Affinity-based XML Fragmentation

Rebeca Schroeder
Univ. Federal do Paraná
Curitiba, PR, Brazil
rebecas@inf.ufpr.br

Ronaldo dos S. Mello
Univ. Fed. de Santa Catarina
Florianópolis, SC, Brazil
ronaldo@inf.ufsc.br

Carmem S. Hara
Univ. Federal do Paraná
Curitiba, PR, Brazil
carmem@inf.ufpr.br

ABSTRACT

In this paper we tackle the fragmentation problem for highly distributed databases. In such an environment, a suitable fragmentation strategy may provide scalability and availability by minimizing distributed transactions. We propose an approach for XML fragmentation that takes as input both the application's expected workload and a storage threshold, and produces as output an XML fragmentation schema. Our workload-aware method aims to minimize the execution of distributed transactions by packing up related data in a small set of fragments. We present experiments that compare alternative fragmentation schemas, showing that the one produced by our technique provides a finer-grained result and better system throughput.

1. INTRODUCTION

Recently, it has been shown that the throughput of a system executing distributed transactions is significantly worse than executing them on a single node, even considering only short transactions [8]. Although this is a well-known problem, the dissemination of large-scale distributed systems has shown that the traditional solutions for data distribution, which involves both data fragmentation and allocation, are not suitable [17]. In this context, data distribution is a fundamental issue to provide a scalable and available database service [12]. Nevertheless, most cloud datastores do not provide any control over data placement and randomly allocate them among sites in order to provide load balancing and low maintenance costs.

In order to avoid distributed transactions, data items required by a transaction should be ideally located in the same fragment or in a set of few fragments. Indeed, we have shown [5, 15] that the performance of a distributed datastore has a direct correspondence with the number of data requests issued across the network. As an example, consider a database schema consisted of seven data items as illustrated in Figure 1. Here, nodes represent data items and edge values denote the frequency in which two nodes are accessed by the same transaction. Assume that the storage threshold is set such that each fragment can keep up to 4 nodes. Figure 1 shows two fragmentation schemas. The fragmentation represented as dashed lines keeps strongly connected nodes in the same fragment (*a*-*c*-*d* and *b*-*e*-*f*-*g*), while the one on dotted lines partitions data without considering the workload (fragments *a*-*b*-*c*-*d* and *e*-*f*-*g*). Assume that each fragment is allocated at distinct servers in the network. For the first fragmentation schema, a transaction involving data items *b*, *f* and *g* requires access to a single fragment. On the other hand, the second fragmentation schema results in a distributed transaction involving both servers.

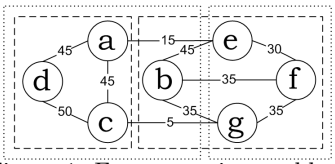
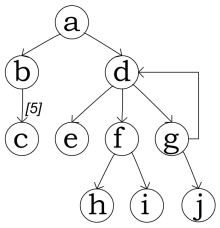


Figure 1: Fragmentation problem

In this paper we propose a workload-aware technique for XML fragmentation which minimizes the execution of distributed transactions. XML has been chosen because it is a flexible data model which supports several types of applications, including systems in the cloud [2, 13]. We consider a graphical notation to characterize the workload, similar to the one in Figure 1, where nodes denote XML elements and attributes, and edge values represent the frequency in which XML items are accessed together. We identify related XML items in the workload graph based on a technique originally proposed for the relational model [14]. However, as opposed to this work, which considers cycles to partition the input graph into fragments, we have chosen a storage threshold as the basis for our partitioning technique. This is because our goal is to minimize the execution of distributed transactions, packing up as much as possible in a given fragment size.

Some XML partitioning strategies have been proposed in the literature. Most of them focused on providing parallel query scans over large datasets through fragmentation [9, 10] and are based on traditional partitioning methods, where a fragment could be a document or a document subgraph. In this paper, we propose a finer-grained partitioning strategy to support more complex accesses over the schema, such as structural joins. This kind of access pattern characterizes *Online Transaction Processing* (OLTP). Our experimental study shows that, compared to alternative fragmentation schemas, the one produced by our technique presents better performance in terms of query response time and



(a)

T	a	b	c	d	e	f	g	h	i	j	f
t_1	1	1	1	1	1	0	0	0	0	0	120
t_2	0	0	0	0	0	1	0	1	1	0	112
t_3	1	1	1	1	1	0	0	0	1	0	11
t_4	1	0	0	1	0	1	1	1	0	1	27
t_5	0	0	0	0	0	0	1	0	0	1	15

(b)

Figure 2: XGraph and Workload

throughput. In the experiences, we considered a scenario in which XML data is available in a cloud datastore. Although some of the experimental results are particular for the cloud, the fragmentation technique proposed in this paper can be applied for any transactional application that requires distributed storage of XML data.

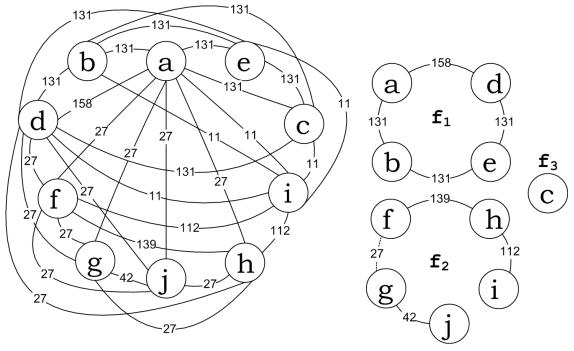
We present our technique considering fragmentation as a twofold problem. The first consists of workload characterization, that is, how to annotate an XML schema with workload information. This is the subject of Section 2. The second concerns data partitioning. In other words, how to minimize the number of fragments, while keeping related data in the same fragment. Our algorithm for fragmenting XML schemas, which is the main contribution of this paper, is presented in Section 3. Section 4 presents an experimental study on a cloud datastore, comparing our results with two others: the original technique proposed by Navathe et al[14] and *XS*, a close related fragmentation technique proposed for XML [6]. Section 5 is dedicated to related work and Section 6 concludes the paper by outlining future work.

2. WORKLOAD CHARACTERIZATION

In this section we present a method for representing workload information on an XML schema. It is based on the work introduced by Navathe et al[14], which has been proposed for the relational model, but here we consider the XML model and access language.

We start by defining a graph representation of XML Schema, containing both the structure of the document and the expected size of its instances. Given an XML Schema S, an *XGraph* is a directed cyclic graph defined as a tuple $G = (X, c, r, l, s, o)$, where (1) X is a set of nodes which consists of the set of elements and attributes in S; (2) c gives the schema hierarchical structure, where $c(x)$ is the set of nodes directly connected to x and there exists an edge (x, y) in G if y corresponds to a subelement or attribute of x in S; (3) r is the node for the root element; (4) l assigns a label to each node in X; (5) s is a function that assigns a size for nodes in X; and (6) o is a function that gives the expected number of occurrences for a subelement x within its parent. In order to allow schema-based reasoning, we assume $o(x)$ to represent the *average* number of occurrences for x.

Figure 2a shows an XGraph where nodes are named with their labels, a is the root node, and edges represent the parent-child relationships. The size of an attribute or text element node consists of the expected size of their values, while for a complex element, the size is given by the structure of its children, but not the actual values of its attributes and subelements. In the example, $s(d)$ consists of the storage size required for keeping a structure composed of e, f and g, while $s(e)$ consists of the expected size of values for e.



(a) (b)

Figure 3: Affinity Graph and Fragments

To simplify the discussion, in the examples of Figures 2 and 3 we consider that for any node n, $s(n) = 1$. Also, in the example, $o(c) = 5$ because c is a multi-valued subelement of b and the average number of occurrences of subelements c in an element b is 5. We assume that for the remaining nodes n in the example, $o(n) = 1$.

Given a representation of an XML Schema and expected size of its instances, we now turn to a definition of workload. We characterize a workload based on the frequency of transactions and the set of paths traversed in each of them. More specifically, a workload is a triple $W = (T, f, u)$, where T is a set of transactions, f defines the expected frequency of transactions in T, and u maps each transaction t to a set of nodes traversed by t. Following the notation introduced in [14], a workload may be represented as a usage matrix as depicted in Figure 2b. In the matrix, each transaction in T is represented as a row, and each data item (nodes in XGraph) as columns. If a given transaction t traverses a node n (that is, $n \in u(t)$), then $M[t, n] = 1$; otherwise, $M[t, n] = 0$.

As an example, consider that a transaction t_1 is expected to be executed 120 times in a given period of time involving instances of the XGraph depicted in Figure 2a, and that t_1 involves paths $a/b/c$ and $a/d/e$. Then, $f(t_1) = 120$, $u(t_1) = \{a, b, c, d, e\}$ and thus $M[t_1, a] = M[t_1, b] = M[t_1, c] = M[t_1, d] = M[t_1, e] = 1$, and $M[t_1, f] = M[t_1, g] = M[t_1, h] = M[t_1, i] = M[t_1, j] = 0$.

Given a workload W on schema graph G, we can now define the affinity of two nodes n_i and n_j as the frequency they are accessed together by any transaction in the workload. That is, the affinity function $aff(n_i, n_j)$ takes as input a set of transactions T and computes the sum of frequencies of transactions that involve both n_i and n_j. More formally, we define $T_{ij} = \{t \in T \mid \{n_i, n_j\} \subseteq u(t)\}$, and $aff(n_i, n_j) = \sum f(t),\ t \in T_{ij}$. As an example, consider the workload given in Figure 2b. The affinity between a and b consists the sum of frequencies of transactions t_1 and t_3. Thus, $aff(a, b) = f(t_1) + f(t_3) = 120 + 11 = 131$. Observe that the affinity function can be used to label edges in a complete graph involving all nodes of an XGraph, as depicted in Figure 3a. We refer to this graph as an *affinity graph* defined as a tuple $XF = (X, E, aff)$, where X is the set of nodes in the *XGraph* and E is a set of edges which relates two nodes n_i and n_j by an affinity value ($aff(n_i, n_j)$). It is worth noticing that both the *XGraph* and the *affinity graph* are built based on workload data given as input to the fragmentation process. Estimating workload information is outside the scope of this paper.

2.1 The Fragmentation Problem

Given the definitions of the previous section, we are now ready to state the fragmentation problem we are considering. Intuitively, our goal is to partition nodes of an XML schema, such that partitions contain as many correlated nodes as possible that can fit in a given storage size. That is, given $G = (X, c, r, l, s, o)$ and an affinity graph $XF = (X, E, aff)$, we are interested in obtaining a fragmentation schema $F = \{f_1, ..., f_n\}$, $n \geqslant 1$, such that $f_i \subseteq X$, $\bigcup_{i=1}^{n}(f_i) = X$, and for every $i \neq j$, $f_i \cap f_j = \emptyset$. Given that our fragmentation schema is based on a storage threshold Γ, we also need the notion of the size of a fragment $f_i \in F$, defined as follows. Let $f_i = \{v_1, \ldots, v_m\}$, the size of f_i is given by the sum of the expected number of occurrences of nodes multiplied by their sizes. That is, $size(f_i) = \sum_{k=1}^{m}(o(v_k) * s(v_k))$.

In order to formally state our problem, we need the notion of a strongly correlated set *scs* for a node in the affinity graph, defined as follows: $scs(n) = \{n' | aff(n, n') \geq aff(n', n'')$ for every node n'' directly connected to $n'\}$. Intuitively, *scs* determines which nodes have stronger affinity with n than with others in the graph. We denote by scs^+ the transitive closure of the *scs* relation.

We can now state our fragmentation problem: Find F such that the following conditions are satisfied:

1. $size(f_i) \leq \Gamma$ for every $f_i \in F$;
2. $size(f_i) > \Gamma - size(f_j)$, for every $f_i \neq f_j$;
3. if n_1 and n_2 are nodes in the same fragment then either
   - $n_2 \in scs^+(n_1)$ or
   - $n_2 \notin scs^+(n_1)$ and $size(scs^+(n_1)) + size(scs^+(n_2)) \leq \Gamma$.

The first condition defines that all fragments in F must fit in Γ and the last two minimize the number of fragments, while generating fragments that are related by affinity values higher than the values with nodes in other fragments. As an example, consider $\Gamma = 5$ and the affinity graph depicted in Figure 3a. The fragmentation in Figure 3b satisfies our conditions because (1) the size of fragments fits in the storage threshold, that is $size(f_1) = 4$ and $size(f_2) = size(f_3) = 5$; (2) there is no free space in fragments to allocate another fragment, e.g., $size(f_2) > \Gamma - size(f_1)$; and (3) the affinity between any node in f_1 with any node in f_2 is lower than the affinity between any pair of nodes in the same fragment, for example, $aff(a, f) < aff(f, i)$.

In short, the XML fragmentation problem is to find a fragmentation schema which optimizes the number of fragments generated while allocating the most related data items in the same fragment.

3. XML SCHEMA FRAGMENTATION

In this section we propose an algorithm for solving the XML fragmentation problem. Algorithm *xAFFrag* takes as input an XGraph G with information on element sizes (s) and number of occurrences (o), an affinity graph XF and a storage threshold Γ. The algorithm is divided in two major steps: the first step computes fragments basely solely on strongly connected sets; in the second step these initial fragments are merged if their sizes lie within Γ.

In order to compute the *scs*'s, we consider some auxiliary variables: *allNodes*, which is initially set to the nodes in the affinity graph, and used to maintain the nodes that have not been assigned to a fragment; *allEdges*, also initially set to the edges in the affinity graph, and from which elements are extracted whenever they are traversed to compose a strongly connected set. Nodes and edges in the current *scs frag* being computed are inserted in variables *fNodes* and *fEdges*, respectively, while its size is kept in *fSize*. Variable *border* consists of edges that can potentially be traversed from nodes in the fragment in order to compute the *scs* transitivity closure.

The first step of the algorithm (Lines 1 to 25) processes *allEdges* in descending order of affinity. Given an edge (n_1, n_b), the goal of this step is to compute $scs(n_1)$. However, since we have to consider the storage threshold Γ, before inserting new nodes in the fragment we check whether it is possible to do so within the size of Γ (Line 15). A fragment is generated after all edges (n_1, n_b) in *border* are processed as follows: n_b is only considered to be inserted in the current fragment *frag* if it is related with higher affinity to some element in *frag* than to any other outside the fragment (Lines 13-14). Observe that the candidates are processed in descending order of affinity in order to fill up the fragment with those with highest correlation. At the end of the first step, all nodes have been assigned to some fragment.

For the second step, observe that edges that have been traversed to create the initial fragments in Step 1 have been removed from *allEdges*. Thus, in order to determine whether fragments can be merged and keep those with higher affinity together, we keep processing *allEdges* in descending order of affinity (Lines 26 to 33). For each edge, we check whether it relates nodes from distinct fragments and if they can be combined into a single one without violating the storage threshold. If so, they are merged (Lines 28 to 32).

As an example, consider the affinity graph of Figure 3a and $\Gamma = 5$ as the input to our Algorithm. The first edge to be processed is the one with highest affinity (a, d). Node a is inserted into a fragment *frag*. To simplify the discussion, we consider that all nodes have size 1. Thus, at this point, $fSize = 1$, given that a has a single occurrence. Since this is below the threshold, we keep trying to insert new nodes to *frag* among those connected to a, which are kept in *border*. The one with highest affinity is d. Node d is inserted in *frag*, since it is not connected to any other node with higher affinity and its insertion in *frag* does not exceed the value of Γ. The same happens for inserting both nodes b and e into *frag*. However, the addition of c, which has a multiple occurrence of 5, would make the size of the fragment be set to 9. This would exceed the storage threshold and thus c is not inserted into *frag*. Also, nodes f, g, h, i, j are not inserted in the fragment, but for a distinct reason: their affinities are higher with nodes that are not in the current fragment a-b-d-e. A similar process creates two other fragments: f-h-i and g-j. However, they are merged by the second major step which generates the fragment f-g-h-i-j. Observe that node c generates a fragment by itself given that its expected size prevents it to be combined with any other node. Thus, the final fragmentation schema generated is $\{\{a, b, d, e\}, \{c\}, \{f, g, h, i, j\}\}$ as depicted in Figure 3b.

The strategy adopted in our algorithm is similar to the approach *MakePartition* proposed in [14] for generating a linearly connected tree from an affinity graph. However, they differ on the criteria for deciding when they should stop inserting nodes in a fragment: while *xAFFrag* is based on a storage threshold, *MakePartition* identifies affinity cycles.

Algorithm xAFFrag

Input: XGraph $G = (X, c, r, l, s, o)$, Affinity Graph $XF = (X, E, aff)$ and Γ
Output: F fragmentation schema

```
1  F ← {};
2  allNodes ← X;
3  allEdges ← E;
4  repeat
5      (n1, nb) ←edge in allEdges with highest affinity;
6      fNodes ← {n1};
7      fEdges ← {};
8      fSize ← o(n1).s(n1);
9      border ← {(n1, nb)|nb ∈ allNodes};
10     allNodes ← allNodes − {n1};
11     while fSize < Γ and border! = {} do
12         (n1, nb) ← extract edge from border with highest
           affinity, where n1 ∈ fNodes and nb ∉ fNodes;
13         nbEdges ← {(nb, n) ∈ allEdges|n ∈ allNodes};
14         if for all edges e ∈ nbEdges: aff(e) ≤ aff(n1, nb) then
15             if o(nb).s(nb) + fSize ≤ Γ then
16                 fNodes ← fNodes ∪ {nb};
17                 fEdges ← fEdges ∪ {(n1, nb)};
18                 boder ← border ∪ nbEdges;
19                 allNodes ← allNodes − {nb};
20             end
21         end
22     end
23     F ← F ∪ {fNodes};
24     allEdges ← allEdges − fEdges;
25 until allNodes == {};
26 repeat
27     (n1, n2) ← extract edge from allEdges with highest
       affinity;
28     if n1 is an element of f1 ∈ F, n2 is an element of f2 ∈ F
       andf1 ≠ f2 then
29         if Γ − size(f1) ≥ size(f2) then
30             F ← F − {f1} − {f2} ∪ {(f1 ∪ f2)};
31         end
32     end
33 until allEdges == {};
34 output F;
```

In *xAFFrag*, such affinity cycles may be split or extended according to a storage threshold. Therefore, our algorithm maximize storage, while keeping related data items in the same fragment.

4. EXPERIMENTAL STUDY

We have conducted an experimental study for determining the effect of our XML fragmentation approach on the performance of queries over a distributed datastore. Specifically, we compare *xAFFrag* with *MakePartition* algorithm proposed by Navathe et al [14] as well as with the *XS Partition* [6], a close related algorithm for fragmenting XML.

4.1 Experimental setting

We apply the XBench benchmark [1] which has been proposed for applications categorized as text-centric and that involve multiple documents. The benchmark provides an XML Schema that models part of the Springer digital library, a workload with 19 queries and a database generator that can be configured to output datasets containing a user-defined number of articles. Among the 19 queries defined for the benchmark, we have chosen 11, because the remaining ones (3, 4, 5, 7, 9, 10, 11 and 18) target at XQuery processors evaluation. Unfortunately, XBench does not provide the frequency for each query and we have adopted representative query frequencies for the workload on the Springer digital library. Detailed information on the experiments described in this paper can be found in [16].

Given the workload and size of nodes provided by XBench, we generated XML fragmented schemas according to algorithms *xAFFrag*, *MakePartition* and *XS*. From each schemata, a database was populated in a distributed datastore in order to evaluate the query performance and system throughput for each fragmentation approach. Scalaris[20] has been chosen as the transactional key-value cloud datastore in our experiments. Such datastore currently supports only in-memory persistence based on a DHT over a P2P network. Besides Scalaris servers, that compose the storage network, there are also Scalaris clients. Clients are responsible for processing queries by issuing requests to a server (using a simple `put-get-remove` DHT interface) in order to obtain the required data fragments. The communication between Scalaris clients and servers uses remote procedure calls over HTTP, while communication among Scalaris servers uses a native message passing layer in Erlang.

The experiments were carried out on Amazon EC2 instances. Eight large instances were allocated, each of them with 7.5 GB of memory and 4 EC2 compute units on a 64-bit platform. Each instance runs a Scalaris server and a set of Scalaris clients. The system was configured to keep 4 replicas for each data item stored.

4.2 Implementation

In order to store XML fragments in the distributed datastore, we have considered a simple and generic model based on `key-value` pairs. That is, given an XML fragmentation schema, we populate a datastore as follows: a `key-value` pair is generated for each fragment, such that in each pair the `key` contains a system-generated unique identifier and `value` contains the XML fragment itself.

```
(f1, <article id="1" lang="en">
       <prolog>
           <title>Brazil Beaches and Waterfalls</title>
           <keywords><hole>f1</hole></keywords>
           <authors>
                 <author>
                    <name>Brian Lawrenson</name>
                    <contact><hole>f3</hole></contact>
                 </author>
           </authors>
        </prolog>
        <body><hole>f2</hole></body>
        <epilog><hole>f4</hole></epilog>
      </article>)
```

Figure 4: Fragments stored as key-value pairs

As an example, consider the pair in Figure 4. It corresponds to a fragment identified by f_1, and a value containing an XBench *article* instance. Here, `key`s are simple fragment identifiers, f_1, f_2, f_3, f_4. In order to connect XML fragments, we use the idea of *holes* and *fillers* proposed by XFrag [7]. A *hole* represents a placeholder in a subtree, populated by another subtree, called *filler*. In the example, nodes *keywords*, *contact*, *body* and *epilog* refer to *holes* that are filled by elements stored in other fragments.

4.3 Experimental Results

The goal of the experiments reported in this section is to determine the effect our partitioning algorithm has on the system performance, and compare it with both *MakePartition* and *XS Partition*. The comparison is based on two metrics: rate of transactions/queries per second (TPS) and query response time in milliseconds.

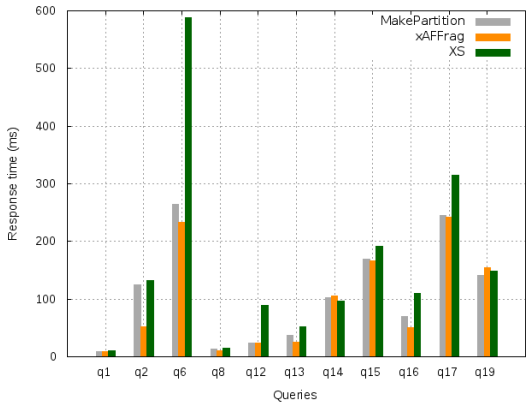


Figure 5: Query Performance - 160 clients on size 1

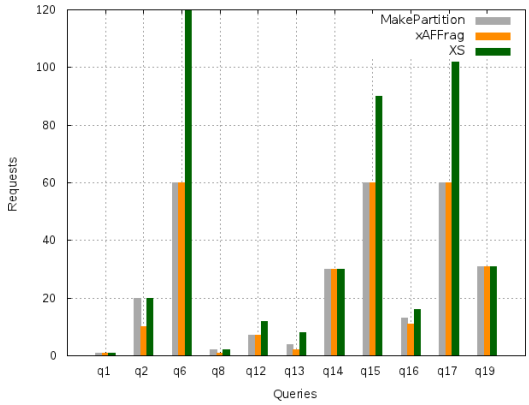


Figure 6: Query Requests

4.3.1 Query Response Time

For this experiment we generated an XBench dataset with 26 documents, and deployed 160 Scalaris clients. The average response time collected from 30 executions are presented in Figure 5. We have also collected the total number of requests Scalaris clients sent to servers, which also corresponds to the total number of fragments required to execute each query in the benchmark. Figure 6 presents the average value for 30 executions. As expected, these two graphs show that there is a direct correspondence between the number of requests to the datastore and the query response time. Indeed, observe that for executing q_6 on the *xAFFrag* generated database, we need 60 accesses to the datastore, which takes 233.47 ms. The execution of the same query on the *XS* generated database doubles the number of requests and has the same effect on the response time (588.63 ms).

The difference between the results reported for *xAFFrag* and *MakePartition* can be explained as follows. Although both are based on the affinity graph, they differ on their criteria for generating fragments. While *xAFFrag* is based on a storage threshold, *MakePartition* creates fragments based on cycles in the graph, regardless the number of data items or their size. Given that this approach does not focus on maximizing the storage capacity of the fragments, the number of fragments for a given dataset tends to be larger, which is what happened in this experiment. With smaller fragments, the number of requests to process a query is also likely to be larger, which reflects on the results of Figures 5 and 6.

In contrast, the number of fragments generated by *XS* and by *xAFFrag* are similar. Nevertheless, *XS* presents the worst performance among the three. The affinity among XML data is restricted to that related by parent-child, previous-sibling, and next-sibling relationships. Since *XS* does not consider the affinity of all sibling nodes, it may place strongly related nodes in distinct fragments. Consequently, it requires a higher number of requests to retrieve them.

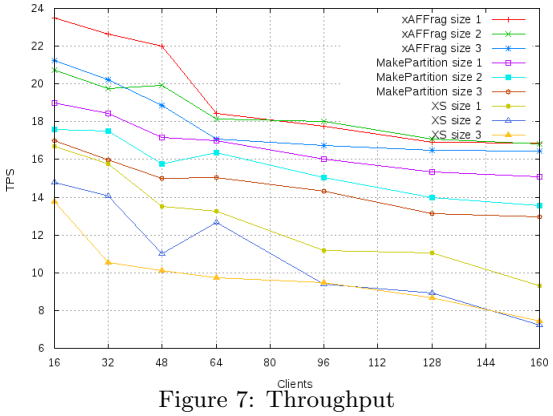


Figure 7: Throughput

4.3.2 System Throughput

In order to determine the impact of the three fragmentation approaches on the system throughput, we generated XBench datasets with 26, 266 and 466 XML documents. They are referred to as datasets of size 1, 2 and 3, respectively. For each size, XBench queries were executed with increasing number of Scalaris clients, ranging from 16 to 160 in increments of 16. The results are depicted in Figure 7. They present the average value of 30 executions for each combination of dataset size and number of clients.

For all fragmentation schemas, an increase on the database size generates a degradation on the system performance by roughly 2 queries per second. With respect to the number of clients, we observed that the degradation remains almost stable for *xAFFrag* and *MakePartition* when the number of clients is between 128 and 160. However, this is not the case for *XS* schema. We have noticed a similar behavior on additional experiences we have performed for assess the effect on query response time with increasing number of clients: while the response time remained almost constant for *xAFFrag* and *MakePartition*, for *XS* schema it almost doubles. We believe that the high number of requests generated by *XS* associated with a high number of clients increase the competition for resources, leading to CPU overhead and transaction conflicts, which impacts the system performance.

The experiments show that our maximum-storage affinity-based approach for fragmenting XML is effective to improve the performance of a distributed datastore, compared to previous proposals.

5. RELATED WORK

There are several works that tackle the problem of data fragmentation. Decades of relational optimizer research generated database design tools that provide optimizer extensions to execute *what-if* analysis in order to select suitable data fragments [4]. *What-if* analysis is still applied by recent works, like Shinobi [18]. In fact, *what-if* analysis is an effective way to identify workload trends. However, a lot

of information is required to support this analysis. Usually, such information is not available in the first stages of database design. Besides, fragmentation of distributed databases is known to be an NP-hard problem [11] and, therefore, heuristic-based approaches become more attractive.

Most of XML fragmentation approaches are limited to horizontal and vertical fragmentation techniques [3, 11]. In the XML model, horizontal fragments usually refer to sets of document instances and vertical fragments consist of subtrees of a document. Recent research efforts present heuristics for horizontal fragmentation of XML documents. Most of them consider Online Analytical Processing (OLAP) applications [9, 10] which exploit selection predicates in order to allow parallel query scans through fragmentation. Given the ad-hoc structure of these queries, they usually do not consider the workload. OLTP transactions, on the other hand, are well-structured and usually involve few data items. Instead of a parallel scan, these transactions require a fragmentation technique that provides low response time on queries by packing up data in a small set of fragments. To this end, the workload must be considered to provide a finer-grained fragmentation as presented in this paper.

Similar to our approach, the goal of Schism [8] is to avoid distributed transactions by keeping related data in the same fragment. While Schism targets the relational model and is based on instances, our technique targets XML and is based on schemas. We focus on a schema-based approach in order to avoid an exhaustive analysis process on instances of very large databases. To the best of our knowledge, the work most related to ours is the *XS* partitioning algorithm [6]. *XS* is an instance-based method to generate fragments from workload data. However, the approach to cluster related data is not entirely effective, given that related XML nodes may be placed in different fragments. Our experimental results show that *xAFFrag* is more effective in keeping related data in the same fragment, providing lower response time on queries than *XS*.

6. CONCLUSION

We have proposed an approach for partitioning XML schemas according to an application workload. Our technique focuses on analyzing the workload in order to identify nodes that must be kept together and arrange them in fragments according to a storage threshold. As the final result, transactions involving data items spread among a set of distributed servers are reduced. This work makes contributions in the context of highly distributed databases, where communication costs must be reduced to provide a scalable service.

We have conducted an experimental study based on the XBench benchmark. It shows that our approach to maximize the fragments' storage usage can improve the system throughput by roughly 22%, compared to the fragmentation provided by a traditional method[14]. Besides, our experiments show that *xAFFrag* can improve the performance by expressive 55%, compared to *XS* algorithm - a close related approach for XML fragmentation[6]. Although *xAFFrag* and *XS* have similar goals, the results show that the clustering strategy in *xAFFrag* is more effective.

There are several issues we plan to consider in the future. One of them is to extend the proposed approach for considering both data fragmentation and data locality issues based on the application workload. We also intend to investigate optimizations on the proposed algorithm by taking into consideration the structure of XML Schema and recent graph-based partitioning algorithms [19]. A related issue is metadata management [17], and a high-level definition language for specifying both data fragmentation and locality, which may be needed for fine tuning. Other issues that deserve further investigation include: indexing structures and query optimization strategies that consider the specificities of large distributed datastores.

7. ACKNOWLEDGMENTS

This work was partially supported by CNPq (Proc. 484366/2011-4-Ed.Universal) and by AWS in Education research grant award.

8. REFERENCES

[1] Xbench - a family of benchmarks for xml dbmss. Available at: http://se.uwaterloo.ca/ ddbms/projects/xbench/, 2012.

[2] 28msec. Sausalito: a scalable xml database designed for the cloud. Available at: http://www.28msec.com/, 2012.

[3] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. Xml processing in dht networks. In *Proceedings of the IEEE 24th ICDE*, pages 606–615, 2008.

[4] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of ACM SIGMOD*, pages 359–370, 2004.

[5] D. E. M. Arnaut, R. Schroeder, and C. S. Hara. Phoenix - A Relational Storage Component for the Cloud. In *4th IEEE CLOUD*, pages 684 – 691, 2011.

[6] R. Bordawekar and O. Shmueli. An algorithm for partitioning trees augmented with sibling edges. *Inf. Process. Lett.*, 108(3):136–142, Oct. 2008.

[7] S. Bose and L. Fegaras. XFrag: A Query Processing Framework for Fragmented XML Data. In *WebDB*, pages 97–102, 2005.

[8] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3:48–57, 2010.

[9] A. Cuzzocrea, J. Darmont, and H. Mahboubi. Fragmenting very large xml data warehouses via kmeans clustering algorithm. *Int. J. Bus. Intell. Data Min.*, 4:301–328, November 2009.

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

[11] P. Kling, M. T. Özsu, and K. Daudjee. Distributed xml query processing: Fragmentation, localization and pruning. Technical report, University of Waterloo, September 2010.

[12] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the SIGMOD '10*, pages 579–590, 2010.

[13] W.-S. Li, J. Yan, Y. Yan, and J. Zhang. Xbase: cloud-enabled information appliance for healthcare. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 675–680. ACM, 2010.

[14] S. Navathe and M. Ra. Vertical partitioning for database design: a graphical algorithm. *ACM SIGMOD International Conference on Management of Data*, 18:440–450, 1989.

[15] E. A. Ribas, R. Uba, A. P. Reinaldo, A. de Campos Jr., D. Arnaut, and C. Hara. Layering a dbms on a dht-based storage engine. *Journal of Information and Data Management*, 2(1):59–66, 2011.

[16] R. Schroeder, R. Mello, and C. Hara. Affinity-based xml fragmentation: web supplement. Available at: http://www.inf.ufpr.br/rebecas/xaffrag/, 2012.

[17] A. Tatarowicz, C. Curino, E. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *Inter. Conference on Data Engineering*, 2012.

[18] E. Wu and S. Madden. Partitioning Techniques for Fine-grained Indexing. In *ICDE*, 2011.

[19] J. Zhou, N. Bruno, and W. Lin. Advanced partitioning techniques for massively distributed computation. In *ACM SIGMOD International Conference on Management of Data(to appear)*, 2012.

[20] Zuse Institute Berlin and onScale solutions GmbH. Scalaris - distributed transactional key-value store. Available at http://code.google.com/p/scalaris/, 2012.