CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Evaluation of XPath queries over XML documents using SparkSQL framework
Student:	Bc. Radoslav Hricov
Supervisor:	Ing. Adam Šenk
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2016/17

Instructions

SparkSQL framework enables distributed and parallel data processing of various formats using SQL-like query language. The main goal of the master thesis is to use the SparkSQL framework to implement a subset of expressions from the XPath query language, which is used for querying XML data.

1. Get acquainted with the Apache Spark engine, mainly focus on its SparkSQL framework.
2. Study the works related to the process of mapping the XML database technology (XML documents) to the relational database technology.
3. Based on your knowledge, design a query engine that will be able to evaluate XPath queries over XML documents.
4. Implement a prototype of the designed solution using the SparkSQL framework.
5. Perform suitable testing on the implemented prototype, primarily aim on its functional properties.
6. Create a summary of the performed testing and assess the possibility of its deployment in a highly distributed environment.

References

Will be provided by the supervisor.

L.S.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague September 23, 2015

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

Evaluation of XPath queries over XML documents using SparkSQL framework

Bc. Radoslav Hricov

Supervisor: Ing. Adam Šenk

5th February 2016

Acknowledgements

I would like to express sincere thanks to my supervisor Ing. Adam Šenk for his helpful advices and comments that helped me to finish this master's thesis.

Also I would like to thank Prof. Dr. Wolfgang Benn and Johannes Fliege from Technische Universität Chemnitz, Faculty of Computer Science for all their help and the opportunity to work on my master's thesis abroad at university in Chemnitz.

Last but not the least, I would like to express my heartfelt gratitude to my parents, all my family and friends for the support not only during the work on the thesis, but during my whole university study.

Declaration

Citation of this thesis

Hricov, Radoslav. *Evaluation of XPath queries over XML documents using SparkSQL framework*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstract

The main goal of this thesis is to use Spark SQL framework to implement a subset of expressions from XPath query language. The first part of this thesis is focused on introducing the Apache Spark project. The second part covers analysis of mapping XML documents into the tabular form using an encoding of nodes that keeps a document order. Also the approach to the solution that uses Spark's features is described in the second part. The third part of the thesis is focused on implementation and testing of designed solution.

Keywords XML, XPath, SQL, Spark, Spark SQL, DataFrame, Dewey order encoding

Abstrakt

Cieľom tejto práce je implementovať podmnožinu výrazov jazyka XPath pomocou systému Spark SQL. Prvá časť práce je zameraná na predstavenie projektu Apache Spark. Druhá časť pokrýva analýzu možnosti mapovania

XML dokumentov do formy tabuľky použitím kódovania prvkov, ktoré zachováva ich poradie v rámci dokumentu. V druhej časti je taktiež popísaných niekoľko spôsobov riešenia, ktoré využívajú funkcie systému Spark. Tretia časť tejto práce je zameraná na implementáciu a testovanie navrhnutého riešenia.

Kľúčové slová XML, XPath, SQL, Spark, Spark SQL, DataFrame, Dewey order encoding

Contents

List of Figures

List of Tables

Introduction

Nowadays XML is a popular language for its platform independent way of storing data. By using the XPath query language it is possible to create queries to select data stored in XML documents. In the IT world there exist solutions that map XML semistructured data into the relational tables to be processed via SQL. In connection with it, one of the options how to evaluate XPath queries is to map them to the SQL queries.

Apache Spark is a fast evolving engine for in-memory big data processing that powers several modules. One of the modules is Spark SQL that allows working with structured data using SQL-like query language or domain-specific language of DataFrame.

Since Spark SQL works with the structured data, firstly the process of mapping XML tree data structure to the structured tabular format must be realized.

The main goal of this master's thesis is to use Spark SQL framework to implement a subset of expressions from XPath query language.

Apache Spark is a quite new technology. Through this thesis we would like to get acquainted with the Spark engine mainly with its module SQL and recognize its possibilities and potential limitations.

Chapter 1

Technologies

This chapter covers the basic technologies that relate to this thesis. We describe an XML as a popular language used for data storing and sharing them over the web. In connection with XML there exists a query language called XPath that is used to locate and process data stored in XML documents. In this thesis we translate XPath queries to another query language, which is allowed in Spark SQL engine, called SQL.

1.1 XML

The XML is an abbreviation of Extensible Markup Language [4]. It was designed as a platform-independent easy to read language for storing data and exchanging them over the web. Extensible Markup Language is focused on the meaning of data. Unlike HTML, XML has no predefined tags. Tags are created by author of the XML document according to his needs. The XML document is built by two component units:

- Element
- Attribute

Element is the basic logical unit of XML document. Every element starts with an opening tag and ends with a closing tag. The name of the element is given by tag. Tags always begin with `<` and end with `>`. Between these two characters is the name of element, for example `<element>`. The closing tag is a bit different. It has backslash before the element name such as `</element>`. Elements can contain a text node, attributes or other elements. A nesting of element nodes creates a tree structure of the XML document where inner nodes are XML elements and leaves of tree represent text nodes. According to the XML specification there also exist elements without the closing tag. It is called a self-closing tag and it is used as an alternative expression for an empty element. In this case the element `<element></element>` is equivalent to the self-closing tag `<element />`. There exists one more special element type called a mixed content element. The mixed content means that it can contain attributes, text nodes and other elements nested within element. The XML elements are simply extensible. It means that elements can be extended to carry more information without the impact on the application that processes the XML document.

Attributes carry additional information about the element. It is a pair of an attribute's name and value that relate to the particular element. The attribute's specification is located in the opening tag of the element and it is always behind the element name. There can be more attributes in a row, but the value of each attribute must be quoted such as `<element attribute1="attr1" attribute2="attr2"> </element>`. It depends on the author of XML document whether the new element or attribute is used for adding information to extend the XML document.

Following example shows a simple XML document containing elements, attributes and text nodes.

```
<bookstore>
  <book category="web">
    <title lang="en">XQuery Kick Start</title>
    <author>James McGovern</author>
    <author>Per Bothner</author>
    <author>Kurt Cagle</author>
    <author>James Linn</author>
    <author>Vaidyanathan Nagarajan</author>
    <year>2003</year>
    <price>49.99</price>
    <edition>2</edition>
    <pages>765</pages>
  </book>
  <cd>
    <title>Love, Lust, Faith and Dreams</title>
    <author>30 Seconds to Mars</author>
    <year>2013</year>
    <price>25.55</price>
    <genre>Alternative rock</genre>
  </cd>
  <book category="web" cover="paperback">
```

```
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

Extensible Markup Language is derived from SGML and it is a W3C recommendation. Nowadays XML is very popular, because it can carry data and it is often used for a data exchange within web services. For example a SOAP protocol for exchanging messages between independent applications is fully based on the XML.

XML documents can be processed by XSL technologies including XSLT, XPath and XQuery. In the following chapter XPath query language is described because it is a technology used in this thesis.

1.2 XPath

One of the approaches that allow processing the XML documents is XPath. XPath is a query language for selecting and extracting information from the XML document defined by W3C [5]. The name XPath is an abbreviation which stands for XML Path language and it is derived from ability to address the nodes of XML tree in hierarchical way. For this purpose, the path notation is used to navigate through the tree structure. The path is a sequence of steps separated with a slash or double slash, where every step is a move in axis level and points to one or more XML component units. Each path step consists of three parts:

- Axis
- Node test
- One or more non-compulsory optional predicates

An axis defines a direction for a step move and it is possible to navigate by forward or reverse steps. In the following Figure 1.1 there are shown some XPath axes in a hierarchical arrangement and also their relations to each other. Forward axes are child, descendant, attribute (not shown in Figure 1.1), self, descendant-or-self, following-sibling, following, namespace (not shown in Figure 1.1). Axes parent, ancestor, preceding-sibling, preceding and ancestor-or-self belong into the second group – reverse axes. Forward axes provide access to the nodes which are below or on the same level within XML document according to the document order. The reverse axes provide access to the nodes which are above, or in case of ancestor-or-self axis, on the same level as the context node. The context node is currently processed node from a set of nodes that are processed by XPath processor.

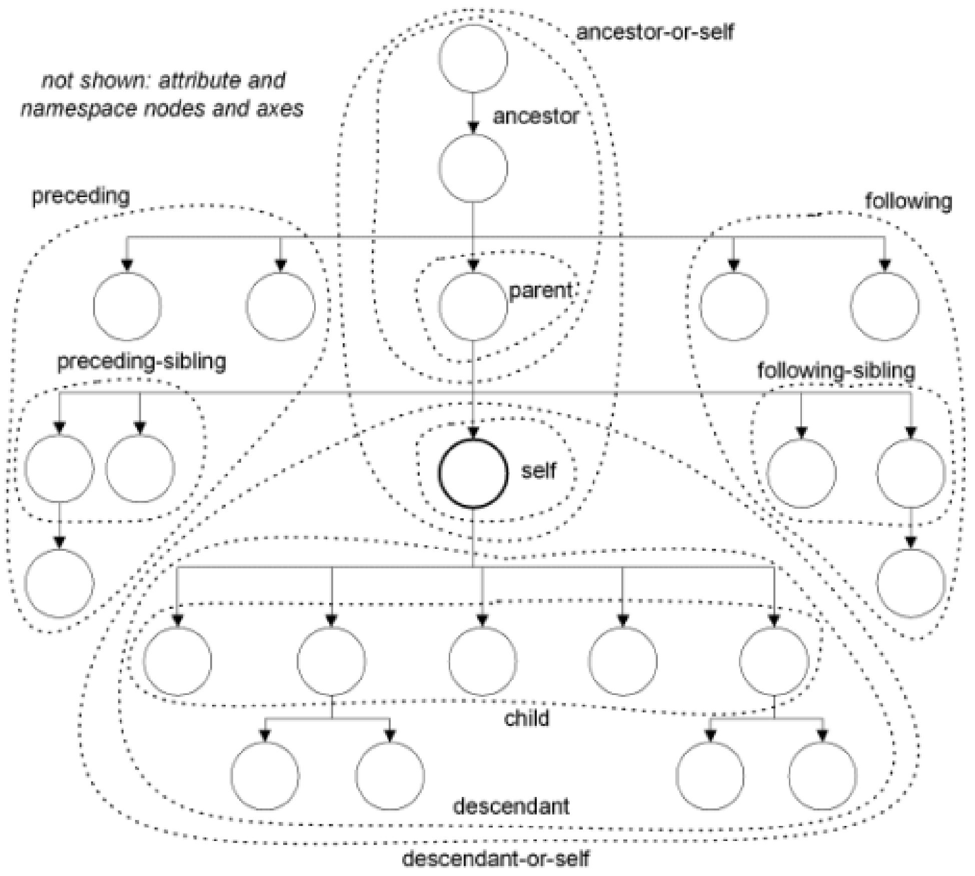

Figure 1.1: XPath Axes [1]

A node test provides a filtration based on the name or kind from the desired axis. By name, filtered are all elements from the context node which have the same name as it is desired. Instead of the concrete name, a wildcard character asterisk to select all element nodes relating to the desired axis can be also used. The filtration based on kind means that it is possible to select a specific component items of XML document, for example all components which are either comment, node of any type, text, element, attribute or processing instruction can be selected.

Predicate is optional part of the XPath expression and makes the node test more specific. Predicate is written in square brackets at the position following the node test part. For example if the context node has more children with the same name it is possible to point to the certain node by using predicate such as

```
/child::bookstore/child::book[2]
```

for selecting only the second *book* node that is a child element of the element *bookstore*. In another case, for example, it is possible to select all descendant elements whose element price is lower than 100 as follows

```
/descendant::*[attribute::price < 100]
```

XPath query steps can be written in a normal – unabbreviated or in abbreviated syntax. In unabbreviated syntax there is always defined the axis before the node test delimited by `::`. In abbreviated syntax, in some cases the axis name can be omitted. For example the child axis can be omitted, so the

```
/child::bookstore/child::book
```

in unabbreviated form can be abbreviate to

```
/bookstore/book
```

Similarly works the descendant axis. Its abbreviation is `//`, so expression

```
/child::bookstore/descendant::author
```

can be alter with

```
/bookstore//author
```

The selection of an attribute can also be done without defining the axis. Axis `attribute::` can be replaced by `@`, so

```
//author/@id
```

is short for

```
/descendant::author/attribute::id.
```

For addressing parent axis from the context node the abbreviation `..` can be used with the same meaning as `/parent::node()`.

The result of XPath step and also of whole query may be one or more XML component units. The result may not contain duplicates and it is always returned in the document order.

1.3 SQL

A database management system allows user to store, retrieve, modify or delete of data in an efficient way by using a special query language. The abbreviation SQL stands for a Structured Query Language [6]. By means of SQL it is described what should be done with data, not how to do it. A concept of Structured Query Language is based on the tuple relational calculus (relational algebra) and its queries are built as the simple and understandable English sentences. Originally the SQL was made by IBM in 1970 and it was developed for managing relational databases and manipulating data. The SQL consists of four types of commands – DDL, DML, DCL and TCL.

DDL stands for Data Definition/Description Language. Database administrators use these commands of the language to create databases and define a structure of the tables in database. DDL contains CREATE, ALTER, RENAME and DROP commands.

DML is Data Manipulation Language that is mainly used for insert, obtain and modify data. The language involves commands like SELECT, INSERT INTO, UPDATE, DELETE. These commands are often used by database users.

DCL is Data Control Language and it is a part of SQL language that is used by database administrators to protect the database. GRANT, DENY and REVOKE are DCL's commands.

TCL stands for Transaction Control language and it is a subset of the SQL language that allows managing of transactions in database. For this purpose the commands such as COMMIT, ROLLBACK and SAVEPOINT are used.

Some of the mentioned commands can require additional keywords or clauses to make them more comprehensive.

The Structured Query Language also provides several built in functions. We can divide them into two groups – aggregate functions and scalar functions. Aggregate functions consume data from a column and return the single–value result. This group includes functions like `COUNT()` which returns count of inputted rows or function `MAX()` that returns the maximal value from inputted column. Scalar functions process an input value and return a single value. The function is applied on each record in a column. For example function `UCASE()` which converts text to upper case or function `LEN()` which returns the length of text are the scalar functions.

Although SQL is an ANSI standard, various versions exist with differences depending on the vendor of database. All of these variations support at least the set of basic SQL operations and functions, but sometimes they are extended by features and capabilities which are not portable among different RDBMSs.

A part of the SQL standard is functionality that allows a creation of custom functions. This capability is called User-Defined Functions. As it is written above, vendors customize SQL and not every vendor allows functions defined by user. For implementing User Defined Functions, procedural languages are often used, for example PL/SQL, Java and C# are used in Oracle database, Java and C# in IBM DB2, Transact-SQL, C and group of .NET languages are used in Microsoft SQL Server.

SQL queries created by database users or administrators are processed by RDBMS. There are several steps which are done by RDBMS before the query is executed:

- **Query parsing and validating** – query is split to the words and then it is checked if query has correct syntax. A validation is process in which the semantic errors are checked and it also checks if desired names of tables and names of columns exist and are in database catalog.
- **Execution plan generating** – in this process an access plan for the query is generated. RDBMS can optimize generated access plan according its statistic.
- **Execution plan executing** – in the last step the query is executed by running the execution plan.

Chapter 2

Apache Spark

In this chapter the Spark engine and its module Spark SQL are described since they are the main technologies we have worked with. We describe relation between two popular engines that are aimed to the processing of big data – Spark and Hadoop. Both are considered as top–level Apache projects. In the end of this chapter we introduce a cluster mode of Spark.

2.1 Spark Core

Apache Spark is a multipurpose cluster computing system for a large-scale data processing. Spark is open source engine originally developed by UC Berkley AMPLab and later in 2010 adopted by Apache Software Foundation. Spark provides a fast in memory computing and its ecosystem consists of higher–level combinable tools including Spark Streaming, Dataframes and SQL, MLlib for machine learning and GraphX for graph processing [2].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.

Running Spark provides a web user interface for checking and monitoring statuses, settings or results of Spark and Spark jobs [7].

Spark has become popular among the companies such as Amazon, Baidu, Databricks, eBay, TripAdvisor, Yahoo! and others that use Spark within various business spheres [8].

To use Spark engine a driver program is needed. Developers implement the driver program to define a processes that should run in parallel on a cluster. A brain of the driver program is SparkContext object that is a connection to Spark and it is often used for creating RDDs, broadcast variables or accumulators on computing cluster.

RDD that is an abbreviation of Resilient Distributed Dataset is a Spark's main abstraction. It is a collection of objects that can be processed in parallel. More about RDD is written in Chapter 2.1.1. We also describe accumulators and broadcast variables in the separate Chapter 2.1.2.

In processing of large datasets speed plays an important role. Spark deals with speed mainly via in–memory computing, but the platform is also efficient on a disk. In 2014 Spark won the competition in large–scale sorting and beat the previous record held by Hadoop MapReduce that used 2100 nodes to sort around 100TB of data in 72 minutes. Spark was able to do sorting on ten times fewer machines and it was approximately three times faster [9].

According to the official web page of Spark project, in comparison with Hadoop MapReduce, the applications on Spark can be 10x faster on disk and up to 100x faster in memory [2].

2.1.1 RDD

The main Spark's abstraction is an immutable collection called RDD. RDD stands for Resilient Distributed Dataset and it is a collection of objects that may be processed in parallel [7]. RDD makes Spark fault resilient, because every RDD has information about its creation from the other RDD, so it is easy to reconstruct just the lost partitions [10].

There are four ways how to create the resilient distributed dataset collection in Spark:

- **File system** – Spark provides a possibility to create a dataset from the same storages as Hadoop does. It includes a local file system, Hadoop Distributed File System, HBase and others. Also type of supported files is the same as in Hadoop including text files or SequenceFiles.

- **Parallelizing** – it is a process of transforming of existing collection in driver program into the parallelized collection. In this process the resilient distributed dataset which can be processed in parallel is created.

- **Transforming** – seeing that the RDD is immutable collection of objects, every operation that returns RDD always returns a new distributed collection and the already processed collection stays unchanged.

- **Changing persistency** – for a faster running of actions on RDDs, Spark allows persisting and caching of existing RDD in any step of driver program. It means that RDD collection stored in memory across the nodes can be used for further actions and can avoid repeating of lazy operations on RDD. It is mostly used within iterative algorithms or in a really fast interactive use. It is possible to store RDD either into the memory or on the disk of executor. The caching in Spark is also fault resilient, so if something went wrong with the stored partition of RDD it would be recreated by applying operations on the original RDD.

It is important to know that a transformation on RDD is a lazy operation. New RDD is not computed immediately, instead of it, the called operations are remembered and computed at the moment when the result of operations is needed. So the result is a new RDD derived from the original RDD by executing all the called operations. The efficiency is expected from this design, it is because Spark does not return large dataset back to the driver program after each operation. It returns only the final dataset after executing all operations.

As it was mentioned, two types of operations, transformations and actions, can be performed on the RDD. A typical example of transformation is *map* that calls a function on each element of dataset and returns a new RDD due to its immutability. Other transformations are for example *filter*, *coalesce*, *repartition*, *union*, *join*, *Cartesian*, *groupByKey/sortByKey* and others. On the other hand, the operations called actions such as *reduce* or *count* aggregate all elements of dataset and send result to the driver program. The other typical actions are *collect*, *take*, *saveAsTextFile* or *foreach*. The actions are those operations that invoke the computation of lazy transformations.

2.1.2 Variables sharing

A driver program usually contains functions that are passed to the Spark to be executed on the cluster nodes. Each variable in a function is newly copied to the cluster nodes, so the changes performed on the variables are not propagated back to the driver program or to the other executors. To deal with this problem Spark provides broadcast variables and accumulators as a way for sharing variables [11].

Broadcast variables

Broadcast variables provide a possibility to effective sending of variables to all the executors. For the executors these shared broadcast variables are just read-only values. Also in this case Spark works fast by using effective broadcast algorithms. If we know that we will use same data in several operations, the broadcast variables are the way how to cache data such as datasets on executors and minimize a communication load.

Accumulators

On the other hand Spark provides accumulators. The accumulators enable the driver program to aggregate values from the worker nodes. The accumulators are often used in case of a need to count some event. For example, for a debugging purpose sometimes it is useful to count how many times the function was called. From the executors' point of view, the accumulators are write–only variables, so workers cannot access to the value of accumulator. Only driver program is able to see the value of accumulator. The efficiency is expected from this design, because not every change of accumulator is propagated to the other executors. Accumulators of primitive data types such as *int*, *double*, *float* and *long* are available in Spark. Spark also allows to use custom accumulators types and user defined aggregation function. Instead of the addition operation the aggregate function can be altered, for example, it can be altered by finding the average from the accumulated values. This altered aggregating function must be commutative and associative.

2.1.3 Spark stack of library

A project Spark consists of Spark core engine and several built-in modules. It is possible to make a combination of more modules within the one driver program. The interoperability of several modules is a crucial advantage of Spark. If the Spark core is improved it can also have an enhanced impact on the performance of modules. On the other hand, instead of maintaining more independent systems we can have just Spark for the same purpose. In this section we describe Spark's modules shown in the Figure 2.1.

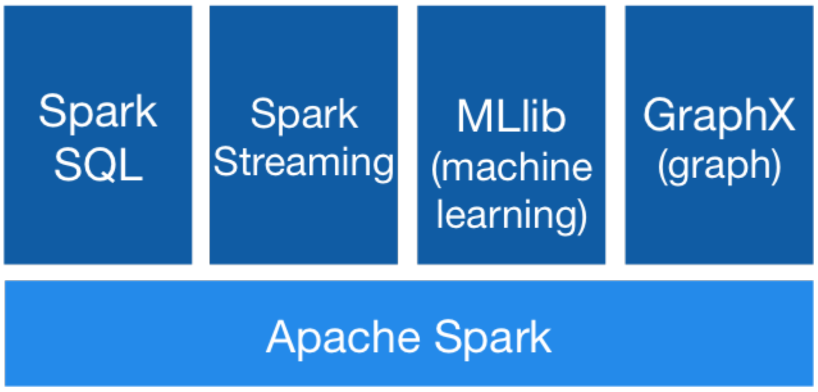

Figure 2.1: Spark Stack [2]

Spark Streaming

Spark module Spark Streaming enables a live data stream processing. Spark Streaming supports sources such as Kafka, Flume Twitter, Kinesis and TCP sockets. Streams can be processed by the functions included in Streaming API and the processed streams can be stored in databases or filesystems, or be processed by another Spark extension. Under the hood Spark Streaming works as follows as it was mentioned in [12], that follows original paper [13]. It receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches. The main abstraction is discretized stream called Dstream that is a sequence of RDDs.

DataFrames and SQL

Spark SQL enables processing of structured data via SQL and by using DataFrame API. A DataFrame is a distributed collection of data that is similar to database table, so each item of collection is a row representing a record. Spark SQL plays a big role in this thesis so we decided to describe it and its main abstraction DataFrame more detailed in a separate Chapter 2.3.

MLlib for machine learning

By MLlib extension Spark provides a machine learning functionality. It consists of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering and dimensionality reduction [14]. MLlib consists of two packages *mllib* and *ml*. The first is built on the top of RDD, the second is built on the top of DataFrames and it is recommended to use *ml* for its greater versatility and flexibility and ability that allows constructing machine learning pipelines.

GraphX

GraphX is an extension of Spark for manipulating graphs that provides graph parallel computations. Directed multigraph with properties attached to each edge and vertex is GraghX abstraction called Graph which extends RDD [15].

2.2 Spark's relation to Hadoop

In community of people who want to process larger amounts of data, a frequently asked question is whether it is better to use Hadoop MapReduce or Spark. In our case, the usage of Spark engine and its module Spark SQL was determined in the assignment of this work. In this section we compare two Apache projects Spark and popular Hadoop. We describe the essential similarities and differences between Spark and Hadoop's MapReduce as engines for a parallel – big data processing. We show why Spark could be better, in meaning that it should be faster, than Hadoop MapReduce module. Hadoop and Spark are both considered as the top-level Apache projects which means, that the projects are managed and developed by principles of Apache Software Foundation.

2.2.1 Hadoop versus Spark

Hadoop is an open–source framework that provides a distributed computing across the cluster and its computations are based on a map/reduce programming model. Hadoop consists of several modules whereby one of them is a MapReduce module. Spark extends and improves the MapReduce model and allows efficient computations, querying, stream and graph processing and machine learning. It must be said that Spark is not dependent on Hadoop however it can load data from and store them to Hadoop's HDFS.

Hadoop MapReduce is a disk based engine for processing and analyzing big data. To use the engine, developers write directly map and reduce jobs, on the other hand to run computation on Spark, it allows using the functions of API [11].

2.2.1.1 Execution model

MapReduce jobs consist of the processing step *map*, the synchronization *shuffle* and collecting the results from worker nodes called *reduce* step. During the *shuffle* phase, the result of mapping is merged and prepared for reducing. If user needs complicated data processing and more map/reduce jobs have to be executed consecutively, MapReduce may be slow [16]. It happens when the result of each following job depends on the result of the previous job and a following job cannot start until the preceding is not done. In Hadoop MapReduce computed data are stored to the disk after each map or reduce operation, so MapReduce jobs are executed sequentially. Reason why Spark can be really faster is that it provides in–memory caching, it does in–memory processing of data and within a sequence of operations it does not use the two–stage map/reduce paradigm. Spark brings an alternative approach to the problem of sequences of operations that is referred as Directed Acyclic Graphs [10].

From a cost point of view there exists a disadvantage of Spark. While it performs an in–memory computing, data must fit into memory for the optimal performance.

2.2.1.2 Fault tolerance

This two projects that are being described use different approaches to the failure avoidance. Hadoop uses persisting data on a disk and replication whilst Spark uses Resilient Distributed Datasets that exist mostly in memory and may be reconstructed when failure occurs since RDD has information about its creation. RDDs may also be persisted on disk on demand.

2.2.2 Directed acyclic graphs

In Spark a directed acyclic graph is a graph where vertices are transformations called on distributed dataset. DAG represents a model for scheduling work. An acyclicity is required seeing that the already executed transformations should not be executed once again, and via directionality it indicates an order of jobs. Unlike the map/reduce paradigm, DAG can be optimized and executing of independent DAG steps may be run in parallel. Since DAG contains information about processing data from the one state to the other, it can be easily used for reconstructing data in case of data loss. Advantage of systems using DAGs over map/reduce model is that DAG is executed whole at once on data in memory and does not require repeatedly reading data from disk [16].

Spark extends the idea of DAG and supports in–memory data sharing across DAGs. It can increase the speed of different jobs on the same data.

A DAG execution in Spark relates with the lazy evaluation on RDD and DataFrame, and it can reduce network communication. Figure 2.2 shows simple DAG that was created by Spark web UI.

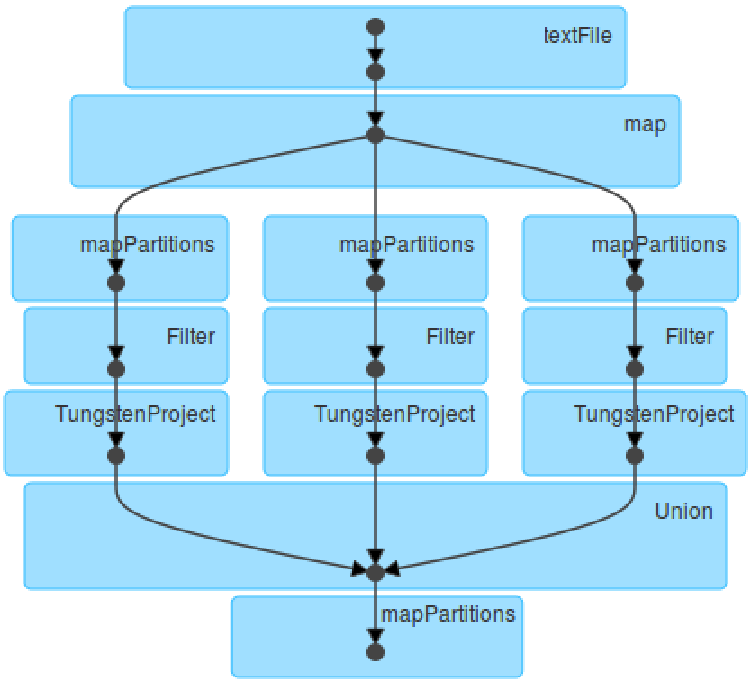

Figure 2.2: Directed Acyclic Graph

2.2.3 Summary

Although Hadoop has more disadvantages compared to the newer Spark, it is still useful in some cases. On the other hand, Spark comes with an admirable performance provided by in–memory data processing. Thanks to the tight integration with Hadoop, Spark is able to use the same data sources and formats as Hadoop does. If the user of Hadoop decides to start using Spark he does not need to migrate his large amounts of data elsewhere, seeing that Spark may use Hadoop Distributed File System. Text written above shows why we decided for using Spark instead of the Hadoop MapReduce engine.

2.3 Spark SQL

For scientists, analytics or general business users it is easy to use the Structured Query Language and create queries to examine data because it is a standard language that they know. Spark SQL, as the Spark module, enables these users to query and process structured and semistructured data via SQL or HiveQL languages [3]. The Spark SQL module has background in project Shark. It is an old project SQL–on–Spark from University of California, Berkeley, which altered Apache Hive to run on Spark.

Apache Hive is data warehouse software which facilitates querying and managing large datasets residing in distributed storage. HiveQL is SQL-like language that is used to query structured data in Hive [17].

Spark SQL inherited only the best features of Shark. With the new Spark SQL module came better integration with the Spark core and other modules.

Spark SQL consists of three components. The first, a Catalyst Optimizer is responsible for a query optimization and code generation. The second, a Spark SQL Core executes queries as RDDs and allows reading parquet, CSV, JSON and other data formats. The third is a Hive support that enables using Hive on Spark.

Catalyst optimizer is a basic component of Spark SQL and it supports both a rule and a cost based optimization. It executes four phases:

- Analysis of logical plan
- Logical plan optimization
- Physical plan generation (sometimes it may create more plans, but based on the cost just one is chosen)
- Code generation

The computation of Spark SQL starts with the analysis of an abstract syntax tree built from SQL or DataFrame object. Then, the unresolved (not know yet) attributes such as columns names and their types, or whether the desired columns belong to some table, are resolved by using Catalog object. By logical optimizations the rule based optimizations are performed to the logical plan. During this phase a constant folding, a projection pruning, Boolean expressions simplification, null propagation and other optimizations are realized [3]. Physical planning generates one or more physical plans. It takes optimized logical plan and applies operators of Spark execution engine. Based on the cost it chooses the best plan. Note that cost based optimization is used only within join algorithms. For a small relations a broadcast join is automatically used. In the last step of query optimization a Java bytecode is generated to be executed on separated machines [3]. Following Figure 2.3 shows how the queries are being planned.

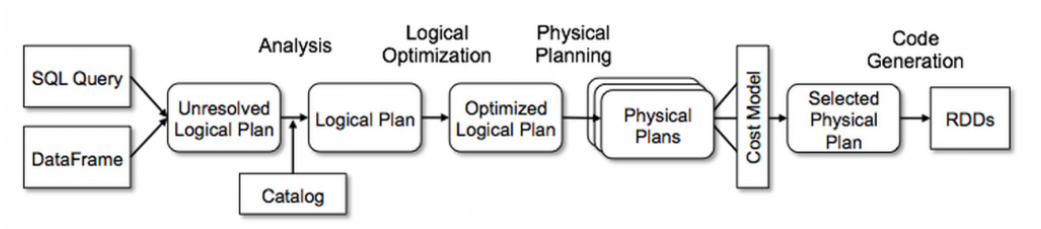

Figure 2.3: Spark SQL query planning [3]

Spark SQL works with a special kind of RDD called DataFrame. DataFrame is also known as the SchemaRDD that is its older version. General RDD is a collection of objects. SchemaRDD is very similar to database table, so object is a row representing a record. Especially, the SchemaRDD allows using the SQL queries, it also provides a more efficient way of working with data by utilizing its schema. Renaming from SchemaRDD to DataFrame happened by upgrading to Spark 1.3 version. DataFrame still represents a distributed data collection but it does not inherit from RDD anymore. DataFrame can be always converted to RDD collection and by own implementation it provides most of the functionality of RDD. A big advantage of DataFrame is its higher performance of computation against the RDD. A DataFrame can be built from several sources such as Hive tables, existing RDD or text files.

There exist two possibilities of converting RDD to DataFrame:

- **Reflection** – calling *createDataFrame* function on RDD requires Class object that provides a schema for a newly created DataFrame.
- **Construction of schema** – a schema may be represented as a *StructType* object that infers structure encoded in a *String*. A new DataFrame is created by applying schema on RDD.

DataFrame also provides a domain-specific language. It means, that it is possible to use built-in functions from API instead of direct evaluation of SQL query. For example query

```
SELECT * FROM dataFramePeople WHERE age>19
```

can be directly written in driver program as follows

```
dataFramePeople.filter(dataFramePeople.col("age").gt(19)).show()
```

DataFrame admittedly provides more domain-specific functions, for more detailed description read documentation [18].

Such as RDDs, also DataFrames are lazy evaluated, so operations on DataFrame are executed when the result is required by invoking action.

On the top of each Spark SQL application is *SQLContext* or *HiveContext* which use a *SparkContext* and bring all SQL functionality to the Spark. Spark SQL offers a feature that extends its functionality by registering user-defined functions which may be used in SQL queries. User defined functions are registered via *SQLContext*.

2.4 Running on cluster

Spark running in a distributed mode uses master–slave architecture where one node called driver coordinates numerous distributed workers [7].

Firstly, the terms that are used in this work are described.

- **Driver** – main process that runs *Main()* function of driver program, it creates *SparkContext* and prepares and schedules tasks.
- **Worker** – node of cluster that executes received application code.
- **Executor** – process created on the worker node, it executes individual tasks. Every executor is run in separate process.
- **Task** – the smallest piece of work that is executed by executor.
- **Job** – if an action in driver program is called, a job consisting of multiple tasks is computed in parallel.
- **Stages** – job divided into smaller tasks that are depended on each other.

Spark provides a possibility to run applications on a cluster via cluster managers such as Mesos and Yarn.

Apache Mesos is a cluster manager that abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively [19]. Hadoop Yarn is one of the Hadoop's modules, it is a framework for job scheduling and cluster resource management [20].

Spark also provides its own built–in standalone cluster manager. For running applications on the standalone cluster the Spark engine currently allows two deploy modes:

- **Client mode** – driver is run in the same process as a client that submitted the application. So if you run your application from a local machine, the driver process is running on your local machine. This can cause an unwanted effect since a frequent transfer of data among local driver and workers may be slower than in cluster mode. This mode is Spark's default mode.

- **Cluster mode** – application is submitted to the cluster and driver program is run from one of the worker nodes.

A brain of the driver program is a *SparkContext* object that manages processes on a cluster. It also notifies executors that are created on the cluster nodes and sends them JAR (in case Java and Scala) containing the application code. Finally it creates tasks for executors to be executed by them. On the worker nodes transformations are executed. Actions usually activate a transfer data from the worker nodes to the driver node, so if total amount of data on the worker nodes is larger than the available memory on the driver node it can be a problem that causes collapse of driver node.

Instead of adjusting the required setting parameters of *SparkContext* to run application according to them, the Spark submit script may be used. With the Spark submit script, the application packed in JAR is automatically submitted to the worker nodes. Through *SparkContext* and by submit script, the resources such as memory or number of CPU cores of executors and driver may be scheduled.

However just modes using cluster were described, the Spark engine provides a local mode too. In local mode the Spark driver and executor run in the same Java process [7].

Chapter 3

Analysis and design of solution

XML has become one of the most popular data format for data storing and exchanging them over the web. Normally, the XML documents are mostly processed by using the appropriate XML processing languages such as XPath or XQuery. However there exist attempts to process XML documents by using relational databases. Relational databases store non–structured data, so several approaches that map XML document to the relational table were invented. For this purpose both a schema mapping and an order mapping can be used. Depending on the approach, with the schema mapping usually more than one table is created due to different structures of XML subtrees. In this paper [21], there is described an algorithm for lossless schema mapping to generate a database schema from a DTD, which makes several improvements over existing algorithms. Also the other strategies for mapping XML to the relation table have been proposed by researchers such as [22] [23] or [24].

3.1 Transforming of XML document to relational table

The main programming abstraction of Spark SQL is DataFrame, distributed collection that is similar to the relational table. In this thesis we would like to process XML documents, so we need some mechanism that allows a transformation of an XML tree data model to the unordered relational data model. Tree structure of XML document is ordered data model based on the document order, or more precisely said, it is based on the order of each element within XML document. In this thesis 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 the insertion or deletion of nodes. The transformation of XML document must fulfil a requirement of possibility to transform from the unordered relational model back to the XML document. Hence we decided to follow the paper [24] since it shows that XML's ordered data model can indeed be efficiently supported by a relational database system. This is accomplished by encoding order as a data value.

In the following sections three methods to the transformation of XML documents into the tables are presented and one of them will be chosen. Also the creation of the Edge table that includes order information as a result of the transformation, is described below.

3.2 Global order encoding

In a global order encoding, each node is assigned a number according to the pre–order tree traversal. The pre–order traversal function assigns number to the root node then calls recursively the pre–order assignation to its child nodes from the left to the right. The result of this assignation is number representing an absolute position of node within XML document. Figure 3.1 shows how global order encoding is working. Note that the light–blue ellipses are element nodes and the light–green ellipses are text nodes.

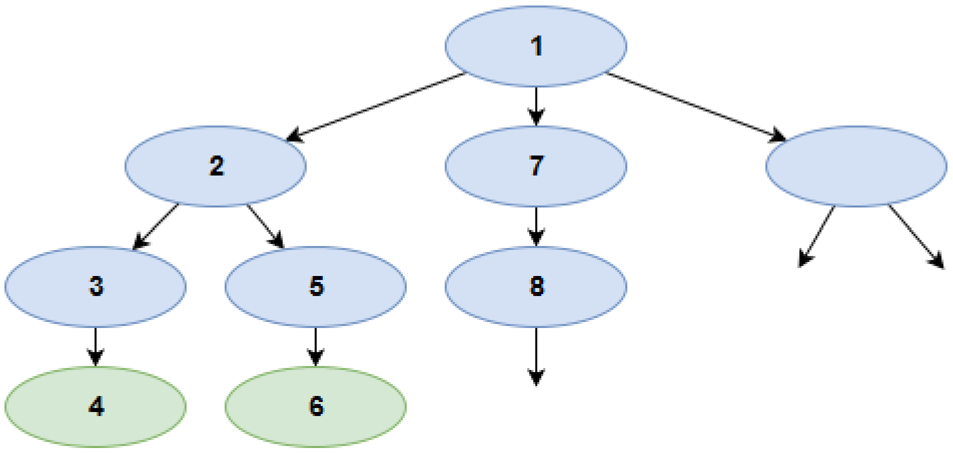

Figure 3.1: Global order encoding

With the global order encoding it is easy to find a result of queries like following-sibling or following because its Edge table may contain an ID of the last descendant of a context node. So the Edge table can be designed as `Edge(id, parentId, lastDescId, pathId, value)`. Global order encoding is the best on the queries evaluation, but not so good in the insertion of new nodes. When a new node is inserted, the IDs of nodes following the new inserted ones must be actualized. Although it looks like that the floating–point values could solve renumbering problem, actually it could just partially. In fact, *integer* and *real* are both stored by the same count of bits. The floating–point values could improve the performance of inserting, but in the worst case, when the count of inserted nodes is greater than available values nevertheless the renumbering must be performed.

3.3 Local order encoding

Nodes in a local order encoding are marked by number depending on their order among their siblings. For a better picture see Figure 3.2. It is easy to insert new node because just siblings following the new node have to be incremented. As it was mentioned in global order encoding, the floating–point values may also help, but with the same limitation. On the other hand, evaluation of queries, mainly following and preceding axis are difficult to execute while there is no available global order information. Edge table of local order encoding can be designed as `Edge(id, parentId, sIndex, pathId, value)`. Since the ID assigned to the node does not provide information about the position among its siblings, the sIndex as position must be added. In this case ID is unique identifier that does not relate to the document order, so it is not assigned according to the document order.

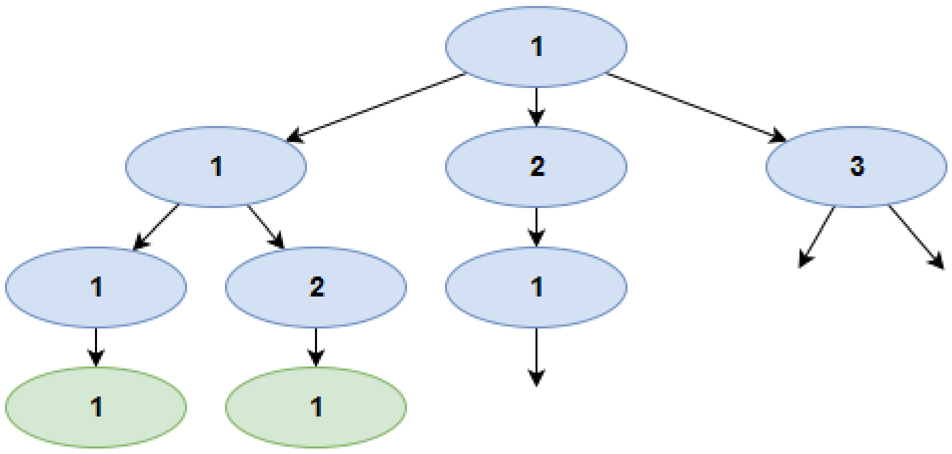

Figure 3.2: Local order encoding

3.4 Dewey order encoding

A dewey order encoding is a mix of the global and the local order encodings. In dewey order encoding each node is not assigned just by a single number, but it is assigned by path. The path represents a traverse from the root node of document to any concrete node of XML document. Each step of path represents a local order position information of ancestor node, but the complex path represents absolute node position within XML document. An evaluation of query within dewey encoding is very similar to the querying with global order encoding. Otherwise, inserting of new nodes cause that axis following–siblings and also their descendants must be updated. Despite that the dewey order encoding uses advantages of two preceding methods, one disadvantage should be mentioned. If the XML tree is too deep, the dewey path may require more space to be stored while it is not a single value anymore. It must be stored as a *vector* or *string*. Seeing that Dewey path include enough information about the position in XML document, the Edge table may be defined as `Edge(dewey, pathId, value)`. Dewey order encoding is shown in the Figure 3.3.

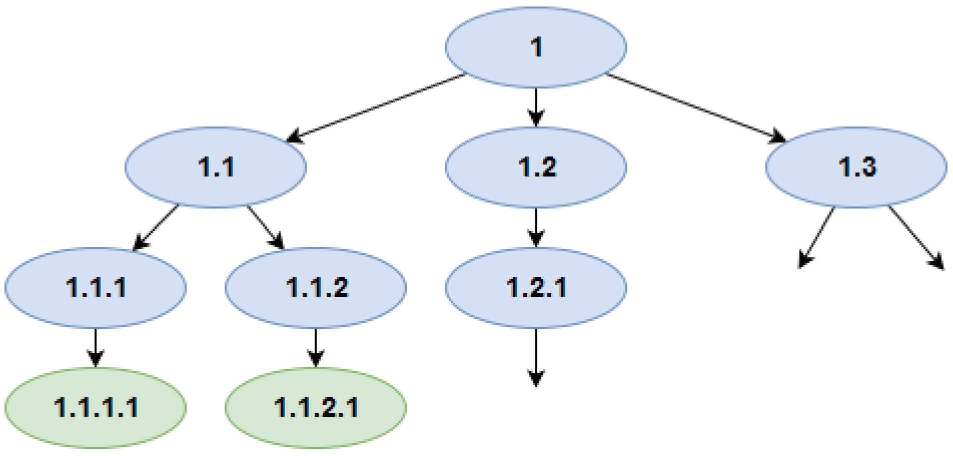

Figure 3.3: Dewey order encoding

3.5 Summary

Although Spark SQL supports JSON files it is not an advantage for this thesis, because it is a special type of JSON file with an atypical design. This special JSON consist of separated valid JSON objects that have the same schema. Same schema is important for a DataFrame creation since the schema of DataFrame is derived directly from JSON and it does not need to be defined in a program code. As a consequence, a regular multi-line JSON file will most often fail [25]. Instead of JSON we decided to use a simple text file that includes rows of transformed XML. From the three mentioned encodings the dewey order encoding will be used since it is the universal solution and the information stored in dewey path are sufficient. In compare with the global encoding it can be a bit slower depending on the comparison of the paths. Dewey path implicitly contains information about the node's ancestor nodes and also the position among the siblings. This thesis could be extended in a future by implementing a possibility to perform updates on really large XML files. Thus dewey encoding is the best option for our purposes.

In the further sections we use terms *edge table* and *nodes table* in the same meaning.

Chapter 4

Our approach

In this chapter five methods are described. Firstly we introduce a trivial method and then in the next sections it will be improved. First two ways use the SQL queries to evaluate XPath queries. The others use a domain specific language of Spark SQL API. The single methods are compared and we provide the results of local performance testing. The reason why we provide just local testing is that our early idea was to test functionality of Spark SQL API and find out whether it is usable to solve our problem. To ensure consistency of experiments each experiment was executed 10 times, whereby the first attempt was excluded since it was absolutely different from the others, and other measured times were averaged.

All local experiments were run on virtual machine hosted on an Intel Core i3 350M 2.27 GHz processor, with 8GB DDR3 RAM and 100Mbps LAN network, and with installed Windows 8.1 Pro 64-bit operating system. Virtual machine has allocated 2 CPU cores and 5GB RAM, and operating system Ubuntu 14.04 64-bit has been being run on it. All experiments were run on Spark in version 1.5.2, for which 512MB of memory has been allocated. A complex performance comparison of all methods is in the Chapter 6.3. Information about tested tables containing transformed XML documents are in Table 5.1.

4.1 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. The partial table of nodes can be found in Chapter 5.1. In all our tests we performed translations of simple XPath queries that covered all XPath axes. The following example shows how XPath query:

```
//book/author
```

was translated to the SQL query using our first method:

```
(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)
```

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 is 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 last extra step their content is appended.

On our small testing file it was relatively fast to compute a result, but problems with performance began during the processing larger table of nodes containing 791922 rows. After we examined an execution plan we found out, that for this naive method the Cartesian product followed by filtration was executed. Cartesian product makes a set of all pairs from the first and the second table. Joined table contains `n*m` (`n` and `m` represent number of rows in tables) records so applied filter has to run over the all records and it takes too much time. In this case the Cartesian product was bottleneck of the first method. The following physical plan is generated for the previously referred SQL query:

```
TungstenProject [dewey#0,pathId#1,type#2,value#3]
 Filter ((dewey#35 <= dewey#0) && UDF(dewey#35,dewey#0))
  CartesianProduct
   Scan PhysicalRDD[dewey#0,pathId#1,type#2,value#3]
```

```
ConvertToSafe
 TungstenProject [dewey#35]
  Filter ((dewey#31 < dewey#35) && UDF(dewey#31,dewey#35))
   CartesianProduct
    ConvertToSafe
     TungstenProject [dewey#35]
      Filter ((type#37 = 1) && (value#38 = author))
       Scan PhysicalRDD[dewey#35,pathId#36,type#37,value#38]
    ConvertToSafe
     TungstenProject [dewey#31]
      Filter ((dewey#21 <= dewey#31) && UDF(dewey#21,dewey#31))
       CartesianProduct
        ConvertToSafe
         TungstenProject [dewey#31]
          Filter ((type#33 = 1) && (value#34 = book))
           Scan PhysicalRDD[dewey#31,pathId#32,type#33,value#34]
        ConvertToSafe
         TungstenProject [0 AS dewey#21]
          Scan OneRowRelation[]
```

A User Defined Function (marked as UDF in physical plan) checks desired relation between two dewey paths.

4.2 Join-based SQL

In this case the best solution is to avoid Cartesian product and apply a SQL JOIN clause instead. Hence the JOIN ON conditions were defined to join results of single XPath steps. The idea is to select nodes that are candidates for a next context node, then combine them with the current context node and based on the relation do filtration of 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 further chapters.

By analyzing Spark's execution plans we finally decided for RIGHT JOIN (LEFT JOIN is also acceptable but it depends on the order in which XPaths steps are joined). Although the type of JOIN was defined, in some cases Spark has generated Cartesian product because joins conditions were not strong enough. 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 user defined function we add required UDF into the join condition. It must be said that also a JOIN, whose condition is based only on the user defined function that require arguments from the both left and right table, invokes the Cartesian product because all pairs must be processed by UDF. After the mentioned changes were applied, for XPath query:

```
//book/author
```

the SQL query is generated as follows:

```
SELECT p2.dewey, p2.pathId, p2.type, p2.value FROM nodes p2
  RIGHT JOIN
 (SELECT p1.dewey, p1.pathId, p1.type, p1.value FROM nodes p1
     RIGHT JOIN
  (SELECT p0.dewey, p0.pathId, p0.type, p0.value FROM nodes p0
        RIGHT JOIN
   (SELECT '0' as dewey) n0
  ON p0.type=1 AND n0.dewey < p0.dewey
               AND p0.value='book'
               AND isPrefix(n0.dewey, p0.dewey)
               WHERE p0.dewey IS NOT NULL) n1
 ON p1.type=1 AND n1.dewey < p1.dewey
              AND p1.value='author'
              AND isChild(n1.dewey, p1.dewey)
              WHERE p1.dewey IS NOT NULL) n2
ON n2.dewey <= p2.dewey AND isPrefix(n2.dewey, p2.dewey)
                        WHERE p2.dewey IS NOT NULL
```

The following physical plan was generated before the last mentioned SQL query was evaluated':

```
TungstenProject [dewey#0,pathId#1,type#2,value#3]
 Filter isnotnull(dewey#0)
  BroadcastNestedLoopJoin BuildLeft, RightOuter,
     Some(((dewey#16 <= dewey#0) && UDF(dewey#16,dewey#0)))
   ConvertToUnsafe
    Scan PhysicalRDD[dewey#0,pathId#1,type#2,value#3]
   TungstenProject [dewey#16,pathId#17,type#18,value#19]
    Filter isnotnull(dewey#16)
     BroadcastNestedLoopJoin BuildRight, RightOuter,
        Some(((dewey#12 < dewey#16) && UDF(dewey#12,dewey#16)))
      ConvertToUnsafe
      Filter ((type#18 = 1) && (value#19 = author))
       Scan PhysicalRDD[dewey#16,pathId#17,type#18,value#19]
     TungstenProject [dewey#12,pathId#13,type#14,value#15]
      Filter isnotnull(dewey#12)
       BroadcastNestedLoopJoin BuildRight, RightOuter,
          Some(((dewey#11 < dewey#12) && UDF(dewey#11,dewey#12)))
        ConvertToUnsafe
         Filter ((type#14 = 1) && (value#15 = book))
          Scan PhysicalRDD[dewey#12,pathId#13,type#14,value#15]
        TungstenProject [0 AS dewey#11]
         Scan OneRowRelation[]
```

After the changes were done the performance was admittedly better than the performance of method using the Cartesian product. By Table 4.1 we attach a comparison of two previously mentioned methods.

Table 4.1: 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

As we can see, the usage of OUTER JOIN and proper definition of JOIN conditions have crucial impact on the performance. After twenty minutes of computing we were forced to cancel the third measurement marked with asterix. We realized that Cartesian product in Spark is really slow.

On a Spark cluster we did a test. We applied the Cartesian product on two tables with size 8MB and 9.8MB and both contained about 265 000 rows. After 24 hours less than one third of result was computed and Spark stored more than 60GB of data on disk.

4.3 SQL query via DataFrame API

So far we have worked only with pure SQL queries, but 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. By using API we changed the order of processed axes, so instead of RIGHT JOIN the LEFT JOIN was applied. Generated physical plan is similar to the previous one:

```
TungstenProject [dewey#13,pathId#14,type#15,value#16]
 BroadcastNestedLoopJoin BuildRight, LeftOuter,
    Some(((dewey#9 <= dewey#13) && UDF(dewey#9,dewey#13)))
  TungstenProject [dewey#9]
   Filter AtLeastNNulls(n, dewey#9)
    BroadcastNestedLoopJoin BuildRight, LeftOuter,
       Some(((dewey#5 < dewey#9) && UDF(dewey#5,dewey#9)))
     TungstenProject [dewey#5]
      Filter AtLeastNNulls(n, dewey#5)
       BroadcastNestedLoopJoin BuildLeft, LeftOuter,
          Some(((dewey#4 < dewey#5) && UDF(dewey#4,dewey#5)))
        TungstenProject [00 AS dewey#4]
         Scan OneRowRelation[]
        ConvertToUnsafe
         Filter ((value#8 = bookstore) && (type#7 = 1))
          Scan PhysicalRDD[dewey#5,pathId#6,type#7,value#8]
     ConvertToUnsafe
      Filter ((value#12 = book) && (type#11 = 1))
       Scan PhysicalRDD[dewey#9,pathId#10,type#11,value#12]
  ConvertToUnsafe
   Scan PhysicalRDD[dewey#13,pathId#14,type#15,value#16]
```

Since we know that SQL and DataFrame share the same optimization pipeline (shown in Figure 2.3), the physical plans vary in small details depending on 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 broadcasts a smaller one across the workers. Table 4.2 shows the time comparison of the computation using SQL and DataFrame.

Table 4.2: 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

As it was expected the duration of the computations are almost the same since optimizer generates the same physical plan.

4.4 Alternative methods without joins

After the previous findings and since we assumed that a computation could be faster, we decided to restrict the usage of JOIN in our further ideas. Several alternatives such as nested queries, SQL IN operator or SQL UNION statement have been tested but they led into the failure.

We had some ideas that essentially do not use joins. 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 such as:

```
//book/author
```

expressed by SQL:

```
SELECT * FROM nodes_table WHERE value='author' AND parent(dewey) IN
(SELECT dewey FROM nodes_table WHERE value='book')
```

In order for Spark to be able to understand the query, it must be rewritten to the following form:

```
SELECT * FROM nodes_table WHERE value='author' AND parent(dewey) IN
(SELECT dewey FROM (SELECT * FROM nodes_table WHERE value='book') A)
```

Parent() is a user defined function that cuts the last part of inputted dewey path, so 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, but unfortunately Spark does not think the same. Spark is not able to execute nested SELECT following the WHERE clause. Neither by using Spark SQL API it is possible because Spark evaluates it in different way as it is expected. For example, we translated previous SQL query by using Spark SQL API such as:

```
schemaNodes.where(
   schemaNodes.filter("value='author'").col("dewey").isin(
      schemaNodes.filter("value='book'").col("dewey")));
```

Spark understands it wrong and generates the following physical plan:

```
Filter dewey#0 IN (dewey#0)
 Scan PhysicalRDD[dewey#0,pathId#1,type#2,value#3]
```

According to the physical plan Spark does not see the required, based on name filters that are important in this case and in addition, the filter shown in physical plan is always evaluated as true, so whole table is returned. Also using two separated DataFrames on IN statement had no effect, while without a join Spark had no information about the nodes from the second table.

Since there is no direct column to join two DataFrames we realized that instead of the join, a union of DataFrames could work for us. The idea was based on the union of the results of filtrations of two successively evaluated XPath steps. Then, as we had one united DataFrame we applied filtrations again and we used IN operator, such as in the following example with separated DataFrames:

```
DataFrame a = sqlContext.createDataFrame(dewey, Node.class).
        filter("value='author'");
DataFrame b = sqlContext.createDataFrame(dewey, Node.class).
        filter("value='book'");
DataFrame c = a.unionAll(b);
DataFrame result = c.where(c.filter("value='author'").
        col("dewey").isin(
                c.filter("value='book'").col("dewey")));
```

Also this example should return a DataFrame with no rows because it should check if each author's dewey path is somewhere in a set of dewey paths of all books. But it again led to unexpected behavior of Spark that ignored WHERE condition and checked if each dewey path is the same dewey path (it does not matter if it is author or book) so condition was always fulfilled. Generated physical plan shows the problem where `dewey#4` and `dewey#8` point to the same field of each record:

```
Union
 Filter ((value#7 = author) && dewey#4 IN (dewey#4))
  Scan PhysicalRDD[dewey#4,pathId#5,type#6,value#7]
 Filter ((value#11 = book) && dewey#8 IN (dewey#8))
  Scan PhysicalRDD[dewey#8,pathId#9,type#10,value#11]
```

4.5 Left semi join

As we mentioned, the IN clause may be used just with joined table. We realized that Spark SQL provides LEFT SEMI JOIN that is more effective than our all previous attempts. Firstly we have to 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 on existence of records in right table. It means that if there exists a record in the right table that fulfill JOIN ON condition, just a record from the left table is returned.

By this method we implemented a translation of XPath steps only for parent, child, ancestor, ancestor-and-self, descendant and descendant-or-self axes. Other axes have to be implemented by different manner that does not use LEFT SEMI JOIN, but uses, for example user defined functions. It is because the implemented axes are based on prefixes. The Table 4.3 shows the duration comparison of computation using LEFT JOIN and LEFT SEMI JOIN both via DataFrames, and both are applied without caching.

Table 4.3: LEFT JOIN versus LEFT SEMI JOIN

	Table	**Method**	**Query**	**Time[s]**
1	Books	LEFT JOIN	//book/author	8.313
2	Books	LEFT SEMI JOIN	//book/author	4.687
3	Nasa	LEFT JOIN	//author/suffix	231.989
4	Nasa	LEFT SEMI JOIN	//author/suffix	18.748
5	Nasa	LEFT JOIN	//suffix	219.012
6	Nasa	LEFT SEMI JOIN	//suffix	13.336

During the implementation of this method we have found a new faster solution that for the XPath steps evaluation does not use the joins. Hence we decided to not continue on development of this method and we rather wanted to improve the new solution. The method that in a principle does not use the joins is described in Chapter 4.6. It must be noted, that in the next comparison of methods, the LEFT SEMI JOIN that uses caching is applied.

4.6 Broadcasted lookup collection

In various Spark tutorials while using the JOIN statement 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 consider as a lookup table.

Since when we know that it is impossible to work with two DataFrames in the same time without joining them together, we had to find out how to deal with this limitation. We adapt the idea of lookup table, but since we had bad experience with the joins, we wanted to avoid them. Our idea of avoidance of the JOIN clause is creation of a collection from the context node by applying *collect* action on the DataFrame. Firstly, the action *collect* creates a collection of *Strings* where each element is dewey path. Then we register a user defined function, and during the registration, the broadcast variable from the collection is created. Input parameter of the user defined function is a dewey path of candidate for a member of new context node. The candidates for a new context node are all rows whose value of column *value* fulfill the node test of XPath step. Called UDF checks whether the relationship between inputted dewey path and the dewey paths in the collection of context node is as it is desired. If UDF is evaluated as true, a currently checked node will be member of next lookup table. 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 method mentioned in Chapter 4.1. We created UDF separately for each axis. The difference is that these functions firstly create broadcast variable and then, according to the axis specifier they detect whether examined node belongs to the axis that was desired. Instead of two input parameters, just one is required by UDFs in this method, seeing that they use the broadcast variable. Principle of UDF relation checking is described in Chapter 5.2.1.

Also in this method the evaluation starts with a selection of parent node of root node and then, the independent XPath steps are evaluated step by step. By the evaluation of the last XPath step the result nodes are gotten, but they still do not contain their content such as text nodes or other descendant elements. The last step of the evaluation that returns nodes that are the correct result of XPath query is described in Chapter 4.7.

Table 4.4 shows a comparison of the method using LEFT SEMI JOIN and the method using lookup collections. Unlike the previous measurement in Chapter 4.5, in this case both methods firstly cache partitions of processed DataFrame into the memory.

Table 4.4: 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

4.7 Getting result

By evaluation of the last XPath step we still do not have the whole result. We have just nodes that do not contain their nested – descendant nodes. Methods mentioned in this chapter cannot be used for this purpose since they traverse just through the element nodes. Their problem is that their results do not contain text nodes and they return distinct results. In the final we want to have a set of the result nodes, so the fact must be considered that the absolute evaluation of some axes can return duplicates. It is because the result is set that contains all result nodes and their nested nodes.

We tried more methods such as Cartesian product, OUTER JOINS or sequential search of descendant nodes but they always were bottlenecks. Mostly, the computation of the final result took more than the evaluation of XPath steps.

We finally create a user defined function and via the UDF one more column is added to the DataFrame that is being processed. The new column contains a number that indicates how many times current node is in the result set. Then the *flatMap* transformation, that returns RDD containing duplicates according to the number in the last column, is invoked. RDD is then easily converted back to the DataFrame.

This method of getting the final result is used within methods using LEFT SEMI JOIN and broadcasted lookup table. Trivial SQL method used the Cartesian product, and the OUTER JOIN is used within methods that use LEFT/RIGHT JOIN within SQL and DataFrame API.

4.8 Summary

We introduced five methods of our approach. We have started with the trivial method that uses the Cartesian product to join the single XPath steps. From our measurements we can see that Cartesian product is usable just with the small tables.

By analysis of the physical plans of our methods we were able to improve them.

We also designed multiple methods, but Spark showed its limitations and some of them could not be realized. Finally we designed the method that uses the one of the biggest advantage of Spark called broadcast variables. As it is shown in our measurements it is the fastest method from the all that we have introduced.

Chapter 5

Implementation

On the beginning of the problem we have an XML file and an XPath query as input values. We are working with DataFrames that are the main abstraction of Spark SQL. It is a distributed collection that may be considered as a relational table.

Hence, firstly the XML document must be transformed into the relational table to be able to be processed via Spark SQL API. In Chapter 3.1 we have already described chosen encoding that will be used for the XML document transformation.

Then we have an XPath query that must be translated into the SQL query to be able to be evaluated in the SQL module of Spark. According to the facts written above we developed two applications. The first is XML processor that transforms XML document into the relational table. The second one is a driver program for Spark that processes XPath query, by using SQL query or via Spark SQL API applies it on DataFrame built from the transformed XML document, and returns the final table of nodes that may be further processed.

The Figure 5.1 shows how two mentioned applications locally cooperate to return a result of XPath query. In the following figure, the transformed XML document is stored as a text file. Both, the text file and the XPath query input as parameters into the driver program that is then run on Spark. A result of evaluation of XPath query in driver program is a JSON file stored in HDFS.

In the following sections we describe our approach that is used to accomplish the goals of this thesis and also issues that happened during the realization are mentioned.

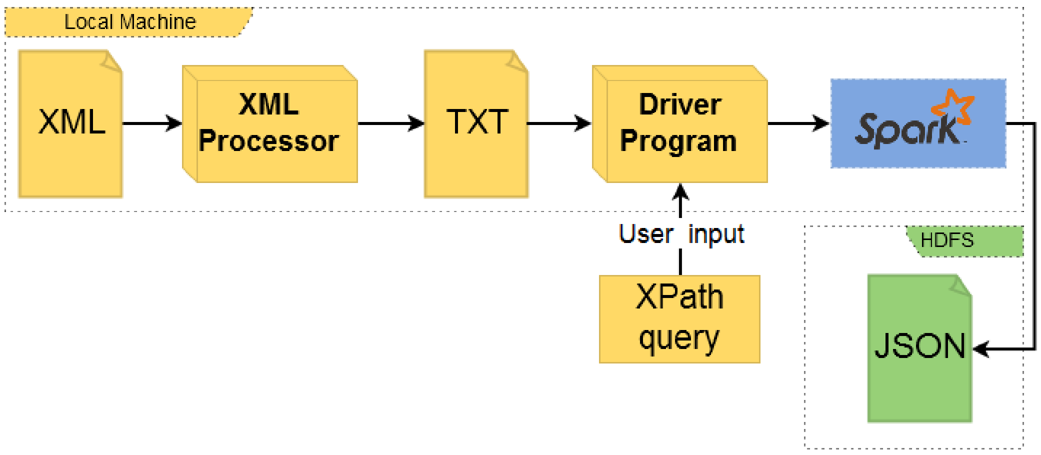

Figure 5.1: Local cooperation of applications

5.1 XML processor

At first, the XML processor is introduced. It was developed according to the analysis in Chapter 3.1. The XML processor was created as a standalone application. It means, that it may be used also without using our second application or the Spark engine. In our prototype application we decided to omit some XML components such as attributes, namespaces and others since they are just another types of XML nodes. Therefore we process just elements and text nodes.

The process of transformation begins with a numbering of elements and text nodes. Based on the pre–order traversal of 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*. For example, a comparison of dewey paths `0.1.2.15.7` and `0.1.2.2.7` gives a result where `0.1.2.15.7` is lexicographically less than `0.1.2.2.7` because on the same position 1 is less than 2 and it violates the document order. Hence, the path `0.1.2.15.7` is in the second phase recomputed to the `00.01.02.15.07` and `0.1.2.2.7` to the `00.01.02.02.07`. Now each part (parts are separated by dots) of dewey paths has the same length. The number of zeros in prefix depends on the number of digits of the highest value of dewey path part among all dewey paths. This also helps in SQL ORDER BY operation.

During the first – numbering phase also the paths to the certain nodes built from the names of nodes are created and it depends on user whether the table of paths will also be generated and saved as a file. Figures 5.2 and 5.3 shows how XML document from Chapter 1.1 is transformed into the relational table. We provide just uncompleted tables. The complex tables are available in Appendix D.

```
+-------------+------+----+--------------------+
|        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|
|  00.01.01.02|     4|   1|              author|
|00.01.01.02.01|    5|   3|      James McGovern|
|          ...|   ...| ...|                 ...|
|  00.01.01.07|     6|   1|                year|
|00.01.01.07.01|    7|   3|                2003|
|  00.01.01.08|     8|   1|               price|
|00.01.01.08.01|    9|   3|               49.99|
|  00.01.01.09|    10|   1|             edition|
|00.01.01.09.01|   11|   3|                   2|
|  00.01.01.10|    12|   1|               pages|
|00.01.01.10.01|   13|   3|                 765|
|     00.01.02|    14|   1|                  cd|
|  00.01.02.01|    15|   1|               title|
|00.01.02.01.01|   16|   3|Love, Lust, Faith...|
|          ...|   ...| ...|                 ...|
|     00.01.03|     1|   1|                book|
|  00.01.03.01|     2|   1|               title|
|00.01.03.01.01|    3|   3|        Learning XML|
|          ...|   ...| ...|                 ...|
+-------------+------+----+--------------------+
```

Figure 5.2: Nodes' table of transformed XML

Note that a type 1 represents an element node and a type 3 represents a text node according to the W3C node representation. However it is not shown in the example above, but the mixed content elements are also supported. Since we need to have one record in one row because of the usability in our second application, a new line character in value string had to be escaped, so `\n` is replaced by `*//n*`. It is certain that escaping string is unique, so it is possible to reconstruct the value to its original state.

The second generated table – the table of paths is usually smaller since one path can address more than one XML node. The Figure 5.3 shows the table containing paths.

```
+------+------------------------------+
|pathId|                          path|
+------+------------------------------+
|     0| bookstore                    |
|     1| bookstore/book               |
|     2| bookstore/book/title         |
|     3| bookstore/book/title/#text   |
|     4| bookstore/book/author        |
|     5| bookstore/book/author/#text  |
|     6| bookstore/book/year          |
|     7| bookstore/book/year/#text    |
|     8| bookstore/book/price         |
|     9| bookstore/book/price/#text   |
|    10| bookstore/book/edition       |
|    11| bookstore/book/edition/#text |
|    12| bookstore/book/pages         |
|    13| bookstore/book/pages/#text   |
|    14| bookstore/cd                 |
|    15| bookstore/cd/title           |
|    16| bookstore/cd/title/#text     |
|   ...| ...                          |
+------+------------------------------+
```

Figure 5.3: Paths' table of transformed XML

Our early approach used Document Object Model paradigm, so while small XML documents were processed, the processing was successful since XML document fits into the memory. A problem occurred during the processing larger XML document that was about 700MB because the whole DOM object did not fit into the memory. It was necessary to use another technique for working with larger XML files, so a different transformation that uses a SAX parser was implemented.

The SAX parser is an event driven parser that is alternative to DOM. Unlike DOM, it does not process an XML document at once. It reads an XML sequentially and allows processing of separated XML shreds.

As it has been mentioned, the XML document is processed in two phases. With the SAX's sequential processing, not the whole document is in memory, so the result of the first phase must be stored into a temporary file and then, the second phase may start.

To run transformation, our application requires two parameters. First is a path to the XML file on local disk. This path is also used to create the directory where the file containing the result table will be stored. A new created file is text file and its original name is extended by suffix *_dewey*.

The second parameter sets a mode, how the application should be run. It has three running strategies. Input parameter set as 0 activates a transformation that uses the SAX parser. It is recommended to use this mode for larger XML documents. It generates a table with structure `Edge(deweyId, type, value)`. *DeweyId* represents a dewey path from the root node of XML document to a single node. In the further work we use term *dewey path* that better expresses meaning of the *deweyId*. Next two modes use DOM, so the processed document must fit into the memory. If parameter is set to 1, a structure of generated rows is the same as in the previous mode. On the other hand, if the second parameter is set to anything else except 0 and 1, rows are stored as `Edge(deweyId, type, value, pathId)` and additionally, the second text file that contains paths to the specific nodes with structure `Path(pathId, path)` is created.

We also decided to keep the transformation using DOM, because by using DOM we implemented the transformation of XML document into the relational table according to the original paper [24] that was followed.

Seeing that each XML node of XML document is extended by the dewey path and the node type, the size of output file is bigger than the size of original XML file. In following Table 5.1 we provide a view to the size comparison of the original XML file and the file containing the result of transformation. The sizes of files do not include the size of table of paths. All mentioned XML and first two text files are available on attached CD.

Table 5.1: 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

In our further work the Edge table without the paths is used. For a testing purpose the XML document was transformed by option 0 and for smaller files by option 1.

5.2 XPath executor

Our second application is a driver program for Spark. We will use the Edge table generated by the previously mentioned transforming application by option 0 or 1. It means that a column *pathId* is not important for our processing seeing that all necessary information are included in the dewey path and *pathIds* can help just with the processing of child axis. However we chose the dewey order encoding due to its sufficient performance in nodes selection and in updating, in this work just a node selection will be implemented. Although the transforming application supports the mixed content nodes, this feature is not supported by XPath executor. Spark provides API in Scala, Java, Python and R. We decided to implement our driver program in Java. Since Spark SQL provides two alternatives of querying data by using SQL queries and/or DataFrame API, we decided to explore them both. In this chapter we describe an implementation of driver program for Spark that is used to accomplish the goals of thesis. We also describe specific problems we had to face during the implementation of Spark's driver program.

5.2.1 Supported XPath features

XPath is a complex query language with many possibilities. In this thesis we decided to focus on XPath axes. Here is a description of our ideas that are a basis for a translation of XPath axes into the SQL queries. The ideas are applied in our driver program. The translation of particular axes is based on the comparison of dewey paths. It is necessary to say that it is a view from the context node. On the beginning of translation of every XPath query the context node is the document root, then context node is the result of the most recently executed XPath step. We use term context node that represents set of resultant nodes after the XPath step evaluation. In the following list the approaches to the various XPath axes are described.

- **Child** – A translation of the child axis means, that nodes with the dewey path greater than dewey paths of the nodes within the context node should be found and returned. A potential child's dewey path starts with one of the node's dewey path from the context node. The dewey path of child node of some node is lexicographically greater and it contains one more path part, such as `00.01.01` is the child element of `00.01`.

- **Descendant** – All nodes whose dewey path is lexicographically greater than dewey paths of the context node are selected. Their dewey path also starts with one of the node's dewey path from the context node. Difference is that descendant nodes may have more than one extra path part.

- **Descendant or self** – Translation of this axis is based on the Descendant axis, but the self nodes are included into the selection. Hence the nodes whose dewey path is lexicographically greater than, or equal to the context node are taken.
- **Following sibling** – Nodes with the dewey path lexicographically greater than the paths of the context node are selected. For following sibling axis is important to select nodes with the same path length as the context node. Dewey paths differ in their last part such as `00.01.01` and `00.01.02`. That also means that they have the same parent.
- **Preceding sibling** – This translation is the same as in the previous case, but firstly, the lexicographically lower nodes are chosen.
- **Following** – Nodes whose dewey path is lexicographically greater than context node are filtered in this case. Common attribute of nodes paths of following axis is that they do not contain dewey paths of context node as a prefix of their paths.
- **Preceding** – Dewey paths of nodes in preceding axis are lexicographically less than those in context node. They are not prefixes of any path of context node and additionally also paths of context node are not prefixes of desired nodes.
- **Self** – It could be naive to translate a self axis by "do nothing". It should be considered that XPath allows filtering based on the node name, in this case just nodes where dewey path equals to the previous chosen nodes – context node are extracted.
- **Parent** – Nodes with dewey path lexicographically lower then paths of context node are chosen in parent axis. Relation between a node and its parent node is the same as in child axis but in inverted meaning. The parent nodes are prefixes for the context node.
- **Ancestor** – All nodes with lexicographically less dewey path than context nodes, and those that are prefixes to the paths from the context node are selected.
- **Ancestor or self** – This axis is similar to the ancestor axis, but firstly, the nodes whose path is lexicographically less than, or equal to context node are chosen.

If a concrete name in the node test part of desired axis is set, firstly, a dataset is filtered by the name and then the filtration through axis is applied.

5.2.2 Reading files

A file containing table of transformed XML document is stored on a disk. If we want to see advantages of Spark we have to run our driver program on a cluster and in the cluster mode. It means that the file should be accessible for Spark driver and all worker nodes. Easier way to do this is usage of Hadoop Distributed File System since it is supported by Spark. Since when the file is accessible for members of Spark cluster it may be read. Firstly we used 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. Spark SQL is working mainly with the DataFrames. DataFrame may be created directly from RDD, but it is necessary to define a schema of 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.

5.2.3 XPath parser

We implemented a simple XPath parser that parses XPath queries which were inputted as a parameter of 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. Also the wildcard `*`, that is an alias for any element node may be used in the queries that are parsed by parser.

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 indirect recursive algorithm. It means that every next step is dependent on the result of previously evaluated step.

This parser is working just with axes that were mentioned in the Chapter 5.2.1 and does not support predicates.

5.2.4 User Defined Functions

Since Spark SQL allows the usage of the User Defined Functions, we created several own functions that are useful in the translations of queries. This functions are used for the better nodes filtering based on the comparison of nodes' relations. In our SQL queries the following user defined functions are used:

- **isChild** – UDF *isChild* checks if two input arguments differ in a length of dewey path (children node has one more path part) and if the second argument includes entire path of its potential parent. If conditions are fulfilled the true value is returned. By changing the order of parameters the user defined function *isChild* may be used for checking whether node is parent node of some other node.
- **isPrefix** – Two dewey paths are taken as arguments to this function. True value is returned if the first argument is lexicographically less than, or equals to the second argument and the first is included within second. We can simply match if the first path is prefix of the second because every dewey path starts with the part containing one or more zeros such as `00.01.02`. Zeros are located always as the first part of dewey path and not at the other position.
- **isSibling** – This user defined function returns true if two inputted dewey paths have the same length and differ just in the last part of their paths. It also means that two parameters belongs under the same parent.

Note that the user defined functions implemented in the broadcast lookup collection method extend the main idea of the functions mentioned above.

5.2.5 Driver program modes

Using the strategy design pattern, three strategies of application run was created. The strategies are characterized in more detail in Chapter 4. In the final driver program three modes are available:

- **SQL** – uses pure SQL that is evaluated directly. This mode is based on the RIGHT JOIN as it was described in Chapter 4.2.
- **DSL with JOIN** – uses a domain specific language of Spark SQL. Translation of XPath steps is implemented by using LEFT SEMI JOIN according to the Chapter 4.5.
- **DSL without JOIN** – uses Spark SQL API to translate XPath query using broadcasted collection as it is mentioned in Chapter 4.6.

The first input parameter of the driver program indicates a path to the text file containing a transformed XML document. The second parameter expects an XPath query that will be evaluated, and the third parameter sets a running mode.

5.2.6 Result of evaluated XPath query

Our first idea was to build a new XML document after the successful evaluation of XPath query over the transformed XML document. Admittedly a newly created document should have been constructed in the document order according to the original XML document.

If we want to build an XML document in a parallel way, it is hardly possible and impractical. Even though each executor has its own partitions and theoretically can build XMLs from them, the final XML document cannot be built because executors do not have information about the nesting of XML elements.

Seeing that some actions called on DataFrame, we consider mainly collect, can require a transfer data from executors to a single driver, it could be a problem when data are larger than available driver's memory. Hence we decided to store results in a way that do not require collecting data to the driver. Accordingly, the result is not stored as a single XML file, but it is stored as a table including schema of stored data.

5.2.6.1 Output formats

Spark offers a possibility to save DataFrame in various formats that keep a schema of data such as Parquet file or JSON. Note that it is not JSON of XML, but it is a JSON of rows from the DataFrame where each line is a valid JSON object [2]. Spark SQL also supports writing data to Apache Hive that is data warehouse software which facilitates querying and managing large datasets residing in distributed storage [17]. Since that DataFrame can be easily converted to the RDD, this ability extends the available output formats by text file and Hadoop SequenceFile.

5.2.6.2 Saving of result

On the beginning of processing text file of transformed XML document, the file is read from a Hadoop Distributed File System because it is easily accessible for members of cluster. The same idea was applied to the storing of result. After the processing, the result is stored back to the HDFS, to the directory *result* that is located in the same directory whence the text file was read. By default the result is not saved into the single file, but into the set of numerous files according to the count of partitions. Implicitly one partition may be set and then result will be stored in one file. During the storing process a method *toString()* is called on each element of DataFrame and elements are stored one by one per line. The result stored in Hadoop Distributed File System may be easily processed again by Spark or by other technologies that support HDFS.

After executing all transformations and actions, the final result is ordered set of XML nodes. This set can contain duplicates. Normally, XPath processors that evaluate XPath queries always return a set of unique nodes and returned nodes can contain other nested nodes. By evaluation of queries that use for example preceding or following axis, the returned nodes can also be nested in some other returned nodes. Reason why we have duplicates is, that we work with a set of self-contained nodes (without any nested nodes). Even though we have duplicates, they will be stored on different levels in potential resulting XML document.

Finally we decided to store result as a JSON file. Even though it is the specially formed JSON it still can be processed by other technologies that do not support other output formats available in Spark.

5.2.6.3 Building XML

While we were working with Spark just locally and we processed just small XML documents we used Document Object Model to build an XML file from the resultant nodes. We implemented methods that transform nodes back to the XML file and keep document order according to the original XML. The first method uses DOM representation and as we mentioned in Chapter 5.1 it is sufficient only when DOM object fits into the memory. So with larger results it fails. We provide an example of the reverse transformation back to the XML file that uses DOM.

This method takes the first node from the result and creates a new element in DOM. Then all its child nodes are selected and assigned to their parent. If some node was already assigned it is deleted from the set of resultant nodes. These steps are repeated for each child until there is no other child node or the set is empty. If there are still some nodes in the resultant set, the algorithm continues and creates following sibling for the first created element.

Since it is easy to rewrite DOM method to sequential manual creation of XML elements, we provide also an algorithm for processing of larger results. The difference is, that the manual creation uses a collection for nodes that are out of the document order and that will be processed later, and each processed node is directly written into the file. In the manual creation the nodes are read sequentially.

Both methods do not require resultant nodes to be in document order, but the resultant set must be sorted. Note that nodes in the document order and sorted nodes are not the same since resultant set can contain duplicates.

5.2.7 Performance improvements

In this section we describe how to speedup Spark's driver program. Spark has various options to be configured to run application faster. They are mostly set via SparkContext or via Spark's submit script. For example, number of driver's and workers' cores and their total memory, maximum size of transported data, maximal size of result, buffer size of Kryo serializer and others may be set to appropriate values. However, the speedup may be also done programmatically by using Spark's features such as broadcast variables, persistence and repartition. In the following sections three methods that could be helpful are described.

5.2.7.1 Shuffle and broadcast variables

Execution of certain operations in Spark such as *join* or *groupBykey* and others can invoke shuffle of data across the network. The shuffle usually causes data transfer across executors which is expensive operation since it is performed as all–to–all operation and it involves data serialization and network I/O. Too much network communication has a negative impact on the performance [25]. Working with two DataFrames where one of them is smaller than the second, and that fits into the memory, the broadcast variable can be useful to avoid the shuffle operation. It is preferable to broadcast smaller DataFrame and use it as an immutable lookup table. In the following Table 5.2 we compare a time of computations with and without using broadcast variable. We tested broadcast variables on the simple join of two tables where one was more than hundred times smaller.

Table 5.2: Performance improvement by using broadcast variable

	Broadcast variable	**Time[s]**
1	True	134.255
2	False	147.193

As can be seen there is a significant difference between usages of broadcast variable.

5.2.7.2 Partitions

In Spark, read data are split into the several partitions that are distributed over executors. An advantage is that the total number of partitions is configurable. Having numerous executors and working with just one partition has no sense. By default Spark splits data at least to the number of partitions which relates to the number of cores that are available on all executor nodes whereby default maximal size of partition is 128MB in a cluster mode and 32MB in a local mode. Very large partitions and on the other hand very small partitions might have bad impact on the performance. If required count of partitions is set to the bigger value than the count of read items, the empty partitions are created, so it also is not a good idea. Count of partitions should be balanced. In Table 5.3 we show how the number of partitions can affect the performance. Some transformations such as join can automatically increase number of partitions when it is needed. Hence Spark provides a coalesce method that decreases the number of partitions and does not shuffle data over the network. We tested various size of partitions on our biggest table by using early version of broadcasted lookup table method that used a join as the last step of algorithm. Table 5.3 shown below presents an impact of number of partition on the performance.

Table 5.3: Performance improvement by repartition

	Partitions count	**Time[s]**
1	17	1163
2	100	744
3	200	713
4	500	841
5	1000	824
5	5000	976

Table 5.3 shows that it has sense to consider number of partitions but it depends on the used transformations and actions. We did one more test in which rows were just filtered and modified, so these operations did not require a transfer data to other executors, since executors have worked only with their partitions. Result of the test showed that in this case it was better to keep a default partitioning.

5.2.7.3 Caching and persistence

RDD and DataFrame as distributed collections in Spark are always computed in the moment when the action is called. So if more actions are called on the same collection, by default it is recomputed repeatedly. From a time point of view it can be costly to repeat the same computation more times. Spark provides a possibility to make a checkpoint for the further computations. It is allowed by persisting RDD and DataFrame into the memory or on a disk. Spark allows several storage levels such as memory, disk, memory and disk (if data do not fit into memory they are persisted to disk) in serialized or not serialized form. Persisting is not invoked by calling applicable method, but it is invoked after the first action that is called on collection. We provide a Table 5.4 containing a time of computations with and without caching. We were testing on 2.1GB file and we used the fastest storage level – memory without serialization. To show the difference, we evaluated longer XPath query.

Table 5.4: Performance improvement by using caching

	Cached	**Time[s]**
1	True	1144
2	False	2178

Caching has the sense within data that are repeatedly used. As we can see from our measurement, when actions are invoked frequently, caching has really positive impact on performance.

5.2.8 Issues

We have started to work with a 1.1.0 version of Spark and our driver program was finally completed in version 1.5.2 released in November 2015, so we had to face up to a version upgrading. The biggest changes were realized between versions 1.2 and 1.3 and they led to the code refactoring. In the following list the changes that had the biggest effect on our application are mentioned.

- SchemaRDD was renamed to DataFrame, it provides the same functionality, but it does not inherit directly from RDD.
- Creation and registration of UDFs were changed.
- Some changes in SQL API where some methods were marked as deprecated and replaced by another method.

We developed our driver program in Java. After some time we realized that it probably was not the best option. Tons of tutorials and also solutions of frequently problems are showed in Scala, sometimes in Python, but rarely in Java. Sometimes it was difficult to translate code from Scala or Python to Java since APIs differ in some details.

5.3 Working with cluster

During the implementation of our third method we got an access to the Spark cluster. From this moment we were able to run our driver program in the cluster mode and see all advantages of parallel computing.

At the beginning of this chapter we showed Figure 5.1 that illustrates local cooperation of two implemented applications. By using a cluster, the cooperation is a bit different since the text file to be processed and the driver program must be available for all members of cluster. The Figure 5.4 shows, how the standalone Spark cluster was used to get the result of XPath query evaluation.

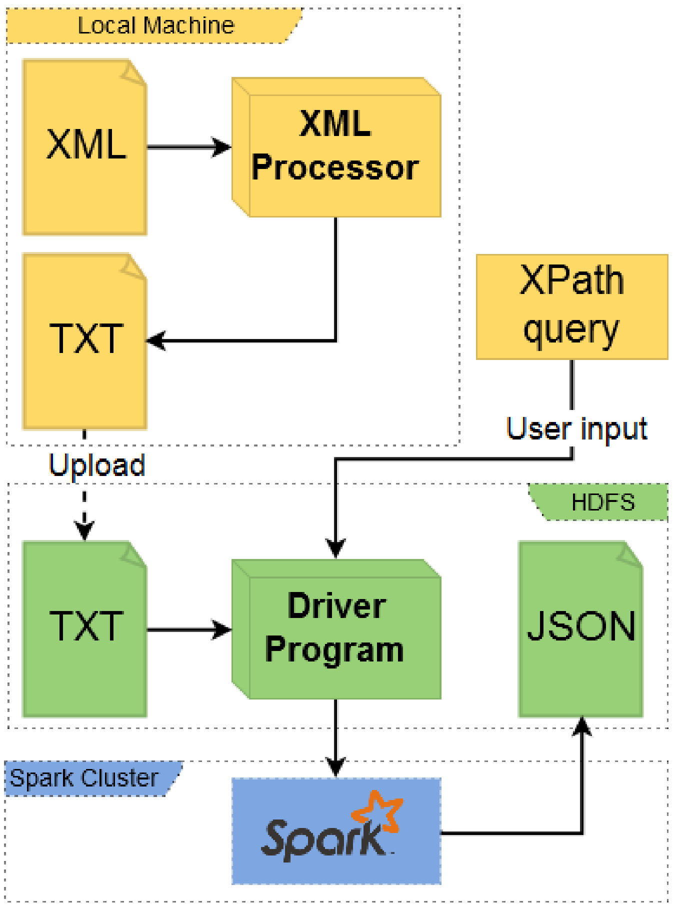

Figure 5.4: Cooperation of applications and standalone Spark cluster

With a small input file it is not possible to see benefits of computation on cluster since in some cases communication load can take more time then computation. The computation on cluster forced us to use the Hadoop Distributed File System to make our text files visible for workers. On cluster we continued our test attempts with the bigger files since sufficient amount of memory was available among the worker nodes.

The parameters of the local testing machine that was used for experiments were described at the beginning of the Chapter 4. The available cluster on that the experiments were run consists 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 32GB DDR2 RAM and 2x 1Gbps LAN. Each virtual machine has allocated 6GB RAM (of which 4,8GB were used by Spark) and 2 CPU cores. Virtual machines were connected via 10Gbps VMXNet3 LAN and installed operating system was Ubuntu 14.04. Version of Spark that was used for the experiments was 1.5.2, so the same as in experiments performed locally.

Comparison of computation in local mode and cluster mode brought expected results. Admittedly computation on cluster with enabled cluster mode was faster in some cases. Table 5.5 shows time comparison of local and cluster mode. In these experiments the fastest method that uses nested lookup collection was used.

Table 5.5: 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

As it may be seen in Table 5.5 the processing of smaller table can be faster when it is done locally. It can be caused by the cluster overhead expenses such as serialization and transporting data among other workers.

A graph in the Figure 5.5 shows measured times from Table 5.5.

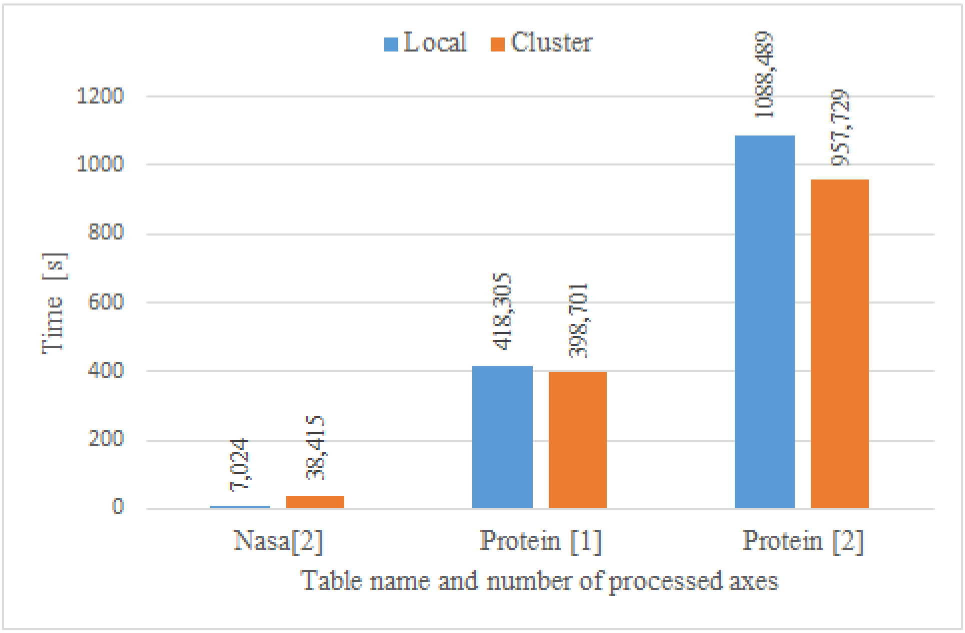

Figure 5.5: Performance of cluster and local mode

5.4 Summary

In this chapter two applications were described. The first application prepares XML documents to the form to be able to be processed by the second application. It was developed according to the analysis in Chapter 3.1 and we showed the resultant tables containing nodes and paths.

The second application is the Spark's driver program and within its final version three methods of the approach were implemented. In this chapter performance of the individual methods was compared. The final interpretation of performance testing is in the Chapter 6.3.

Chapter 6

Testing and Experiments

In this chapter the used test methods are described. Tested attempts are focused mainly on functionality of our solution. Also the results of performance testing of implemented methods are presented in this chapter. Within the functionality testing we decided for manual and unit testing.

6.1 Manual testing

Manual testing is considered as a user testing. During the implementation of driver program we were working with couple of XML documents and we created a numerous XPath queries that cover most of query cases. By hand we tested each step of each XPath query and we aim our attention on the count of returned nodes. We used BaseX in version 8.3.1 and we run same partial queries to get number of returned nodes to the comparison. BaseX is an XML database engine, and XPath and XQuery processor. After that we compared count of returned nodes from our application and from BaseX. Comparisons of counts was not sufficient enough since we did not know if correct nodes were returned. We realized that some automatic tests that compare results of complex XPath query are needed.

6.2 Unit testing

Unit testing is based on the testing of small pieces – units of the application. Although we have the separated functions for the translation of each axis, it is a bit complicated to test them since called transformations are evaluated lazily. The main idea of the unit testing is to test each small unit separately before they are integrated into the bigger units. If smaller units work as it is expected, most probably bigger units will also work.

We follow this idea and we deal with lazy evaluation. Firstly we wrote tests just for functions that evaluate child and descendant axes so if these tests pass we can use them in the other testing queries such as parent, preceding or following axes that cannot be tested from a document node. Note that the document node is an alternative to `doc("xmlFile.xml")` in XPath and it is the first step of each query in our driver program. Then we created a set of XPath queries (different from manual testing) that contains two XPath queries for each axis. The first query has a concrete node test value, for example `/child::book` and the second has an undefined node test such as `/child::*`. The same XPath queries are used in the tests for all the implemented methods.

We again used BaseX to evaluate all the created XPath queries. Then each result of evaluation was transformed to the list of nodes via our XML transformer and they were stored to the test environment. So we had the correct results for each XPath query that we would tested. Finally we could evaluate queries in XPath executor and compare their results with the results from BaseX. We created eleven test cases, one for each axis. In each test two queries are tested as it was mentioned.

It is not needed to test the document order of resultant nodes since the algorithms for the XML file creation presented in the Chapter 5.2.6.3 do not require it. During the implementation, the unit tests helped with a bugs finding. All bugs were solved and all the tests have passed.

6.3 Experiments

In this work, in the Chapter 4 we provide several comparisons and in this section we summarize measured times of different methods. We collected all measured values into the one summarizing Table 6.2. Table 6.1 summarizes tested queries and tested tables.

Table 6.1: 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

All the measurements in the table below were realized locally. The parameters of the local testing machine that was used for experiments were described at the beginning of the Chapter 4.

Table 6.2: 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, it is because it has made no sense since the computation was very slow. The second SEMI JOIN marked with ∗ is method that uses caching. The measuring of the value marked with # was interrupted due to its slowness. Measured values are projected on a graph in Figure 6.1. Numbers in header of table are count of processed axes.

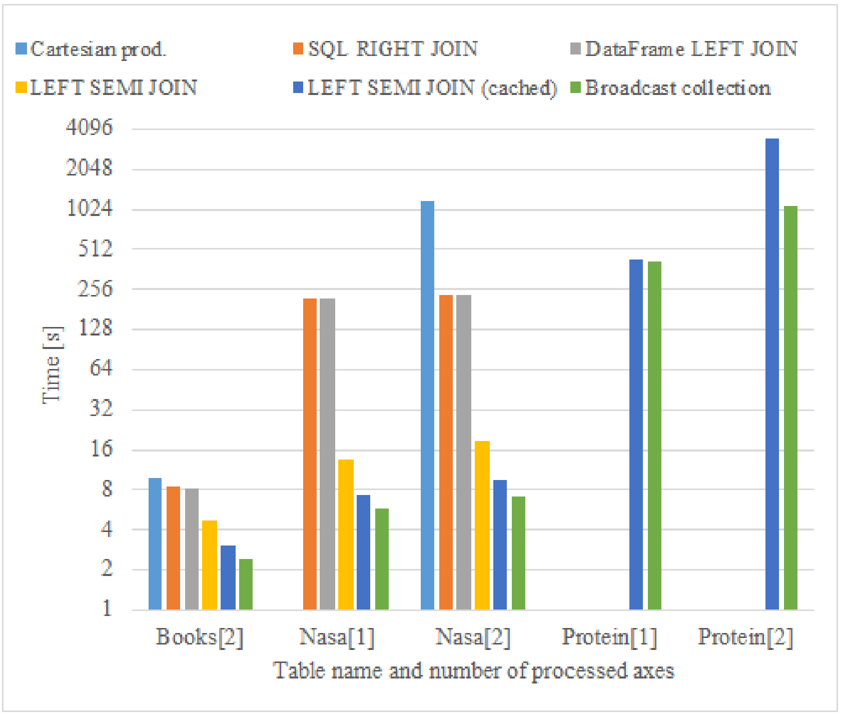

Figure 6.1: Performance of proposed methods

It can be seen that method using Cartesian product is really slow. SQL RIGHT JOIN and DataFrame LEFT JOIN are pulled over the same optimization process 2.3, so if the same physical plan is generated computational time is almost the same. In the graph, there is one more interesting thing. We can see positive impact of caching. According to the Table 6.2, 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, impact of the number of evaluated steps can be seen in the graph. This implies that more XPath steps mean longer computation duration since there is no optimization used, so all steps are evaluated.

6.4 Summary

Functionality of implemented methods was tested by several tests that were separately prepared for each axis. All prepared tests passed so we can suppose that XPath queries are evaluated correctly.

Also the performance of our methods was tested and we found out, that the creation of new methods led to an increased computational speed.

Conclusion

Apache Spark is a new engine that is still in active development. It can be seen on the number of new releases that have been released relatively often during the work on this thesis.

The main goal of this master's thesis was to use Spark SQL framework to implement a subset of expressions from XPath query language. For this purpose two applications were designed and developed. The first takes an XML document and transforms it into the tabular form so it can be used and processed via the second application. The second application is a driver program of Spark.

In this thesis, five different methods of our approach were introduced and also the performance comparison of single methods was presented. However, in the final driver program there are just three methods implemented. It is because one method using Cartesian product was extremely slow, and since it was determined that by executing same query using SQL and DataFrame the same physical plan is generated, just SQL method was implemented.

While working just locally with really small data, also the method using Cartesian product was relatively fast. Working with cluster showed that by processing bigger data, the trivial method have had to be improved.

A subset of expressions from XPath query language that are supported by the implemented methods contains all XPath axes except the axes of attribute and namespace.

A functionality of methods was tested by two independent methods, manually and by unit tests. Several tests were created for single axes. Within implemented methods all tests successfully passed, so the main goal of the thesis was fulfilled.

Sometimes it was not easy to understand why an application on cluster fell down whereas in local mode driver program was successfully executed.

However, thanks to the Spark's web user interface we was able to identify problem. Measurements on cluster shows that for smaller datasets it is better to compute data locally due to big overhead on cluster.

Bibliography

[1] Umbraco. 2016, [Cited 2016-1-8]. Available from: `https://our.umbraco.org/media/upload/0562fd58-c6db-4fa8-a432-68b28f11c3f2/rs/7x1B0.gif`

[2] Spark.apache.org. Apache Spark^{TM}- Lightning-Fast Cluster Computing. 2016, [Cited 2016-1-8]. Available from: `http://spark.apache.org/`

[3] Armbrust, M.; Xin, R. S.; Lian, C.; et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ACM, 2015, pp. 1383–1394.

[4] Bray, T.; Paoli, J.; Sperberg-McQueen, C. M.; et al. Extensible markup language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, volume 16, 1998.

[5] Clark, J.; DeRose, S.; et al. XML path language (XPath) version 1.0. 1999.

[6] Beaulieu, A. *Learning SQL*. O'Reilly Media, 2005, ISBN 9780596552923.

[7] Hamstra, M.; Karau, H.; Zaharia, M.; et al. *Learning Spark: Lightning-Fast Big Data Analytics*. O'Reilly, 2015, ISBN 9781449358624.

[8] Konwinski, A.; Owen, S. Powered By Spark - Spark - Apache Software Foundation. 2015, [Cited 2016-1-9]. Available from: `https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark`

[9] Xin, R. Spark officially sets a new record in large-scale sorting. 2014, [Cited 2016-1-8]. Available from: `https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html`

[10] Zaharia, M.; Chowdhury, M.; Das, T.; et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, USENIX Association, 2012, pp. 2–2.

[11] Zaharia, M.; Chowdhury, M.; Franklin, M. J.; et al. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, 2010, p. 10.

[12] Spark.apache.org. Spark Streaming - Spark 1.5.2 Documentation. 2016, [Cited 2016-1-8]. Available from: `http://spark.apache.org/docs/latest/streaming-programming-guide.html`

[13] Zaharia, M.; Das, T.; Li, H.; et al. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ACM, 2013, pp. 423–438.

[14] Spark.apache.org. MLlib - Spark 1.5.2 Documentation. 2016, [Cited 2016-1-8]. Available from: `http://spark.apache.org/docs/latest/mllib-guide.html`

[15] Xin, R. S.; Crankshaw, D.; Dave, A.; et al. GraphX: Unifying data-parallel and graph-parallel analytics. *arXiv preprint arXiv:1402.2394*, 2014.

[16] Olson, M. MapReduce and Spark - Cloudera VISION. 2013, [Cited 2016-1-9]. Available from: `http://vision.cloudera.com/mapreduce-spark/`

[17] Hive.apache.org. Apache Hive TM. 2016, [Cited 2016-1-7]. Available from: `https://hive.apache.org/`

[18] Spark.apache.org. DataFrame. 2016, [Cited 2016-1-9]. Available from: `http://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/DataFrame.html`

[19] Mesos.apache.org. Apache Mesos. 2016, [Cited 2016-1-9]. Available from: `http://mesos.apache.org/`

[20] Hadoop.apache.org. Welcome to Apache™Hadoop®! 2016, [Cited 2016-1-9]. Available from: `https://hadoop.apache.org/`

[21] Atay, M.; Chebotko, A.; Liu, D.; et al. Efficient schema-based XML-to-Relational data mapping. *Information Systems*, volume 32, no. 3, 2007: pp. 458–476.

[22] 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*, ACM, 2004, pp. 31–38.

[23] 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, 2000. WECWIS 2000. Second International Workshop on*, IEEE, 2000, pp. 134–143.

[24] Tatarinov, I.; Viglas, S. D.; Beyer, K.; et al. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, ACM, 2002, pp. 204–215.

[25] Spark.apache.org. Spark Programming Guide - Spark 1.5.2 Documentation. 2016, [Cited 2016-1-9]. Available from: `http://spark.apache.org/docs/1.5.2/programming-guide.html`

APPENDIX **A**

Acronyms

ANSI	American National Standards Institute
API	Application Programming Interface
CPU	Central Processing Unit
CSV	Comma-Separated Values
DAG	Directed Acyclic Graph
DCL	Data Control Language
DDL	Data Definition Language
DML	Data Manipulation Language
DOM	Document Object Model
DSL	Domain-Specific Language
DTD	Document Type Definition
HDFS	Hadoop Distributed File System
HTML	HyperText Markup Language
JAR	Java Archive
JSON	JavaScript Object Notation
RDBMS	Relational DataBase Management System
RDD	Resilient Distributed Dataset

SAX	Simple API for XML
SGML	Standard Generalized Markup Language
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TCL	Transaction Control Language
TCP	Transmission Control Protocol
UDF	User-Defined Function
UI	User Interface
W3C	World Wide Web Consortium
XML	eXtensible Markup Language
XSL	eXtensible Stylesheet Language
XSLT	eXtensible Stylesheet Language Transformations

Appendix **B**

Contents of enclosed CD

`readme.txt`	the text file with CD contents description
└─ `src`	the directory of source codes
└─ `XML_Processor`	sources of XML processor
└─ `target`	
└─ `XMLProcessor-1.0.jar`	pre-built executable JAR
└─ `XPath_Executor`	sources of XPath executor
└─ `target`	
└─ `XPathExec-1.0.jar`	pre-built executable JAR
└─ `XML`	the directory of processed XML and text files
└─ `thesis`	the directory of LaTeX source codes of the thesis
└─ `thesis.pdf`	the thesis text in PDF format

Appendix C

User manual

This appendix covers a manual for users of two proposed applications. It contains scripts to run applications from command line.

C.1 XML processor

To run the XML document transformation to the text file containing its tabular form use the script as follows:

```
user@ubuntu:~$ java -jar
               ./XMLProcessor.jar
               /home/user/XMLs/bookstore.xml
               0
```

Parameters of XML processor are as follows:

1. Path to the XML document to be transformed
2. Mode (0 - SAX parser; 1 - DOM method; 2 - DOM method with the generation of the table of paths)

After the application evaluation, the resultant text file is stored into the same directory, in this case it is stored as /home/user/XMLs/bookstore_dewey.txt.

C.2 XPath executor

To run driver program on Spark, the Spark's submit script can be used. We provide example of script that was used to run driver program on the cluster in cluster mode. As first, it is needed to upload driver program to the HDFS as follows:

```
user@ubuntu:~$ HADOOP_USER_NAME=hdfs
               ./hadoop-2.6.2/bin/hdfs dfs -put -f
               ./XPathExec-1.0.jar
               hdfs://addressToHDFS:9000/users/rhr/
```

The same script can be used to upload text file containing the transformed XML document to the HDFS.

Then the driver program can be run on cluster as follows:

```
user@ubuntu:~$ ./spark-1.5.2/bin/spark-submit
               --class "dp.xpathexec.Main"
               --master spark://134.109.193.131:6066
               --deploy-mode cluster
               --executor-cores 1
               --executor-memory 2432m
               --driver-memory 4864m
               --driver-cores 2
        hdfs://addressToHDFS:9000/users/rhr/XPathExec-1.0.jar
        hdfs://addressToHDFS:9000/users/rhr/bookstore_dewey.txt
        /bookstore/book/author
        0
```

Parameters of driver program are as follows:

1. Path to the text file containing transformed XML document
2. XPath query
3. Mode (0 - DSL Broadcast collection; 1 - DSL LEFT SEMI JOIN; 2 - SQL RIGHT JOIN)

Directory containing the resultant files is stored into the same directory whence the text file was read, so in this case result is stored into hdfs://addressToHDFS:9000/users/rhr/result. Note that in local mode, if the file is read locally, the result is stored to the local machine.

XPath executor was created as Maven project. If it is needed it can be reinstalled, recompiled or rebuilt by Maven commands.

Appendix **D**

Tables of transformed XML document

This appendix contains full nodes' and paths' table of transformed XML document from Chapter 1.1. Tables are partially shown in Section 5.1.

Nodes' 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|
|   00.01.01.02|     4|   1|              author|
|00.01.01.02.01|     5|   3|      James McGovern|
|   00.01.01.03|     4|   1|              author|
|00.01.01.03.01|     5|   3|         Per Bothner|
|   00.01.01.04|     4|   1|              author|
|00.01.01.04.01|     5|   3|          Kurt Cagle|
|   00.01.01.05|     4|   1|              author|
|00.01.01.05.01|     5|   3|          James Linn|
|   00.01.01.06|     4|   1|              author|
|00.01.01.06.01|     5|   3|Vaidyanathan Naga...|
|   00.01.01.07|     6|   1|                year|
|00.01.01.07.01|     7|   3|                2003|
|   00.01.01.08|     8|   1|               price|
|00.01.01.08.01|     9|   3|               49.99|
|   00.01.01.09|    10|   1|             edition|
|00.01.01.09.01|    11|   3|                   2|
|   00.01.01.10|    12|   1|               pages|
|00.01.01.10.01|    13|   3|                 765|
|      00.01.02|    14|   1|                  cd|
|   00.01.02.01|    15|   1|               title|
|00.01.02.01.01|    16|   3|Love, Lust, Faith...|
|   00.01.02.02|    17|   1|              author|
|00.01.02.02.01|    18|   3|  30 Seconds to Mars|
|   00.01.02.03|    19|   1|                year|
|00.01.02.03.01|    20|   3|                2013|
|   00.01.02.04|    21|   1|               price|
|00.01.02.04.01|    22|   3|               25.55|
|   00.01.02.05|    23|   1|               genre|
|00.01.02.05.01|    24|   3|    Alternative rock|
|      00.01.03|     1|   1|                book|
|   00.01.03.01|     2|   1|               title|
|00.01.03.01.01|     3|   3|        Learning XML|
|   00.01.03.02|     4|   1|              author|
|00.01.03.02.01|     5|   3|         Erik T. Ray|
|   00.01.03.03|     6|   1|                year|
|00.01.03.03.01|     7|   3|                2003|
|   00.01.03.04|     8|   1|               price|
|00.01.03.04.01|     9|   3|               39.95|
+--------------+------+----+--------------------+
```

Paths' table

```
+------+------------------------------+
|pathId|                          path|
+------+------------------------------+
|     0| bookstore                    |
|     1| bookstore/book               |
|     2| bookstore/book/title         |
|     3| bookstore/book/title/#text   |
|     4| bookstore/book/author        |
|     5| bookstore/book/author/#text  |
|     6| bookstore/book/year          |
|     7| bookstore/book/year/#text    |
|     8| bookstore/book/price         |
|     9| bookstore/book/price/#text   |
|     10| bookstore/book/edition        |
|     11| bookstore/book/edition/#text |
|     12| bookstore/book/pages          |
|     13| bookstore/book/pages/#text    |
|     14| bookstore/cd                  |
|     15| bookstore/cd/title            |
|     16| bookstore/cd/title/#text      |
|     17| bookstore/cd/author           |
|     18| bookstore/cd/author/#text     |
|     19| bookstore/cd/year             |
|     20| bookstore/cd/year/#text       |
|     21| bookstore/cd/price            |
|     22| bookstore/cd/price/#text      |
|     23| bookstore/cd/genre            |
|     24| bookstore/cd/genre/#text      |
+------+------------------------------+
```