A Survey on XML Fragmentation

Vanessa Braganholo
Fluminense Federal University, UFF
Brazil
vanessa@ic.uff.br

Marta Mattoso
Federal University of Rio de Janeiro, COPPE/UFRJ
Brazil
marta@cos.ufrj.br

ABSTRACT

Efficient document processing is a must when large volumes of XML data are involved. In such critical scenarios, a well-known solution to this problem is to distribute (map) the data among several processing nodes, and then distribute the processing accordingly, taking advantage of parallelism. This is the approach taken by distributed databases and MapReduce environments. Fragmentation techniques play an important role in these scenarios. They provide a way to "cut" the database into pieces and distribute the pieces over a network. This way, queries can also be "cut" into sub-queries that run in parallel, thus achieving better performance when compared to the centralized environment. However, there is no consensus in the database community as to what an XML fragment is. In fact, several approaches in literature present definitions of XML fragments. In addition to query processing, using XML fragmentation techniques may also be helpful when managing XML documents distributed along the web or clouds. This paper surveys the existing XML fragmentation approaches in literature, comparing their features and highlighting their drawbacks. Our contribution resides in establishing a map of the area.

1. INTRODUCTION

Efficient document processing is a must when large volumes of XML data are involved [35, 56]. To achieve good performance in query processing, lots of initiatives focus on indexing [13, 14, 18, 20, 21, 34, 57] and query optimization [15, 16, 29, 55, 74]. In addition to these initiatives, distributed query processing can further improve the performance when large collections of documents are involved. In critical analytical scenarios, complex queries, such as OLAP (On-Line Analytical Processing), cannot effectively benefit from indexing techniques since queries involve several attributes and are *ad-hoc*.

MapReduce [22, 23] and Hadoop[1] have been extensively used to execute operations in parallel based on *ad-hoc* sub-sets of data. Originally defined for searching web log datasets, MapReduce is being progressively used for other types of datasets. Horizontal data fragmentation, allocation and indexing techniques [26] have been proposed to improve Hadoop's performance and experiments have been performed over a variety of data, including TPC-H OLAP queries [71]. Other types of fragmentation have also been used to process large XML datasets using Hadoop [17]. In fact, MapReduce has been compared [24, 25, 61, 67] and combined [3, 71] with Parallel Databases approaches, always aiming at achieving better response times for query processing. Such strategies can be used to process large XML datasets if XML documents are correctly fragmented and reconstructed.

The term XML *fragment* appeared in the beginning of the last decade denoting a well-formed piece of an XML document [69, 60] (in fact, the first draft related to the W3C candidate recommendation appeared in June of 1999). The main goal at that time was to ease the communication between applications, so they could exchange pieces of documents instead of entire ones. After that, the term XML fragment was used (with several different definitions) in the context of distributed databases [2, 12, 47, 64], and distributed query processing issues began to raise researchers' interest. In essence, these definitions can be summarized as follows: an XML fragment is a (not necessarily well-formed) piece of an XML document or subset of an XML collection.

Fragmentation techniques provide a way to "cut" the database into pieces (fragments). This way, queries can also be "cut" into sub-queries that run in parallel over smaller portions of data, thus achieving better performance when compared to its serial execution over the whole document. Successful parallel query processing in relational databases has been directly achieved through table fragmentation. By using the relational algebra to define

[1] `http://hadoop.apache.org`

fragments, algebraic query processing on these fragments can be done with correctness rules [58]. However, there is no consensus in the database community as to what an XML fragment precisely is. In fact, several approaches in literature present definitions of XML fragments [4, 8, 11, 12, 39, 42, 41, 43, 47, 62]. They can be classified according to the way they fragment the data. Regardless of the fragmentation type, the fragmentation unit is usually a collection of elements, which can be of Multiple Documents (MD) or Single Document (SD) [75].

Given the large amount of approaches that propose fragmentation alternatives in literature, in this paper we survey the existing approaches, comparing their features and establishing a timeline, so the reader can understand the evolution of the proposed solutions, their characteristics, pros and cons. It is important to note that, when we refer to distribution, we are not considering distributed query processing in data integration scenarios [5, 33, 44, 68], since their main goal is to provide data access, not necessarily in a high performance fashion. We also do not consider approaches that fragments the document solely for internal query processing [40, 53, 72]. Instead, we focus on approaches that aim to improve the performance on processing collections of documents by distributing data to several nodes in a network and use parallel processing strategies such as MapReduce.

The remaining of this paper is organized as follows. Section 2 provides intuitive definitions for XML fragmentation, and Section 3 formalizes them. Section 4 discusses fragmentation techniques, while Section 5 discusses their main features, comparing them regarding several aspects. Finally, we discuss open problems in Section 6.

2. AN INTUITIVE GRASP ON XML FRAGMENTATION

Data fragmentation is characterized by physical changes to the dataset, that is, the dataset is fragmented and allocated to multiple computational nodes [4, 8, 12, 39, 42, 41, 43, 47]. In their book [58], Ozsu and Valduriez present a detailed introduction to the subject of XML distributed query processing and XML fragmentation. They classify fragmentation in two groups: *ad-hoc* and structured. *Ad-hoc* fragmentation does not take the document schema into consideration. Edges are arbitrarily removed from the document. Structured fragmentation, on the other hand, is one that uses schema property(ies) to define the fragments [58]. In this work we are focused on large-scale distributed and parallel query processing. In structured fragmentation, each fragment is usually defined by using

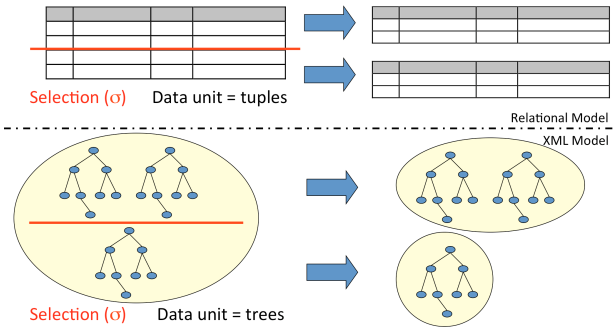

Figure 1: Horizontal fragmentation in the relational model and in the XML model

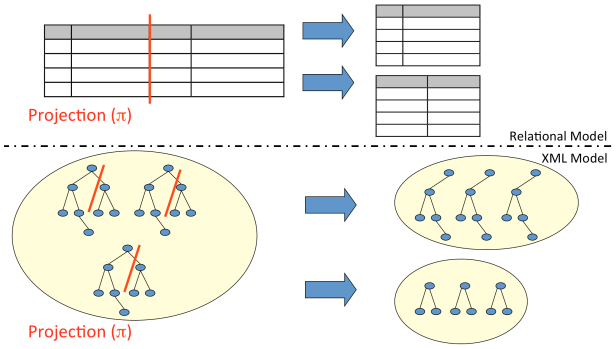

Figure 2: Vertical fragmentation in the relational model and in the XML model

set operations (i.e. algebraic selection, projection) over the dataset. In this case, the specification of queries and fragments share the same operations, which makes it easier to automatically decompose queries to run in parallel over the fragments.

To help the intuitive grasp on XML fragments we refer to fragmentation in the relational model [58], where horizontal fragments are those obtained by table selection operations. This means that horizontal fragments are restricted by the selection predicate and follow the same schema of the original table. Vertical fragments, on the other hand, are defined by projection operations, and thus follow a different schema of the original table. The same principle can be applied to XML databases [4, 42]. Figure 1 and Figure 2 establish a parallel between data fragmentation in the relational and in the XML models.

In distributed databases, queries are executed over one specific target fragment or in parallel by accessing different fragments of the dataset independently. Data fragmentation can be very effective in distributed query processing where there is a well-known set of frequent queries. Such queries must be analyzed so that a corresponding fragmentation design[2] can be achieved [6, 8, 42, 48]. A

[2]The process of deciding on how to fragment the

good fragmentation design is one that benefits the frequent queries, thus allowing for gains in performance. Gains are focused on locality of access and data pruning rather than parallel processing. The price to be paid is poor performance for some of the non-frequent queries and limited opportunities for high performance parallel queries.

To illustrate, assume an XML collection *COrders* that contains information about customer's orders. Queries are usually targeted at regions as follows: South America, North America or Other Continent. Assume also that the collection is physically fragmented into three fragments, based on their location: one containing orders of customers from North America (allocated at node *n1*), one containing orders of customers from South America (allocated at node *n2*), and a third one containing orders from other Continents (allocated at node *n3*).

Now, assume that customers from North America buy items usually over 1,000 dollars. On the other hand, South American customers' orders are usually around 500 dollars. Now, suppose we want to run a query to retrieve the average total of orders with items above 1,000. This query will be distributed to the three fragments: sub-query *s1* will be executed over the North America fragment, sub-query *s2* will query the South America fragment, and sub-query *s3* will run over the remaining fragment. Since there are a lot more orders from customers in North America that satisfy this query predicate, *s2* and *s3* will probably finish their processing way before *s1*. Thus, the total query processing time will be highly influenced by *s1,* and there is nothing *n2* and *n3* can do to help *n1* processing *s1*, since they do not have the required data. If there is a following operation for these intermediate results, skew continues to be propagated. Even if the fragments are replicated, since the sub-query *s1* is already running in *n1* over the whole North America fragment there is no way to stop this subquery (assuming query processing follows a static optimized execution plan) and redistribute the elements of this long processing fragment to idle nodes.

This type of fragmentation favors the distributed processing with the goal of restricting the access to a subset of the data by using pruning strategies. This happens in cases where a query accesses a subset of the fragments (orders from North America with items above 2,000 dollars, for instance). When the goal is high performance, horizontal and vertical fragmentation can still be used. The idea in this case is to generate uniform fragments, with similar number of tuples/elements each. The goal is to favor parallel processing by using the largest possible number of processing units. Still in this case, data skew is a problem, since the query processing time will also be highly influenced by selectivity factors and the node that contains the largest number of tuples/elements that satisfy the query predicate can be overloaded.

database (how each fragment should be defined) is called *fragmentation design.*

3. FORMAL DEFINITIONS

Based on the intuition of XML fragmentation presented in the previous section, we now formalize the main concepts related to XML fragmentation. As mentioned in the introduction, there is no consensus in the literature as to what an XML fragment exactly is. In this section, we present the definitions proposed in [4] as an illustrative example.

XML Document. 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.

Types. 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 3 shows the *SOrders* schema that was used in the example given in Section 2. Over this schema, we can define collections such as $COrders := \langle SOrders, order \rangle$, or $CItems := \langle SOrders, items \rangle$, both of Multiple Documents.

Path Expression. A *path expression* P is a sequence $/\mathtt{e}_1/\ldots/\{\mathtt{e}_k \mid \mathtt{@a}_k\}$, where $\mathtt{e}_x \in \mathcal{L}$, $1 \leq x \leq k$,

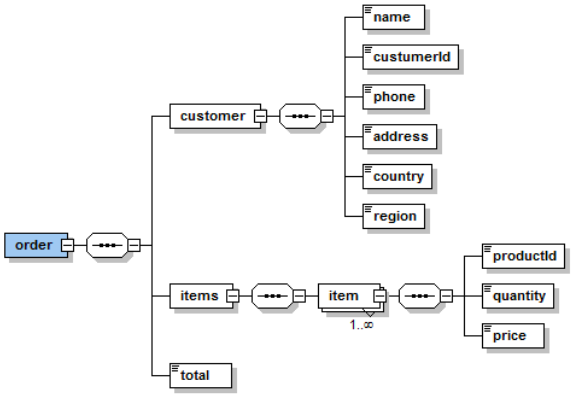

Figure 3: *SOrders* sample schema

and $a_k \in \mathcal{A}$. P may optionally contain "$*$" 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 (i.e., 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).

The following XML fragmentation definition builds on the semantics of the operators from the TLC (Tree Logical Classes) algebra [59], since it is one of the few XML algebras [30, 32, 37, 63, 76, 77] that operates on sets of data trees, which are natural operands for set fragmentation operations.

XML Fragment. A fragment F of a homogeneous collection C is a collection represented by $F := \langle C, \gamma \rangle$, where γ denotes an operator 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.

Instances of a fragment F are obtained by applying γ to each document in C. The collection 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).

Horizontal Fragmentation. A horizontal fragment F of a collection C is defined by the selection operator (σ) [59] 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.

$$FCustomer := \langle COrders, \pi_{/order,\{/order/items\}} \rangle$$
$$FItems := \langle COrders, \pi_{/order/items,\{\}} \rangle$$

Figure 5: Example of vertical fragments

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 μ is true. Figure 4 shows the definition of the horizontal fragments we used in Section 2. Notice that the third fragment is defined as the complement of the other two.

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

Vertical Fragmentation. A vertical fragment is obtained by applying the projection operator (π) [59] 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 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.

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. As an example, we could separate customers from items, placing them in different fragments, as shown in Figure 5. Note that this is an alternative fragmentation, non-related to the one shown in Figure 4.

It is worth mentioning that the path expression P cannot retrieve nodes that may have cardinality greater than one, except when the element order is indicated. 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.

Hybrid Fragmentation. The idea of hybrid fragmentation is to apply a vertical fragmentation followed by a horizontal fragmentation, or vice-versa.

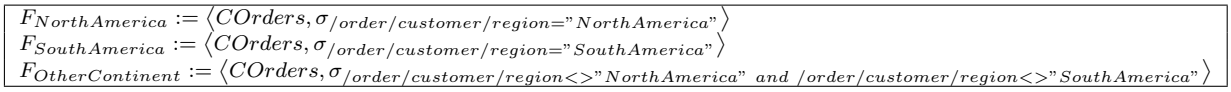

Figure 4: Example of horizontal fragments

An interesting use of this technique is to normalize the schema of XML collections in SD repositories, thereby allowing horizontal fragmentation.

Let σ_μ and $\pi_{P,\Gamma}$ be selection and projection operators, respectively, defined over a collection C. A hybrid fragment of C is denoted 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 μ.

Correctness Rules. 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 correct fragmentation of C:

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 [37] is used (TLC is an extension of TAX [37]), and for vertical fragmentation, the join ($\bowtie$) operator [59] is used.

These rules are important to guarantee that queries are correctly translated from the centralized environment to the corresponding fragmented one, and that results are correctly reconstructed.

4. FRAGMENTATION TECHNIQUES

In the next subsection we discuss *ad-hoc* fragmentation alternatives and then we focus on several approaches for structured fragmentation, following Ozsu and Valduriez [58] fragmentation classification.

4.1 Ad-hoc Fragmentation

Ad-hoc fragmentation approaches do not take the schema into consideration when defining the fragments. To generate the fragments, they either arbitrarily cut the document and mark it in a way that it can be reconstructed later, or use some kind of constraint over the document. We call these alternatives holes and fillers, and constraint-based, respectively.

Holes and Fillers. Bose et al propose a fragmentation model for stream data [11]. Using this fragmentation model, Bose and Fegaras [10] focus on processing fragmented data streams using a system they call XFrag. In this context, the goal is to fragment stream data and send it through the network. XFrag uses the concept of holes and fillers to fragment XML documents that will be sent over the network as data streams. The original document is divided into 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 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.

Lee, Kim and Kang [45] analyze the classical holes and fillers approaches for stream query processing [10, 36] and claim they are inefficient in terms of memory consumption. To overcome this limitation, they propose to use a labeling scheme to connect vertical fragments of an XML document. In a XML tree, a labeling schema (such as global order or Dewey encoding [70]) can be used to connect parent and child. In the same way, Lee, Kim and Kang use a labeling schema to connect vertical fragments, where each fragment is a subtree of the original tree. Despite the ad-hoc nature of this approach, the authors mention the importance of the reconstruction property for correctness. Although there are not an explicit notion of holes and fillers in this approach, each fragment carries an id that is used to connect it to its parent, which is similar to the other holes and fillers approaches.

The query processing technique is also different from the ones applied to stored data. If we were to apply traditional distributed query processing techniques with the fragmentation schema proposed in [11], query processing would be inefficient, since it would be necessary to completely reconstruct the document before processing the query. This is because XFrag does 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.

The *ad-hoc* fragmentation proposed by Abiteboul et al. [1, 2] use remote function calls (web ser-

vices) to fragment XML documents. The approach is called Active XML, and the goal is to guarantee that documents are up-to-date with respect to the ever-augmenting dynamic issues in current distributed and parallel computing environments. From time to time, or at query time, the web services are called and the results are embedded as XML fragments into the Active XML document. Thus, in this approach, web service calls represent cross-fragment edges [58]. Note that this can be seen as a holes and fillers approach, where web service calls are considered holes, which mark fragmentation spots, and their results are considered fillers.

Constraint-based. Bonifati and Cuzzocrea [8] propose a fragmentation approach based on structural constraints of the XML document: size, tree-width, and tree-depth. Given values for these three constraints, their approach finds ways of fragmenting the original tree to satisfy the constraints. The approach is based on a set of heuristics, and it is called SimpleX. There is no separation of fragmentation types, and no correctness rules. The approach was designed to work with stream data, but can be applied over stored data as well. The problem is that there are several possible ways of fragmenting a database while respecting the constraints. In this sense, Waldvogel, Kramis and Graf [73] propose five split algorithms and evaluate their performance, with the aim at finding the one that produces the most effective fragments.

Choi et al [17] use MapReduce to process a set of small XPath queries in parallel over large XML documents in an approach called HadoopXML. The input XML file is fragmented so that data blocks of equal size are produced. This is similar to the size constraint proposed by Bonifati and Cuzzocrea [8], and, as such, there is also no correctness rules or a formal concept of fragment type. The main goal of HadoopXML is to process as many small queries as possible in parallel. High-cost *ad-hoc* queries are not addressed.

4.2 Structured Fragmentation

According to Ozsu and Valduriez, *ad-hoc* fragmentation works well when data is already distributed. However, since there is no clear fragmentation predicate, there is less opportunity for distributed query optimization. An alternative that addresses this issue is structured fragmentation, which is based on the concept of fragmenting an XML data collection according to some properties of the schema [58].

Structured fragmentation can be performed in several ways. In this survey, we classify the existing approaches according to the way they define the fragments, which can be: hybrid, XPath-based, and set-oriented.

Hybrid. Ma and Schewe [47] base their XML fragments definition on ideas from fragmentation of object databases. In fact, a previous publication of Schewe contrasts fragmentation for these two models [64]. In their work, Ma and Schewe propose three types of XML fragmentation: *horizontal*, which groups elements of an XML document according to some selection criteria; *vertical*, which restructures a document by unnesting some elements; and a special type named *split*, that breaks an XML document into a set of new documents. Despite the use of the names *horizontal* and *vertical*, their fragments are not purely based on selection and projection. 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 restructure the document, and artificial attributes to properly connect document fragments (for reconstruction purposes). Even though the user can define fragments from several XML documents, their approach is not suitable for MD repositories. In this case, to define horizontal fragments, the user must first integrate all XML documents into an SD view. It is important to note that they followed up with their work [49, 50] by proposing a cost model to help the fragments design aiming at reducing the query processing time [48, 52, 51].

XPath-based. Bremer and Gertz [12] propose an approach for distributed XML design, covering both data fragmentation and allocation. Fragments are defined using XF, a subset of XPath. Each fragment definition consists of two parts: a *selection fragment* and optionally a set of *exclusion fragments*. The selection fragment is applied to the XML database, resulting in a set of nodes N. Then, the exclusion fragments are applied over N. The results are the desired fragments. It is clear that this formalism does not distinguish between horizontal and vertical fragmentation, which are combined into a hybrid type of fragment definition. Nevertheless, their approach only addresses SD repositories. 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. Although Bremer and Gertz mention correctness rules, they do not present a reconstruction rule, which is crucial for automatic query processing.

Bonifati et al. [9] also define vertical fragments using XPath expressions. In the fragments definition, path expressions may contain child axes and positional filters. One fragment may possibly reference many fragments: a single super fragment, that is an ancestor of the current fragment, and (possibly many) child fragments. Child fragments are connected to their parent through *sub* tags that are artificially inserted into the parent fragment. Each *sub* tag references a child fragment. These definitions are used in a DHT (distributed hash table) P2P (peer-to-peer) system that supports XPath lookup queries. Since no selection predicate is allowed in the fragment definition, this approach does not consider horizontal nor hybrid fragmentation. An extended version of this paper, including experimental results, appeared in a journal paper in 2006 [7]. The main goal of the experiment was to measure the number of hops needed to find a query answer in the DHT, assuming the data is fragmented according to their definitions. Note that this approach has also the idea of holes and fillers, but it is classified as structured since it uses XPath to define the fragments.

The approach of Jeong et al. [38] also resembles the idea of holes and fillers. It splits the XML tree into subtrees with the goal of easing the processing of keyword-based queries. Fragments are defined using XPath and connected in a tree of fragments that is called Fragment Object Tree. There is no clear distinction between fragment types, but it is possible to use filters in the XPath expressions to define fragments (which plays the role of selections), and each of the several XPath expressions that define the Fragment Object Tree plays the role of a projection. Thus, we could classify this approach as providing a hybrid fragmentation.

Fegaras et al. [29] propose a declarative language to specify MapReduce jobs over XML documents called MRQL [28]. They propose a fragmentation technique based on the Hadoop input format. Fragments are defined by synchronization nodes and XPath expressions that are applied over these nodes. There are no correctness rules, but there is an algebra behind MRQL that is used to perform query optimization.

Set-oriented. Andrade et al. [4] defined horizontal, vertical and hybrid XML fragments inspired by the analogous definitions for the relational model [58]. Their approach is called PartiX and explores the analogy between relations and collections of trees (both are *sets*). It supports both SD and MD databases. Fragments are defined by XML algebra expressions [59]. A horizontal fragment is defined by a selection operation, while a vertical fragment is defined by a projection, plus an optional set of path expressions that point to subtrees to be pruned out of the fragment. A hybrid fragment is defined by a selection followed by a projection, or vice-versa. The definitions presented on Section 4 are extracted from this work, which is pioneer in the sense of formally defining correctness rules for the fragmentation definition. As in the relational model, the same algebra is used to define fragments and query predicates, which helps query decomposition and predicate matching. By using the same algebra and correctness rules, they were able to define a query processing methodology that is capable of automatically processing queries over distributed and fragmented databases [59].

Kido, Amagasa and Kitagawa [39] proposed horizontal and vertical fragmentation for XML data that was also inspired by the relational model. In their work, they use Data Guides [34] as the schema definition for the XML database. Their fragments are then specified over the Data Guide, which can be represented as a graph. A vertical fragment is a sub-graph of the graph that represents the database schema. A horizontal fragment is defined over a vertical fragment. When compared to the approach of Andrade et al. [4], the horizontal fragment of Kido, Amagasa and Kitagawa [39] is similar to the hybrid fragment of Andrade et al. They do not have a specific horizontal fragment as presented in PartiX [4], and no reconstruction rule either.

Kling, Ozsu and Daudjee [41] propose vertical fragmentation for XML databases. Their definition of vertical fragments is similar to that of [39]. However, they do not define horizontal nor hybrid fragments, although they mention they should be defined by selections (horizontal fragments) and selection plus projections or vice-versa (hybrid fragments).

This, in fact, is done in their following work [42], where Kling, Ozsu and Daudjee present definitions for horizontal and vertical fragments. The revised ideas in [42] are equivalent to those of Andrade et al. [4]. A horizontal fragmentation algorithm is defined over MD collections by using selection predicates and minterm[3] combinations, resulting in homogeneous fragments. Vertical fragments are defined by partitioning the schema of the documents into disjoint subgraphs, and can be applied to either SD or MD collections. Finally, horizontal and vertical fragments can be combined into hybrid fragments. The focus of their fragmentation design technique, however, is on pruning fragments for distributed

[3]A minterm is a conjunction of simple predicates [58].

query processing rather than on high performance parallel processing. It is targeted on a previously known frequent set of queries.

A common problem to [4] and [42] is that the use of minterms lead to a fixed number of fragments, disregarding the number of available processing units. This approach applies to distributed databases, but not to parallel query processing, since the number of fragments is typically much smaller than the current available processing units, which leads to idle processors and load unbalance. Since the number of processing units can be very dynamic, any design with a fixed number of fragments will require additional techniques to provide for high performance.

5. DISCUSSION

Now that we have discussed the main approaches on physical XML fragmentation, in this section we study the impact they had in literature by establishing a timeline. Finally, we analyze the features of each of the proposed approaches, summarizing their differences. Our goal is to try to obtain a uniform view on the XML fragments definitions and show the benefits of the different fragmentation design techniques so the designer can choose according to the target query-processing scenario.

In the timeline, shown in Figure 6, we analyze when each of the approaches was proposed. Ad-hoc approaches are shown in green, while structured approaches are shown in blue. White rectangles denote derived approaches. We consider a paper to be derived from a previous one if it discusses new features based on previous definitions. For instance, Figueiredo, Braganholo and Mattoso [31] used the definition of Andrade et al. [4] to propose a methodology for distributed XML query processing, and thus it is shown in white in the Figure. For derived approaches, we use a solid line to point to the original approach (either a green or blue rectangle). This way we can analyze the impact and continuity of each of the individual approaches. The dotted line that connects Ma and Schewe's approach with Schewe's approach denotes a previous work that pavements the subject, but do not present the fragments definitions themselves.

Additionally, each approach is positioned into a lane, according to its classification as constraint-based, holes and fillers, XPath-based, set-oriented or hybrid. This helps one to have a better understanding of how each type of approach evolved over time. While the timeline evidences that the subject has been continuously investigated throughout the years, Table 1 and Table 2 summarize the features of each of the approaches for structured (Table 1) and *ad-hoc* (Table 2) fragmentation, shown in Figure 6.

The first interesting feature to note is that, from all of the approaches, only two support fragmentation of MD collections [4, 42]. These two approaches are equivalent in terms of ideas. The only difference resides in the formalism that is used to present them. Ma and Schewe [47] also mention support to MD collections, but this needs to be done by first defining a single SD view, and then applying the fragmentation over this SD view. This maneuver could be applied to the other approaches as well, but it creates an artificial layer that needs to be handled by the user (that needs to create the views), and also at query processing time (it may decrease query performance). All of the approaches support fragmentation of SD collections.

Regarding the fragmentation types, most of the approaches allow a hybrid type of fragment [4, 12, 28, 38, 39, 42, 47]. In fact, in [39], a horizontal fragment is a subset of instances conforming to a vertical fragment. Thus, to be able to achieve horizontal fragmentation, one needs first to vertically fragment the database. Due to that reason, we consider this to be a hybrid fragment instead of a horizontal one. In a similar line of thought, Ma and Schewe [47] define their horizontal fragments by using selection and projection operations. In this paper, we consider them to be hybrid fragments, since horizontal fragments are classically defined only by selection operations [58]. However, we classified both [47] and [39] as allowing horizontal fragments in Table 1, preserving the classification given by the paper authors. According to this discussion and reclassification of fragmentation types, only [4, 42] support pure horizontal fragmentation.

As for vertical fragments, four out of nine structured approaches [4, 39, 42, 41] support pure vertical fragmentation. Ma and Schewe allow a special type of projection that unnests elements and arrange then into new artificial elements in the XML document. Thus, they do not use pure projections to define the vertical fragments. Additionally, they require artificial attributes to be created to support document reconstruction. Such attributes contain references to connecting fragments. Similarly, Bonifati et al. require an artificial element called sub to connect vertical fragments. Thus, we do not consider it to use pure projections.

Since ad-hoc fragmentation does not rely on an explicit fragmentation specification, Table 2 does not present the fragmentation types.

As mentioned before, in the same way as in the relational model [10], correctness rules are needed for

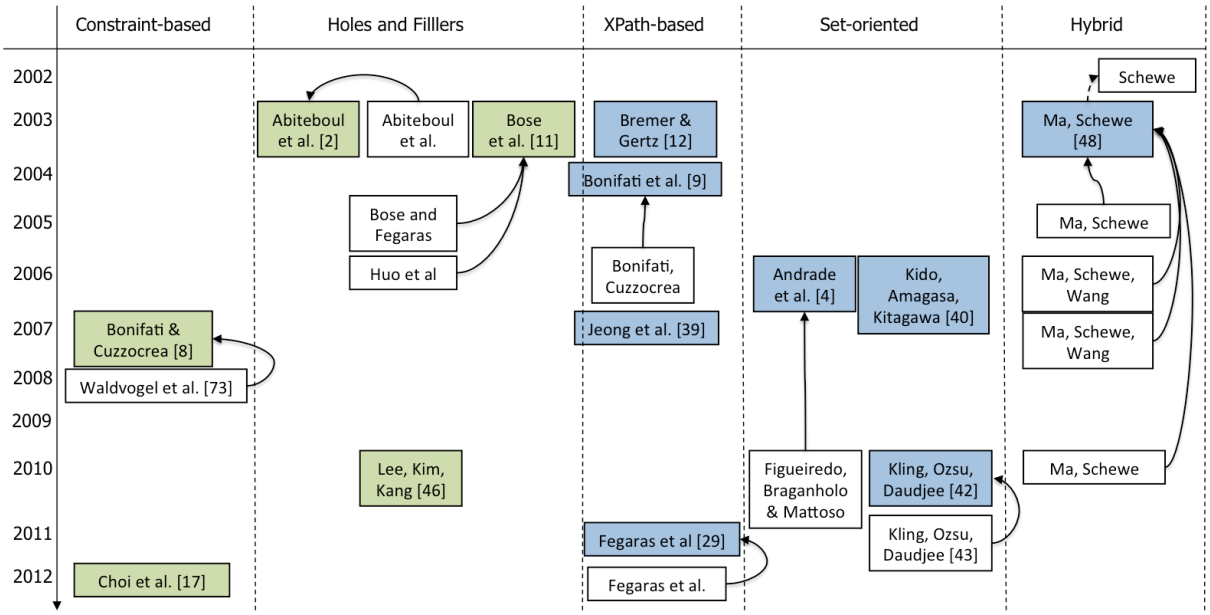

Figure 6: Timeline of the XML fragmentation approaches

automatic distributed query processing. When we use the same XML algebraic operations in fragmentation definition and query processing, automatic mappings from centralized to distributed fragments can be achieved following on correctness rules. Ma and Schewe [47] mention the need of creating artificial attributes for reconstruction purposes. However, only three of the proposed approaches present or mention correctness rules [4, 12, 39]. The approach in [39], however, does not present them formally, and does not include the reconstruction rule, which is crucial for reconstructing the original database from its fragments and vice-versa, thus allowing a correct mapping from centralized database to the corresponding distributed database. This allows for automatic distributed query processing on top of the mapped fragments. Bremer and Gertz's approach [12], on the other hand, does mention the importance of a reconstruction rule, but does not define it. Only [4] formally defines the three correctness rules. In fact, these rules are then explored in their following work [59] to automatically process distributed queries over the fragmented database.

Finally, most of the approaches deal with stored data [2, 4, 8, 12, 28, 38, 39, 42, 41, 47] instead of streams, and use native XML databases to store the data [2, 4, 8, 12, 38, 42, 41, 47]. Approaches based on MapReduce use HDFS to store the data [17, 28].

6. OPEN PROBLEMS

Based on our analysis of previous work in XML fragmentation, we have identified a list of open problems, which we discuss next.

Fragmentation Design. When fragmentation design is used, the most frequent queries must be known to derive a good fragmentation schema - one that benefits the most frequent queries. Experiments in literature show that queries that do not benefit from the fragmentation design suffer large impacts on performance [39, 59]. Only a few work in literature present algorithms for XML fragmentation design [6, 8, 42, 48, 66]. However, these algorithms were not used in parallel query processing and they do not present correctness rules for the resulting fragmentation. This makes it difficult to check the correctness of the fragmentation schema designed by the algorithms.

Even in cases where fragmentation design was performed, it cannot be easily adapted to changes in the query processing environment (for example, the addition of processing nodes). Additionally, pruning irrelevant fragments limits parallelism and performance gains, since several computational nodes remain idle when they have fragments that are irrelevant to the query that is being processed. In fact, experimental results [42] show that the pruning algorithm does not improve query performance.

Implementation. The absence of correctness rules also affects query processing. Most of the approaches show evaluation performance of query processing, and thus they implemented a prototype that is capable of processing queries. However, due to the absence of correctness rules, their prototype becomes a black box, very complex to be re-implemented by people outside their research group. This also prevents these approaches from

Table 1: Summary of the approaches for structured fragmentation

	MD	SD	horizontal	vertical	hybrid	correctness rules	source data	storage
Ma, Schewe [47]	N	Y	selection + projection	artificial elements	–	–	stored	native
Bremer, Gertz [12]	N	Y	–	–	selection fragments + exclusion fragments	disjointness, completeness	stored	native
Bonifati et al. [9]	N	Y	–	projection and artificial elements	–	–	stored	native
Andrade et al. [4]	Y	Y	selection	projection	selection + projection	disjointness, completeness, reconstruction	stored	native
Kido, Amagasa, Kitagawa [39]	N	Y	subset of the vertical fragments	subgraph of a dataGuide	–	disjointness, completeness	stored	relations
Jeong et al. [38]	N	Y	–	–	selection + projection	–	stored	native
Kling, Ozsu, Daudjee [41]	N	Y	–	subset of a schema graph	–	–	stored	native
Kling, Ozsu, Daudjee [42]	Y	Y	selection	projection	selection + projection	–	stored	native
Fegaras et al. [29]	N	Y	–	–	selection + projection	–	stored	HDFS

Table 2: Summary of the approaches for ad-hoc fragmentation

	MD	SD	correctness rules	source data	storage
Abiteboul et at. [2]	N	Y	–	stored or stream	native
Bose at al. [11]	N	Y	–	stream	–
Bonifati, Cuzzocrea [8]	N	Y	–	stored or stream	native
Lee, Kim, Kang [45]	N	Y	–	stream	–
Choi et al. [17]	N	Y	–	stored	HDFS

being used in Map-Reduce settings. We have unsuccessfully searched for publicly available implementations of each of the approaches, but only found one of Fegaras et al. [28]. In fact, it is currently an incubated project at Apache, which is available at `http://lambda.uta.edu/mrql`. As for the ad-hoc approaches, only Active XML is publicly available at the OW2 open source portal (`http://forge.ow2.org/projects/activexml`).

Virtual Fragmentation. The approaches we discuss in this paper can all be classified as physical fragmentation, since they physically break the XML tree into several pieces. Virtual fragmentation [54] is an alternative to physical fragmentation, and consists of replicating the database into several nodes and distributing the query into subqueries so that each node runs over a different portion of the data. Virtual fragmentation approaches [62] do not suffer from problems related to the physical fragmentation design, since data is replicated in all nodes, and queries run over all available nodes, each over a small non-overlapping portion of the data. The sub-queries are run in parallel, thus achieving gains in performance when compared to centralized approaches.

Virtual fragmentation is trickier in the XML model than in the relational model because of the lack of keys in the former. In fact, in the relational model, the sub-queries are built by adding selection predicates that range over the domain of the table key. The XPath function *position()* is a good replacement for the table key because it is unique (a given element in an XML document has a unique position in the context of its parent). However, this is not enough. In the same way that virtual fragmentation in the relational model needs the help of clustered indices, another feature of the Native XML databases is crucial for the virtual fragment to work properly - each element must be indexed by its position, so that the query processor can go directly and only access the desired elements when processing a sub-query. Full scans must be avoided at all costs. In fact, experimental results by Silva et al. [65] show that, in general, native DBMSX index XML elements by their position. Thus, virtual fragmentation is a promising technique, especially in dynamic environments such as clouds, but needs further investigation.

Parallelism. One of the main problems in obtaining acceleration in parallel query processing is load

balance, which is a problem in almost all scenarios. Load balance is very important for efficient query processing, and is not taken into account in existing approaches, not even in virtual fragmentation [62]. MapReduce tries to deal with load balancing by submitting backup tasks when the whole process is close to completion. This way, only the answer provided by the task that finishes first is considered [23]. This does not solve the problem, but it is a start. Recent work has been done on the issue of optimizing XML query processing in MapReduce [27, 28, 29]. Others leave the fragmentation technique open [19]. The authors of [62] suggest using adaptive virtual fragmentation [46] to solve this issue. This, however, has not been done yet.

Acknowledgements. We would like to thank Luiz Augusto Matos da Silva for helping in the bibliographic search. We would also like to thank CNPq and FAPERJ for partially supporting this research.

7. REFERENCES

[1] S. Abiteboul, A. Bonifati, G. Cobena, C. Cremarenco, F. Dragan, I. Manolescu, T. Milo, and N. Preda. Managing distributed workspaces with active XML. In *VLDB*, pages 1061–1064, 2003.

[2] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *SIGMOD*, pages 527–538, 2003.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[4] A. Andrade, G. Ruberg, F. Baião, V. Braganholo, and M. Mattoso. Efficiently processing XML queries over fragmented repositories with PartiX. In *DATAX*, pages 150–163, 2006.

[5] C. Baru, A. Gupta, B. Ludascher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-based information mediation with MIX. *SIGMOD Record*, 28(2):597–599, 1999.

[6] L. Birhanu, S. Atnafu, and F. Getahun. Native XML document fragmentation model. In *SITIS*, pages 233 –240, 2010.

[7] A. Bonifati and A. Cuzzocrea. Storing and retrieving XPath fragments in structured P2P networks. *DKE*, 59(2):247–269, 2006.

[8] A. Bonifati and A. Cuzzocrea. Efficient fragmentation of large XML documents. In *DEXA*, pages 539–550, 2007.

[9] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath lookup queries in P2P networks. In *WIDM*, pages 48–55, 2004.

[10] S. Bose and L. Fegaras. XFrag: a query processing framework for fragmented XML data. In *WebDB*, pages 97–102, 2005.

[11] S. Bose, L. Fegaras, D. Levine, and V. Chaluvadi. A query algebra for fragmented XML stream data. In *DBPL*, pages 195–215, 2003.

[12] J.-M. Bremer and M. Gertz. On distributing XML repositories. In *WebDB*, pages 73–78, 2003.

[13] J.-M. Bremer and M. Gertz. Integrating document and data retrieval based on XML. *The VLDB Journal*, 15(1):53–83, 2006.

[14] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: a hardness study and a principled heuristic solution. *IEEE TKDE*, 16(11):1313 – 1323, 2004.

[15] D. Che, K. Aberer, and T. Ozsu. Query optimization in XML structured-document databases. *The VLDB Journal*, 15(3):263–289, 2006.

[16] D.-R. Che. Accomplishing deterministic XML query optimization. *Journal of Computer Science and Technology*, 20(3):357–366, 2005.

[17] H. Choi, K.-H. Lee, S.-H. Kim, Y.-J. Lee, and B. Moon. HadoopXML: a suite for parallel processing of massive XML data with multiple twig pattern queries. In *CIKM*, pages 2737–2739, 2012.

[18] C.-W. Chung, J.-K. Min, and K. Shim. APEX: an adaptive path index for XML data. In *SIGMOD*, pages 121–132, 2002.

[19] G. Cong, W. Fan, A. Kementsietsidis, J. Li, and X. Liu. Partial evaluation for distributed XPath query processing and beyond. *ACM TODS*, 37(4):32:1–32:43, 2012.

[20] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *VLDB*, pages 341–350, 2001.

[21] D. Dash, N. Polyzotis, and A. Ailamaki. CoPhy: a scalable, portable, and interactive index advisor for large workloads. *PVLDB*, 4(6):362–372, 2011.

[22] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[23] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[24] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *CACM*, 53(1):72–77, 2010.

[25] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1-2):515–529, 2010.

[26] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only aggressive elephants are fast elephants. *PVLDB*, 5(11):1591–1602, 2012.

[27] L. Fegaras. Supporting bulk synchronous parallelism in map-reduce queries. In *SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1068–1077, 2012.

[28] L. Fegaras, C. Li, and U. Gupta. An optimization framework for map-reduce queries. In *EDBT*, pages 26–37, 2012.

[29] L. Fegaras, C. Li, U. Gupta, and J. J. Philip. XML query optimization in map-reduce. In *WebDB*, pages 1–6, 2011.

[30] M. Fernandez, J. Simeon, and P. Wadler. An algebra for XML query. In *FST TCS*, pages 11–45, 2000.

[31] G. Figueiredo, V. Braganholo, and M. Mattoso. Processing queries over distributed XML databases. *JIDM*, 1(3):455–470, 2010.

[32] F. Frasincar, G.-J. Houben, and C. Pau. XAL: an algebra for XML query optimization. *Australasian Computer Science Communications*, 24(2):49–56, 2002.

[33] G. Gardarin, A. Mensch, T.-T. Dang-Ngoc, and L. Smit. Integrating heterogeneous data sources with XML and XQuery. In *DEXA*, pages 839–846, 2002.

[34] R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[35] G. Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE TKDE*, 19(10):1381 –1403, 2007.

[36] H. Huo, G. Wang, X. Hui, R. Zhou, B. Ning, and C. Xiao. Efficient query processing for streamed XML fragments. In *Database Systems for Advanced Applications*, volume 3882 of *Lecture Notes in Computer Science*, pages 468–482. 2006.
[37] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: a tree algebra for XML. In *DBPL*, pages 149–164, 2001.
[38] C.-H. Jeong, Y. Choi, D.-S. Jin, M. Lee, S.-P. Choi, K. Kim, M.-H. Cho, W.-K. Joo, H.-M. Yoon, J.-H. Seo, and J. Kim. Service-centric object fragmentation for efficient retrieval and management of huge XML documents. In *PDCAT*, pages 118–124, 2007.
[39] K. Kido, T. Amagasa, and H. Kitagawa. Processing XPath queries in PC-Clusters using XML data partitioning. In *ICDE Workshops*, pages 114–119, 2006.
[40] J. Kim and H.-J. Kim. A partition index for XML and semi-structured data. *DKE*, 51(3):349–368, 2004.
[41] P. Kling, M. Ozsu, and K. Daudjee. Generating efficient execution plans for vertically partitioned XML databases. *PVLDB*, 4(1):1–11, 2010.
[42] P. Kling, M. Özsu, and K. Daudjee. Scaling XML query processing: distribution, localization and pruning. *Distributed and Parallel Databases*, 29(5):445–490, 2011.
[43] H. Kurita, K. Hatano, J. Miyazaki, and S. Uemura. Efficient query processing for large XML data in distributed environments. In *AINA*, pages 317–322, 2007.
[44] K. Lee, J. Min, and K. Park. A design and implementation of XML-Based mediation framework (XMF) for integration of internet information resources. In *HICSS*, pages 202–202, 2002.
[45] S. Lee, J. Kim, and H. Kang. Memory-efficient query processing over XML fragment stream with fragment labeling. *Computing and Informatics*, 29(5):757–782, 2010.
[46] A. Lima, M. Mattoso, and P. Valduriez. Adaptive virtual partitioning for OLAP query processing in a database cluster. *JIDM*, 1(1):75–88, 2010.
[47] H. Ma and K.-D. Schewe. Fragmentation of XML documents. In *SBBD*, pages 200–214, 2003.
[48] H. Ma and K.-D. Schewe. Heuristic horizontal XML fragmentation. In *CAISE*, pages 131–136, 2005.
[49] H. Ma and K.-D. Schewe. Fragmentation of XML documents. *JIDM*, 1(1):21–34, 2010.
[50] H. Ma and K.-D. Schewe. Revisiting "Fragmentation of XML documents". *JIDM*, 1(1):35–36, 2010.
[51] H. Ma, K.-D. Schewe, and Q. Wang. A heuristic approach to cost-efficient fragmentation and allocation of complex value databases. In *ADC*, pages 183–192, 2006.
[52] H. Ma, K.-D. Schewe, and Q. Wang. A heuristic approach to cost-efficient derived horizontal fragmentation of complex value databases. In *ADC*, pages 103–111, 2007.
[53] I. Machdi, T. Amagasa, and H. Kitagawa. XML data partitioning strategies to improve parallelism in parallel holistic twig joins. In *ICUIMC*, pages 471–480, 2009.
[54] M. Mattoso. Virtual partitioning. In L. Liu and M. T. Ozsu, editors, *Encyclopedia of Database Systems*, pages 3340–3341. 2009.
[55] J. McHugh and J. Widom. Query optimization for XML. In *VLDB*, pages 315–326, 1999.
[56] M. M. Moro, V. Braganholo, C. F. Dorneles, D. Duarte, R. Galante, and R. S. Mello. XML: some papers in a haystack. *SIGMOD Record*, 38(2):29–34, 2009.
[57] W. Ng and J. Cheng. An efficient index lattice for XML query evaluation. In *DASFAA*, pages 753–767, 2007.
[58] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. 3 edition, 2011.
[59] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMO*, pages 71–82, 2004.
[60] Paul Grosso and Daniel Veillard. XML fragment interchange. W3C candidate recommendation 12 february 2001., 2001. W3C Candidate Recommendation 12 February 2001.
[61] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.
[62] C. Rodrigues, V. Braganholo, and M. Mattoso. Virtual partitioning ad-hoc queries over distributed XML databases. *JIDM*, 2(3):495–510, 2011.
[63] C. Sartiani and A. Albano. Yet another query algebra for XML data. In *Database Engineering and Applications Symposium*, pages 106–115, 2002.
[64] K.-D. Schewe. Fragmentation of object oriented and semistructured data. In *BalticDB*, pages 253–266, 2002.
[65] L. Silva, L. Silva, M. Mattoso, and V. Braganholo. On the performance of the position() XPath function. In *DocEng*, 2013.
[66] T. Silva, F. Baião, J. Sampaio, M. Mattoso, and V. Braganholo. Towards recommendations for horizontal XML fragmentation. *JIDM*, 4(1):27–36, 2013.
[67] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *CACM*, 53:64–71, 2010.
[68] D. Suciu. Distributed query evaluation on semistructured data. *ACM TODS*, 27(1):1–62, 2002.
[69] B. Surjanto, N. Ritter, and H. Loeser. XML content management based on object-relational database technology. In *WISE*, pages 70–79, 2000.
[70] I. Tatarinov, E. Viglas, K. Beyer, J. Shanmugasundaram, and E. Shekita. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.
[71] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.
[72] Z. Vagena, M. Moro, and V. Tsotras. Efficient processing of XML containment queries using partition-based schemes. In *IDEAS*, pages 161–170, 2004.
[73] M. Waldvogel, M. Kramis, and S. Graf. Distributing XML with focus on parallel evaluation. In *DBISP2P*, pages 55–67, 2008.
[74] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *ICDE*, pages 443–454, 2003.
[75] B. B. Yao, M. T. Özsu, and J. Keenleyside. XBench - a family of benchmarks for XML DBMSs. In *Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web-Revised Papers*, pages 162–164, 2003.
[76] M. Zhang and J. T. Yao. XML algebras for data mining. In *Data Mining and Knowledge Discovery: theory, tools and technology*, pages 209–217, 2004.
[77] X. Zhang, B. Pielech, and E. A. Rundesnteiner. Honey, i shrunk the XQuery!: an XML algebra optimization approach. In *WIDM*, pages 15–22, 2002.