PartiX: processing XQuery queries over fragmented XML repositories

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

[1]Department of Computer Science and Engineering – COPPE/Federal University of Rio de Janeiro, Brazil
{alexsilv,gruberg,vanessa,marta}@cos.ufrj.br

[2]Department of Applied Informatics – UNIRIO, Brazil
fernanda.baiao@uniriotec.br

Technical Report ES-691
December, 2005

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 proposes an architecture for XQuery processing on top of fragmented XML data. This architecture was implemented in a system prototype named PartiX, which 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

XML and its related standards represent a breakthrough in information integration systems and distributed data management. They have raised the development of a number of XML-enabled and XML-native database systems [11,13,22]. Yet, using XML data still represents many challenges for query processing. Despite the efforts of both academic and industrial database research in the last years [1,5,10,20], efficiently processing queries in large XML repositories remains as an interesting open problem even for centralized scenarios.

Many factors influence the performance of XML query processing. In particular, the size of the database has usually a significant impact on typical tools for retrieving XML data. This happens because a basic functionality of these tools consists of XML parsing, namely verifying the validity of XML data with respect to predefined types described in DTDs or XML Schema definitions [23]. Most XML parsers are based on tree structures such as the Document Object Model (DOM) [23], whose data unit is the XML document. It turns out that type validation in XML often demands in-memory support structures that are usually proportional to the document size. Therefore, such a validation task might become quite time consuming on large XML repositories [21].

Another performance issue comes from the complexity of the standard XML query languages, XPath and XQuery [23], which are very powerful and require sophisticated algorithms. It is worth noting that typical XML queries, even the simplest ones, most often involve expensive operations such as text-based search and path expression evaluation. A basic, widely-exploited method to improve the performance of these queries consists in using pre-computed indexes. Nevertheless, some important classes of XML queries, especially ad-hoc, make it difficult to define specific indexes in advance. For those queries, large data volumes represent a performance burden even for systems with advanced optimization features [19].

Data fragmentation techniques have been used to help database systems to cope with large data volumes, in both relational [26] and object databases [3]. However, these techniques are not suited for XML databases, mainly due to the great flexibility of the semi-structured data model. Incomplete and redundant information are some of the inherent XML characteristics that complicate the definition of XML fragments. Checking for the *correctness* of an XML fragmentation schema (with respect to completeness, disjointness, and reconstruction [26]) is a hard problem in general. Furthermore, this problem is amplified by the complexity of typical XML query operators; for instance, XML selections (necessary to define horizontal fragments) usually involves path expressions containing regular terms, such as '//', and predicates with functions and/or quantifiers.

Guidelines to properly define XML fragments were outlined in [9] and [16]. However, when we tried to use these guidelines for different types of XML repositories, such as Single Document (SD) and Multiple Document (MD), we had difficulty in identifying a clear distinction for horizontal, vertical and hybrid XML fragmentation. For example, in [9] both horizontal and vertical fragmentations are blurred in the specification of an XML fragment. This also happens in [16], where horizontal fragmentation may involve data restructuring. Therefore, it is even harder to check for the correctness of such a combination of fragmentation techniques. On the other hand, the fragment definitions in [9] can only be applied to SD repositories. Similarly, to fragment an MD repository in [16], the user must specify an SD view of the database for each fragment definition. Namely, each XML document has to be properly connected to an intermediary view. Notice that this is a cumbersome approach even for small collections of documents. Moreover, to the best of our knowledge, current works on XML data fragmentation lack a *comprehensive experimental analysis* of the effects of different fragmentation techniques on the performance of query processing over large XML repositories.

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. Our fragmentation model is formal and yet simple when compared to related work. We consider both SD and MD repositories, and we also outline the verification of correctness rules in each case. We present an architecture to process XQuery statements over partitioned XML repositories based on our fragmentation model. The proposed architecture is represented by a system named PartiX. We have run our tests with a PartiX prototype over the XML-native open source DBMS eXist [11]. 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. 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. Section 2 presents some basic concepts on XML data model and query language, while in Section 3 we formalize our fragmentation model. The PartiX architecture is described in Section 4. Our experimental results and corresponding analysis are presented in Section 5. In Section 6, we discuss related work. Section 7 closes this work with some final remarks and research perspectives.

2. Basic Concepts

XML documents are represented by 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 is a function that maps leaf nodes in t to values in $\mathcal{D}$. The root node of Δ is denoted by $root_\Delta$. We assume that nodes in Δ do not have mixed content; if a given node v is mapped into $\mathcal{D}$, than v does not have siblings in Δ. Notice, however, that this is not a limitation. We have chosen not to consider mixed content elements to simplify the presentation of our ideas. 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* of XML documents represents 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 shows the $S_{virtual\_store}$ schema tree, which we use to illustrate 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 2 shows the definition of the homogeneous collections C_{store} and C_{items}, based on $S_{virtual\_store}$.

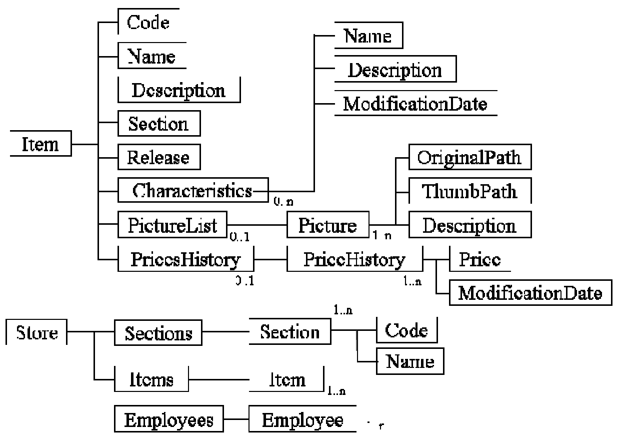

Figure 1: $S_{virtual\_store}$ schema

$$C_{store} := \langle S_{virtual\_store}, Store \rangle, \ C_{store} \ is \ \mathrm{SD}$$
$$C_{items} := \langle S_{virtual\_store}, Item \rangle, \ C_{items} \ is \ \mathrm{MD}$$

Figure 2: Specification of collections Cstore and Citems

We consider two types of XML repositories, as mentioned in [25]. 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 2 corresponds to an MD repository, whereas the collection C_{store} is an SD repository.

A *path expression P* is a sequence $/e_1/\ldots/\{e_k|@a_k\}$, where $e_x \in \mathcal{L}$, $1 \leq x \leq k$, and $a_k \in \mathcal{A}$. *P* may optionally contain the symbols "*" to indicate any element, and "//" to indicate any sequence of descendant elements. Besides, the term 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, $\theta \in \{=, <, >, \neq, \geq, \leq\}$, $value \in \mathcal{D}$, 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 later case, *p* is true if there are nodes selected by *Q* (existential test).

3. XML Data Fragmentation

The subject of data fragmentation is well known in relational [26] and object databases [3]. 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 fragmentation definition follows the semantics of the operators from the TLC algebra [18], since it is one of the few XML algebras [12,14,27,28] that uses collections of documents and it is adequate to the XML data model defined in Section 2. 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 for homogeneous collections.

Definition 1: *A* fragment F *of a homogeneous collection* C *is a collection represented by* F:= ⟨C, γ⟩*, 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 operators* select *and* project.

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*. A fragment *F* is valid if all documents generated by γ are well-formed [23] (in particular, they must have a single root).

We now detail and analyze the main types of fragmentation in XML. However, we first want to make it 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. Moreover, we show how fragment definitions in PartiX can be expressed with TLC operators. We discuss this in details in Section 3.4.

3.1. Horizontal Fragmentation

The main goal of the horizontal fragmentation is to group data that is frequently accessed in isolation by queries with a given selection predicate. A horizontal fragment of a collection *C* is defined by the selection operator (σ) [18] applied over the documents in *C*, where the predicate of σ is a boolean expression with one or more simple predicates. In this case, *F* has the same schema as *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 3 shows the specification of some alternative horizontal fragments for the collection C_{items} of Figure 2. For instance, fragment $F1_{good}$ (Figure 3(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 3(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 $\mathrm{F1}_{with\_pictures}$ cannot be classified as a vertical nor hybrid fragment.

(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)	$F1_{with\_\text{pictures}} := \langle C_{items}, \sigma_{\text{/Item/PictureList}} \rangle$ $F2_{with\_\text{pictures}} := \langle C_{items}, \sigma_{\text{empty(/Item/PictureList)}} \rangle$

Figure 3: Examples of three *alternative* fragments definitions over the collection C_{items}

It is worth noting 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.

3.2. Vertical Fragmentation

Vertical fragmentation is obtained by applying the projection operator (π) [18] to "split" a data structure into smaller parts that are frequently accessed in queries. Observe that, in XML repositories, the projection operator has a more sophisticated semantics than that used in other data models. 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 *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). An exception is made when we indicate the element order (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 4 shows examples of vertical fragments of the collections C_{items} and C_{store}, defined on Figure 2. Fragment $F2_{items}$ represents 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, the nodes that satisfy /Item/PictureList are exactly the ones pruned out the subtrees rooted in /Item in the fragment $F1_{items}$, thus preserving disjointness w.r.t. $F2_{items}$.

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

Figure 4: Examples of vertical fragments definitions over collections C_{items} and C_{store}

3.3. Hybrid Fragmentation

The hybrid fragmentation of an XML collection is trivially defined by applying 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.*

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

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

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

$F4_{items} := \langle C_{store}, \pi_{\text{/Store,\{/Store/Items\}}} \rangle$

Figure 5: Examples of hybrid fragments over collection C_{store}

Note that 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 5.

3.4. TLC and PartiX

In TLC [18], the algebraic operators require an *Annotated Pattern Tree* (APT) to determine which XML trees are going to be handled as input. Although we do not explicitly represent such an APT in our fragments definition, it can be inferred from both the collections specification and the path expressions and predicates used in the fragments. In our case, the schema tree of a collection corresponds to the basic structure of the APT. As an example, Figure 6 shows the APT for the collection C_{Items}.

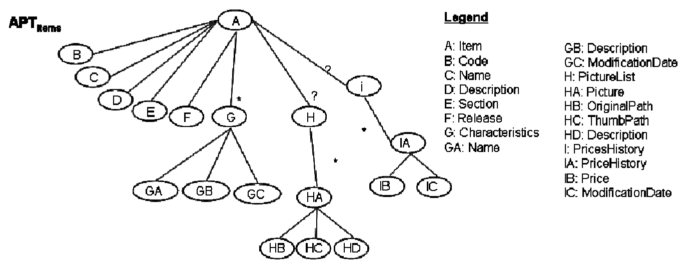

Figure 6: APT for collection C_{Items}

The fragmentation model of PartiX is based on two basic operations: selection and projection. The selection operator S[*apt*](*S*) in TLC takes as input an *apt* and a set of trees *S*, and produces as output the subset of input documents *S* which matches the pattern in *apt*. Besides the tree structure, *apt* can contain predicates in the tree nodes. In PartiX, definitions of horizontal fragments are mapped to the selection operator. As an example, to define a horizontal fragment F = $<C_{items}, \sigma_{\text{/Item/Section="CD"}}>$, we use the TLC selection operator S[apt_{Items2}](C_{items}), where apt_{Items2} is shown in Figure 7.

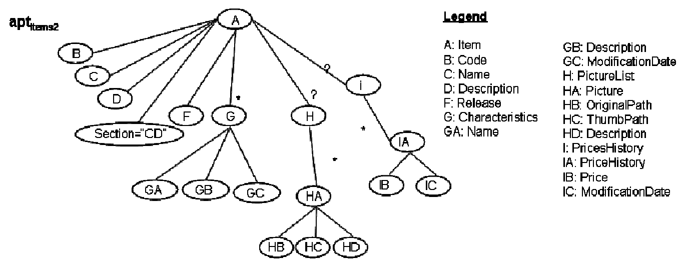

Figure 7: APT with selection criteria

The projection operator P[*nl*](*S*) of TLC takes as input a set of trees *S* and a list *nl* of Logical Class Labels to identify sets of nodes that should be kept in the result. A Logical Class is the set of nodes in the trees resulting from a TLC operator, and it corresponds to a given node in the APT. To each of such classes, it is assigned a unique label which is called Logical Class Label (LCL). As a simplification, usually LCLs are sequential numbers. For example, the LCL corresponding to some of the nodes of the APT of Figure 6 would be:

- Node A: LCL = 1
- Node B: LCL = 2
- Node E: LCL = 5
- Node HA: LCL = 12

In the above example, numbers were assigned following the order between nodes.

The TLC projection operator can be used in PartiX to specify vertical fragments. To do so, we use the list of path expressions that will be pruned, take its complement and use the corresponding LCLs as input for the projection operation. As an example, suppose we want to specify the vertical fragment F = <C_{items}, $\pi_{/\mathrm{Item}, \{/\mathrm{Item/PictureList}\}}$>. In this example, the element PictureList must be pruned out of the result. The corresponding TLC operation is then P[1,2,3,4,5,6,7,8,9,10,16,17,18,19](C_{items}).

3.5. 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 $\Phi := \{F_1, ..., F_n\}$. When fragmenting *C*, the following rules must be verified so that the fragmentation 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 all data item *d*, 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 $C := \nabla F_i$, $\forall F_j \in \Phi$, where ∇ depends on the type of fragmentation. For horizontal fragmentation, the union ($\cup$) operator[1] [14] is used, and for vertical fragmentation, the join ($\bowtie$) operator [18] is used. We keep an ID in each vertical fragment, so we can reconstructed them when needed

Procedures to verify correctness depend on the algorithms adopted in the fragmentation design. As an example, some fragmentation algorithms for the relational model guarantee the correctness of the resulting fragmentation design [26]. Still, others [17] require the use of additional techniques to check for correctness. Such verification is out of the scope of this paper.

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. PartiX works as a middleware between the user application and the DBMS server. 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 in Section 4.1).

[1] TLC is an extension of the TAX algebra [14]. The union operator is defined in TAX, and is thus "inherited" by TLC.

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 8. 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 types used by the distributed collections, while the *XML Distribution Catalog Service* stores the definitions of the fragmented collections. 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 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.

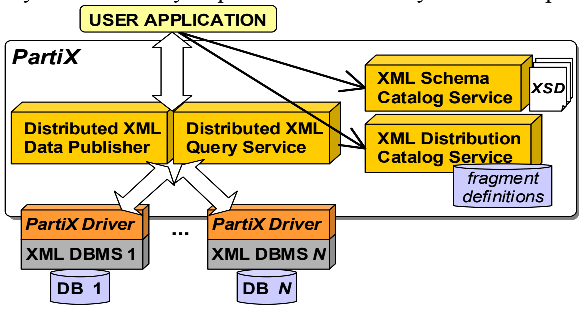
USER APPLICATION
PartiX
XML Schema Catalog Service
XSD
Distributed XML Data Publisher
Distributed XML Query Service
XML Distribution Catalog Service
fragment definitions
PartiX Driver
XML DBMS 1
...
PartiX Driver
XML DBMS N
DB 1
DB N

Figure 8: PartiX architecture

The proposed architecture is implemented in a prototype of the PartiX system. We have developed a PartiX *driver* to the eXist DBMS [11]. The *query service* is capable of coordinating the distributed execution of sub-queries annotated with the location of the data. The *Data Publisher* interprets loading scripts and stores the documents of a collection in the XML DBMSs.

In our evaluation, we have considered the three types of fragmentation in our experiments: horizontal, vertical and hybrid. Section 5 shows the experimental results. For horizontal fragmentation, we have written algorithms to rewrite XQuery in sub-queries and also to compose query results. We describe both of them in Sections 4.1 and 4.2. In our algorithms, we adopt a fragmentation design that follows the correctness rules presented in Section 3.5. We leave the rewrite algorithms for vertical and hybrid fragmentation as future work.

4.1. Query Decomposition

As mentioned before, in this section we focus on query decomposition for horizontal fragmentation only.

XQuery is very expressive and very complex. It is well known that rewriting XQuery queries is a complicated task. For horizontal fragmentation, however, we can make some assumptions that really simplify this task. By definition, the schema of horizontal fragments is preserved in fragmentation – they are exactly the same as the schema of the original collection. Because of this, it is guaranteed that all elements names referred in a query will also be in the fragments. The only exception is when fragments are built using existential tests, like the one in Figure 3(c). Yet, not even this case requires changing elements referenced in the query since it will only be sent to the right fragments. Therefore, we can concentrate only on locating the necessary fragments.

The main point of our algorithm is to find out what are the fragments involved in a query, and change the input parameter of the *collection* function appropriately.

In our experiments, we use the fragmentation design shown in Table 1. Notice that a trivial specification for these fragments would use OR connectors. However, we consider that horizontal fragments can only be defined using AND (i.e. as a conjunction), so we have applied De Morgan to obtain the specifications of Table 1. We will use this fragmentation schema in our examples.

Let *C* be a collection decomposed into a set of *horizontal* fragments $\Phi := \{F_1, \ldots, F_n\}$, $n \geq 1$. By definition, $F_i := \langle C, \sigma_{\mu i} \rangle$, has XML documents that satisfy $\sigma_{\mu i}$. The selection condition μi is a conjunction of simple predicates $PredF_i = \{p_1, \ldots, p_x\}$. When μi is negated, we negate each predicate in the set. As an example, fragment $F2_{4f}$ shown in Table 1 results in the set of predicates $PredF_{F24f}$ = {/Item/Section = "Perfumes", /Item/Section = "Electronics"}.

Table 1: Fragment definitions for databases Items_SHor and Items_LHor

#Frags	Fragments Definition
2	$F1_{2f} := \langle C_{items}, \sigma_{\text{not(/Item/Section≠”Toys” } \wedge \text{ /Item/Section≠”Games” } \wedge \text{ /Item/Section≠”Perfumes” } \wedge \text{ /Item/Section≠”Electronics”)}} \rangle$
	$F2_{2f} := \langle C_{items}, \sigma_{\text{not(/Item/Section≠”Books” } \wedge \text{ /Item/Section≠”CD” } \wedge \text{ /Item/Section≠”DVD” } \wedge \text{ /Item/Section≠”Clothes”)}} \rangle$
4	$F1_{4f} := \langle C_{items}, \sigma_{\text{not(/Item/Section≠”Toys” } \wedge \text{ /Item/Section≠”Games”)}} \rangle$
	$F2_{4f} := \langle C_{items}, \sigma_{\text{not(/Item/Section≠”Perfumes” } \wedge \text{ /Item/Section≠”Electronics”)}} \rangle$
	$F3_{4f} := \langle C_{items}, \sigma_{\text{not(/Item/Section≠”CD” } \wedge \text{ /Item/Section≠”DVD”)}} \rangle$
	$F4_{4f} := \langle C_{items}, \sigma_{\text{not(/Item/Section≠”Books” } \wedge \text{ /Item/Section≠”Clothes”)}} \rangle$
8	$F1_{8f} := \langle C_{items}, \sigma_{\text{(/Item/Section=”Toys”)}} \rangle$
	$F2_{8f} := \langle C_{items}, \sigma_{\text{(/Item/Section=”Games”)}} \rangle$
	$F3_{8f} := \langle C_{items}, \sigma_{\text{(/Item/Section≠”Perfumes”)}} \rangle$
	$F4_{8f} := \langle C_{items}, \sigma_{\text{(/Item/Section=”Electronics”)}} \rangle$
	$F5_{8f} := \langle C_{items}, \sigma_{\text{(/Item/Section=”CD”)}} \rangle$
	$F6_{8f} := \langle C_{items}, \sigma_{\text{(/Item/Section=”DVD”)}} \rangle$
	$F7_{8f} := \langle C_{items}, \sigma_{\text{(/Item/Section=”Books”)}} \rangle$
	$F8_{8f} := \langle C_{items}, \sigma_{\text{(/Item/Section=”Clothes”)}} \rangle$

Let Q be an XQuery query with a set of predicates $Preds = \{p_1, ..., p_k\}$, $k \geq 0$, and a set of path expressions $Paths = \{P_1, ..., P_m\}$, $m \geq 1$,.The expressions in *Paths* are from both the predicates and the return clause of Q.

To identify the fragments to which Q should be sent, we analyze the following items. This analysis builds a set *Frags* with all the fragments required to answer Q.

- ❖ For each p_j in *Preds*, if $p_j \in PredF_i$, then add F_i to *Frags*.

This analyses the predicates used in the query that are also used in the fragments definition.

- ❖ For each P_l in *Paths*, if P_l is prefixed by a simple path $\mathrm{p_j}$ used in some μi, and $\mathrm{p_j}$ has no value comparison and no function application, then Q must be sent to F_i. Place F_i in *Frags*.

This case concerns queries containing path expressions that have a prefix which participates in some fragment definition based on the existential test. More than one fragment may be chosen in this case.

Notice that the previous points are not exclusive. All of them may be used for a single query. If none of them apply, place all fragments in *Frags*. Once the set *Frags* is defined, we need to map the query Q into sub-queries according to the sites where each fragment in *Frags* is allocated. This mapping step is fairly simple. We just generate one sub-query for each fragment $F_i \in Frags$, changing the input parameter of the *collection* operator to the location of F_i. As an example, if we use the fragmentation schema shown in the second line of Table 1 (4 fragments), we find out that the query:

```
for $x in   collection(”/db/Samples/Horizontal/Hete250Mb/Sample1”)
where $x/Item/Description/contains(string(.),”silent”)
and $x/Item/Section = ”CD”
return $x/Item
```

is mapped to fragment $F3_{4f}$, and the sub-query will be stated as:

```
for $x in   collection(”/db3/F3_4f”)
where $x/Item/Description/contains(string(.),”silent”)
and $x/Item/Section = ”CD”
return $x/Item
```

4.2. Query Result Composition

Once all sites involved in the query processing execute their sub-queries, they send their results back to the *Distributed XML Query Service*. This component is responsible for composing partial into the final result, and sending it to the user application.

The final result consists of all sub-queries results returned by each site. When the query has aggregate functions, the results are first processed to calculate the final set of documents.

In both cases, since we are considering horizontal fragments, the final result represents the *union* of the sub-queries' results. This follows the reconstruction property presented in Section 3.5.

5. Experimental Evaluation

This section presents experimental results obtained with PartiX implementation for horizontal, vertical and hybrid fragmentation. We evaluate the benefits of data fragmentation for the performance of query processing in XML databases. The computer used in the tests was a 2.4Ghz Athlon XP with a 512Mb RAM memory.

In the sections that follow, we describe the experimental scenario we have used for each of the fragmentation types: horizontal, vertical and hybrid, and show that applying them in data collections have resulted in a reduction in the query processing times.

We applied the ToXgene [4] XML database generator to create the C_{store} and C_{items} collections, as defined in Figure 1 and Figure 2, and also a collection for the schema defined in the XBench benchmark [25]. Four databases were generated for the experiments: *database Items_SHor,* with document sizes of 2K in average, and elements PriceHistory and ImagesList with zero occurrences (Figure 1); *database Items_LHor*, with document sizes of 80Kb in average (Figure 1); *database XBench_Ver*, with the XBench collections, with document sizes varying from 5Mb to 15Mb each; and *database Store_Hyb* (Figure 1), 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 Items_LHor and Store_Hyb). 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 to be sent to different data fragments. In [**Erro! Fonte de referência não encontrada.**], we describe how this can be done. After each sub-query is executed, we compose the result to compute the final query answer [**Erro! Fonte de referência não encontrada.**].

The parallel execution of a query is simulated assuming that all fragments are placed in different sites and that the sub-queries are executed in parallel in each site. When the query predicates match the fragmentation predicates, the sub-queries are issued only to the corresponding fragment. 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. 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 executed 10 times, and the execution time was calculated by discarding the first execution time and calculating average of the remaining results. We have measured the execution times of each sub-query in the XML DBMS.

5.1. Horizontal Fragmentation

For horizontal fragmentation, the experiments were executed using the set of queries shown in Table 2. These queries illustrate diverse access patterns to XML collections, including the usage of predicates, text searches and aggregation operations.

Table 2: Queries evaluated in experiments with PartiX for horizontal fragmentation

Q1	for $x in collection(”/db/Samples/Horizontal/Hete250Mb/Sample1”)
	where $x/Item/Release = ”T”
	return $x/Item

Q2	for $x in collection("/db/Samples/Horizontal/Hete5Mb/Sample1")
	where $x/Item/Section = "Books"
	return $x/Item/Name

Q3	for $x in collection("/db/Samples/Horizontal/Hete5Mb/Sample1") where $x/Item/Section = "Games" return $x/Item/Name
Q4	for $x in collection("/db/Samples/Horizontal/Hete5Mb/Sample1") where $x/Item/Section = "Books" and $x/Item/Release = "T" return $x/Item/Name

Q5	for $x in collection(”/db/Samples/Horizontal/Hete250Mb/Sample1”)
	where $x/Item/Description/contains(string(.),”silent”)
	return $x/Item

Q6	for $x in collection(”/db/Samples/Horizontal/Hete250Mb/Sample1”)
	where $x/Item/Description/contains(string(.),”silent”)
	and $x/Item/Section = ”CD”
	return $x/Item

Q7	for $x in collection(”/db/Samples/Horizontal/Hete250Mb/Sample1”)
	where count($x/Item/Characteristics) >= 4
	return $x/Item

Q8	for $x in collection(”/db/Samples/Horizontal/Hete250Mb/Sample1”)
	where $x/Item/Section = ”DVD”
	and count($x/Item/Characteristics) >= 4
	return $x/Item

The XML database was fragmented as follows. The C_{items} collection was horizontally fragmented by the "Section" element. We varied the number of fragments (2, 4 and 8) with a non-uniform document distribution between the fragments. The definition of the fragments we have used is shown in Table 1.

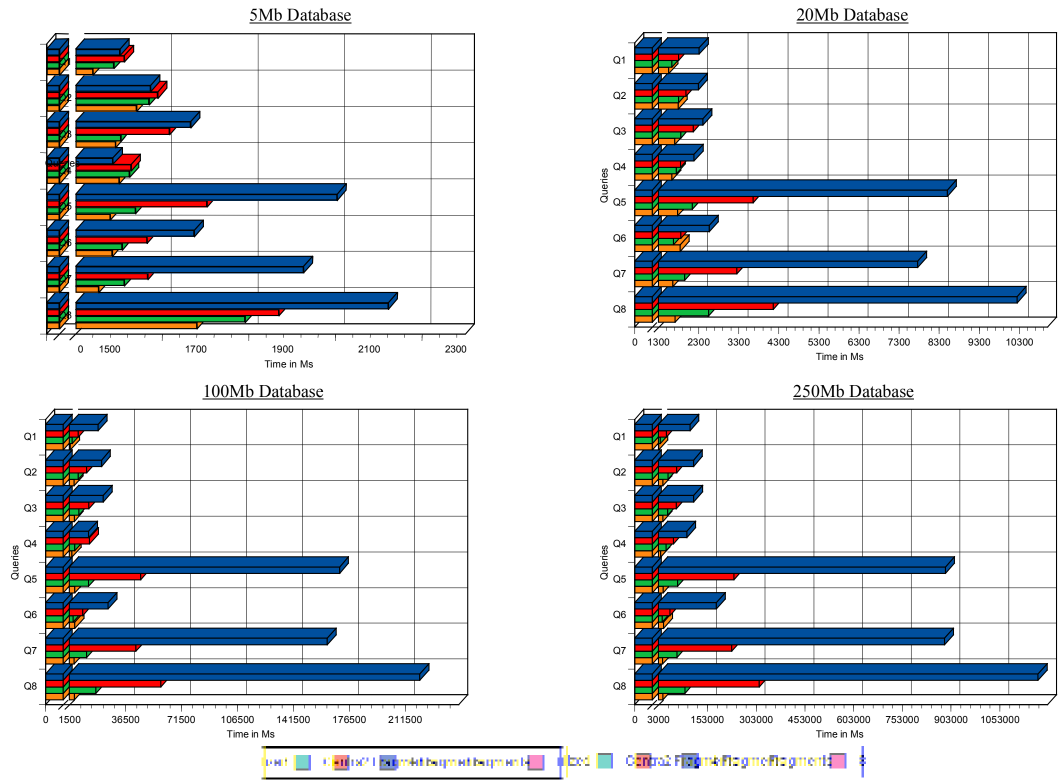

Figure 9: Experimental results for database Items_SHor – horizontal fragmentation

Figure 9 shows the performance results obtained from the PartiX execution on top of database Items_SHor, and Figure 10 on database Items_LHor, 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 Items_SHor and Items_LHor 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, query Q8 is executed in 1200s in the centralized database Items_SHor and in 31s in the centralized database Items_LHor. When using horizontal fragmentation with 2 fragments, these times are reduced to 300s and 14s, respectively.

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, thus not requiring additional time for reconstructing the global query result.

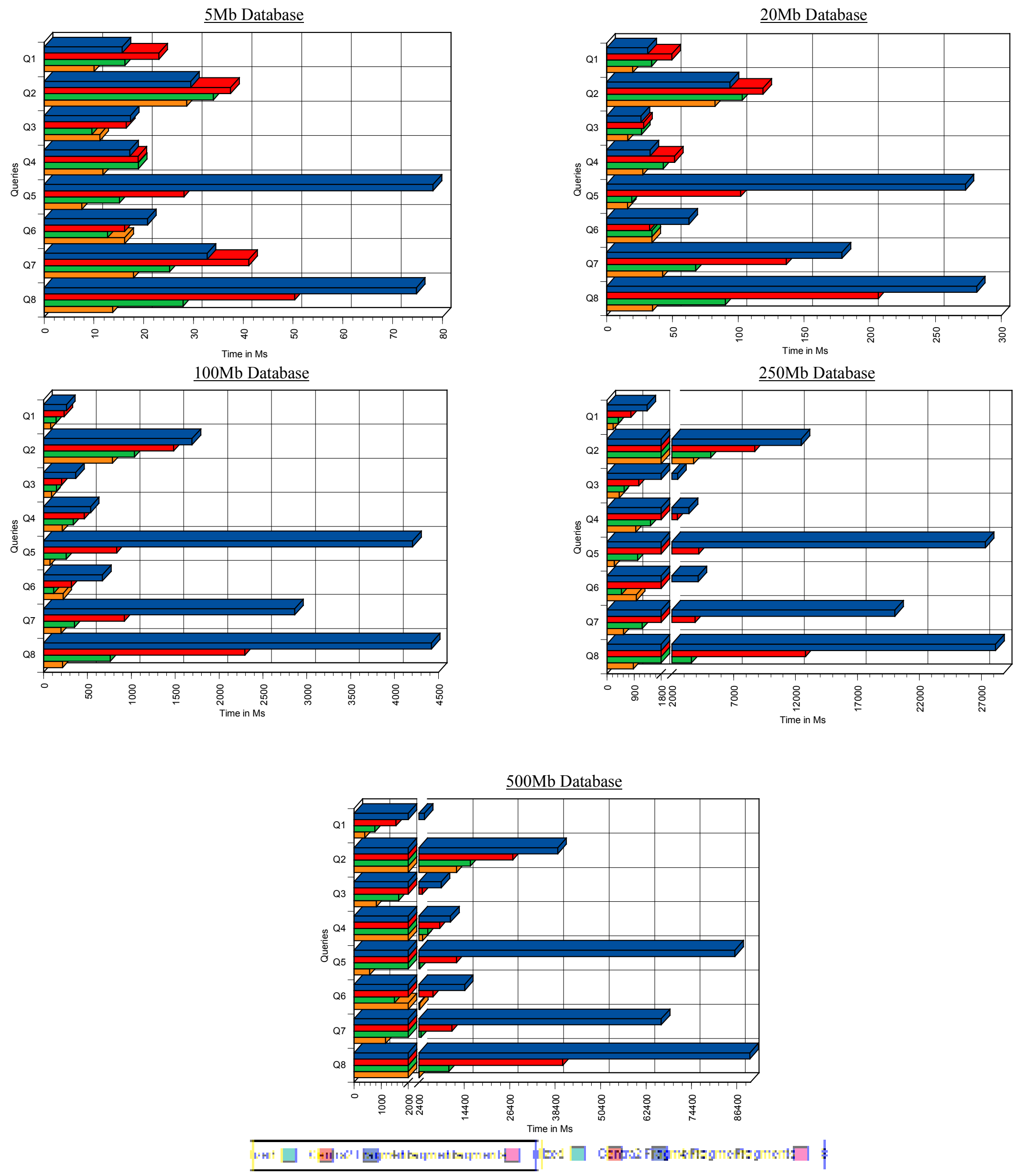

Figure 10: Experimental results for database Items_LHor – horizontal fragmentation

Another interesting result may be observed in the execution of Q6. As the number of fragments increases, the execution time of Q6 increases in some scenarios. This happens because eXist generates different execution plans for each sub-query, which favored the query performance in scenarios with few fragments. However, all the distributed configurations presented better performance when compared to the centralized database.

As expected, in small databases (i.e., 5Mb) the performance gain obtained is not enough to justify the use of fragmentation. Moreover, we concluded that the document size is very important for defining the fragmentation schema. Database Items_LHor (Figure 10) presents better results with few fragments, while database Items_SHor presents better results with many fragments.

5.2. Vertical Fragmentation

For the experiments with vertical fragmentation, we have used database *XBench_Ver* and some of the queries specified in XBench [25], which are shown in Table 3. We have named them Q1 to Q10, although these names do not correspond to the names used in the XBench document.

Table 3: Queries evaluated in experiments with PartiX for vertical fragmentation

Q1	for $art in collection("/db/Samples/Horizontal/TCMD5Mb/Sample1")/article[@id="200"]
	return $art/prolog/title

Q2	for $prolog in collection("/db/Samples/Horizontal/TCMD5Mb/Sample1")/article/prolog
	where $prolog/authors/author/name="Ben Yang"
	return $prolog/title

Q3	for $a in collection("/db/Samples/Horizontal/TCMD5Mb/Sample1")/article[@id="350"]/ body/section[@heading="introduction"], $p in collection("/db/Samples/Horizontal/TCMD5Mb/Sample1")/article[@id="350"] /body/section[. >> $a][1] return <HeadingOfSection>{$p/@heading}</HeadingOfSection>

Q4	for $a in collection("/db/Samples/Horizontal/TCMD5Mb/Sample1")/article
	where some $b in
	$a/body/abstract/p satisfies (contains($b, "the") and contains($b, "hockey"))
	return $a/prolog/title

Q5	for $art in collection("/db/Samples/Horizontal/TCMD5Mb/Sample1")/article[@id="100"]
	return $art/prolog/*/author/name

Q6	for $a in collection("/db/Samples/Horizontal/TCMD5Mb/Sample1")/article/prolog
	where $a/dateline/country="Canada"
	order by $a/dateline/date
	return <Output>{$a/title}{$a/dateline/date}</Output>

Q7	for $a in collection("/db/Samples/Horizontal/TCMD5Mb/Sample1")/article[@id="6"]
	return $a

Q8	for $a in collection("/db/Samples/Horizontal/TCMD5Mb/Sample1")/article[@id="120"]
	return
	<Output>
	  {$a/prolog/title}
	  {$a/prolog/authors/author[1]/name}
	  {$a/prolog/dateline/date}
	  {$a/body/abstract}
	</Output>

Q9	for $a in collection("/db/Samples/Horizontal/TCMD5Mb/Sample1")/article
	where contains ($a//p, "hockey")
	return $a/prolog/title

Q10	for $a in
	collection("/db/Samples/Horizontal/TCMD5Mb/Sample1")/article/prolog/authors/author
	where empty($a/contact/text())
	return <NoContact>{$a/name}</NoContact>

Database *XBench_Ver* was vertically fragmented in three fragments, described in Table 4.

Table 4: Fragments definitions for database XBench_Ver

#Frags	Fragments Definition
3	$F1_{papers} := \langle C_{papers}, \pi_{/article/prolog} \rangle$ $F2_{papers} := \langle C_{atigos}, \pi_{/article/body} \rangle$ $F3_{papers} := \langle C_{artigos}, \pi_{/article/epiog} \rangle$

Figure 11 shows the performance results of PartiX in this scenario. In the 5Mb database, we had gains in all queries, except for Q4 and Q10. With vertical fragmentation, the main benefits occur for those 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 significant performance gains in any case, except in the 100Mb database. In this case, however, the performance gain was not significant. 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 100Mb database, queries Q6 and Q9 perform equivalently to the centralized approach. In the 250Mb database, queries Q6, Q9 and Q3 also 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, which is much more expensive than a union.

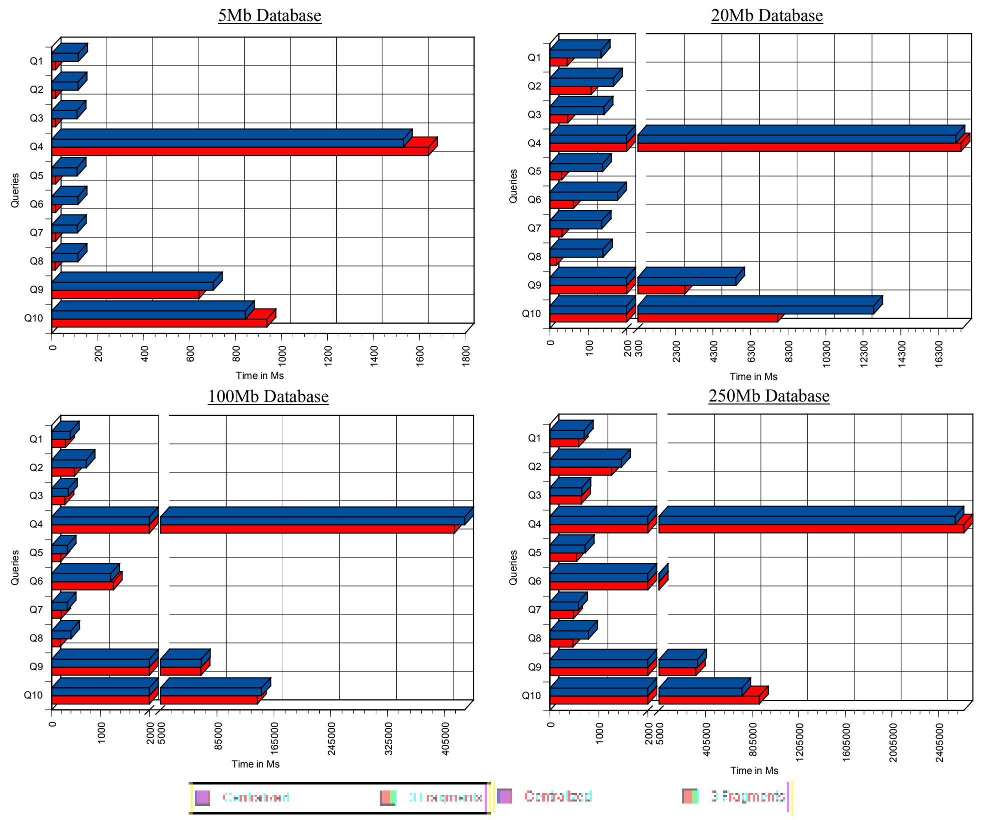

Figure 11: Experimental results for database XBench_Ver – vertical fragmentation

5.3. Hybrid Fragmentation

In the experiments with hybrid fragmentation, we have used the *database Store_Hyb*, and the set of queries shown in Table 5. The database was fragmented according to Table 6.

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 Items_SHor e Items_LHor, 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 form 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 *Fragmentation Mode 1* (Fragment Mode 1 in Figure 12 and Figure 13), 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 a selection operator. We have called this approach *Fragmentation Mode 2* (Fragment Mode 2 in the figures). As we will see, this fragmentation mode will beat the centralized approach in most of the cases.

When we consider the transmission times (Figure 12), Fragmentation Mode 1 performs worst 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. Query Q11 uses an aggregation function that presented a good performance in the 100Mb database and in bigger ones. In the remaining databases, it presented poor performance (5Mb database) or an anomalous behavior (20Mb database).

Table 5: Queries evaluated in experiments with PartiX for hybrid fragmentation

Q1	for $x in collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1")/Store/Items/Item where $x/Release = "T" return $x
Q2	for $x in collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1")/Store/Items/Item where $x/Section = "CD" return $x/Name
Q3	for $x in collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1")/Store/Items/Item where $x/Section = "Supplies" return $x/Name
Q4	for $x in collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1")/Store/Items/Item where $x/Section = "CD" and $x/Release = "T" return $x/Name
Q5	for $x in collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1")/Store/Items/Item where contains(string($x/Description),"execute") return $x
Q6	for $x in collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1")/Store/Items/Item where contains(string($x/Description),"execute") and $x/Section = "CD" return $x
Q7	for $x in collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1")/Store/Items/Item where count($x/Characteristics) >= 4 return $x
Q8	for $x in collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1")/Store/Items/Item where $x/Section = "CD" and count($x/Characteristics) >= 4 return $x
Q9	for $x in collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1") /Store/Employees/Employee return $x

Q10	for $x in collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1")
	/Store/Employees
	return <TotalPagamento>{sum($x/Employee/Salario)}</TotalPagamento>

Q11	Let $f := collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1") /Store/Employees, $s1:=collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1") /Store/Items/Item[Section="CD"], $s2:=collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1") /Store/Items/Item[Section="DVD"], $s3:=collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1") /Store/Items/Item[Section="Supplies"], $s4:=collection("/db/Samples/FragHibrida/StoreItem5Mb/Sample1") /Store/Items/Item[Section="Books"] return <Totais> <TotalEmployee>{count($f/Employee)}</TotalEmployee> <TotalCD>{count($s1)}</TotalCD> <TotalDVD>{count($s2)}</TotalDVD> <TotalSupplies>{count($s3)}</TotalSupplies> <TotalBooks>{count($s4)}</TotalBooks> </Totais>

Table 6: Fragments definition for experiments with hybrid fragmentation

#Frags	Fragments Definition
5	$F1 := \langle C_{Store} ,\ \pi_{/Store,\{/Store/Items\}} \rangle$ $F2 := \langle C_{Store},\ \pi_{/Store/Items} \bullet \sigma_{/Store/Items/Item/Section="CD"} \rangle$ $F3 := \langle C_{Store},\ \pi_{/Store/Items} \bullet \sigma_{/Store/Items/Item/Section="DVD"} \rangle$ $F4 := \langle C_{Store},\ \pi_{/Store/Items} \bullet \sigma_{/Store/Items/Item/Section="Supplies"} \rangle$ $F5 := \langle C_{Store},\ \pi_{/Store/Items} \bullet \sigma_{/Store/Items/Item/Section="Books"} \rangle$

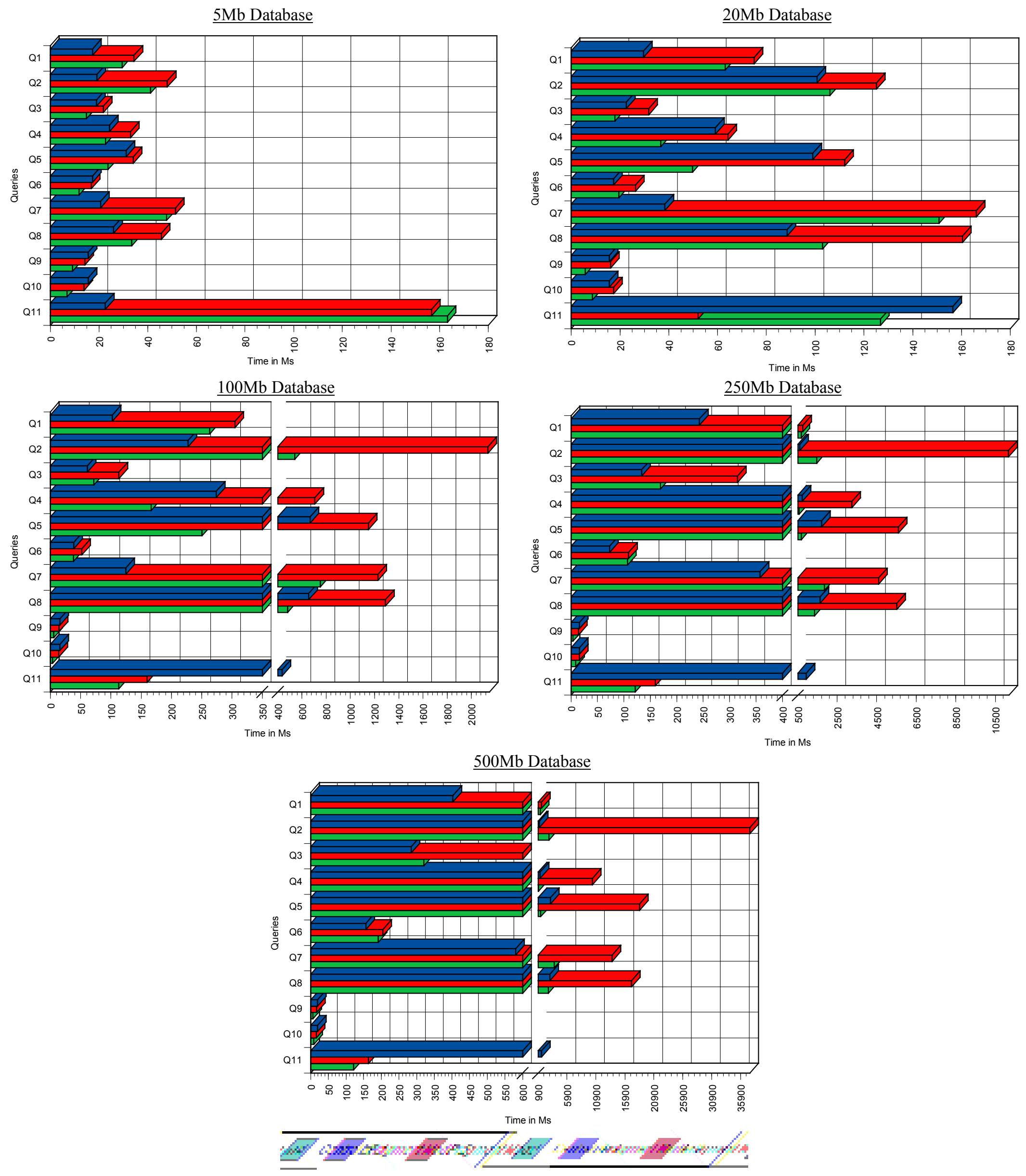

Figure 12: Experimental results for database Store_Hyb – hybrid fragmentation

We notice that the Fragmentation Mode 2 performs better, although it does not win in all cases. In the 5Mb database, it wins in queries Q3, Q4, Q5 and Q6. These queries benefit from the parallelism of the fragments and from the use of a specific fragment. As in the Fragmentation Mode 1, queries Q9 and Q10 always performs better than the centralized case. Query Q11 only looses in the 5mb database.

As the database size increases, the query results also increase. This makes the total query processing time also to increase. 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 worst 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, Fragmentation Mode 2 wins in all databases, in all queries, except for query Q11 in the 5Mb database. However, Fragmentation Mode 1 has shown to be effective in some cases.

Figure 13 shows the experimental results without the transmission times. It shows that hybrid fragmentation reduces the query processing times significantly.

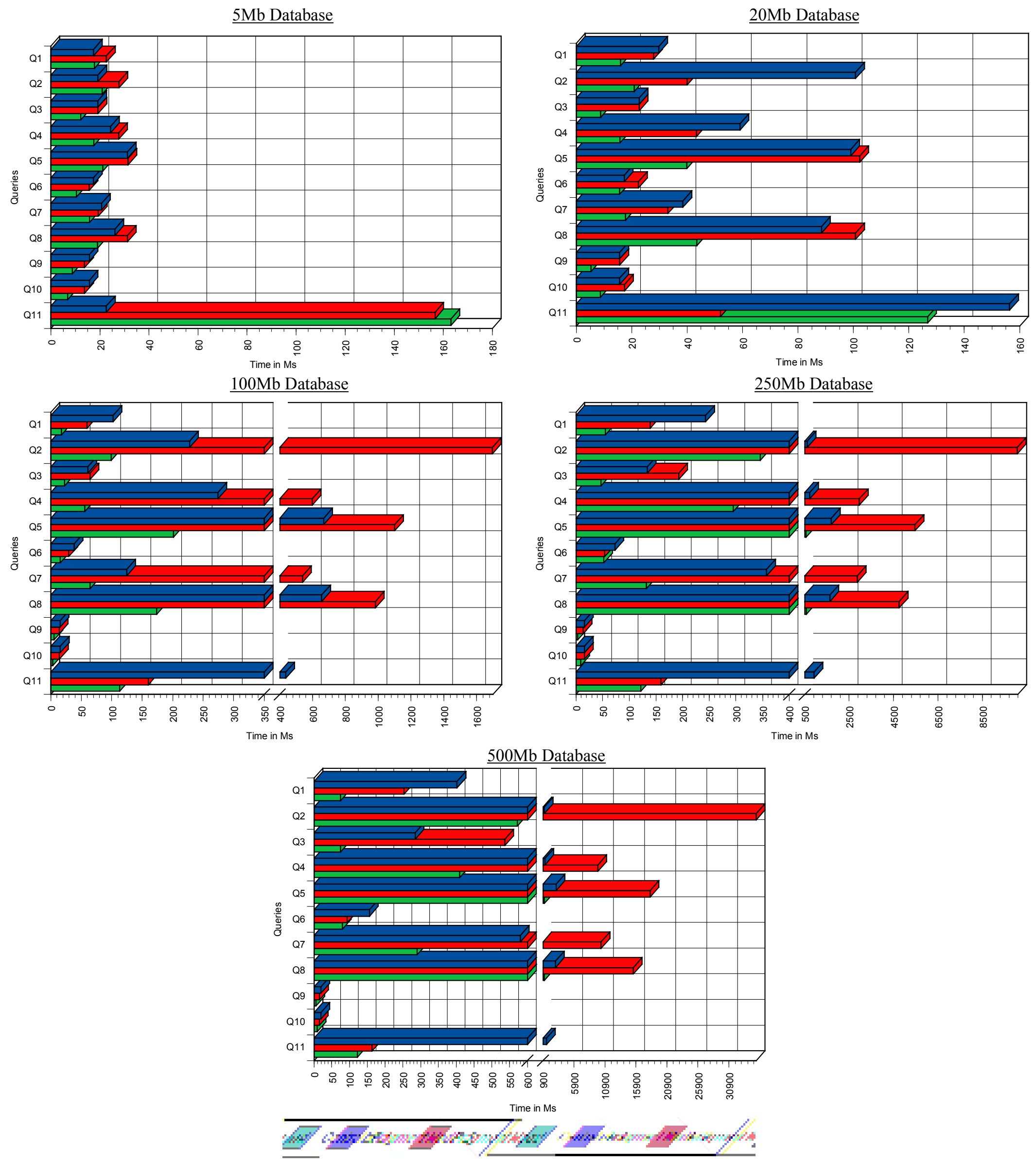

Figure 13: Experimental results for database Store_Hyb – hybrid fragmentation, without transmission times

6. Related Work

Foundations of distributed database design for XML were initially addressed in [9] and [16]. Ma and Schewe [16] extended ideas coming from the fragmentation of object databases to deal with XML data. They propose three types of XML fragmentation: *horizontal*, which groups elements of an 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 concepts are not clearly identified. For example, horizontal fragmentation involves data restructuring and elements projection, thus yielding fragments with different schema definitions. Also, vertical fragmentation requires the specification of artificial elements to properly connect document fragments. Even though the user can define fragments from several XML documents, their approach is not really suitable for MD repositories. In this case, to define horizontal fragments, the user must first integrate all XML documents into an SD view. Therefore, we found it difficult to follow those guidelines since they are tied to a specific representation of the database. In addition, they do not present correctness rules for their fragmentation definitions.

Our definition of vertical XML fragmentation is inspired in the work of Bremer and Gertz [9]. 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 aim to maximize local query evaluation by replicating global information, and distributing some index structures. They present important performance improvements, but their empirical evaluation focuses on the benefits of such indexes.

Recently, P2P systems have been explored for distributed data management in dynamic environments. Most P2P systems are based on special index structures, such as *Distributed Hash Tables* (DHT). Bonifati *et al*. [6] propose an interesting approach to query XML data over a P2P network. Vertical fragments of an XML document can be identified by XPath expressions and distributed in a DHT, such that each peer keeps some data fragments and their related path expressions (that is, the list of path expressions of their child fragments and their super fragments). This way, queries can be efficiently routed in the P2P system. Although adequate for P2P requirements, the solution proposed in [6] does not apply to our scenario since we can rely on a global catalog and we are interested in considering the different alternatives to fragment SD and MD repositories.

Another P2P approach is presented in [1]. It considers the distribution of Active XML (AXML) documents, which are dynamic documents with special elements denoting calls to Web Services. To speedup the access to the contents of an AXML document, they propose a mechanism to locally replicate remote data fragments that are returned by some service calls. Fragments of an AXML document may be distributed such that the original document keeps references to its external fragments (and vice-versa), similarly to the XInclude language [24]. However, they do not distinguish different fragmentation techniques.

In XFrag, Bose and Fegaras [7] foccus on processing fragmented data streams. The fragmentation model is proposed in [8]. In this context, the goal is to fragment data and send it through the network. User applications can query this data as it passes through the network. The goal of this work is completely different of our work. XFrag focus on querying streams of data, while we focus on queering stored data. The query processing techniques completely diverge. If we were to use the fragmentation schema proposed in [8] in our application scenario, the query processing would be inefficient, since it would be necessary to completely reconstruct the document before processing it. This is because [8] do not distinguish between horizontal, vertical and hybrid fragmentation. Also, it is not possible to describe fragments using predicates, which makes it impossible to define horizontal or hybrid fragments.

For scenarios based on Web services, Amer-Yahia and Kotidis [2] explore the publication of XML fragments to reduce communication costs. However, the authors define only vertical fragmentation for SD repositories, and their evaluation scenario does not consider query processing. None of these works clearly define the main possible fragmentation alternatives (i.e., horizontal, vertical, and hybrid) on XML. This significantly complicates checking for correctness of XML fragmentation, and reconstructing query results from fragmented repositories. Also, these works address only SD repositories. 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 we have performed several tests to evaluate the performance of the different fragmentation alternatives. We did not find in the related work any experimental analysis of the response time of query processing on fragmented XML repositories.

To clearly identify the differences of PartiX and these related work, in the next sections we use the theories in [8] and [2] to construct the fragment definitions of Figure 3(a), Figure 4(a) and Figure 5.

6.1. XFrag

XFrag [7,8] uses the concept of fillers and holes to fragment XML documents that will be sent over the network as data streams. The original document is divided in several smaller documents called fillers. Each fragment may contain one or more holes, where other fragments (the fillers) may fit. Such holes are marked with special tags *stream:hole*.which reference the ID of its corresponding filler. The information about the structure of the original document, which is called *Tag Structure*. Roughly speaking, it describes the DTD of the XML document that is to be fragmented and assigns an *id* to each element type. This *id* could be used to replace the element name in the fragments, to achieve data compression.

To represent the fragments of Figure 3(a), Figure 4(a) and Figure 5 using this fragmentation model, it is first necessary to describe the Tag Structure of C_{store}.

```
<stream:structure>
  <tag name="Store" id ="1">
    <tag name="Sections" id="2">
      <tag name="Section" id="3">
        <tag name="Code" id="4"/>
        <tag name="Name" id="5"/>
        </tag>
      </tag>
      <tag name="Items" id="6">
        <tag name="Item" id ="7">
          <tag name="Code" id="2"/>
          <tag name="Name" id="3"/>
          <tag name="Description" id="4"/>
          <tag name="Section" id="5"/>
          <tag name="Release" id="6"/>
          <tag name="Characteristics" id="7">
            <tag name="Name" id="8"/>
            <tag name="Description" id="9"/>
            <tag name="ModificationDate" id="10"/>
          </tag>
          <tag name="PictureList" id="11">
            <tag name="Picture" id="12">
              <tag name="OriginalPath" id="13"/>
              <tag name="ThumbPath" id="14"/>
              <tag name="Description" id="15"/>
            </tag>
          </tag>
          <tag name="PricesHistory" id="16">
            <tag name="PriceHistory" id="17">
              <tag name="Price" id="18"/>
              <tag name="ModificationDate" id="19"/>
            </tag>
          </tag>
        </tag>
      </tag>
    </tag>
</stream:strutcture>
```

Horizontal Fragmentation. The fragments we need to define are: $F1_{CD} := \langle C_{items}, \sigma_{\text{/Item/Section="CD"}}\rangle$ and $F2_{CD} := \langle C_{items}, \sigma_{\text{/Item/Section}\neq\text{"CD"}}\rangle$ (Figure 3(a)). In the fragmentation model of [8], there is no way to define fragments using predicates, as this example requires. Thus, it is sometimes impossible to define certain types of fragments. This is clear when we try to define horizontal fragments over a MD collection, as in the example of Figure 3(a). This is because in the XFrag model, there must be one fragment per document, and there is no way to separate then into two different classes (CDs and not CDs) in the fragment that references each of those fragments. This will be clearer in the example above.

To simplify the example, we will assume the store has only one section, which is devoted to Audio and Video related Items. Also, we will assume there is only 4 items, two that are CDs, and two that are not.

Fragment 1:

```
<Store>
  <Sections>
    <Section>
      <Code>01</Code>
      <Name>Audio and Video<Name>
    </Section>
  </Sections>
  <Items>
    <stream:hole id="10" tsid="7"/>
    <stream:hole id="20" tsid="7"/>
    <stream:hole id="30" tsid="7"/>
    <stream:hole id="40" tsid="7"/>
  </Items>
</Store>
```

This first fragment describes the structure of the Store, and references the remaining fragments.
Now, we have one more fragment for each item. We will show just a piece of two of them:

Fragment 2:

```
<stream:filler id="10" tsid="7">
  <Item>
    <Code>01098</Code>
    <Name>How to Dismantle an Atomic Bomb<Name>
    <Description>U2, Audio CD</Description>
    <Section>CD</Section>
    ...
  </Item>
</stream:filler>
```

Fragment 3:

```
<stream:filler id="20" tsid="7">
  <Item>
    <Code>01076</Code>
    <Name>U2 - Vertigo 2005 - Live From Chicago <Name>
    <Description>U2 DVD, Deluxe Edition</Description>
    <Section>DVD</Section>
    ...
  </Item>
</stream:filler>
```

It is clear, by this example, that it is not possible to separate items that are CDs from items that are not CDs. Thus, it is not possible to define horizontal fragments with the fragmentation model of [8].

Vertical Fragmentation. For vertical fragmentation, we need to define fragments $F1_{items} := \langle C_{items}, \pi_{\text{/Item}, \{\text{/Item/PictureList}\}}\rangle$ and $F2_{items} := \langle C_{items}, \pi_{\text{/Item/PictureList}, \{\}}\rangle$. These are easily defined using the model of [8].

Fragment 1:

```
<Store>
  <Sections>
    <Section>
      <Code>01</Code>
      <Name>Audio and Video<Name>
    </Section>
  </Sections>
  <Items>
    <Item>
      <Code>01098</Code>
      <Name>How to Dismantle an Atomic Bomb<Name>
      <Description>U2, Audio CD</Description>
      <Section>CD</Section>
      <Release>T</Release>
      <Characteristics>...</Characteristics>
      <stream:hole id="11" tsid="11"/>
      <PriceHistory>...</PriceHistory>
    </Item>
    <Item>
      <Code>01076</Code>
      <Name>U2 - Vertigo 2005 - Live From Chicago <Name>
      <Description>U2 DVD, Deluxe Edition</Description>
      <Section>DVD</Section>
      <Release>T</Release>
      <Characteristics>...</Characteristics>
      <stream:hole id="12" tsid="11"/>
      <PriceHistory>...</PriceHistory>
    </Item>
    <Item>
     ...
     </Item>
    <Item>
     ...
     </Item>
  </Items>
</Store>
```

Now, there are four more fragments (since we are assuming there are 4 items) with the PictureList:

Fragment 2:

```
<stream:filler id="11" tsid="11">
```

```
  <PictureList>
    ...
  </PictureList>
</stream:filler>
```

Fragment 3:

```
<stream:filler id="12" tsid="11">
  <PictureList>
    ...
  </PictureList>
</stream:filler>
```

Fragment 4:

```
<stream:filler id="13" tsid="11">
  <PictureList>
    ...
  </PictureList>
</stream:filler>
```

Fragment 5:

```
<stream:filler id="14" tsid="11">
  <PictureList>
    ...
  </PictureList>
</stream:filler>
```

Hybrid Fragmentation. Since it is not possible to express horizontal fragments in the approach, it is also not possible to express hybrid fragments.

By looking at these examples, it is easy to see that in XFrag, there is no distinction between horizontal, vertical and hybrid fragmentation.

6.2. Efficient XML Data Exchange through Fragments Publication

The work presented in [2] is motivated by a typical scenario of data exchange in the Web, where XML views are used as pair-wise agreements for collaboration, and data producers and consumers often perform many operations to build such views from their databases. The goal is to avoid unnecessary data manipulation and transfers by allowing the publication of database fragments and operations that connect them to generate an agreed XML schema. Thereby, users can define mappings between the fragments of producers and consumers, and then only required fragments are accessed when data is requested. The authors also present some criteria to check the validity of a fragmentation schema. Two basic operations are used to define the links among fragments of a partitioned schema: *combine*, to join two fragments; and *split*, to project a fragment. Their evaluation scenario is focused on data transfers between Web services.

Horizontal Fragmentation. It is not possible to represent the horizontal fragments of Figure with the fragmentation model of Figure 3(a), since a fragment is defined as any subtree of the schema, and selections are not allowed. This is enforced by their validity criterion for schema fragmentation, where each element must be used by only one fragment.

Vertical Fragmentation. The proposed fragmentation model does not support pruning subtrees of vertical fragments. In fact, there is not a tight control of the correspondence between the XML schema and the fragment definitions. A possible representation of the fragments $F1_{items} := \langle C_{items}, \pi_{\text{/Item, \{/Item/PictureList\}}}\rangle$ and $F2_{items} := \langle C_{items}, \pi_{\text{/Item/PictureList, \{\}}}\rangle$ in Figure 4(a) would be:

```
<fragmentation name="ITENS-f">
    <fragment name="F1-itens">
        <element name="Item">
            <attribute name="ID" type="string"/>
            <attribute name="PARENT" type="string"/>
        </element>
    </fragment>
    <fragment name="F2-itens">
        <element name="PictureList">
```

```
                    <attribute name="ID" type="string"/>
                    <attribute name="PARENT" type="string"/>
            </element>
        </fragment>
    </fragmentation>
```

Additionally, the connection between these fragments would be expressed by an operation Combine(F1-itens,F2-itens). However, it is worth mentioning that the *combine* operator can link only child elements of a fragment (one cannot specify a path to connect descendants in the subtrees).

Hybrid Fragmentation. Since horizontal fragments are not supported, this model does not cover hybrid fragments.

7. 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 of fragments in XML and define correctness criteria for the different types of fragmentation. By specifying the concept of collections, 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 the experimental evaluation.

Our experiments highlight the potential to significant gains of performance through fragmentation. The reduction in the time of query execution is obtained by intra-query parallelism, and also by the directed execution, avoiding scanning unnecessary fragments. The queries executed by PartiX with eXist over databases generated by ToXgene present an estimated execution time up to 72 times smaller when compared to centralized executions. The PartiX architecture is generic, and can be plugged to any XML DBMS that process XQuery queries. This architecture follows the line of *database clusters* that have been presenting excellent performances over relational DBMSs [15].

As future work, we intend to plan 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 [26], and to implement tools to automate this fragmentation process. We also want to address in more depth the definition of algorithms to automatically rewrite queries to run over the fragmented database.

References

1. S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. SIGMOD Conference 2003, pp. 527–538.
2. S. Amer-Yahia, and Y. Kotidis. A Web-Services Architecture for Efficient XML Data Exchange. ICDE 2004, pp. 523-534.
3. F. Baião, M. Mattoso, and G. Zaverucha. A Distribution Design Methodology for Object DBMS. Distributed and Parallel Databases, 16(1):45–90, 2004.
4. D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a template-based data generator for XML. WebDB 2002, pp. 621–632.
5. E. Bertino, B. Catania, and W. Q. Wang. XJoin Index: Indexing XML Data for Efficient Handling of Branching Path Expressions. ICDE 2004, pp. 828.
6. A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath lookup queries in P2P networks. WIDM 2004, pp. 48–55.
7. S. Bose, L. Fegaras. XFrag: A Query Processing Framework for Fragmented XML Data. WebDB 2005, pp. 97–102.
8. S. Bose, L. Fegaras, D. Levine, and V. Chaluvadi. A Query Algebra for Fragmented XML Stream Data. DBPL 2003, pp. 195–215.
9. J.-M. Bremer and M. Gertz. On Distributing XML Repositories. WebDB 2003, pp. 73–78.
10. Y. Chen, S. B. Davidson, and Y. Zheng. BLAS: An Efficient XPath Processing System. SIGMOD Conference 2004, pp. 47–58.
11. eXist: Open Source Native XML Database. http://exist.sourceforge.net.
12. M.F. Fernández, J. Siméon, and P. Wadler. An Algebra for XML Query. FSTTCS 2000, pp. 11-45.
13. T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Natix: A Technology Overview. Web, Web-Services, and DB Systems 2002, pp. 12–33.
14. H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. DBPL 2001, pp 149-164.
15. A. Lima, M. Mattoso, and P. Valduriez. Adaptive Virtual Partitioning for OLAP Query Processing in a Database Cluster. SBBD 2004, pp. 92–105.
16. H. Ma and K.-D. Schewe. Fragmentation of XML Documents. SBBD 2003, pp. 200–214.

17. S. Navathe, K. Karlapalem, and M. Ra, "A mixed fragmentation methodology for initial distributed database design," Journal of Computer and Software Engineering, vol. 3, no. 4, 1995.
18. S.Paparizos, Y.Wu, L.V.S.Lakshmanan, H.V.Jagadish. Tree Logical Classes for Efficient Evaluation of XQuery. SIGMOD 2004, pp. 71–82. .
19. J. Rittinger. Pathfinder/MonetDB: A High Performance Relational Runtime for XQuery. BTW 2005, pp. 5–7.
20. F. Rizzolo and A. Mendelzon. Indexing XML Data with ToXin. WebDB 2001, pp. 49–54.
21. N. Ruberg, G. Ruberg, and I. Manolescu. Towards cost-based optimization for data-intensive web service computations. SBBD 2004, pp. 283–297.
22. H. Schöning. Tamino - A DBMS designed for XML. ICDE 2001, pp. 149–154.
23. World Wide Web Consortium (W3C). http://www.w3.org.
24. XML Inclusions (XInclude) 1.0. http://www.w3.org/TR/2004/PR-xinclude-20040930.
25. B. Yao, M. T. ¨Ozsu, and N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. ICDE 2004, pp. 621–632.
26. Özsu and P. Valduriez. Principles of Distributed Database Systems. Prentice Hall, 1999.
27. M. Zhang and J.T. Yao. XML Algebras for Data Mining. Data Mining and Knowledge Discovery: Theory, Tools and Technology, 2004, pp. 209-217.
28. X. Zhang, B. Pielech, and E.A. Rundesnteiner. Honey, I shrunk the XQuery!: an XML algebra optimization approach. WIDM 2002, pp. 15-22.