HadoopXML: A Suite for Parallel Processing of Massive XML Data with Multiple Twig Pattern Queries

Hyebong Choi[‡]
hbchoi@dbserver.kaist.ac.kr

Kyong-Ha Lee[‡]
bart7449@gmail.com

Soo-Hyong Kim[‡]
kimsh@dbserver.kaist.ac.kr

Yoon-Joon Lee[‡]
yoonjoon.lee@kaist.ac.kr

Bongki Moon[§]
bkmoon@cs.arizona.edu

[‡]Computer Science Dept., KAIST, Daejeon, 301-781, Korea
[§]Dept. of Computer Science, University of Arizona, Tucson, Arizona, 85721, USA

ABSTRACT

The volume of XML data is tremendous in many areas, but especially in data logging and scientific areas. XML data in the areas are accumulated over time as new data are continuously collected. It is a challenge to process massive XML data with multiple twig pattern queries given by multiple users in a timely manner. We showcase *HadoopXML*, a system that simultaneously processes many twig pattern queries for a massive volume of XML data with Hadoop. Specifically, HadoopXML provides an efficient way to process a single large XML file in parallel. It processes multiple twig pattern queries simultaneously with a shared input scan. Users do not need to iterate M/R jobs for each query. HadoopXML also saves many I/Os by enabling twig pattern queries to share their path solutions each other. Moreover, HadoopXML provides a sophisticated runtime load balancing scheme for fairly assigning multiple twig pattern joins across nodes. With synthetic and real world XML dataset, we demonstrate how efficiently HadoopXML processes many twig pattern queries in a *shared* and *balanced* way.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing*; D.1.3 [**Software**]: Programming technique—*concurrent programming*

General Terms

algorithms, experimentation, performance

Keywords

XML, parallel processing, query optimization, MapReduce

1. INTRODUCTION

XML is one of the most prominent data formats and many data have been produced and transformed into the format. Specifically, scientific data and log messages are often kept in the form of XML. Such XML data are large and also growing very quickly. For example, UniprotKB, which provides the collection of functional information on proteins, now hits more than 108GB a file [2]. Moreover, new elements and attributes are continuously appended to existing XML files as they are generated over time. In a typical scenario, users prepare their queries in advance even when XML data are not completely produced. This is akin to the background of XML pub/sub systems, but different in that the data is sometimes stored in a huge XML file in disk and the volume is continuously growing. This makes it difficult to process the data within XML pub/sub systems or single-site XML databases. It is because conventional XML pub/sub systems are mainly devised to consider a series of small-size XML documents and XML databases are not optimized for such a big XML file that also must be appended or even substituted by a new XML file frequently. Thus, it is prudent to process user queries over XML data with MapReduce [4].

To address this issue, we devise HadoopXML which provides facilities to efficiently process a massive volume of XML data in parallel. HadoopXML is a set of applications developed on the popular MapReduce framework, Hadoop [1]. Main features of HadoopXML are as follows: First, it provides an efficient means to process a massive volume of XML data in parallel. It partitions XML data into blocks with no loss of structural information. Second, HadoopXML processes multiple twig pattern queries simultaneously. There is no need to iterate M/R jobs for each query in a query set. Third, HadoopXML enables query processing tasks to share input scans and their intermediate results with each other. A path solution is shared by multiple twig pattern queries that contain the common path pattern. Moreover, it saves many I/Os by removing many redundant intermediate results as we substitute many path patterns that include `//`, $\star$ to distinct root-to-leaf paths. Lastly, HadoopXML provides a sophisticated runtime load balancing scheme for evenly distributing twig joins across nodes. The rest of this proposal is organized as follows. Section 2 describes our system architecture. Section 3 explains features of HadoopXML. Section 4 presents implementation details. Section 5 describes our demonstration scenarios.

2. SYSTEM ARCHITECTURE

HadoopXML processes XML data in 3 steps: preprocessing and 2 consecutive M/R jobs. In preprocessing step (shown in Fig. 1), XML data are partitioned into equal-sized blocks and then loaded into HDFS. Also, elements are labeled for the use in twig pattern joins and the labels are written in label blocks separate from XML blocks. In the stage, HadoopXML also decomposes a given set of queries into linear path patterns. Then, it builds an NFA-style query index and a table that holds mapping information between given queries and the decomposed path patterns.

In the 1st M/R job, the query index is loaded into each mapper via `distributedCache` mechanism in Hadoop beforehand. After that, mappers read XML blocks as SAX streams and filter out only the labels of elements matched with the decomposed path patterns. Then, reducers group the labels by `PathId` and count the number of labels for each `pathId`. The path solutions and size in-

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.
ACM 978-1-4503-1156-4/12/10.

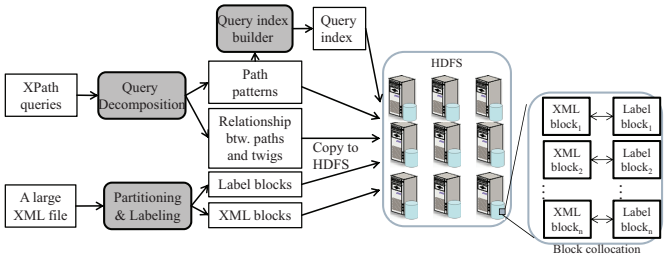


Figure 1: Preprocessing step in HadoopXML

formation are stored in HDFS. After that, our multi-query optimizer decides which reducer in the next M/R job will perform which twig pattern join for balancing workloads across nodes, based on size information for the path solutions and the mapping table.

In the 2nd M/R job, mappers read grouped path solutions and tag reduce ids to the grouped path solutions as keys. Since mapped outputs are shuffled by intermediate keys, path solutions tagged by the same reducer id go to the same reducer together. Finally, reducers perform twig pattern joins and output final results to HDFS. Fig. 2 illustrates data flows in two M/R jobs in HadoopXML.

3. FEATURES OF HADOOPXML

HadoopXML has many features for efficient XML data processing. Since fault-tolerance and scalability are its primary goal, Hadoop is not optimized for I/O efficiency [6]. Thus, we endeavor to increase I/O efficiency but without modification of Hadoop internals.

Partitioning with no loss of structural information
With labeled values, each label block records a root-to-leaf path that represents the structural information for the start of the corresponding XML block. For example, consider an XML document with four elements: `<a><b><c></c><d></d></b></a>`. If the XML file is partitioned into two blocks and the second block contains an XML fragment `</d></b></a>`, we keep a root-to-leaf path `/a/b/d` for the start of the block. When a map task reads the second block, a query index is first fed with the SAX stream restored from the root-to-leaf path string `/a/b/d` before actual block reading. This guarantees that a query index in each map task starts with correct internal states when processing XML blocks that lie in the "middle" of the SAX stream.

Collocating XML blocks and label blocks
HadoopXML reads both XML blocks and their corresponding label blocks during query processing. If two blocks are stored separately in two nodes, additional network I/Os occur as the system reads blocks via network, delaying map stage. To increase spatial locality, we extend block placement policy in HDFS so as to put an XML block and its corresponding label block together into the same node.

Multiple twig pattern matchings in parallel
In HadoopXML, multiple join operations are distributed across nodes and executed in parallel as many as the number of reducers. We also implement each join operation with an I/O optimal holistic twig pattern join algorithm for improving I/O efficiency in HadoopXML [3].

Sharing input scan and path solutions
MapReduce's batch nature makes it difficult to support ad-hoc queries like DBMS. To iterate the same M/R job from input scan to reduce stage for each query is wasteful in many cases. Moreover, many twig patten queries share linear path patterns with each other in practice. Sharing path solutions reduces redundant processing of path patterns and saves many I/Os [8]. In this respect, we borrow the concept of path sharing from YFilter [5]. Moreover, path solutions are shared by multiple twig pattern joins in HadoopXML. While joining path solutions for processing twig patterns, a group of join operations assigned to the same reducer share the path solutions each other if the path patterns are shared by the twig patterns. This helps reduce the overall I/O cost of join operations.

Converting to distinct path patterns
Many path patterns may be matched with a single root-to-leaf path in practice. For example, assume that path patterns `/a//c`, `//c` and `/a/*/c` are matched with a root-to-leaf path `/a/b/c` in an XML file. If the paths are treated as different each other, three path solutions are redundantly produced for a single distinct path during query processing. By converting redundant path patterns to root-to-leaf paths which are distinct in an input XML, we nicely reduce the sizes of path solutions and save many I/Os. In order to support this feature, HadoopXML extracts distinct root-to-leaf paths during data loading in preprocessing step.

Runtime load balancing and multi query optimization
A straggling task lags overall job execution in Mapreduce. This problem becomes more severe if it happens in reduce stage. MapReduce's native runtime scheduling does not work well especially for reducers. HadoopXML rather uses *dynamic shuffling* scheme that balances workloads across reducers at runtime. To achieve this, HadoopXML estimates the cost of each twig join operation before actual joining. It is achieved by computing the cost of each join operation, as the worst-case I/O and CPU time complexities of twigStack algorithm is linear in the sum of sizes of input path solutions. The sizes of path solutions is counted in the 1st reduce stage. The cost estimation also considers the sizes of path solutions shared by multiple twig pattern queries. Then, it assigns join operations into reducers at runtime such that every reducer has the same overall cost of join operations each other.

4. IMPLEMENTATION

We implemented HadoopXML with Hadoop version 0.21.0. Our cluster consists of 9 nodes, running on CentOS 6.2. A master is equipped with an AMD Athlon II x4 620 processor, 8GB memory and a 7200RPM HDD. The other nodes are designated as slaves, each of which has an Intel i5-2500k processor, 8GB memory and a 7200RPM HDD. All nodes are connected via Gigabit switching hub. We use default settings for our Hadoop cluster for fair comparison. Region numbering scheme [7] is used for labeling XML, but modified for the support of big XML files. Since *end* values in the numbering scheme are generated in postorder, labels are kept in memory until we meet `endElement()`, causing a memory space problem in such a big XML file. Our scheme reads an XML block, then promptly appends labels into the corresponding label block. After data loading, HadoopXML sorts labels by *start* in preorder. For path filtering, we use the NFA-style query index in YFilter [5]. We also use TwigStack algorithm [3] to implement holistic twig pattern joins in the 2nd M/R jobs, but other holistic join techniques can be used in HadoopXML with no loss of generality. Finally, we also extend `DataPlacementPolicy` class in HDFS in order to collocate XML blocks and their corresponding label blocks.

5. DEMONSTRATION SCENARIO

Table 1 presents statistics for XML datasets used in our experiments. The demonstration will use only a small fraction of one syn-

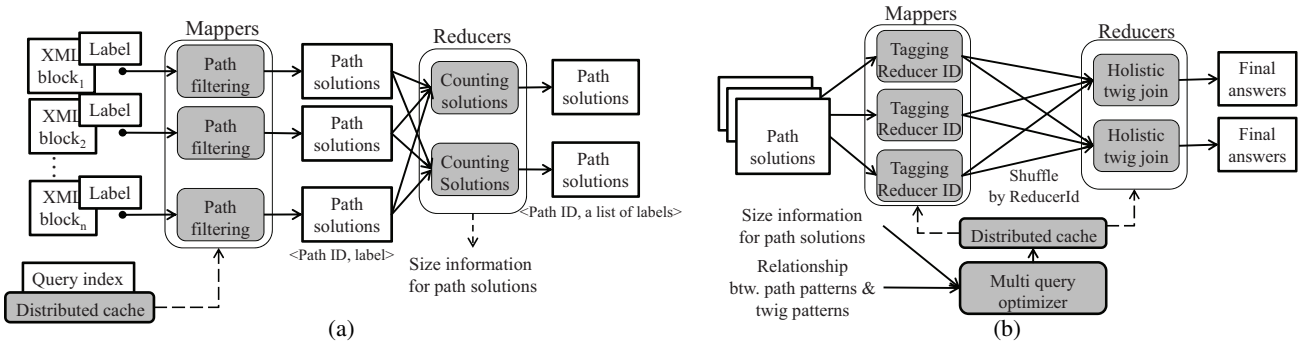
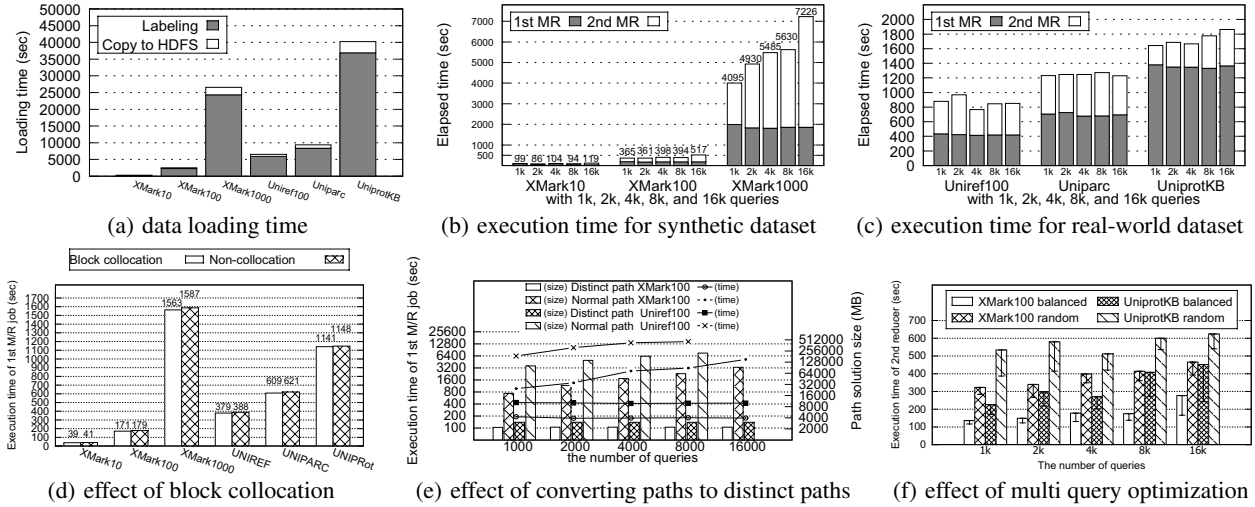


Figure 2: (a) path query processing in the 1st M/R job (b) twig pattern joins in the 2nd M/R job



(a) data loading time (b) execution time for synthetic dataset (c) execution time for real-world dataset

(d) effect of block collocation (e) effect of converting paths to distinct paths (f) effect of multi query optimization

Figure 3: Experimental results

Table 1: Statistics of XML dataset

Filename	UniRef100	UniParc	UniProtKB	XMark1000
File size(KB)	25,088,663	38,334,953	108,283,066	117,159,962
# of elements	335,153,446	360,376,852	2,110,330,358	1,670,594,672
# of attributes	589,568,839	1,215,063,103	383,127,024	2,783,354,175
Depth in avg.	4.5649	3.7753	4.3326	4.7375
Max depth	6	5	7	12
# distinct paths	30	24	149	548

thetic and one real-world data set due to the limited demonstration time and the nature of MPP(Massive Parallel Processing) applications. However, we still present our experimental results done with all the dataset in fig. 3. Currently, HadoopXML supports a subset of XPath 1.0 language, *i.e.* {/,//,*, @, []}.

In our demonstration, users will be given a list of sample XPath queries generated from DTDs for the datasets in Table 1. Users can also edit the queries with their tastes. Users are then allowed to load sample XML files into Hadoop XML and run their queries themselves. During the processing, users will be explained step by step with Hadoop GUI how the system processes a massive volume XML data. Users will also check how features of HadoopXML affect the overall performance of the system as they can turn on and off the features, *e.g.* block collocation, sharing input scan & path solutions, load balancing and so on.

Acknowledgments

We thank to Jiaheng Lu for providing us with Java version of twig join algorithms. This work was partly supported by NRF grant funded by the Korea government (MEST)(No. 2011-0016282).

6. REFERENCES

[1] Hadoop. *http://hadoop.apache.org*, Apache Software Foundation.

[2] A. Bairoch et al. The universal protein resource (uniprot). *Nucleic acids research*, 33(suppl 1):D154–D159, 2005.

[3] N. Bruno et al. Holistic twig joins: optimal xml pattern matching. In *Proceedings of ACM SIGMOD*, pages 310–321. ACM, 2002.

[4] J. Dean et al. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] Y. Diao et al. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Transactions on Database Systems*, 28(4):467–516, 2003.

[6] K. Lee et al. Parallel data processing with mapreduce: a survey. *ACM SIGMOD Record*, 40(4):11–20, 2012.

[7] Q. Li et al. Indexing and querying xml data for regular path expressions. In *Proceedings of VLDB*, pages 361–370, 2001.

[8] T. Nykiel et al. Mrshare: Sharing across multiple queries in mapreduce. *Proceedings of the VLDB Endowment*, 3(1-2):494–505, 2010.