Distributed processing of XPath queries using MapReduce *

Matthew Damigos[1], Manolis Gergatsoulis[1], and Stathis Plitsos[2]

[1] Database and Information Systems Group (DBIS),
Department of Archives and Library Science, Ionian University, Corfu, Greece
mgdamig@gmail.com, manolis@ionio.gr.
[2] Department of Management Science and Technology,
Athens University of Economics and Business, Athens, Greece
stathisp@aueb.gr.

Abstract. In this paper we investigate the problem of efficiently evaluating XPath queries over large XML data stored in a distributed manner. We propose a MapReduce algorithm based on a query decomposition which computes all expected answers in one MapReduce step. The algorithm can be applied over large XML data which is given either as a single distributed document or as a collection of small XML documents.

1 Introduction

XML is a widespread format used for exchanging information on the Web, and, in general, for representing semi-structured data. The efficient querying and analysing large amount of web data is now being broadly recognized as a significant challenge in many areas, such as system designing, data analysis, decision-making, marketing and biology research. The management of the information appearing in a collection of XML data is achieved by using XML administrative languages such as XPath, XSLT and XQuery [6]. In this paper, we focus on XPath, which constitutes the basis for most of the other XML administrative languages. Furthermore, we use the MapReduce distributed framework [4] to process and manage large amount of XML data. MapReduce is widely used for processing large amount of data using a cluster of commodity machines. Boasting a simple, fault-tolerant and scalable paradigm, it has established itself as dominant in the area of massive data analysis.

In this paper we investigate the problem of evaluating XPath queries over large XML data which is stored in a cluster of commodity machines. The XML query evaluation in the MapReduce framework has been little investigated in the past. Query decomposition to sub-queries depending on the accessibility of data distributed to a fixed number of sites has been exploited in [7]. However, this

* This research was supported by the project "Handling Uncertainty in Data Intensive Applications", co-financed by the European Union (European Social Fund - ESF) and Greek national funds, through the Operational Program "Education and Lifelong Learning", under the research funding program THALES.

approach is agnostic to data distribution thus adhering to the distributed file system paradigm. Partial evaluation of XPath queries over a distributed XML document is also the subject of [3]. However, this approach assumes a sole coordinator entrusted with the joining of the partial results. In addition, the authors also propose a MapReduce algorithm which evaluates boolean queries. Note that this algorithm uses a single reduce task to compute the final answer. The problem of evaluating twig pattern queries on distributed XML data is investigated in [2], where the authors propose a system which computes the result based on indexing. In [5], MRQL, a query language over MapReduce, is introduced for the analysis of XML data. Finally, [9] explores the idea of a parallelized processing workflow of XML fragments in a MapReduce environment.

The main contribution of this paper is that it proposes a MapReduce algorithm, called HoX-MaRe, which computes all expected answers in one MapReduce step (Section 3.1). Our algorithm is based on query decomposition and a horizontal fragmentation method for the XML document and ensures that it computes all answers that would be resulted when the query is evaluated in a single machine. Preliminary experimental results concerning the performance of our algorithm are presented in Section 3.2.

2 Preliminaries

In this section we present the preliminary definitions of the concepts used in the subsequent sections. Consider a directed, rooted, labelled tree t (*tree* for short), where its labels come from an infinite set Σ. We denote $\mathcal{N}(t)$ and $\mathcal{E}(t)$ the set of nodes and edges, respectively, of t, and we write $label(n)$ to denote the label of a node n of t. We refer to the unique path through which n is reachable from root of t (denoted by $root(t)$) as *reachable path* of a node n in $\mathcal{N}(t)$. If there is an edge (n_1, n_2) in $\mathcal{E}(t)$, the node n_2 is a *child* of n_1. A node n'_2 of t is a *descendant* of a node n'_1 of t if t has a path from n'_1 to n'_2. A *branching node* is each node in $\mathcal{N}(t)$ having at least two children.

We consider two types of trees that represent XML documents and queries in XPath, respectively. An XML document is represented by a tree (also called *XML tree*) having text, or numbers, associated with leaf-nodes; while the XPath queries are different from XML trees in four aspects. First, the labels of a query come from the set $\Sigma \cup \{*\}$, where $*$ is the "wildcard" symbol. Second, a query P has two types of edges: $\mathcal{E}_{/}(P)$ is the set of child edges (represented by a single line) and $\mathcal{E}_{//}(P)$ is the set of descendant edges (represented by a double line). Third, a query P has an output node (or output, for short), denoted by $out(P)$, and is represented by a circled node. Fourth, each leaf-node which is not the output node may be associated with a condition; instead of text and numbers that are associated with leaf-nodes of an XML tree. The *selection path* of a non-boolean query Q is the path from the root to the output node. A *subquery* of Q is an XPath query having a subset of both the nodes and the edges of Q.

The result of applying a query Q on an XML tree t is based on a set of mappings from the nodes of Q to the nodes of t, called embeddings. An *em-*

bedding from Q to t is a mapping $e : \mathcal{N}(Q) \rightarrow \mathcal{N}(t)$ with the following properties:(1) Root preserving: $e(root(Q)) = root(t)$, (2) Label preserving: For all nodes $n \in \mathcal{N}(Q)$, either $label(n) = *$ or $label(n) = label(e(n))$, (3) Child preserving: For all edges $(n_1, n_2) \in \mathcal{E}_{/}(Q)$, we have that $(e(n_1), e(n_2)) \in \mathcal{E}(t)$, (4) Descendant preserving: For all edges $(n_1, n_2) \in \mathcal{E}_{//}(Q)$, the node $e(n_2)$ is a proper descendant of the node $e(n_1)$, and Leaf preserving: For each leaf-node $n_\ell \in \mathcal{N}(Q)$ associated with a condition either of the form "$text() = str$" or of the form "$val()$ *op num*" we have that either the text associated with $e(n_\ell)$ is identical to *str* or the number C associated with $e(n_\ell)$ satisfies the condition "C *op num*", respectively. We recall that *op* stands for one of the arithmetic comparison operators $=$, $\neq$,$<$,$>$, $\leq$,$\geq$, and *num* is a number. The result $Q(t)$, now, of applying a non-boolean query Q on a tree t is formally defined as follows: $Q(t) = \{e(out(Q))|e \textit{ is an embedding from } Q \textit{ to } t)\}$. If Q is a boolean query then the result $Q(t)$ is "*true*", only if there is an embedding from Q to t.

2.1 MapReduce framework

The MapReduce is the programming model for processing large datasets in a distributed manner. The storage layer for the MapReduce framework is a Distributed File System (DFS), such as the Hadoop Distributed File System (HDFS), and is characterized by the block size which is typically 16-128MB in most of DFSs. Creating a MapReduce job is straightforward. The user defines two functions, the *Map* and the *Reduce* function, which run in each cluster node, in isolation. The map function is applied on one or more files, in DFS, and results `<key,value>` pairs. This process is called *Map task*. The nodes that run the Map tasks are called *Mappers*. The *master controller* is responsible to route the pairs to the *Reducers* (i.e., the nodes that apply the reduce function on the pairs) such that all pairs with the same key initialize a single reduce process, called *reduce task*. The reduce tasks apply the reduce function in the input pairs and also result `<key,value>` pairs. This procedure describes a *MapReduce step*. Furthermore, the output of the reducer can be set as the input of a map function, which gives to the user the flexibility to create procedures of multiple steps.

2.2 Fragmentations of XML data

Considering an arbitrary method of attaching ids to element-nodes of an XML tree t we define, in this section, a fragmentation of an XML tree which preserves the structure of the initial tree. We say that a set $\mathcal{T}$ of XML trees (called *fragments*) forms a *horizontal fragmentation* of t if for each fragment $\mathcal{F}$ in $\mathcal{T}$ the following hold. (1) For each node $n_\mathcal{F}$ of $\mathcal{F}$, there is a node n of t having the same reachable path to that of $n_\mathcal{F}$, (2) for each node n of t, there is a node $n_\mathcal{F}$ of a document $\mathcal{F}$ in $\mathcal{T}$ having the same reachable path to that of n, and (3) each fragment in $\mathcal{T}$ contains at least one leaf-node of t, which is not contained in other fragment in $\mathcal{T}$. For example, in Fig. 1, the XML trees $\mathcal{F}_1$ and $\mathcal{F}_2$ represent the fragments of a horizontal fragmentation of the XML tree t. To verify this, notice that $\mathcal{F}_1$ and $\mathcal{F}_2$ are obtained by splitting t at the node m_7, and the path

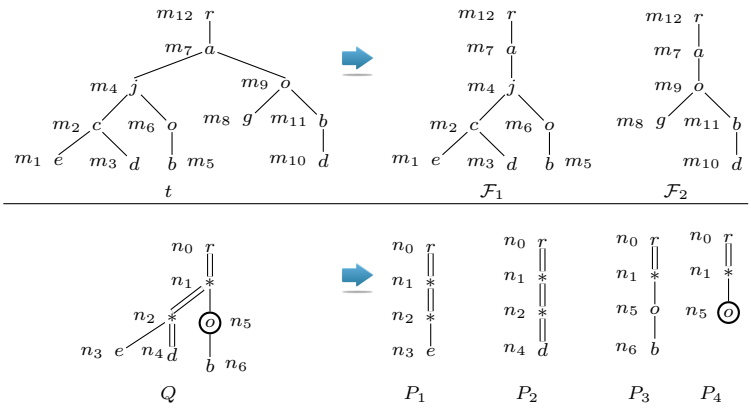


Fig. 1: (a)XML fragmentation, (b) XPath Query Decomposition

from the root of t to m_7 is included in both fragments. It is easy to see that the nodes of t keep their ids after the fragmentation. In this way the child-parent relationship, as well as the structure of t, are preserved in each fragment.

The horizontal fragmentation can be used to partition the information of an XML tree to several XML fragments. In the following, we consider that the size of each fragment does not exceed the maximum block size.

3 XPath Evaluation on MapReduce framework

The problem of efficiently evaluating XPath queries over a large amount of XML data using the MapReduce framework is formally stated as follows. Considering a distributed collection $\mathcal{T}$ of XML trees which form a horizontal fragmentation of a large XML tree t and an XPath query Q, we want to compute the answers that would be resulted by $Q(t)$, in parallel, using the MapReduce framework. In Section 3.1, we propose an algorithm, called $HoX - MaRe$ *Algorithm*, which deals with this problem in one MapReduce step.

Consider the horizontal fragmentation of t illustrated in Fig. 1. It is easy to verify that evaluating Q over each fragment ($\mathcal{F}_1$ and $\mathcal{F}_2$), in isolation, using one of the conventional methods of XPath evaluation, we get only the node m_6; i.e., we miss the node m_9. This node, however, should be included in $Q(t)$. Intuitively, the reason why the node m_9 is not resulted from the evaluation of Q on the fragments of t, is that the XML data needed to obtain the embedding giving m_9 is split in the two different fragments. To deal with this problem we define the concept of *query decomposition*. Let D_Q be the set of XPath queries obtained from an XPath query Q as follows. For each leaf node n of Q which is not output we add the reachable path of n to D_Q. Then we also add to D_Q the query consisting of the selection path of Q; which is the only non-boolean query in D_Q. The set D_Q gives the *decomposition* of Q.

Example 1. The XPath query Q illustrated in Fig. 1 has 3 leaf nodes; the n_3, n_4 and n_6. From these nodes we obtain the queries P_1, P_2 and P_3, respectively, and add them to a set D_Q. Notice that each query is given by the reachable path of each leaf node. In addition, we add to D_Q the selection path of Q, which is given by the query P_4. The set D_Q gives the decomposition of Q.

The following theorem describes how we can find an embedding from a XPath query Q to a tree t when we have already found a set of embeddings from the queries in the decomposition of Q to t.

Theorem 1. *Let Q be an XPath query in $\mathcal{XP}^{\{*,[\,],//,\wedge\}}$, t be an XML tree and $D_Q = \{P_1, \ldots P_n\}$ be the decomposition of Q such that P_n is the selection path of Q. Then for each node $o \in \mathcal{N}(t)$ the following are equivalent:*
(1)There is a set of embeddings $M = \{e_1, \ldots, e_n\}$ such that (a) for each i, with $1 \leq i \leq n$, e_i is an embedding from P_i to t, (b) for each $n \in \mathcal{N}(Q)$ there aren't two embedding e_i, $e_j \in M$ such that $e_i(n) \neq e_j(n)$, and (c) $e_n(out(P_n)) = o$.
(2) There is an embedding e from Q to t such that $e(out(Q)) = o$.

The Condition 1 in Theorem 1 can be easily extended to cover the case that each query $R_i \in D_Q$ maps on a fragment in a horizontal fragmentation $\mathcal{T}$ of t (instead of a mapping from P_i directly to t).

Corollary 1. *If we replace the Condition 1(a) in Theorem 1 with "(a') for each i, with $1 \leq i \leq n$, e_i is an embedding from P_i to a fragment $\mathcal{F}$, where $\mathcal{F}$ is contained in a horizontal fragmentation $\mathcal{T}$ of t", then Theorem 1 still holds.*

Corollary 1 implies a two-steps method for computing all nodes in $Q(t)$ in a distributed manner. Particularly, we firstly compute, in parallel, the embeddings from each query in D_Q to each fragment of $\mathcal{T}$, then these embeddings are emitted, along with the answers of the non-boolean queries, and in the second phase, the embeddings are combined properly in order to give the output nodes in $Q(t)$.

3.1 HoX-MaRe Algorithm

In this section, we present a MapReduce algorithm, called $HoX - MaRe$ Algorithm, which computes the answer of an XPath query Q when Q is posed on a distributed XML tree given by a horizontal fragmentation $\mathcal{T}$. The fragments in $\mathcal{T}$ are stored in a DFS. In the following we consider that the branching nodes of Q are mapped, using a bijection h, on an integer between 1 and $|\mathcal{N}_B(Q)|$, where $|\mathcal{N}_B(Q)|$ is the number of branching nodes of Q. In addition, we suppose that h has initially been sent to all mappers. The Map and the Reduce function of the algorithm are formally depicted in Fig. 2.

Map function: The Map function gets as input a fragment $\mathcal{F}$ in $\mathcal{T}$ and performs the following operations. Initially, the decomposition D_Q of Q is properly generated as described in previous paragraph. For each query P in D_Q we compute the set $M_{P,t}$ containing all embeddings from P to $\mathcal{F}$. Then for each embedding e in $M_{P,t}$ we create an array K_e, denoted as *embedding-array*, as

```
- Map: <XML fragment F, XPath query Q >
D_Q = getDecomposition(Q);//Return the decomposition of Q
N_B = getBranchingNodes(Q);//Return the branching nodes of Q. |N_B| is the size of N_B
n_DB = get1stbranchingnode(Q)//Return the first branching node of Q.
For each query P in D_Q do
  For each embedding e from P to F do
    K_e[j] = *;//Initialize the array K_e of the images of the branching nodes,
    of size |N_B| (i.e., j = 0, ..., |N_B| − 1).
      For each branching node n of P appearing in the position ℓ of N_B do
        K_e[ℓ] = e(n)
        If P is the selection path of Q then
          output ← < e(n_DB), [K_e, P, e(out(Q))] >
        else
          output ← < e(n_DB), [K_e, P, true] >

- Reduce:< Key K, Collection Values >
D_Q = getDecomposition(Q);
B = getBuckets(Values)
For each T in B
  For each [c1, c2, c3] in T
    If c2 is either the selection path of Q or the query Q
      output ← < K, c3 >
```

Fig. 2: HoX-MaRe Algorithm

follows. For each branching node n of Q which is included in P we put the node $e(n)$ to the position $h(n)$ of K_e, while we put the symbol "*" to each position of K_e which is not mapped by a node of P.

To define the key of each pair we distinguish the first branching node n_B of Q, traversing the selection path of Q from the root to the output. It is easy to verify that this node is included in each query in D_Q. Finally, for each embedding e, the map function outputs a key-value pair, where the key is $e(n_B)$, and the value is a triple of the form $[K_e,\ true,\ P]$, when the query P is not the selection path of Q or a triple of the form $[K_e,\ out(P),\ P]$, otherwise. Note here that if the input query does not have any branching node the key of each pair is given by null values; consequently all pairs are routed in a single reducer.

Example 2. Consider the query Q, the XML tree t, the horizontal fragmentation $\mathcal{T}$ of t and the decomposition D_Q of Q illustrated in Fig. 1. Notice that Q has 2 branching nodes; the n_1 and n_2. Running the map function of the HoX-MaRe algorithm over $\mathcal{T}$, two map tasks are performed; one on each fragment. Applying the map function on $\mathcal{F}_1$ we compute the embedding e_1 from P_1 to $\mathcal{F}_1$, where $e_1(n_0) = m_{12}$, $e_1(n_1) = m_7$, $e_1(n_2) = m_2$ and $e_1(n_4) = m_1$, as well as its embedding-array $K_{e_1} = [m_7, m_2]$. The embedding-array $K_{e_2} = [m_7, m_4]$ of the embedding e_2 from P_2 to $\mathcal{F}_1$ is computed similarly in the same task, while the second task computes the embedding-array $K_{e_4} = [m_7, *]$ of e_4 from P_4 to $\mathcal{F}_2$. For K_{e_1} and K_{e_2} the first task outputs the key-values pairs $< m_7, [K_{e_1},\ true,\ P_1] >$ and $< m_7, [K_{e_2},\ true,\ P_2] >$, while the second task outputs the pair $< m_7, [K_{e_4},\ m_9,\ P_4] >$ for e_4. We follow the same procedure for each embedding computed in each task.

Reduce Function: The reduce function performs over the input key-value pairs as follows. A function *getBuckets* is initially applied over the input values.

This function groups properly the triples included in the value of the input pair, based on the concept of *unification* of the embedding-arrays. We say that two embedding-arrays K_1, K_2 are *unifiable* if there is not any integer i such that $K_1[i] \neq K_2[i]$ and $K_1[i], K_2[i] \neq *$. In particular, the function *getBuckets* returns every set B (denoted *bucket*) containing tuples such that for each query P in the decomposition of Q there is a tuple in B which describes an embedding from P and for every two tuples in B their embedding-arrays are unifiable. Then, for each bucket B, the reducer locates the tuple obtained by the selection path and outputs the output of the selection path.

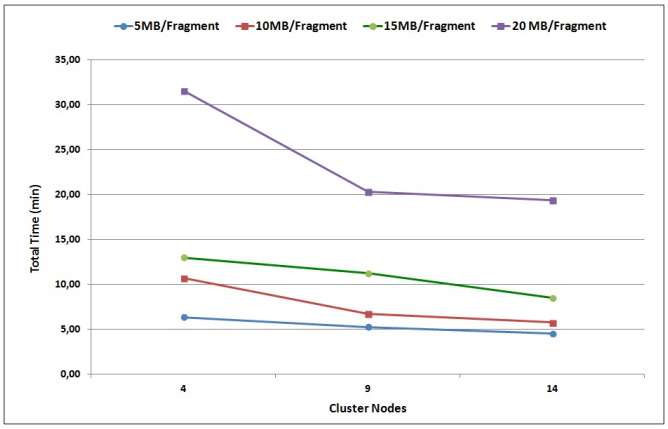
Example 3. Continuing the Example 2 and considering that all the key-value pairs have been emitted from the mappers, it is easy to verify that there is a reduce task which receives all the pairs having m_7 as a key. The reduce task initially calls the function *getBuckets*. Notice that there is not any bucket returned by *getBuckets* which contains both the values $[K_{e_1},\ true,\ P_1]$ and $[K_{e_2},\ true,\ P_2]$, since K_{e_1} and K_{e_2} are not unifiable (notice that $K_{e_1}[1] \neq K_{e_1}[2]$)). However, the values $[K_{e_1},\ true,\ P_1]$ and $[K_{e_4},\ m_9,\ P_4]$ will contained in a returned bucket. For this bucket, the reduce function will output the image of the output of P_4, which is given by the node m_9.

3.2 Preliminary Experimental Results

In this section, we present a set of preliminary experiments performed on a Hadoop cluster of 14 nodes of the following characteristics: Pentium(R) Dual-Core CPU E5700 @ 3.00GHz, 4GB RAM and 30GB available disk space. We run the HoX-MaRe algorithm (see Fig. 2) over an XML document of 1.5GB given by the XML generator of the XMark project[1]. We assigned to each node of the XML document a new attribute indicating a unique ID value and posed a set of XPath queries over several horizontal fragmentations, in terms of maximum fragment size, as well as we run the queries using 4, 9 and 14 nodes. Our results are summarized in the Fig. 3. where the table includes the average of the evaluation time, in min, of the queries, for each fragment-size and each number of cluster nodes. Note that, the time values depicted in this table correspond to the total time for evaluating queries. From the results of the table we conclude that the more nodes we use the faster we get the result of a query; which shows the load balancing feature of the algorithm. Furthermore, we can easily notice that the evaluation time is reduced when we use small XML fragments. This can be explained by the fact that our implementation has been built using a DOM package which requires loading of the whole XML document, in each mapper, into memory. In order to compare our approach with the evaluation of XPath queries in a single computer we tried to run queries using the DOM package. However this was not possible for XML files greater than 100MB.

4 Conclusions

In this paper we presented an algorithm for distributed XPath query evaluation based on MapReduce. Our preliminary experiments shows that the algorithm



Nodes	5MB/Fragment	10MB/Fragment	15MB/Fragment	20 MB/Fragment
4	6.33	10.67	13.00	31.50
9	5.25	6.75	11.25	20.33
14	4.50	5.75	8.50	19.33

Fig. 3: Preliminary Experimental Results

is scales well to large XML datasets. As future work, we plan to investigate heuristics in order to improve further the performance of our algorithm, and make use of hash function in order to improve load balancing of the algorithm.

References

1. XMark: An XML Benchmark Project. `http://www.xml-benchmark.org`.
2. 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.
3. G. Cong, W. Fan, A. Kementsietsidis, J. Li, and X. Liu. Partial evaluation for distributed XPath query processing and beyond. *ACM Trans. Database Syst.*, 37(4):32, 2012.
4. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
5. L. Fegaras, C. Li, U. Gupta, and J. Philip. XML query optimization in Map-Reduce. In *WebDB*, 2011.
6. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2008.
7. D. Suciu. Distributed query evaluation on semistructured data. *ACM Transactions on Database Systems*, 27:2002, 1997.
8. I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD Conference*, pages 204–215, 2002.
9. D. Zinn, S. Khler, S. Bowers, and B. Ludscher. Parallelizing XML processing pipelines via MapReduce. Technical report, 2009.