso, M. Parallel OLAP query processing in database clusters with data replication. *Distributed and Parallel Databases* 25 (1-2): 97–123, 2009.

Mahboubi, H. and Darmont, J. Enhancing XML data warehouse query performance by fragmentation. In *ACM Symposium on Applied Computing (SAC)*. Honolulu, Hawaii, pp. 1555–1562, 2009.

Mattoso, M. Database clusters. In *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu (Eds.). Springer, pp. 700–704, 2009a.

Mattoso, M. Virtual partitioning. In *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu (Eds.). Springer, pp. 3340–3341, 2009b.

Moreira, L., Sousa, F. R., and Machado, J. C. A distributed concurrency control mechanism for XML data. *Journal of Computer and System Sciences* 77 (6): 1009–1022, 2011.

Paes, M., Lima, A. A. B., Valduriez, P., and Mattoso, M. High-Performance query processing of a Real-World OLAP database with ParGRES. In *High Performance Computing for Computational Science (VECPAR)*, J. M. Palma, P. R. Amestoy, M. Daydé, M. Mattoso, and J. a. C. Lopes (Eds.). Lecture Notes in Computer Science, vol. 5336. Springer, pp. 188–200, 2008.

Park, B., Han, H., and Song, I. XML-OLAP: a multidimensional analysis framework for XML warehouses. In *Data Warehousing and Knowledge Discovery*, A. M. Tjoa and J. Trujillo (Eds.). Lecture Notes in Computer Science, vol. 3589. Springer, pp. 32–42, 2005.

Pokorný, J. XML Data Warehouse: Modelling and querying. In *Baltic Conference on Databases and Information Systems (BalticDB&IS)*. Tallinn, Estonia, pp. 267–280, 2002.

Rohm, U., Bohm, K., and Schek, H. OLAP query routing and physical design in a database cluster. In *International Conference on Extending Database Technology: Advances in Database Technology*. London, UK, pp. 254–268, 2000.

Sousa, D. X., Lifschitz, S., and Valduriez, P. Blast distributed execution on partitioned databases with primary fragments. In *High Performance Computing for Computational Science (VECPAR)*. Lecture Notes in Computer Science, vol. 5336. pp. 544–554, 2008.

Vela, B., Blanco, C., Fernández-Medina, E., and Marcos, E. Model driven development of secure XML data warehouses: a case study. In *International Conference on Extending Database Technology and International Conference on Database Theory (EDBT/ICDT Workshops)*. Lausanne, Switzerland, pp. 10:1–10:8, 2010.

Waldvogel, M., Kramis, M., and Graf, S. Distributing XML with focus on parallel evaluation. In *International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P)*. Auckland, New Zealand, pp. 55–67, 2008.

Walton, C. B., Dale, A. G., and Jenevein, R. M. A taxonomy and performance model of data skew effects in parallel joins. In *International Conference on Very Large Data Bases (VLDB)*. San Francisco, CA, USA, pp. 537–548, 1991.

Wiederhold, G. Mediators in the architecture of future information systems. *IEEE Computer* 25 (3): 38–49, 1992.