Volume 5, Issue 7, July 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Parsing Large XES Files for Discovering Process Models: A Big Data Problem

Yosvanys Aponte Báez[*]
Agrarian University of Havana
San José de Las Lajas, Cuba

Alexander Sánchez Díaz
Agrarian University of Havana
San José de Las Lajas, Cuba

Manuel Marco Such
University of Alicante
Alicante, Spain

Abstract— Process mining is a group of techniques for retrieving de-facto models using system traces. Discovering algorithms can obtain mathematical models exploiting the information contained into list of events of activities. Completeness of the traces is relevant for the accuracy of the final results. Noiseless traces appear as an ideal scenario. The performance of the algorithms is significant reduce if the log files are not processed efficiently. XES is a logical model for process logs stored in data centric xml files. In real processes the sizes of the logs increase exponentially. Parsing XES files is presented as a big data problem in real scenarios with dense traces. Lazy parsers and DOM models are not enough appropriate in scenarios with large volumes of data. We discuss this problematic and how to use indexing techniques for retrieving useful information for process mining. An XES compression schema is also discussed for reducing the index construction time.

Keywords—Indexing, Big data, XES, Hadoop, Map Reduce

I. INTRODUCTION

XML is used to exchange information in a self-describing manner. This makes XML verbose thus introducing significant amount of redundancy. The resulting size has negative effect on both query processing and data exchange. The de-facto standard model XES [1] for data representation in process mining stores log information in XML files. This is the input of the discovering algorithms which efficiency is proportional to these files size. In practice, traditionally parsing techniques are not suitable for managing input traces. The real process is dense with large amount of process instances. Also, the most used information retrieval techniques require to be optimized for querying large structured data in xml files.

Process mining help organizations to discover, monitor and improve real processes, by extracting knowledge from event logs. The data is collected from all kind of systems and analysed to see the deviations from the standard procedures and where the bottlenecks are. This helps to improve processes.

In real scenarios (like big management system or banking system with millions of user traces) where anomalies should be predicted in a prudent period of time, parsing is a critical task. That is why process mining is related with big data area regarding the lack of adequate techniques for managing large XES files. We discuss this as a big data problem which solution is crucial for process mining model discovering algorithms. In the following it is presented the difficulties with structural indexes. In section III we explain briefly the main XES concepts and the compression scheme strategy. A map and reduce strategy are presented in section IV. In section V the experimental results over the compression schema strategy. And finally, in section VI appear conclusions and ongoing work.

II. STRUCTURED DATA INDEXING AND QUERYING

Different indexes approaches and queries structures have been proposed in the literature [2][3]. There are two principal types of structural indexes: structural summary and structural join. The structural summaries indexing methods [4][5][6] merge the same sub-structures in a XML document and obtain a smaller tree structure, which is used as the index of the XML document. On the other hand, the structural join strategies are based on the decomposition-matching-merging processes.

There are a lot of published papers about XML indexing and most of them are confined to small data examples running in centralized system [7][8]. The *PCIM* [7] (Path Clustering Indexing Method) clusters paths with the same root-to-leaf nodes and reduces the space cost of the index using two hash tables, the Structural Index and the Content Index, with tag names as hashing keys for efficient searching. The *PCIM* reduces the index space with a high compression ratio and efficiently process complex queries. But it has the following disadvantages: a long time index construction and the adoption of a regional numbering scheme. The *NCIM* [8] (Node Clustering Indexing Method) is another indexing scheme, which differs from the *PCIM* by clustering the nodes with the same tag names and storing them in hash tables. Showing a good compression ratio and supporting complex queries efficiently. A limitation is that they assume that the indexes can fit into main memory and it is an issue when dealing with large XML documents. Both the *PCIM* and *NCIM* encode nodes by means of regional numbering scheme. However, the *PCIM* uses strings to represent labels and the *NCIM* uses integers where possible. In most cases, the *NCIM* outperforms the *PCIM*, because the *PCIM* stores text content in the other tables, whereas the *NCIM* stores them in the leaf node index under the corresponding tag name, which results in reduced search time for processing queries with the selection predicates.

All these approaches and others suffer some difficulties. They require a huge size of memory for storing indexing structures, which could be bigger than the original XML document in some cases. In order to minimize the size of the index structure is necessary a critical execution time. Generally their performance is reduced when dealing with large XML documents.

In [9] we describe how a suffix array can be used as the data structure which stores the structural index in a retrieval system and provides a virtual index of all sub-paths in a digital collection. We also show how an auxiliary ternary search tree can accelerate the resolution of structural queries with only a marginal increase in memory usage. For the case of XES files having highly repetitive structure and small textual content, these structures tend to be very efficient in speed and storage. Otherwise in [10] is introduced a succinct data structure like Rank and Select to index the text content.

Suffix arrays were introduced as a simple, space efficient alternative to suffix trees. Using the lexicographical ordering, these suffixes will be grouped together in the suffix array and can be found efficiently with two binary searches. Otherwise ternary search trees run best when given several similar strings, especially when those strings share a common prefix. Also ternary search trees are effective when storing a large number of relatively short strings and the average-case running time for lookup and insertion is logarithmic.

As cloud computing becomes popular, the issues of parallel xml parsing have been discussed recently. However, to the best of our knowledge, there is a very little work that addresses the problem of indexing as well as querying XML documents on large distributed environments.

III. EVENT LOGS IN PROCESS MINING

XES is an XML based standard for event logs. It provides a structured format for interchange event log data between applications. Its structure is based on a well defined sequence of events and activities. It encapsulates a log as a sequence of process instances named traces. A trace is an ordered sequence of events. All of them are represented as XML elements. The attributes store properties like timestamps, activities, originators, descriptions, etc.

```
<global scope="event">
    <string key="name" value="">
    <string key"description" value="">
</global>
<trace>
    <string key="name" value="t1"/>
    <string key="description" value="trace one"/>
    <event>
        <string key="name" value="e1"/>
        <string key="description" value="event one"/>
    </event>
    <event>
        <string key="name" value="e2"/>
        <string key="description" value="event two"/>
    </event>
</trace>
```

Fig. 1 XES example

The XES format makes events comparable and configurable by introducing the concept of event classifiers. An event classifier assigns to each event an identity. It can be used by a parser to identify the types of queries that can be made. The log holds a list of global attributes for both traces and events. They are understood to be available and properly defined for each element described for traces and events. They are mandatory elements. In figure 1 appears an XES fragment using XML elements. We can look the repetitive definitions of events into a trace. Similar occurs with traces into a log. As we explain in the following section is possible to compress the original XES file reducing it to a smaller structure.

A. *XES compression schema*

We use the global attributes for grouping the similar tags elements and attributes for every trace and event. The data type could be used as a prefix in the attribute name. For instance, in the above example we can join name and description keys in the unique trace attribute *eventstringKeys*. In this case it should appear as a global parameter for traces.

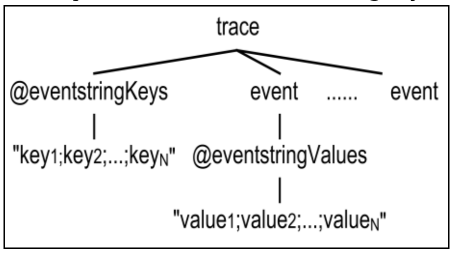


Fig. 2 Generic XES file

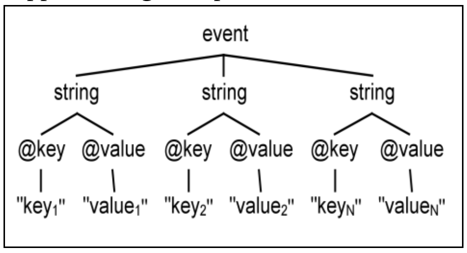


Fig. 3 Generic XES compressed file

In the global section are specified those string keys that are common for every event in all traces. The attributes having as a common value a string are joined. The resulting tree for constructing the index is reduced. The original sub tree for a unique event appears in figure 2. In figure 3 we can observe how the number of nodes for this event at the resulting tree is reduced when compared with the original. Similar analysis can be made for different elements data types.

IV. SCALED INDEX WITH HADOOP PLATFORM

Apache HADOOP[1] is an open-source software framework written in Java for distributed storage and distributed processing of very large data sets on computer clusters. HADOOP consists of two main services: high-performance parallel data processing using a technique called Map-Reduce and reliable data storage using the HADOOP Distributed File System (HDFS).

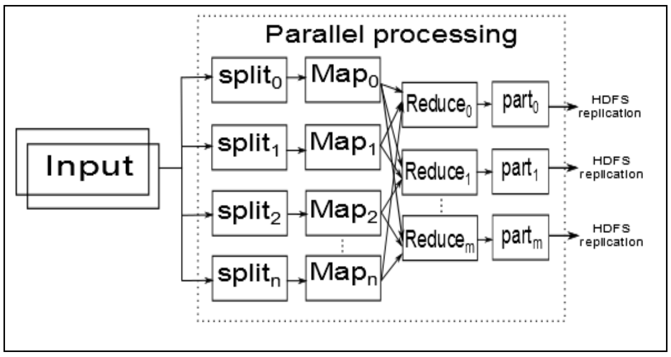


Fig. 4 The processing of Map-Reduce

Map-Reduce [11] is a model for parallel and distributed computation. As we can see in figure 4 it has two computation phases, map and reduce. In the map phase, an input is split into independent chunks which are distributed to the map tasks. The mappers implement compute-intensive task in a completely parallel manner. The output is of the form **<*key, value*>** pairs. The framework sorts the outputs of the mappers, which are then passed to the second phase, the reduce phase. Later the reducers process and sort the **<*key, value*>** pairs according to the ***key*** value and make the final output.

The HDFS is a distributed file system designed to store and process big data sets. Is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS has a master /slave architecture with a single Name Node and a number of Data Nodes. On the one hand the Name Node, the master of the HDFS, maintains the critical data structures of the entire file system. On the other hand the Data Nodes, usually one per node in the cluster, manage storage attached to the nodes that they run on. Inwardly, a file is split into one or more blocks that are stored in the Data Nodes with replications.

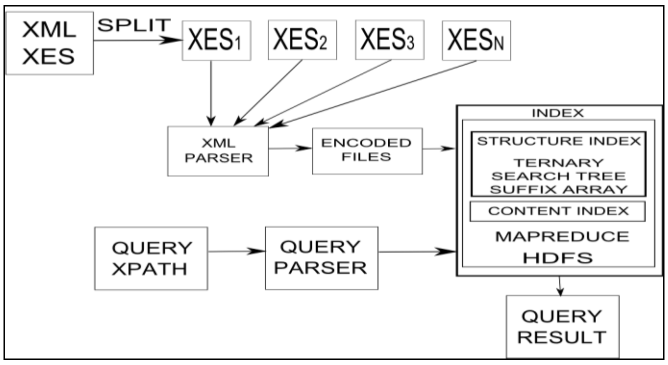


Fig. 5 Architecture of the proposed system

A. The proposed system

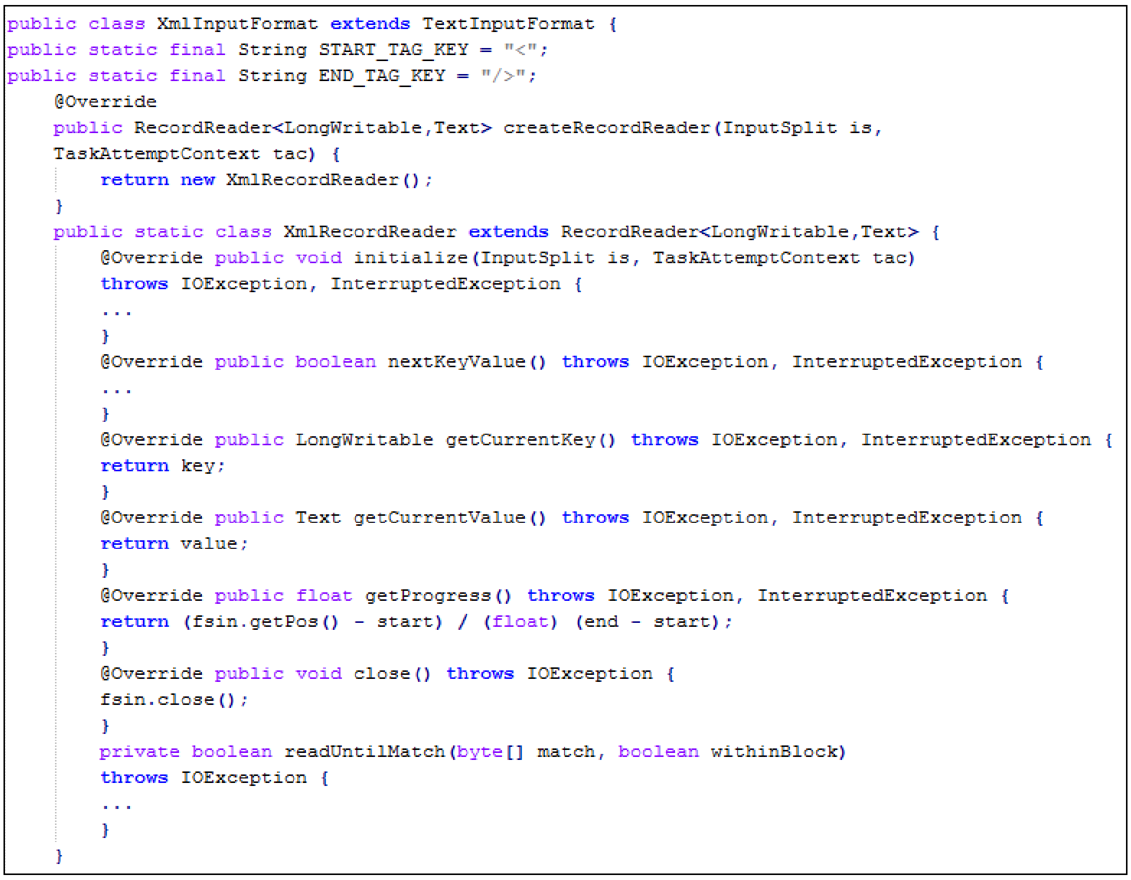
We scaled the system implemented in [9] using the Hadoop platform. The purpose is to implement an efficient solution for processing large scale XML files using Map-Reduce computing framework.

One important challenge is the decision of splitting policy. For this we propose an architecture that first of all split the XES files in multiples parts before indexing (see figure 5). The splits are then processed in parallel. The main idea of our XML reader is to read each element between *open* and *end* tags ("<" and "/>") as an input record in our Map-Reduce application. In our case, the *XmlInputFormat* class (see figure 6) reads records from files and defines a factory method for *RecordReader* implementations as shown.

The *RecordReader* implementation is where the input file data is read and parsed. This is implemented on the *xml RecordReader* class which make use of the *LineRecordReader* class. This class, is the implementation used by *TextInputFormat* to read the lines from files and return them. During this parsing phase each ***Map*** function produces a list of ***key, value*** pairs, where ***key*** is path from root to leave nodes and ***value*** is the set of intervals expanded by nodes that are reached following the labels in the path from the root to leaves. The ***Reduce*** function is then applied in parallel to

[1] https://hadoop.apache.org

each group, which in turn produces a collection of values in the same ***key***. For the flow index construction see figure 7. Later with these encoded files the structure index can be constructed with the suffix array and ternary search tree data structures.



```
public class XmlInputFormat extends TextInputFormat {
public static final String START_TAG_KEY = "<";
public static final String END_TAG_KEY = "/>";
    @Override
    public RecordReader<LongWritable,Text> createRecordReader(InputSplit is,
    TaskAttemptContext tac) {
        return new XmlRecordReader();
    }
    public static class XmlRecordReader extends RecordReader<LongWritable,Text> {
        @Override public void initialize(InputSplit is, TaskAttemptContext tac)
        throws IOException, InterruptedException {
        ...
        }
        @Override public boolean nextKeyValue() throws IOException, InterruptedException {
        ...
        }
        @Override public LongWritable getCurrentKey() throws IOException, InterruptedException {
        return key;
        }
        @Override public Text getCurrentValue() throws IOException, InterruptedException {
        return value;
        }
        @Override public float getProgress() throws IOException, InterruptedException {
        return (fsin.getPos() - start) / (float) (end - start);
        }
        @Override public void close() throws IOException {
        fsin.close();
        }
        private boolean readUntilMatch(byte[] match, boolean withinBlock)
        throws IOException {
        ...
        }
    }
```

Fig. 6 XML input format class

After building XML indexes, users may start sending two types of queries: parent-child query (P-C) and ancestor-descendant (A-D) query, for both types it can included content or not. For all queries first of all the Job Tracker decides how many map task are required and distributes these tasks to the chosen Task Tracker nodes. Due to the indexes are stored in HDFS, the Job Tracker schedules tasks on the nodes where data is present or nearby.

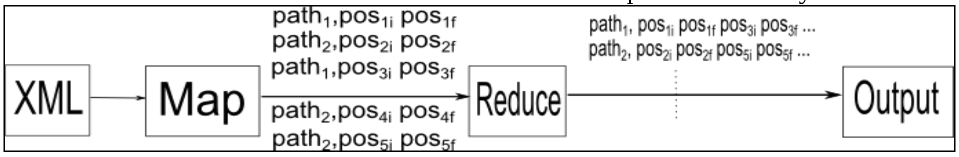


Fig. 7 The flow index construction

V. EXPERIMENTAL RESULTS

This section describes the experimental results for tests performed to check the compression strategies raised in section III-A. We performed the experiments on a Window 8.1 Operating System with an Intel(R) Celeron (R) CPU G1830 @ 2.80 Ghz with 4 Gigabytes of Ram.

First we design and implement a tool that generates the XES files, which allows varying different parameters such as number of traces, originators, activities, events by trace, among others. With this tool we generate a total of 12 XES files, creating two groups of six files: the first varying the traces amount and keeping in constant the other parameters and the second increasing the number of events by traces. The results can be seen in Tables I and II

TABLE I XES GENERATION FILES WITH THE VARIABLE TRACES BY EVENTS

Traces	Activities	Originators	Events by traces	Size (GB)	Compressed size (GB)
100000	100	100	110	3,38	2,04
100000	100	100	120	3,74	2,25
100000	100	100	130	4,07	2,45
100000	100	100	140	4,42	2,67
100000	100	100	150	4,75	2,87
100000	100	100	160	5,09	3,07

TABLE II XES GENERATION FILES WITH THE VARIABLE TRACES

Traces	Activities	Originators	Events by traces	Size (GB)	Compressed size (GB)
10000	100	100	1000	1,35	0,82
20000	100	100	1000	2,71	1,65
30000	100	100	1000	4,08	2,47
40000	100	100	1000	5,45	3,32
50000	100	100	1000	6,83	4,10
60000	100	100	1000	8,20	4,99

Having all XES files generated ready we implement a tool that compresses the XES scheme following the approach described in section 3.1. In the last column of each statistics table we show the sizes of the compressed schemes. Furthermore, we draw the graphs shown in figures 8 and 9, for all cases the graphic of the compressed size is always below the size to the original size, reaching an average compression within 60 %.

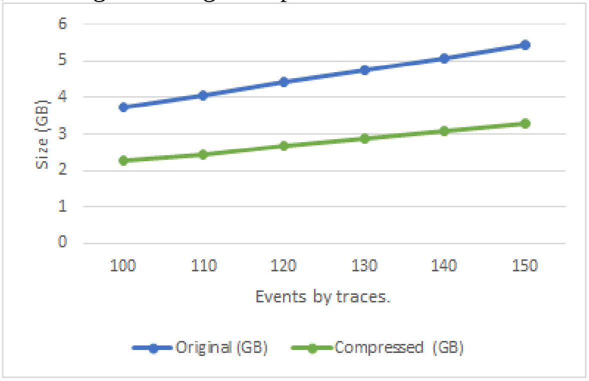


Fig. 8 Original and Compressed size (Table I)

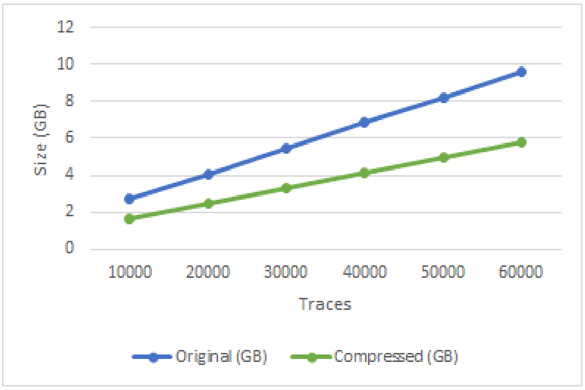


Fig. 9 Original and compressed size (Table II)

VI. CONCLUSIONS AND ONGOING WORK

In this paper, we proposed a system that builds indexes from large XES files using suffix array and ternary search tree for the document structure implemented in a Hadoop platform. We had explained how a ternary search tree, which indexes only the sub-paths ending in textual nodes, could accelerate the response. Also we presented an strategy to compress XES files, that can achieve average compression within 60%, in our case dealing with files with hundreds of thousands of traces heavily impacts in the efficiency of file analysis process and therefore in the index construction time. In addition we discussed how high performance computing could be a feasible technique for managing XES indexes. By this moment we are doing experimental tests with large XES files with different types of nodes and checking the index construction time and scalability factor following the architecture described in section IV-A.

REFERENCES

[1] H. Verbeek, J. C. Buijs, B. F. Van Dongen, and W. M. Van Der Aalst, "Xes, xesame, and prom 6," in Information Systems Evolution, pp. 60-75, Springer, 2011.

[2] S.-C. Haw and C.-S. Lee, "Data storage practices and query processing in xml databases: A survey," Knowledge-Based Systems, vol. 24, no. 8, pp. 1317-1340, 2011.

[3] A. Sayed, "Xml information retrieval systems:A survey," arXiv preprint arXiv:1111.6349, 2011.

[4] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon, "A fast index for semistructured data," in VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy (P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, eds.), pp. 341-350, 2001.

[5] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases," in VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece (M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, eds.), pp. 436-445, Morgan Kaufmann, 1997.

[6] T. Milo and D. Suciu, "Index structures for path expressions," in Proceedings of the 7th International Conference on Database Theory, ICDT '99, (London, UK, UK), pp. 277{295, Springer-Verlag, 1999.

[7] I.-E. Liao, W.-C. Hsu, and Y.-L. Chen, "An efficient indexing and compressing scheme for xml query processing," in Networked Digital Technologies -Second International Conference, NDT 2010, Prague, Czech Republic, July 7-9, 2010. Proceedings, Part I, pp. 70-84, 2010.

[8] W.-C. Hsu and I.-E. Liao, "Cis-x: A compacted indexing scheme for efficient query evaluation of xml documents," Inf. Sci., vol. 241, pp. 195-211, Aug. 2013.

[9] Y. A. Baez and R. C. C. Jimenez, "Indexing structured documents with suffix arrays," in 12th International Conference on Computational Science and Its Applications, ICCSA 2012, Salvador, Bahia, Brazil, June 18-21, 2012, pp. 43-48, 2012.

[10] Y. A. Baez , A. S. Diaz and M. M. Such "Optimized indexes for data structured retrieval," *International Journal of Advanced Research in Computer Science and Software Engineering,* Vol. 5, pp. 124–129, Abril. 2015.

[11] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation -Volume 6, OSDI'04, (Berkeley, CA, USA), pp. 10-10, USENIX Association, 2004.