Efficiently Processing XML Queries over Fragmented Repositories with PartiX

Alexandre Andrade[1], Gabriela Ruberg[1], Fernanda Baião[2], Vanessa P. Braganholo[1], and Marta Mattoso[1]

[1] Computer Science Department, COPPE/Federal Univ. of Rio de Janeiro, Brazil
[2] Applied Informatics Department, University of Rio de Janeiro, Brazil
{alexsilv,gruberg,vanessa,marta}@cos.ufrj.br,
fernanda.baiao@uniriotec.br

Abstract. The data volume of XML repositories and the response time of query processing have become critical issues for many applications, especially for those in the Web. An interesting alternative to improve query processing performance consists in reducing the size of XML databases through fragmentation techniques. However, traditional fragmentation definitions do not directly apply to collections of XML documents. This work formalizes the fragmentation definition for collections of XML documents, and shows the performance of query processing over fragmented XML data. Our prototype, PartiX, exploits intra-query parallelism on top of XQuery-enabled sequential DBMS modules. We have analyzed several experimental settings, and our results showed a performance improvement of up to a 72 scale up factor against centralized databases.

1 Introduction

In the relational [15] and object-oriented data models [4], data fragmentation has been used successfully to efficiently process queries. One of the key factors to this success is the formal definition of fragments and their correctness rules for transparent query decomposition. Recently, several fragmentation techniques for XML data have been proposed in literature [1, 2, 6–8, 12]. Each of these techniques aims at a specific scenario: data streams [7], peer-to-peer [1, 6], Web-Service based systems [2], etc.

In our work, we focus on high performance of XML data servers. In this scenario, we may have a single large document (SD), or large collections of documents (MD) over which XML queries are posed. For this scenario, however, existent fragmentation techniques [1, 2, 6–8, 12] do not apply. This is due to several reasons. First of all, they do not clearly distinguish between horizontal, vertical and hybrid fragmentation, which makes it difficult to automatically decompose queries to run over the fragments. Second, none of them present the fragmentation correctness rules, which are essential for the XML data server to verify the correctness of the XML fragments and then apply the reconstruction rule to properly decompose queries. Also, for large XML repositories, it is important to have a fragmentation model close to the traditional fragmentation techniques, so it can profit as much as possible from well-known results. Third, the query processing techniques are specific for the scenarios where they were proposed, and thus do not apply to our scenario. For instance, the model proposed in [7] for stream data

does not support horizontal fragmentation. The same happens in [2], where fragmentation is used for efficient XML data exchange through Web services. Finally, the lack of distinction between SD and MD prevents the distributed query processing of the MD collection [6, 12].

Thus, to efficiently answer queries over large XML repositories using an XML data server, we need a precise definition of XML fragmentation and a high performance environment, such as a cluster of PCs. This way, queries can be decomposed in subqueries which may run in parallel at each cluster node, depending on how the database is fragmented. In this paper, we are interested in the empirical assessment of data fragmentation techniques for XML repositories. We formalize the main fragmentation alternatives for collections of XML documents. We also contribute by defining the rules that verify the correctness of a fragment definition. Our fragmentation model is formal and yet simple when compared to related work. We consider both SD and MD repositories. To address the lack of information on the potential gains that can be achieved with partitioned XML repositories, we present experimental results for horizontal, vertical and hybrid fragmentation of collections of XML documents. The experiments were run with our prototype named PartiX. Sketches of algorithms for query decomposition and result composition are available at [3]. Our results show substantial performance improvements, of up to a 72 scale up factor compared to the centralized setting, in some relevant scenarios.

This paper is organized as follows. In Section 2, we discuss related work. Section 3 presents some basic concepts on XML data model and query language, and formalizes our fragmentation model. Section 4 shows the architecture of PartiX. Our experimental results and corresponding analysis are presented in Section 5. Section 6 closes this work with some final remarks and research perspectives.

2 Related Work

In this section, we briefly present related work. A more detailed discussion can be found in [3]. Foundations of distributed database design for XML were first addressed in [8] and [12]. Ma and Schewe [12] propose three types of XML fragmentation: *horizontal*, which groups elements of a single XML document according to some selection criteria; *vertical*, to restructure a document by unnesting some elements; and a special type named *split*, to break an XML document into a set of new documents. However, these fragmentation types are not clearly distinguished. For example, horizontal fragmentation involves data restructuring and elements projection, thus yielding fragments with different schema definitions.

Our definition of vertical XML fragmentation is inspired in the work of Bremer and Gertz [8]. They propose an approach for distributed XML design, covering both data fragmentation and allocation. Nevertheless, their approach only addresses SD repositories. Moreover, their formalism does not distinguish between horizontal and vertical fragmentation, which are combined in a hybrid type of fragment definition. They maximize local query evaluation by replicating global information, and distributing some indexes. They present performance improvements, but their evaluation focuses on the benefits of such indexes.

Different definitions of XML fragments have been used in query processing over streamed data [7], peer-to-peer environments [1, 6], and Web-Service based scenarios [2]. However, they either do not present fragmentation alternatives to SD and MD [6], or do not distinguish between the different fragmentation types [1, 2, 6, 7]. In PartiX, we support horizontal, vertical and hybrid fragmentation of XML data for SD and MD repositories. Furthermore, we have implemented a PartiX prototype, and performed several tests to evaluate the performance of these fragmentation alternatives. No work in the literature presents experimental analysis of the query processing response time on fragmented XML repositories.

3 XML Data Fragmentation

3.1 Basic Concepts

XML documents consist of trees with nodes labeled by element names, attribute names or constant values. Let $\mathcal{L}$ be the set of distinct element names, $\mathcal{A}$ the set of distinct attribute names, and $\mathcal{D}$ the set of distinct data values. An *XML data tree* is denoted by the expression $\Delta := \langle t, \ell, \Psi \rangle$, where: t is a finite ordered tree, ℓ is a function that labels nodes in t with symbols in $\mathcal{L} \cup \mathcal{A}$; and Ψ maps leaf nodes in t to values in $\mathcal{D}$. The root node of Δ is denoted by $root_\Delta$. We assume nodes in Δ do not have mixed content; if a given node v is mapped into $\mathcal{D}$, then v does not have siblings in Δ. Notice, however, that this is not a limitation, but rather a presentation simplification. Furthermore, nodes with labels in $\mathcal{A}$ have a single child whose label must be in $\mathcal{D}$. An XML document is a data tree.

Basically, names of XML elements correspond to names of data types, described in a DTD or XML Schema. Let S be a schema. We say that document $\Delta := \langle t, \ell, \Psi \rangle$ *satisfies* a type τ, where $\tau \in S$, iff $\langle t, \ell \rangle$ is a tree derived from the grammar defined by S such that $\ell(root_\Delta) \rightarrow \tau$. A *collection* C of XML documents is a set of data trees. We say it is *homogeneous* if all the documents in C satisfy the same XML type. If not, we say the collection is *heterogeneous*. Given a schema S, a homogeneous collection C is denoted by the expression $C := \langle S, \tau_{root} \rangle$, where τ_{root} is a type in S and all instances Δ of C satisfy τ_{root}.

Figure 1(a) shows the $S_{virtual_store}$ schema tree, which we use in the examples throughout the paper. In this Figure, we indicate the minimum and maximum cardinalities (assuming cardinality 1..1 when omitted). The main types in $S_{virtual_store}$ are `Store` and `Item`, which describe a virtual store and the items it sells. Items are associated with sections and may have descriptive characteristics. Items may also have a list of pictures to be used in the virtual store, and a history of prices. Figure 1(b) shows the definition of the homogeneous collections C_{store} and C_{items}, based on $S_{virtual_store}$.

We consider two types of XML repositories, as mentioned in [17]. An XML repository may be composed of several documents (*Multiple Documents*, MD) or by a single large document which contains all the information needed (*Single Document*, SD). The collection C_{items} of Figure 1(b) corresponds to an MD repository, whereas the collection C_{store} is an SD repository.

A *path expression* P is a sequence /e_1/. . ./{e_k | @a_k}, where $\mathsf{e}_x \in \mathcal{L}$, $1 \leq x \leq k$, and $\mathsf{a}_k \in \mathcal{A}$. P may optionally contain the symbols "$*$" to indicate any element, and "//"

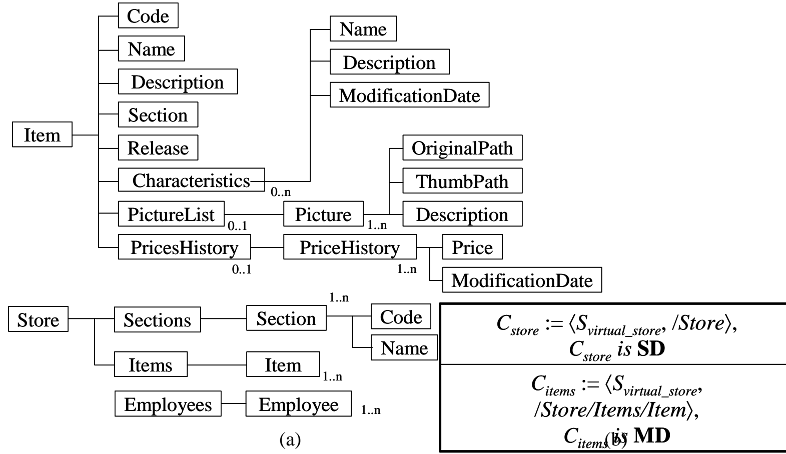

Fig. 1. (a) $S_{virtual_store}$ schema (b) Specification of collections C_{Store} and C_{Items}

to indicate any sequence of descendant elements. Besides, the term $\mathsf{e}[i]$ may be used to denote the i-th occurrence of element e. The evaluation of a path expression P in a document Δ represents the selection of all nodes with label e_k (or a_k) whose steps from $root_\Delta$ satisfy P. P is said to be *terminal* if the content of the selected nodes is simple (that is, if they have domain in $\mathcal{D}$). On the other hand, a *simple predicate* p is a logical expression: $p := \ \ P\ \theta\ value\ \mid\ \phi_v(P)\ \theta\ value\ \mid\ \phi_b(P)\ \mid\ Q$, where P is a terminal path expression, $\theta \in \{=,<,>,\neq,\leq,\geq\}$, $value \in \mathcal{D}$, ϕ_v is a function that returns values in $\mathcal{D}$, ϕ_b is a boolean function and Q denotes an arbitrary path expression. In the latter case, p is true if there are nodes selected by Q (existential test).

3.2 XML Fragmentation Techniques

The subject of data fragmentation is well known in relational [15] and object databases [4]. Traditionally, we can have three types of fragments: *horizontal*, where instances are grouped by selection predicates; *vertical*, which "cuts" the data structure through projections; and/or *hybrid*, which combines selection and projection operations in its definition. Our XML fragmentation definition follows the semantics of the operators from the TLC algebra [16], since it is one of the few XML algebras [9, 10, 18] that uses collections of documents, and thus is adequate to the XML data model defined in Section 3.1. In [3], we show how fragment definitions in PartiX can be expressed with TLC operators. In XML repositories, we consider that the fragmentation is defined over the schema of an XML collection. In the case of an MD XML database, we assume that the fragmentation can only be applied to homogeneous collections.

Definition 1. *A fragment F of a homogeneous collection C is a collection represented by $F := \langle C, \gamma \rangle$, where γ denotes an operation defined over C. F is horizontal if γ denotes a selection; vertical, if operator γ is a projection; or hybrid, when there is a composition of select and project operators.*

(a)	$F1_{CD} := \langle C_{items}, \sigma_{\text{/Item/Section="CD"}} \rangle$ $F2_{CD} := \langle C_{items}, \sigma_{\text{/Item/Section}\neq\text{"CD"}} \rangle$	(b)	$F1_{good} := \langle C_{items}, \sigma_{\text{contains(//Desciption, "good")}} \rangle$ $F2_{good} := \langle C_{items}, \sigma_{\text{not(contains(//Desciption, "good"))}} \rangle$
		(c)	$F1_{with_pictures} := \langle C_{items}, \sigma_{\text{/Item/PictureList}} \rangle$ $F2_{with_pictures} := \langle C_{items}, \sigma_{\text{empty(/Item/PictureList)}} \rangle$

Fig. 2. Examples of three alternative fragments definitions over the collection C_{items}

Instances of a fragment F are obtained by applying γ to each document in C. The set of the resulting documents form the fragment F, which is valid if all documents generated by γ are well-formed (i.e., they must have a single root).

We now detail and analyze the main types of fragmentation in XML. However, we first want to make clear our goal in this paper. Our goal is to show the advantages of fragmenting XML repositories in query processing. Therefore, we formally define the three typical types of XML fragmentation, present correctness criteria for each of them, and compare the performance of queries stated over fragmented databases with queries over centralized databases.

Horizontal Fragmentation. This technique aims to group data that is frequently accessed in isolation by queries with a given selection predicate. A horizontal fragment F of a collection C is defined by the selection operator (σ) [10] applied over documents in C, where the predicate of σ is a boolean expression with one or more simple predicates. Thus, F has the same schema of C.

Definition 2. *Let μ be a conjunction of simple predicates over a collection C. The horizontal fragment of C defined by μ is given by the expression $F := \langle C, \sigma_\mu \rangle$, where σ_μ denotes the selection of documents in C that satisfy μ, that is, F contains documents of C for which σ_μ is true.*

Figure 2 shows the specification of some alternative horizontal fragments for the collection C_{items} of Figure 1(b). For instance, fragment $F1_{good}$ (Figure 2(b)) groups documents from C_{items} which have `Description` nodes that satisfy the path expression `//Description` (that is, `Description` may be at any level in C_{items}) and that contain the word "good". Alternatively, one can be interested in separating, in different fragments, documents that have/have not a given structure. This can be done by using an existential test, and it is shown in Figure 2(c). Although $F1_{with_pictures}$ and C_{items} have the same schema, in practice they can have different structures, since the element used in the existential test is mandatory in $F1_{with_pictures}$. Observe that $F1_{with_pictures}$ cannot be classified as a vertical nor hybrid fragment.

Notice that, by definition, SD repositories may not be horizontally fragmented, since horizontal fragmentation is defined over documents (instead of nodes). However, the elements in an SD repository may be distributed over fragments using a hybrid fragmentation, as described later in this paper.

Vertical Fragmentation. It is obtained by applying the projection operator (π) [16] to "split" a data structure into smaller parts that are frequently accessed in queries. Observe that, in XML repositories, the projection operator has a quite sophisticated semantics: it is possible to specify projections that exclude subtrees whose root is located in any level of an XML tree. A projection over a collection C retrieves, in each document of

(a)	$F1_{items} := \langle C_{items}, \pi_{\text{/Item, \{/Item/PictureList\}}} \rangle$ $F2_{items} := \langle C_{items}, \pi_{\text{/Item/PictureList, \{\}}} \rangle$	(b)	$F1_{sections} := \langle C_{store}, \pi_{\text{/Store/Sections, \{\}}} \rangle$ $F2_{section} := \langle C_{store}, \pi_{\text{/Store, \{/Store/Sections\}}} \rangle$

Fig. 3. Examples of vertical fragments definitions over collections C_{items} and C_{store}

$$F1_{items} := \langle C_{store}, \pi_{\text{/Store/Items, \{\}}} \bullet \sigma_{\text{/Item/Section="CD"}} \rangle$$
$$F2_{items} := \langle C_{store}, \pi_{\text{/Store/Items, \{\}}} \bullet \sigma_{\text{/Item/Section="DVD"}} \rangle$$
$$F3_{items} := \langle C_{store}, \pi_{\text{/Store/Items,\{\}}} \bullet \sigma_{\text{/Item/Section}\neq\text{"CD"} \wedge \text{/Item/Section}\neq\text{"DVD"}} \rangle$$
$$F4_{items} := \langle C_{store}, \pi_{\text{/Store,\{/Store/Items\}}} \rangle$$

Fig. 4. Examples of hybrid fragments over collection C_{store}

C (notice that C may have a single document, in case it is of type SD), a set of subtrees represented by a path expression, which are possibly pruned in some descendant nodes.

Definition 3. *Let P be a path expression over collection C. Let $\Gamma := \{E_1, \ldots, E_x\}$ be a (possibly empty) set of path expressions contained in P (that is, path expressions in which P is a prefix). A vertical fragment of C defined by P is denoted $F := \langle C, \pi_{P,\Gamma} \rangle$, where $\pi_{P,\Gamma}$ denotes the projection of the subtrees rooted by nodes selected by P, excluding from the result the nodes selected by the expressions in Γ. The set Γ is called the prune criterion of F.*

It is worth mentioning that the path expression P cannot retrieve nodes that may have cardinality greater than one (as it is the case of `/Item/PictureList/Picture`, in Figure 1(a)), except when the element order is indicated (e.g. `/Item/PictureList/Picture[1]`). This restriction assures that the fragmentation results in well-formed documents, without the need of generating artificial elements to reorganize the subtrees projected in a fragment.

Figure 3 shows examples of vertical fragments of the collections C_{items} and C_{store}, defined on Figure 1(b). Fragment $F2_{items}$ represents the documents that contain all `PictureList` nodes that satisfy the path `/Item/PictureList` in the collection C_{items} (no prune criterion is used). On the other hand, nodes that satisfy `/Item/PictureList` are exactly the ones pruned out the subtrees rooted in `/Item` in the fragment $F1_{items}$, thus preserving disjointness with respect to $F2_{items}$.

Hybrid Fragmentation. The idea here is to apply a vertical fragmentation followed by a horizontal fragmentation, or vice-versa. An interesting use of this technique is to normalize the schema of XML collections in SD repositories, thereby allowing horizontal fragmentation.

Definition 4. *Let σ_μ and $\pi_{P,\Gamma}$ be selection and projection operators, respectively, defined over a collection C. A hybrid fragment of C is represented by $F := \langle C, \pi_{P,\Gamma} \bullet \sigma_\mu \rangle$, where $\pi_{P,\Gamma} \bullet \sigma_\mu$ denotes the selection of the subtrees projected by $\pi_{P,\Gamma}$ that satisfy σ_μ.*

The order of the application of the operations in $\pi_{P,\Gamma} \bullet \sigma_\mu$ depends on the fragmentation design. Examples of hybrid fragmentation are shown in Figure 4.

3.3 Correctness Rules of the Fragmentation

An XML distribution design consists of fragmenting collections of documents (SD or MD) and allocating the resulting fragments in sites of a distributed system, where each

collection is associated to a set of fragments. Consider that a collection C is decomposed into a set of fragments $\Phi := \{F_1, ..., F_n\}$. The following rules must be verified to guarantee the fragmentation of C is correct:

- *Completeness*: each data item in C must appear in at least one fragment $F_i \in \Phi$. In the horizontal fragmentation, the data item consists of an XML document, while in the vertical fragmentation, it is a node.
- *Disjointness*: for each data item d in C, if $d \in F_i$, $F_i \in \Phi$, then d cannot be in any other fragment $F_j \in \Phi$, $j \neq i$.
- *Reconstruction*: it must be possible to define an operator ∇ such that $C := \nabla F_i$, $\forall F_i \in \Phi$, where ∇ depends on the type of fragmentation. For horizontal fragmentation, the union ($\cup$) operator [10] is used (TLC is an extension of TAX [10]), and for vertical fragmentation, the join ($\bowtie$) operator [16] is used. We keep an ID in each vertical fragment for reconstruction purposes.

These rules are important to guarantee that queries are correctly translated from the centralized environment to the fragmented one, and that results are correctly reconstructed. Procedures to verify correctness depend on the algorithms adopted in the fragmentation design. As an example, some fragmentation algorithms for relations guarantee the correctness of the resulting fragmentation design [15]. Still others [14] require use of additional techniques to check for correctness. Such automatic verification is out of the scope of this paper.

Query Processing. By using our fragmentation definition, we can adopt a query processing methodology similar to the relational model [15]. The query expressed on the global XML documents can be mapped to its corresponding fragmented XML documents by using the fragmentation schema definition and reconstruction program. Then an analysis on query specification versus schema definition can proceed with data localization. Finally global query optimization techniques can be adopted [10, 16].

Figure 5 sketches the query processing in PartiX. The overall idea is that PartiX works as a middleware between the user application and a set of DBMS servers, which actually store the distributed XML data. Information on data distribution is kept by PartiX: when a query arrives, PartiX analyzes the fragmentation schema to properly split it into sub-queries, and then sends each sub-query to its respective fragment. Also, PartiX gathers the results of the sub-queries and reconstructs the query answer. Notice some queries may involve a single fragment, and that in this case, no result reconstruction is

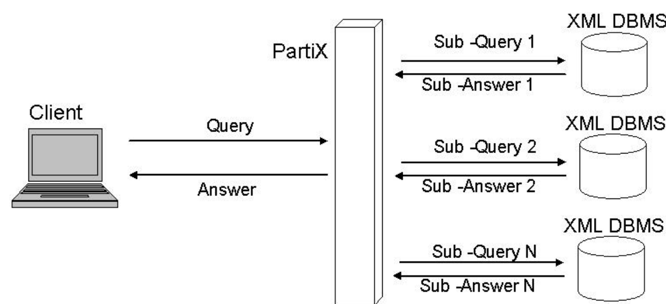

Fig. 5. Query Processing in PartiX

needed. In general, defining query rewriting and data localization is a complex research issue, which can benefit from our formal fragmentation model. Yet, we leave such a definition as future work. In the next section, we detail the PartiX architecture.

4 The PartiX Architecture

We propose an architecture to process XQuery queries in distributed XML data sets. Our architecture uses DBMS with no distribution support, and applies our XML fragmentation model, shown in Section 3. The goal of this architecture, named PartiX, is to offer a system which coordinates the distributed processing of XQuery queries. In our distributed environment, a sequencial XML-enabled DBMS is installed at all nodes, which are coordinated by PartiX. In this way, there is no need of buying a specific distributed DBMS.

Generally speaking, PartiX intercepts an XQuery query before it reaches the XML DBMS. PartiX analyzes the definition of the fragments and rewrites the query as sub-queries accordingly (see details for horizontal fragmentation in [3]). Then, it sends these sub-queries to the PartiX components installed in the corresponding DBMS nodes, and collects the partial results. Our architecture is illustrated in the PartiX system, shown in Figure 6. It is composed of three main parts: (i) *catalog services*, which are used to publish schema and distribution metadata; (ii) *publishing service* for distributed XML data; and (iii) distributed *query service*.

The *XML Schema Catalog Service* registers the data types used by the distributed collections, while the *XML Distribution Catalog Service* stores the fragment definitions. The *Distributed XML Data Publisher* receives XML documents from users, applies the fragmentation that was previously defined to the collections, and sends the resulting fragments to be stored in the remote DBMS nodes. XQuery queries are submitted to the *Distributed XML Query Service*, which analyzes their path expressions and identifies the fragments referenced in each query. It writes the sub-queries that are sent to the corresponding DBMS nodes, constructs the result, and sends it to the user.

Our architecture considers that there is a *PartiX Driver*, which allows accessing remote DBMSs to store and retrieve XML documents. This driver provides a uniform communication interface between the PartiX modules and the XML DBMS nodes that

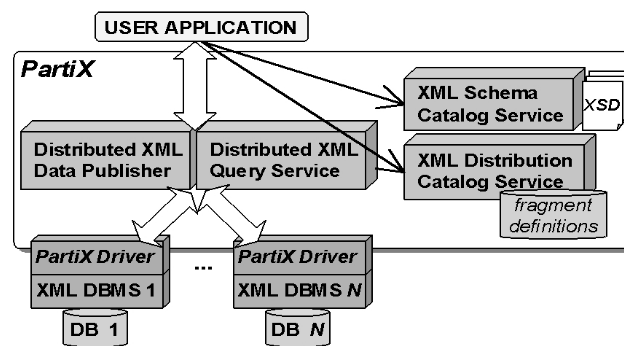

Fig. 6. PartiX Architecture

host the distributed collections. The *PartiX driver* allows different XML DBMSs to participate in the system. The only requirement is that they are able to process XQuery.

The proposed architecture is implemented in a prototype of the PartiX system. We have developed a PartiX *driver* to the eXist DBMS [13]. The *Data Publisher* interprets loading scripts and stores the documents of a collection in the XML DBMSs. In the PartiX prototype, we did not implement automatic query decomposition, and we consider that data location is provided along with sub-queries. However, given a decomposed query, the *query service* is capable of coordinating the distributed execution of the sub-queries annotated with the location of the required data fragments.

In the next section we show a performance evaluation of queries over fragmented repositories using PartiX.

5 Experimental Evaluation

This section presents experimental results obtained with the PartiX implementation for horizontal, vertical and hybrid fragmentation. We evaluate the benefits of data fragmentation for the performance of query processing in XML databases. We used a 2.4Ghz Athlon XP with 512Mb RAM memory in our tests. We describe the experimental scenario we have used for each of the fragmentation types: horizontal, vertical and hybrid, and show that applying them in XML data collections have reduced the query processing times.

We applied the ToXgene [5] XML database generator to create the C_{store} and C_{items} collections, as defined in Figures 1(a) and (b), and also a collection for the schema defined in the XBench benchmark [17]. All of them were stored in the eXist DBMS [13]. Four databases were generated for the tests: *database ItemsSHor*, with document sizes of 2K in average, and elements `PriceHistory` and `ImagesList` with zero occurrences (Figure 1(a)); *database ItemsLHor*, with document sizes of 80Kb in average (Figure 1(a)); *database XBenchVer*, with the XBench collections, with document sizes varying from 5Mb to 15Mb each; and database StoreHyb (Figure 1(a)), with document sizes from 5Mb to 500Mb. Experiments were conducted varying the number of documents in each database to evaluate the performance of fragmentation for different database sizes (5Mb, 20Mb, 100Mb and 250Mb for all databases, and 500Mb for databases ItemsLHor and StoreHyb). (Due to space restrictions, in this paper we show only the results for the 250Mb database. The remaining results are available at [3].) Some indexes were automatically created by the eXist DBMS to speed up text search operations and path expressions evaluation. No other indexes were created.

Each query was decomposed in sub-queries (according to [3]) to be processed with specific data fragments. When the query predicates match the fragmentation predicates, the sub-queries are issued only to the corresponding fragments. After each sub-query is executed, we compose the result to compute the final query answer [3]. The parallel execution of a query was simulated assuming that all fragments are placed at different sites and that the sub-queries are executed in parallel in each site. For instance, in Figure 7(a) with 8 fragments, we can have at most 8 sub-queries running in parallel. We have used the time spent by the slowest site to produce the result. We measured the communication time for sending the sub-queries to the sites and for transmitting their partial results, since there is no inter-node communication. This was done by calculating the

average size of the result and dividing it by the Gigabit Ethernet transmission speed. For all queries we have measured the time between the moment PartiX receives the query until final result composition.

In our experiments, each query was submitted 10 times, and the execution time was calculated by discarding the first execution time and calculating the average of the remaining results. We have measured the execution times of each sub-query. More details on our experiments are available in [3].

Horizontal Fragmentation. For horizontal fragmentation, the tests were run using a set of 8 queries [3], which illustrates diverse access patterns to XML collections, including the usage of predicates, text searches and aggregation operations. The XML database was fragmented as follows. The C_{Items} collection was horizontally fragmented by the "Section" element, following the correctness rules of Section 3.3. We varied the number of fragments (2, 4 and 8) with a non-uniform document distribution. The fragments definitions are shown in [3].

Figure 7(a) contains the performance results of the PartiX execution on top of database ItemsSHor, and Figure 7(b) on database ItemsLHor, in the scenarios previously described. The results show that the fragmentation reduces the response time for most of the queries. When comparing the results of databases ItemsSHor and ItemsLHor with a large number of documents, we observe that the eXist DBMS presents better results when dealing with large documents. This is due to some pre-processing operations (e.g., parsing) that are carried out for each XML tree. For example, when using a 250Mb database size and centralized databases, query Q8 is executed in 1200s in ItemsSHor, and in 31s in ItemsLHor. When using 2 fragments, these times are reduced to 300s and 14s, respectively. Notice this is a superlinear speedup. This is very common also in relational databases, and is due to reduction of I/O operations and better use of machine resources such as cache and memory, since a smaller portion of data is being processed at each site.

An important conclusion obtained from the experiments relates to the benefit of horizontal fragmentation. The execution time of queries with text searches and aggregation operations (Q5, Q6, Q7 and Q8) is significantly reduced when the database is horizontally fragmented. It is worth mentioning that text searches are very common in XML applications, and typically present poor performance in centralized environments, sometimes prohibiting their execution. This problem also occurs with aggregation operations. It is important to notice that our tests included an aggregation function (count) that may be entirely evaluated in parallel, not requiring additional time for reconstructing the global result.

Another interesting result can be seen in the execution of Q6. As the number of fragments increases, the execution time of Q6 increases in some cases. This happens because eXist generates different execution plans for each sub-query, thus favoring the query performance in case of few fragments. Yet, all the distributed configurations performed better than the centralized database.

As expected, in small databases (i.e., 5Mb) the performance gain obtained is not enough to justify the use of fragmentation [3]. Moreover, we concluded that the document size is very important for defining the fragmentation schema. Database ItemsL-

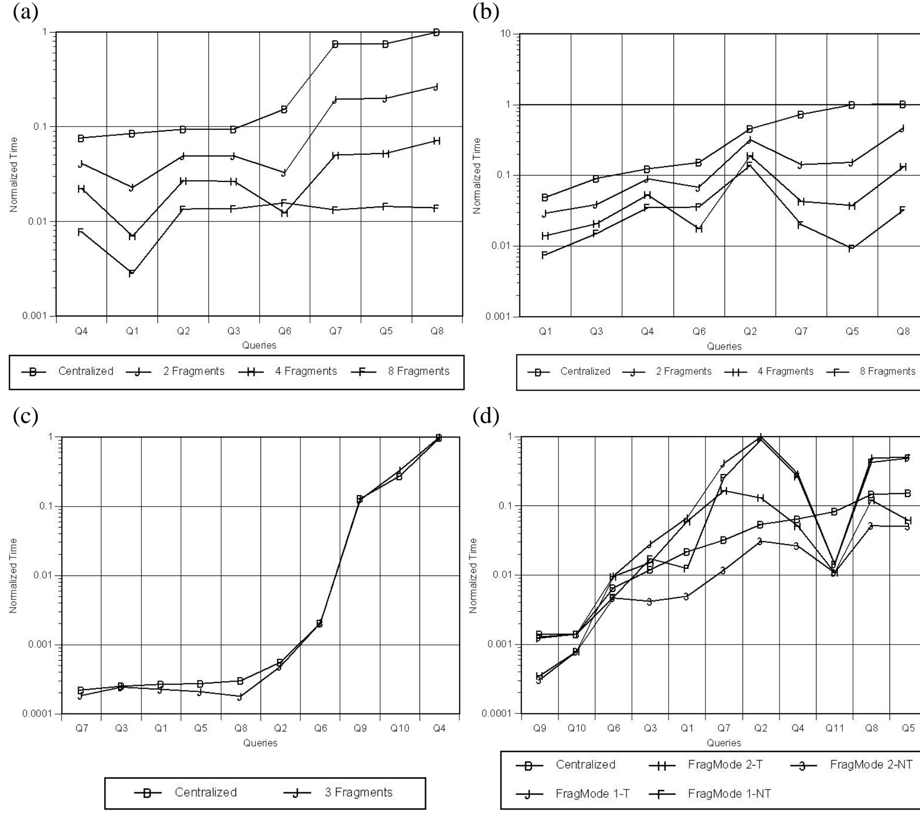

Fig. 7. Experimental results for databases (a) ItemsSHor and (b) ItemsLHor - horizontal fragmentation; (c) XBenchVer - vertical fragmentation; (d) StoreHyb with and without transmission times - hybrid fragmentation

Hor (Figure 7(b)) presents better results with few fragments, while database ItemsSHor presents better results with many fragments.

Vertical Fragmentation. For the experiments with vertical fragmentation, we have used the XBenchVer database and some of the queries specified in XBench [17], which are shown in [3]. We have named them Q1 to Q10, although these names do not correspond to the names used in the XBench document.

Database *XBenchVer* was vertically fragmented in three fragments:

- $F1_{papers} := \langle C_{papers}, \pi_{/article/prolog} \rangle$,
- $F2_{papers} := \langle C_{atigos}, \pi_{/article/body} \rangle$, and
- $F3_{papers} := \langle C_{artigos}, \pi_{/article/epilog} \rangle$.

Figure 7(c) shows the performance results of PartiX in this scenario. In the 5Mb database, we had gains in all queries, except for Q4 and Q10 [3]. With vertical fragmentation, the main benefits occur for queries that use a single fragment. Since queries

Q4, Q7, Q8 and Q9 need more than one fragment, they can be slowed down by fragmentation. Query Q4 does not present performance gains in any case, except for a minor improvement in the 100Mb database [3]. We believe once more that some statistics or query execution plan has favored the execution of Q4 in this database. In the 20Mb database, all queries presented performance gains (except for Q4), including Q10, which had presented poor performance in the 5Mb database.

As the database size grows, the performance gains decreases. In the 250Mb database, queries Q6, Q9 and Q3 perform equivalently to the centralized approach. With these results, we can see that vertical fragmentation is useful when the queries use few fragments. The queries with bad performance were those involving text search, since in general, they must be applied to all fragments. In such case, the performance is worse than for horizontal fragmentation, since the result reconstruction requires a join (much more expensive than a union).

Hybrid Fragmentation. In the experiments with hybrid fragmentation, we have used the C_{Store} collection fragmented into 5 different fragments. Fragment F_1 prunes */Store/Items*, while the remaining 4 fragments are all about Items, each of them horizontally fragmented over */Store/Items/Item/Section*. We call this database *StoreHyb*, and the set of queries defined over it is shown in [3].

As we will see later on, the experimental results with hybrid fragmentation were heavily influenced by the size of the returning documents. Because of this, we show the performance results with and without the transmission times.

We consider the same queries and selection criteria adopted for databases ItemsSHor e ItemsLHor, with some modifications. With this, most of the queries returned all the content of the "Item" element. This was the main performance problem we have encountered, and it affected all queries. This serves to demonstrate that, besides a good fragmentation design, queries must be carefully specified, so that they do not return unnecessary data. Because XML is a verbose format, an unnecessary element may carry a subtree of significant size, and this will certainly impact in the query execution time.

Another general feature we have noticed while making our tests was that the implementation of the horizontal fragment affects the performance results of the hybrid fragmentation. To us, it was natural to take the single document representing the collection, use the prune operation, and, for each "Item" node selected, to generate an independent document and store it. This approach, which we call *FragMode1*, has proved to be very inefficient. The main reason for this is that, in these cases, the query processor has to parse hundreds of small documents (the ones corresponding to the "Item" fragments), which is slower than parsing a huge document a single time. To solve this problem, we have implemented the horizontal fragmentation with a single document (SD), exactly like the original document, but with only the item elements obtained by the selection operator. We have called this approach *FragMode2*. As we will see, this fragmentation mode beats the centralized approach in most of the cases.

When we consider the transmission times (*FragModeX-T* in Figure 7(e)), FragMode1 performs worse for all database sizes, for all queries, except for queries Q9, Q10 and Q11. Queries Q9 and Q10 are those that prune the "Items" element, which makes the parsing of the document more efficient. Query Q11 uses an aggregation function that presented a good performance in the 100Mb database and in larger ones. In the

remaining databases, it presented poor performance (5Mb database) or an anomalous behavior (20Mb database) [3].

Notice the FragMode2 performs better, although it does not beat the centralized approach in all cases. In the 5Mb database, it wins in queries Q3, Q4, Q5 and Q6, which benefit from the parallelism of the fragments and from the use of a specific fragment. As in the FragMode1, queries Q9 and Q10 always performs better than the centralized case; query Q11 only looses in the 5Mb database.

As the database size grows, the query results also increase, thus increasing the total query processing time. In the 20Mb database, query Q6 performs equivalently to the centralized approach. In the 100Mb database, this also happens to Q3 and Q6. In the 250Mb database, these two queries perform worse than in the centralized approach. Finally, in the 500Mb database, query Q4 also performs equivalently to the centralized case, and the remaining ones loose.

As we could see, the transmission times were decisive in the obtained results. Without considering this time, FragMode2 wins in all databases, in all queries, except for query Q11 in the 5Mb database. However, FragMode1 has shown to be effective in some cases. Figure 7(e) shows the experimental results without the transmission times (*FragModeX-NT*). It shows that hybrid fragmentation reduces the query processing times significantly.

6 Conclusions

This work presents a solution to improve the performance in the execution of XQuery queries over XML repositories. This is achieved through the fragmentation of XML databases. We present a formal definition for the different types of XML fragments, and define correctness criteria for the proposed fragmentation model. By specifying the concept of collections of XML documents, we create an abstraction where fragment definitions apply to both single and multiple document repositories (SD and MD). These concepts are not found in related work, and they are fundamental to perform query decomposition and result composition.

Our experiments highlight the potential for significant gains of performance through XML fragmentation. The reduction in the time of query execution is obtained by intra-query parallelism, and also by the local execution, avoiding scanning unnecessary fragments. The queries executed by PartiX with the eXist DBMS present an estimated execution time up to 72 times smaller (for horizontal fragmentation) when compared to centralized executions. The PartiX architecture [3] is generic, and can be plugged to any XML DBMS that process XQuery queries. This architecture follows the approach of *database clusters* that have been presenting excellent performance results over relational DBMSs [11].

As future work, we intend to use the proposed fragmentation model to define a methodology for fragmenting XML databases. This methodology could be used define algorithms for the fragmentation design [18], and to implement tools to automate this fragmentation process. We are also working on detailing algorithms to automatically rewrite queries to run over the fragmented database.

Acknowledgements. The authors thank CNPq for partially funding this work. V. Braganholo thanks FAPERJ and G. Ruberg thanks the Central Bank of Brazil for their support. The contents of this paper express the viewpoint of the authors only.

References

1. S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *SIGMOD*, pages 527–538, 2003.
2. S. Amer-Yahia and Y. Kotidis. A web-services architecture for efficient XML data exchange. In *ICDE*, pages 523–534, 2004.
3. A. Andrade, G. Ruberg, F. Baião, V. Braganholo, and M. Mattoso. Partix: Processing XQueries over fragmented XML repositories. Technical Report ES-691, COPPE/UFRJ, 2005. `http://www.cos.ufrj.br/~vanessa`.
4. F. Baião, M. Mattoso, and G. Zaverucha. A distribution design methodology for object DBMS. *Distributed and Parallel Databases*, 16(1):45–90, 2004.
5. D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a template-based data generator for XML. In *WebDB*, pages 621–632, 2002.
6. A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath lookup queries in P2P networks. In *WIDM*, pages 48–55, 2004.
7. S. Bose, L. Fegaras, D. Levine, and V. Chaluvadi. XPath lookup queries in p2p networks. In *WIDM*, pages 48–55, 2004.
8. J.-M. Bremer and M. Gertz. On distributing XML repositories. In *WebDB*, 2003.
9. M. Fernández, J. Siméon, and P. Wadler. An algebra for XML query. In *FSTTCS*, pages 11–45, 2000.
10. H. Jagadish, L. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *DBPL*, pages 149–164, 2001.
11. A. Lima, M. Mattoso, and P. Valduriez. Adaptive virtual partitioning for olap query processing in a database cluster. In *SBBD*, pages 92–105, 2004.
12. H. Ma and K.-D. Schewe. Fragmentation of XML documents. In *SBBD*, 2003.
13. W. Meier. eXist: Open source native XML database, 2000. Available at: `http://exist.sourceforge.net`.
14. S. Navathe, K. Karlapalem, and M. Ra. A mixed fragmentation methodology for initial distributed database design. *Journal of Computer and Software Engineering*, 3(4), 1995.
15. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.
16. S. Paparizos, Y. Wu, L. Lakshmanan, and H. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD*, pages 71–82, 2004.
17. B. Yao, M. Ozsu, and N. Khandelwal. Xbench benchmark and performance testing of XML DBMSs. In *ICDE*, pages 621–632, 2004.
18. X. Zhang, B. Pielech, and E. Rundesnteiner. Honey, I shrunk the XQuery!: an XML algebra optimization approach. In *WIDM*, pages 15–22, 2002.