Evaluation of XPath Queries Over XML Documents Using SparkSQL Framework

Radoslav Hricov, Adam Šenk, Petr Kroha, and Michal Valenta(✉)

Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic
{hricorad,senkadam,krohapet,valenta}@fit.cvut.cz

Abstract. In this contribution, we present our approach to querying XML document that is stored in a distributed system. The main goal of this paper is to describe how to use Spark SQL framework to implement a subset of expressions from XPath query language. Five different methods of our approach are introduced and compared, and by this, we also demonstrate the actual state of query optimization on Spark SQL platform. It may be taken as the next contribution of our paper. A subset of expressions from XPath query language (supported by the implemented methods) contains all XPath axes except the axes of attribute and namespace while predicates are not implemented in our prototype. We present our implemented system, data, measurements, tests, and results. The evaluated results support our belief that our method significantly decreases data transfers in the distributed system that occur during the query evaluation.

Keywords: Spark · SQL · XML · XPath · Big data

1 Introduction

Currently, XML is a very popular language for its platform independent way of storing data. The XPath query language is one of many possibilities to formulate queries over data stored in XML documents. Having big data stored in XML documents, the problem is how to retrieve data efficiently, i.e., how to implement the XPath query language for big data.

In this paper, we investigate the case in which a single unit of data, usually a file, can be processed by in-memory processing methods. This limitation is often met, because it is currently possible to use computers that have 128 GB operating memory.

The main topic of this paper is to describe how to use Apache Spark SQL framework to implement a subset of expressions from XPath query language under conditions described above. Apache Spark is a fast evolving engine for in-memory big data processing that powers several modules. One of the modules is Spark SQL. It works with structured data using SQL-like query language or domain-specific language of DataFrame. We investigate its possibilities and potential limitations.

XML documents can be processed by XSL technologies including XSLT, XPath, and XQuery. We choose XPath technology, as we worked with it successfully before [7,11]. We map XML data into the relational tables, and we map XPath queries to the SQL queries.

The paper is organized as follows. In Sect. 2, we present work related to our investigations. Concretely, we start with Spark SQL in Sect. 3.1. We explain why we need to transform XML documents into relational tables in Sect. 3.2.

Section 3.3 is dedicated to the investigated methods.

Our approach to query process analysis and the architecture of our system are given in Sect. 3. In Sect. 4, we describe the data we use and measurements we made, and we evaluate the results obtained. Finally, in Sect. 5, we draw conclusions and discuss possible future work.

2 Related Work

In this section, we evaluate work related to our paper. We focus on three inherently related topics: XML-to-relation mapping, Storing XML data in NoSQL databases and distributed evaluation of XPath and XQuery queries.

Starting with strategies for mapping XML to relations, various ways have been proposed in works such as [1,3].

On the other hand, the paper [10] introduces mapping methods that preserve XML node order.

In this work, several indexing methods are described, and comparisons of creating, updating, and reading cases are given.

Paper [8] introduces mapping of XML data to quasi-relational model. The authors propose a simple, yet efficient algorithm that translates XML data into structure savable in relational columns. Stored data can be queried by SQL syntax based language - SQLxD.

The following papers describe mapping into various NoSQL database systems. Specifically, the paper [9] introduces XML format mapping into a key-value store. Three possible ways of mapping are compared, however, query evaluation was not investigated. The paper [4] describes a distributed query engine used in Amazon Cloud. Three possible index strategies are introduced, and a subset of XPath and XQuery is implemented. However, the aim of the present work is to scale queries over a big set of XML documents. It does not investigate parallel processing of a single document. The implemented subset of XPath includes fewer operators than our solution.

Finally, distributed evaluation of XPath and XQuery language over a single document is investigated in [2]. The query engine introduced in this work can compute XML queries including only a small subset of XPath axis identifiers. Some of the papers published in recent years use MR framework for efficient XML query processing. The MRQueryLanguage is introduced in [6]. The query language is designed for querying distributed data, but it brings a new syntax different from known XML query languages like XPath and XQuery. HadoopXML

[5] is a suite enabling parallel processing of XPath queries. Queries are evaluated via twig join algorithm. Unfortunately, the XPath subset (that can be evaluated in HadoopXML) includes only root-to-leaf axis identities: child, and descendant or descendant-or-self. Evaluation of all XPath axes using MR framework is described in [11]. Authors focus on mapping complete XPath axes set to a bash of MR queries. The results are satisfying, but evaluation of some axes is highly inefficient.

3 Our Approach

First, we describe the Spark technology that we used. We focus both on the Spark core and on the SparkSQL framework. Then, we introduce the architecture of our system. In this part, we show how to map the XML data in data processable by SparkSQL, how to store it, how to translate the XPath queries and evaluate them, and how to reconstruct the results back to XML.

Finally, we describe our original approach to XPath query translation. We introduce five different methods that can be used for processing XPath queries using SparkSQL. In the last part of this section, we evaluated the methods and compared them.

3.1 Used Technology - Spark

Apache Spark is a multipurpose cluster computing system for a large-scale data processing. Spark is an open source engine originally developed by UC Berkley AMPLab and later adopted by Apache Software Foundation in 2010. Spark provides fast in-memory computing, and its ecosystem consists of higher-level combinable tools including Spark Streaming, Dataframes and SQL, as well as MLlib for machine learning, and GraphX for graph processing. The core engine of Spark provides scheduling, distributing, and monitoring of applications across the computing cluster. Spark is implemented in Scala that runs on Java Virtual Machine. API of Spark and its tools are available in Scala, Java, Python and R.

The Programming of distributed operations is based on RDD (abbr. Resilient Distributed Dataset). It is a Spark's main abstraction. It is a collection of objects that can be processed in parallel.

In our work, we use the SparkSQL module. This module enables to query RDDs using SQL-like syntax, so even programmers not familiar with Spark API can query data in parallel using Spark. However, SparkSQL implements only a subset of SQL language, which brings severe limitations.

3.2 Transformation of XML Document to Data Frames

The main programming abstraction of SparkSQL is DataFrame. This is a distributed collection that is similar to the concept of relational table. The tree structure of XML document is an ordered data model based upon the order of each element within the XML document. We mainly focus on the selection of nodes, and we want to be able to reconstruct selected nodes back to the valid and ordered XML. Accordingly, we are not interested in insertion or deletion of nodes. The transformation of XML document must be able to transform data from an unordered relational model back to the XML document. Hence, we decided to follow the paper [10], since it shows that XML ordered data model can indeed be efficiently supported by a relational database system. This is accomplished by encoding the data order as a data value. There are three methods for transformation of XML documents in tables [1] global order encoding, local order encoding, and Dewey order encoding.

From the three encodings mentioned above, the Dewey order encoding will be used here, since it is the universal solution, and the information stored in Dewey path is sufficient. Compared with the global encoding, it can be a bit slower (depending upon the comparison of the paths). Dewey path implicitly contains information about the node's ancestor nodes and also about its position among the siblings. Thus, Dewey encoding is the best option for our purposes.

The process of transformation begins with a numbering of elements and text nodes. Based on the pre-order traversal of the XML tree, a Dewey path is assigned to each node.

In the second phase of the transformation, the Dewey paths are recomputed to preserve the document order information, and it also makes the paths comparable as *String*. Now, each part (parts are separated by dots) of Dewey paths has the same length. The number of zeros in its prefix depends upon the number of digits of the highest value of Dewey path part among all Dewey paths. This is also helpful in SQL ORDER BY operation. Additionally, during the first phase, i.e. during the numbering phase, the paths to the certain nodes built from the names of nodes are created and assigned to each stored node.

A file containing serialized, transformed XML document is stored on a disk. When we query the document, it has to be loaded into memory and serialized back to Data Frame. We use Spark core to create RDD of XML nodes. Each row is read and split to the Node object. Node object is then passed to RDD. DataFrame may be created directly from RDD, but it is necessary to define a schema of a table. We created a class Node, and by reflection, the schema defined through Node class was applied on the RDD. Finally, a DataFrame was created from the RDD.

We experimented with datasets shown in Table 1. See Sects. 3.3 and 4 for results.

Table 1. Size comparison of generated tables

	XML file	Text file	Number of rows
books.xml	1.1 kB	1.4 kB	60
nasa.xml	25.1 MB	45.8 MB	791 922
proteins.xml	716.9 MB	2.3 GB	37 260 927

To make it easier for the reader to follow the discussion, let us show a sample of an XML file translated into a (relational) table.

```
+--------------+------+----+--------------------+
|         dewey|pathId|type|               value|
+--------------+------+----+--------------------+
|         00.01|     0|   1|           bookstore|
|      00.01.01|     1|   1|                book|
|   00.01.01.01|     2|   1|               title|
|00.01.01.01.01|     3|   3|   XQuery Kick Start|
|           ...|   ...| ...|                 ...|
+--------------+------+----+--------------------+
```

Fig. 1. Nodes' table of transformed XML

Optionally, an additional table containing the `pathId` and `Path` columns may be generated. It may be helpful for a particular query evaluation, but in principle, the table in Fig. 1 is enough. The column `type` contains the node type according to the W3C classification.

3.3 XPath Queries Evaluation

In this section, we describe our approach to processing of XPath queries. We have an XPath query and that, subsequently, must be translated into the SQL query that can be evaluated in the SQL module of Spark. According to the description given above, we developed two applications. The first one is an XML processor: it transforms an XML document into a relational table, and we introduced it in the previous section. The second one is a driver program for Spark: it processes an XPath query (using SQL query or via Spark SQL API), and it applies it on DataFrame built from the transformed XML document. The result of the driver program operation is the final table of nodes; it may be further processed.

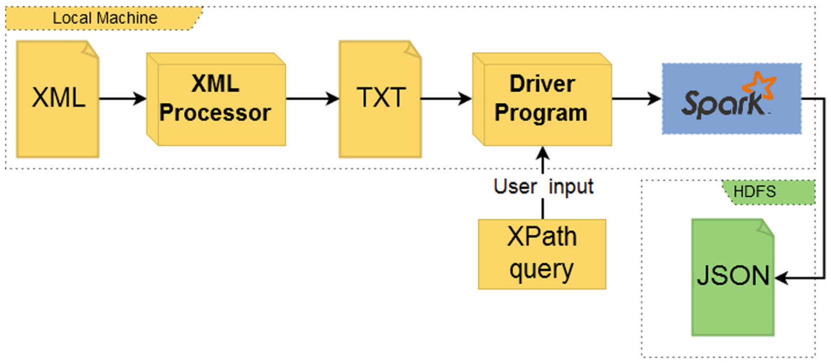

Fig. 2. Local cooperation of applications

Figure 2 shows how the applications locally cooperate to return XPath query result. The transformed XML document is stored as a text file. Both the text file and the XPath query are used as parameters of a driver program that is then running on Spark. The result of evaluation of a XPath query by the driver program is a JSON file stored in HDFS.

We implemented a simple XPath parser that parses XPath queries that were inputted as parameters of the driver program. A support for abbreviated forms of some XPath steps was added to the parser. In this prototype application, the abbreviated forms of child axis as `/` and descendant as `//` are allowed. Further, the wildcard `*` is an alias for any element node, which may be used in the queries that are parsed by a parser.

The whole query is split to the separated XPath steps, and the abbreviated forms are resolved. Then, all steps are evaluated step by step according to the desired axis. The step by step evaluation is implemented by an indirect recursive algorithm. It means that every next step is dependent upon the result of the previously evaluated step. This parser is working just with axes that were mentioned above and does not support predicates.

In following paragraphs, we describe five translation methods we investigated:

- Pure SQL method
- Join-based SQL method
- SQL query via DataFrame API
- Left semi join method
- Broadcasted lookup collection.

First, we introduce a native, trivial method, and then it will be improved in the next subsections. The next two methods of use apply the SQL queries to evaluate XPath queries. The other methods use a domain specific language of Spark SQL API.

Pure SQL Method. In the early familiarization with the Spark SQL module, we tried to directly translate an XPath query to the SQL query. For a faster local testing, we were working with a small table of nodes containing 60 rows. In all our tests, we performed translations of simple XPath queries that covered all XPath axes.

The generated SQL query starts with a selection of an auxiliary node that represents a parent of the root node. It is an alternative to a *document* statement `doc(‘‘xmlFile.xml’’)` in XPath. Then, the inputted XPath query is translated step by step. After the translation of the last step, one more selection and filtration are needed. It completes the result of query by selecting all descendant or self nodes of previously selected nodes. It is because the XPath steps traverse through the nodes, so by the last extra step, their content is appended.

Let us show a basic example to illustrate our approach and to support better understanding of following improvements. The example implements an XPath expression `//book/author/`:

```
SELECT p2.dewey, p2.pathId, p2.type, p2.value FROM nodes p2,
 (SELECT p1.dewey, p1.pathId, p1.type, p1.value FROM nodes p1,
  (SELECT p0.dewey, p0.pathId, p0.type, p0.value FROM nodes p0,
   (SELECT '0' as dewey) n0
  WHERE p0.type=1 AND n0.dewey <= p0.dewey
                  AND p0.value='book'
                  AND isPrefix(n0.dewey, p0.dewey) ) n1
 WHERE p1.type=1 AND n1.dewey < p1.dewey
                 AND p1.value='author'
                 AND isChild(n1.dewey, p1.dewey) ) n2
WHERE n2.dewey <= p2.dewey AND isPrefix(n2.dewey, p2.dewey)
```

We start with the parent of document root, i.e., `doc(''xmlFile.xml'')` – label `n0`, followed by its descendant nodes named `book`, etc. Functions `isChild` and `isPrefix` are based on Dewey encoding string, and their meaning is obvious.

Using our small testing file, it was relatively fast to compute a result; however, when we started with processing larger table of nodes (containing 791922 rows), problems with performance occurred.

After we examined an execution plan, we found out that for this naive method, it was actually the Cartesian product followed by filtration that was executed. It was the bottleneck of this method.

Remember that details of the following improvements can be found in [7].

We cannot discuss the problem fully given the limited space.

Join-Based SQL. In this case, the best solution is to avoid Cartesian product and apply an SQL JOIN clause instead. Hence, the JOIN ON conditions were defined to join results of a single XPath step. The idea is to select nodes that are candidates for the next context node, then combine them with the current context node and, based upon the relation, filtrate suitable nodes from joined pairs by using user defined functions. Note that the context node is a set of nodes returned by executing one step of XPath query. We use this term in the next sections.

By analyzing Spark's execution plans, we finally decided for RIGHT JOIN (LEFT JOIN is also acceptable, but it depends upon the order in which XPaths steps are joined). Although the type of JOIN was defined, Spark has generated Cartesian product in some cases because joins conditions were not strong enough. The conditions were based on non-equality of Dewey paths, and the user defined functions were used in a filter condition. To solve this, the join condition had to be enhanced, so instead of filtering based on a user defined function, we add required UDF into the join condition. For the sake of completeness, let us add that a JOIN, whose condition is based only upon the user defined function (that requires arguments both from left and right tables), invokes the Cartesian product: all pairs must be processed by UDF.

After the changes were done, the performance was admittedly better than the performance of the Cartesian product using method.

We compare the two previously discussed methods in Table 2.

As we can see, the usage of OUTER JOIN and the proper definition of JOIN conditions have a crucial impact upon the performance. After twenty minutes

Table 2. Performance of translated queries - Cartesian product versus Right JOIN

	Table	Method	Query	Time [s]
1	Books	Cart. prod	//book/author	9.790
2	Books	RIGHT JOIN	//book/author	8.372
3	Nasa	Cart. prod	//author/suffix	1200*
4	Nasa	RIGHT JOIN	//author/suffix	232.695

of computing, we were forced to cancel the third measurement marked with asterisk. We realized that Cartesian product in Spark is really slow.

SQL Query via DataFrame API. So far, we have worked only with pure SQL queries. However, DataFrame contains its own API that may be used to obtain the same results as by using SQL queries. We rewrote the previous SQL query that used RIGHT JOIN by calling a certain combination of functions from API. Using API, we changed the order of processed axes, so instead of RIGHT JOIN, the LEFT JOIN was applied.

Since we know that SQL and DataFrame shared the same optimization pipeline, the physical plans vary in small details - depending upon the implementation - and they actually do the same work. Broadcast Nested Loop Join is realized in Spark for OUTER JOINs. It compares the sizes of tables to be joined, and it broadcasts the smaller one across the workers. As it was expected, the durations of the computations of SQL and DataFrame are almost the same, since the optimizer generates the same physical plan.

Alternative Methods Without Joins. After the previous findings, we decided to restrict the usage of JOIN in further experiments. Several alternatives - such as nested queries, SQL IN operator or SQL UNION statement - had been tested, but the results were not satisfiable.

Instead, we decided to avoid joins altogether. To simplify the previous methods, we wanted to select those nodes of some XPath step that are in desired relation with at least one node from the nodes of previously evaluated XPath step. For this purpose, the best option is to use IN operator.

We wrote a user defined function *Parent()* that cuts the last part of inputted Dewey path, and since Dewey path contains information about all ancestors, this function returns the Dewey path of its parent node. This is the valid SQL query and both - the query and the nested query - are executable, though, as it turned out, Spark does not think the same.

Unfortunately, Spark SQL is not able to execute nested SELECT following the WHERE clause. Using Spark SQL API is not applicable either: Spark evaluates it in a different way - as expected.

Left Semi Join. As mentioned above, the IN clause may be used just with a joined table. We realized that Spark SQL provides LEFT SEMI JOIN - and it turned out to be more effective than all of our previous attempts. First, let us explain how LEFT SEMI JOIN works.

"Semi" means that the result contains just rows returned from one table. In case of LEFT SEMI JOIN, just the rows from the left table are returned. LEFT SEMI JOIN is based upon the existence of records in the right table. It means that if there is a record in the right table that fulfills the JOIN ON condition, just this one record from the left table is returned.

Using this method, we implemented a translation of XPath steps for parent, child, ancestor, ancestor-and-self, descendant, and descendant-or-self axes. Other axes have to be implemented in a different manner, a one that does not use LEFT SEMI JOIN, but it uses, for example, user defined functions (UDF). It is because the implemented axes are based on prefixes.

Table 3 compares the computation times using SQL and DataFrame.

Table 3. SQL versus DataFrame

	Table	Method	Query	Time [s]
1	Books	DataFrame	//book/author	8.313
2	Books	SQL	//book/author	8.372
3	Nasa	DataFrame	//author/suffix	231.989
4	Nasa	SQL	//author/suffix	232.695
5	Nasa	DataFrame	//suffix	219.012
6	Nasa	SQL	//suffix	217.818

Broadcasted Lookup Collection. Concerning the JOIN statement, in Spark tutorials, it is recommended to set a table that is repeatedly used in joins as a broadcast variable, and then join it. This table is often considered as a lookup table.

Since when we know that it is impossible to work with two DataFrames at the same time without joining them together, we had to find a way out of this loophole.

We adapt the idea of lookup table, but since we had bad experience with the joins, we wanted to avoid them. To do that, we create a collection from the context node by applying *collect* action on the DataFrame.

First, the action *collect* creates a collection of *Strings* where each element is a Dewey path. Then, we register a user defined function, and during the registration, the broadcast variable from the collection is created. The input parameter of the user defined function is a Dewey path of a candidate for a member of the new context nodes. The candidates for a new context node are all rows whose values of column *value* fulfill the node test of XPath step. The called UDF (User Defined Function) checks whether the relationship between inputted Dewey path and the Dewey paths in the collection of the context node is as it is desired. If the UDF is evaluated as true, the currently checked node will be a member of the next lookup table. The advantage of this method is that each executor may have its own partitions of input file in memory, and just lookup collections are collected to the driver and then broadcasted among other executors.

The user defined functions used in this method are different from those that are used in the Pure SQL discussed method in Sect. 3.3. We created UDF separately for each axis. The difference is that these functions, firstly, create a broadcast variable and then, according to the axis specifier, they detect whether the examined node belongs to the desired axis that was desired. Instead of two input parameters, just one is required by UDFs in this method, given that they use the broadcast variable.

Also, in this method, the evaluation starts with a selection of a parent node of a root node, and then independent XPath steps are evaluated step by step.

By the evaluation of the last XPath step, the result nodes are obtained, but still, their content is not text nodes or other descendant elements. So, the last step of the evaluation is to get them in the required format.

Table 4 shows a comparison of the method using LEFT SEMI JOIN and the method using lookup collections. Unlike the previous measurement in Sect. 3.3, in this case, both methods do as a first step caching of partitions of processed DataFrame into the memory.

Table 4. Performance of translated queries - LEFT SEMI JOIN versus Lookup collection

	Table	Method	Query	Time [s]
1	Books	Lookup col	//book/author	2.442
2	Books	LEFT SEMI JOIN	//book/author	3.064
3	Nasa	Lookup col	//author/suffix	7.024
4	Nasa	LEFT SEMI JOIN	//author/suffix	9.492
5	Nasa	Lookup col	//suffix	5.856
6	Nasa	LEFT SEMI JOIN	//suffix	7.235
7	Protein	Lookup col	//formal	418.305
8	Protein	LEFT SEMI JOIN	//formal	423.819
9	Protein	Lookup col	//organism/formal	1088.489
10	Protein	LEFT SEMI JOIN	//organism/formal	3441.569

4 Experimental Results and Discussion

In Sect. 3, we provided several comparisons, and in this section, we summarize measured times of different methods.

Local Mode. All experiments of local performance testing were run on a virtual machine hosted on an Intel Core i3 350 M 2.27 GHz processor, with 8 GB DDR3

RAM and 100 Mbps LAN network, and with installed Windows 8.1 Pro 64-bit operating system. The virtual machine has allocated 2 CPU cores and 5 GB RAM under the operating system Ubuntu 14.04 64-bit. All experiments were run on Spark in version 1.5.2, for which 512 MB of memory has been allocated. Information about tested tables containing transformed XML documents is in Table 1. Table 5 summarizes tested queries and tested tables. All measured values are summarized in Table 6. All the measurements in Table 6 were realized locally.

Table 5. Summarizing table of tested queries

	Query	XML file	Text file	Rows count
Books 2	//book/author	1.1 kB	1.4 kB	60
Nasa 1	//suffix	25.1 MB	45.8 MB	791 922
Nasa 2	//author/suffix	25.1 MB	45.8 MB	791 922
Protein 1	//formal	716.9 MB	2.3 GB	37 260 927
Protein 2	//organism/formal	716.9 MB	2.3 GB	37 260 927

Table 6. Performance of proposed methods in seconds

Method/Tab.	Books 2	Nasa 1	Nasa 2	Protein 1	Protein 2
Cartesian	9,79	-	1200#	-	-
SQL JOIN	8,372	217,818	232,70	-	-
DF JOIN	8,313	219,012	231,99	-	-
SEMI JOIN	4,687	13,336	18,75	-	-
SEMI JOIN∗	3,064	7,235	9,49	423,819	3441,569
Broadcast coll.	2,442	5,856	7,02	418,305	1088,489

Note that some values were not measured in Table 6, because the computation was very slow. The second SEMI JOIN marked with ∗ is the caching using method. The measuring of the value marked # was so slow that it had to be interrupted. Measured values are shown graphically in Fig. 3.

It can be seen that the method using Cartesian product is really slow. SQL RIGHT JOIN and DataFrame LEFT JOIN are use the same optimization process, so given the same physical plan is generated, the computational time is almost the same. In Fig. 3, one more thing is noteworthy: a positive impact of caching. According to Table 6, the evaluation of XPath query `//author/suffix` over the table Nasa is currently (with broadcast lookup collection method) more than 150 times faster in comparison with the Cartesian product. Since our methods evaluate an XPath query step by step, the impact of the number of evaluated steps can be seen in Fig. 3. This implies that more XPath steps mean longer computation duration since there is no optimization used, so all steps are evaluated.

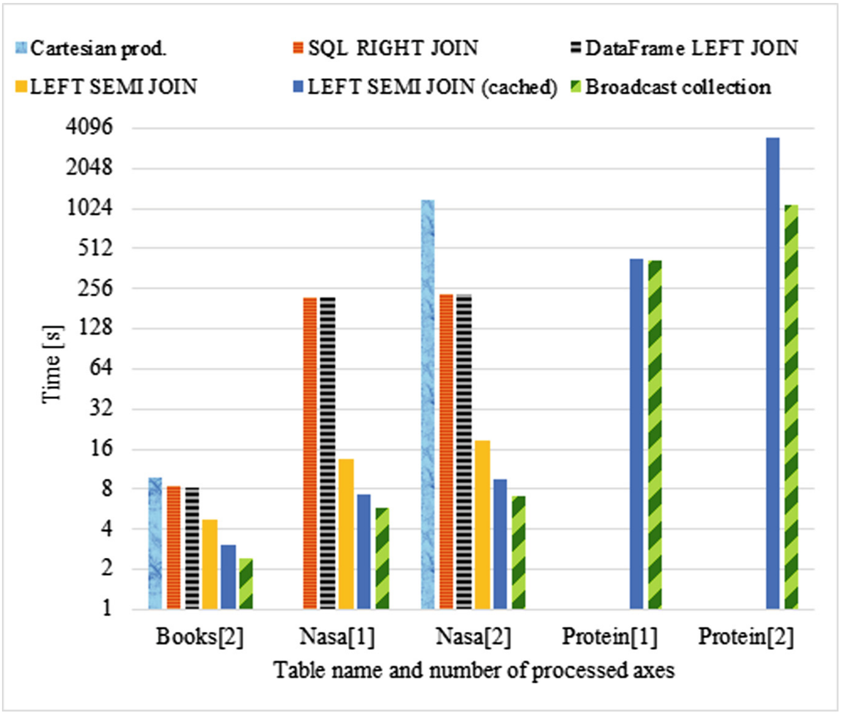

Fig. 3. Performance of proposed methods

Cluster Mode. The experiment ran on a cluster of 4 virtual machines hosted on four processors Intel Xeon 3.4 GHz (each 2 physical cores and 4 logical cores, with enabled Hyper Threading), with 32 GB DDR2 RAM and 2×1 Gbps LAN. Each virtual machine has allocated 6 GB RAM (of which 4,8 GB were used by Spark) and 2 CPU cores. Virtual machines were connected via 10 Gbps VMXNet3 LAN and the installed operating system was Ubuntu 14.04.

Using a small input file, it is impossible to see the benefits of cluster computation: in some cases, the communication load took more time then the actual computation. The cluster computation forced us to use the Hadoop Distributed File System to make our text files visible for workers. On our cluster, we continued our test attempts with bigger files, since a sufficient amount of memory was available for the worker nodes.

Comparison of Performance in Local and Cluster Mode. Comparison of computation in local mode and in cluster mode brought expected results. Admittedly, the computation on the cluster with enabled cluster mode was faster in some cases. Table 7 shows time comparison of local and cluster mode. In these experiments, it turned out that the fastest method is the one that uses nested lookup collection.

Table 7. Performance of cluster and local mode

	Table	Mode	Query	Time [s]
1	Nasa	Local	//author/suffix	7.024
2	Nasa	Cluster	//author/suffix	38.415
3	Protein	Local	//formal	418.305
4	Protein	Cluster	//formal	398.701
5	Protein	Local	//organism/formal	1088.489
6	Protein	Cluster	//organism/formal	957.729

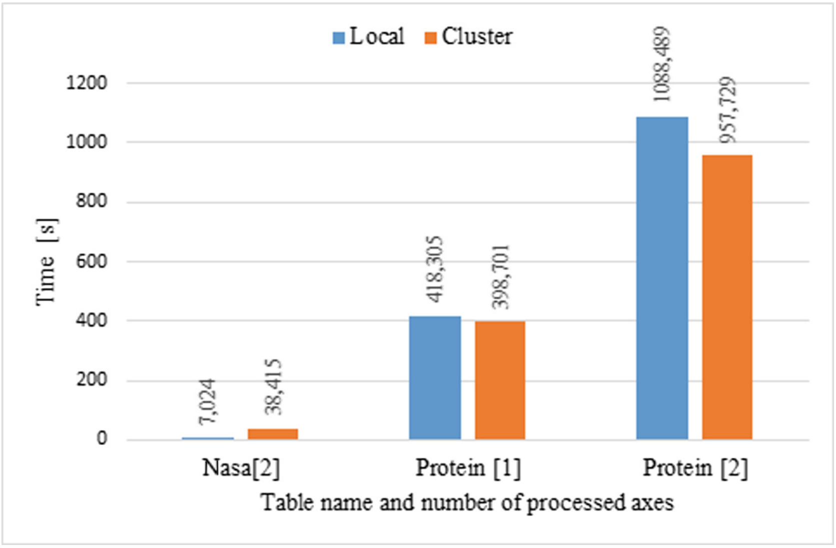

Fig. 4. Performance of cluster and local mode

As we can see in Table 7, a processing of a smaller table can be faster when it is done locally. The reason behind it are cluster overhead expenses such as serialization and transporting data among other workers.

In Fig. 4, we can see the measured times from Table 7.

5 Conclusions and Future Work

We analyzed possibilities of applying XPath queries on XML-documents by using framework Spark SQL. Additionally, we implemented, tested, and measured our initial statements concerning querying process of XML documents using the Spark SQL system - its advantages and disadvantages. We designed multiple methods and compared their efficiency. Bases upon our experiments, we conclude that the efficiency of some of the tested methods is limited. So, not all of the proposed methods can be used to query large data. We compared the method efficiency in both Spark modes, i.e. in the local mode and in the cluster mode.

One of the biggest advantages of Spark is the broadcast variables method. As our measurements show, the broadcast variables method is the fastest (and by far, the best) method from all the methods we investigated.

In the future, we plan to experiment with larger files and with more powerful clusters to evaluate scalability of our methods. We want focus on utilization using various Spark tuning possibilities to optimize computational cost and time. Finally, we plan to compare parallel XPath queries evaluation (using Spark) to other framework for distributed computing.

References

1. Amer-Yahia, S., Du, F., Freire, J.: A comprehensive solution to the XML-to-relational mapping problem. In: Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management, pp. 31–38 (2004)
2. Bidoit, N., Colazzo, D., Malla, N., Sartiani, C.: Partitioning XML documents for iterative queries. In: Proceedings of the 16th International Database Engineering & Applications Symposium, pp. 51–60. ACM (2012)
3. Bourret, R., Bornhövd, C., Buchmann, A.: A generic load/extract utility for data transfer between XML documents and relational databases. In: Advanced Issues of E-Commerce and Web-Based Information Systems, WECWIS 2000, pp. 134–143 (2000)
4. Camacho-Rodríguez, J., Colazzo, D., Manolescu, I.: Building large XML stores in the Amazon cloud. In: 2012 IEEE 28th International Conference on Data Engineering Workshops (ICDEW), pp. 151–158. IEEE (2012)
5. Choi, H., Lee, K.H., Kim, S.H., Lee, Y.J., Moon, B.: HadoopXML: a suite for parallel processing of massive XML data with multiple twig pattern queries. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, pp. 2737–2739. ACM (2012)
6. Fegaras, L., Li, C., Gupta, U., Philip, J.: XML query optimization in Map-Reduce. In: WebDB (2011)
7. Hricov, R.: Evaluation of XPath queries over XML documents using SparkSQL framework - Master thesis. FIT CTU - Master thesis (2016)
8. Marcjan, R., Siwik, L.: The concept of transformation of XML documents into quasi-relational model. In: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kostrzewa, D. (eds.) BDAS 2014. CCIS, vol. 424, pp. 569–580. Springer, Cham (2014). doi:10.1007/978-3-319-06932-6_55
9. Strnad, P., Macek, O., Jira, P.: Mapping XML to key-value database. In: The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDA 2013, pp. 121–127 (2013)
10. Tatarinov, I., Viglas, S.D., Beyer, K., Shanmugasundaram, J., Shekita, E., Zhang, C.: Storing and querying ordered XML using a relational database system. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pp. 204–215. ACM (2002)
11. Šenk, A., Valenta, M., Benn, W.: Distributed evaluation of XPath axes queries over large XML documents stored in MapReduce clusters. In: Proceedings of the 2014 International Semiconductor Laser Conference, ISLC 2014, pp. 253–257. IEEE Computer Society, Washington, DC (2014). http://dx.doi.org/10.1109/DEXA.2014.59