XML Parsers - A comparative study with respect to adaptability

Bachelor Degree Project in Information Technology
Basic level 30 ECTS
Spring term 2018

Johan Holm
Mats Gustavsson

Supervisor: Yacine Atif
Examinator: Alan Said
Abstract

Data migration is common as information needs to be moved and transformed between services and applications. Performance in the context of speed is important and may have a crucial impact on the handling of data. Information can be sent in varying formats and XML is one of the more commonly used. The information that is sent can change in structure from time to time and these changes need to be handled. The parsers’ ability to handle these changes are described as the property “adaptability”.

The transformation of XML files is done with the use of parsing techniques. The parsing techniques have different approaches, for example event-based or memory-based. Each approach has its pros and cons. The aim of this study is to research how three different parsers handle parsing XML documents with varying structures in the context of performance.

The chosen parsing techniques are SAX, DOM and VTD. SAX uses an event-based approach while DOM and VTD uses a memory-based. Implementation of the parsers have been made with the purpose to extract information from XML documents an adding it to an ArrayList.

The results from this study show that the parsers differ in performance, where DOM overall is the slowest and SAX and VTD perform somewhat equal. Although there are differences in the performance between the parsers depending on what changes are made to the XML document.

**Keywords:** XML document, Parsing, Benchmark, Adaptability
Acknowledgement

We would like to thank our supervisor Yacine Atif for the help and support during the writing of this thesis, it would not have been possible without him. We also would like to thank Asitis AB for the support and the opportunity to discuss data migration with developers working with financial data.
Table of contents

1 Introduction ........................................................................................................ 1

2 Background and related work .......................................................................... 2
  2.1 Data migration .......................................................................................................... 2
  2.1.1 Databases ....................................................................................................................... 2
  2.1.2 SQL ................................................................................................................................. 2
  2.1.3 Migration .......................................................................................................................... 3
  2.2 XML to SQL transformation ...................................................................................... 3
  2.2.1 DOM Parsing ................................................................................................................... 3
  2.2.2 SAX Parsing .................................................................................................................... 4
  2.2.3 VTD-XML Parsing ........................................................................................................... 4
  2.3 XML for Software Engineering .................................................................................. 5
  2.4 Related Work ............................................................................................................ 6

3 Problem formulation .......................................................................................... 7
  3.1 Aim ........................................................................................................................... 7
  3.2 Motivation ................................................................................................................. 7
  3.3 Research questions and hypothesis ......................................................................... 8
  3.4 Objectives ................................................................................................................. 8
  3.5 Work contribution ...................................................................................................... 8
  3.6 Methodology ............................................................................................................. 8
    3.6.1 Experiment ...................................................................................................................... 8
    3.6.2 Alternative methods ........................................................................................................ 9
    3.6.3 Selection of parsers ........................................................................................................ 9
    3.6.4 Validity threats .............................................................................................................. 10

4 Implementation ................................................................................................ 11
  4.1 Experiment Design ................................................................................................. 11
    4.1.1 Dependent variables ..................................................................................................... 11
  4.2 Experiment implementation .................................................................................... 12
    4.2.1 SAX parser .................................................................................................................... 12
    4.2.2 DOM parser ................................................................................................................... 13
    4.2.3 VTD parser .................................................................................................................... 14
    4.2.4 XML document generator .............................................................................................. 15
    4.2.5 Data Collection .............................................................................................................. 16

5 Results .............................................................................................................. 17
  5.1 Evaluation ............................................................................................................... 17
  5.2 Analysis .................................................................................................................. 19
  5.3 Discussion .............................................................................................................. 21

6 Conclusion ....................................................................................................... 23
  6.1 Future Work ........................................................................................................... 23
  6.2 Ethical aspects ........................................................................................................ 23

References .............................................................................................................. 24
1 Introduction

XML, eXtensible Markup Language, is used to describe the logical structure of documents. The document can be viewed as a tree of elements, which contain information (Bray et al., 2006). Companies working with financial data can receive data in form of XML and stores this data in SQL, Structured Query Language, databases. The data can be invoices, customer data or account ledgers. This results in that the company must be able to handle different structures of XML documents.

Information needs to be sent in a format that is structured so it can be read by other applications and be able to be read fast as a lot of documents are being sent. Migrating data from XML to a database requires transformation of the data. By using a parsing methodology, the data can be transformed into the desired data format. There are different parsing methodologies that work in varying ways, leading to some parsing methodologies being faster to parse specific kind of structured data.

A document can be extracted into or from a database. However, this operation is expected to be done without the need to perform extensive changes or performance implications to the original applications using the information stored in XML document. This property is referred to as "adaptability". This thesis addresses the adaptability issue in software development and investigates approaches to process XML data.

The three approaches, DOM, SAX and VTD, are investigated experimentally. These three approaches have diverse ways to access and extract the information from the XML document. DOM builds an object model, and from this object model the information is extracted. SAX is an event-based method that reads the document from start to finish and acts on defined events. VTD uses tokens to access the XML document that is present in the memory as a complete document.

The approach is split up into 4 experiments, each experiment testing different structures of XML documents. Each experiment measures the execution time of reading and migrating the data into an ArrayList in Java. The data is presented on a line-chart together with an analysis of the result.
2 Background and related work

This section explains the background of XML parsing and data migration. An explanation of the three chosen XML parsers for the research are given and the software engineering aspect is presented on how it can be used to evaluate parsers. Finally, related work is brought up, discussing previous work done in the field of XML parsing.

2.1 Data migration

2.1.1 Databases

Relational databases are collections of related data. Elmasri and Navathe, (2010) refers to data as facts that can be recorded and have an explicit meaning. As an example, they describe data in the form of an address book where names, addresses and telephone numbers are recorded and stored on a storage media using software. This collection of related data can implicitly be viewed as a database. This example of database is quite general, and they further describe three implicit properties for a database:

- It represents some aspect of the real world. This world can sometimes be called a miniworld or the universe of discourse (UoD). When changes are made to this miniworld, the changes are reflected in the database.
- It is a logically coherent collection of data with some inherent meaning. If data is randomly assorted, this cannot correctly be referred to as a database.
- It is designed, built, and populated with data for a specific purpose. The database is intended for a group of users and applications which the users are interested in.

In summary a database has a source to derive data from, some degree of interaction with real world events and an audience that are interested in its content.

An example of database usage is the storage of invoice data. The invoice data could be tabular, non-recursive and contain information like sender, receiver, date and invoice amount registered in a company’s invoice system. Information in the database would represent the real-world debt in a miniworld context. Transactions would alter the data, like partial payment that would reduce the invoice amount.

2.1.2 SQL

SQL is the abbreviation of structured query language and is the standard language for commercial relational database-management systems (DBMs) (Elmasri and Navathe, 2010). SQL is used to communicate with databases, making it possible to manipulate, retrieve and store data in the databases. There are multiple different DBMs that use SQL as a base but add extensions on top of SQL to add more functionality, for example MySQL, Oracle and Microsoft SQL. Even though they add more functionality, the basic functions Select, Insert, Update, Delete, Create and Drop are still enough to accomplish most of the tasks needed in a database.
2.1.3 Migration
According to Morris, (2006) the definition of data migration is as follows:

Data migration is the selection, preparation, extraction, transformation and permanent movement of appropriate data that is of the right quality to the right place at the right time and the decommissioning of legacy data stores.

(Morris, 2006, s. 7)

As data needs to be moved from different logical and physical locations, migration is performed as a part of a movement process. Yeddula, Das and Reddy (2015) describes that the process starts with a business analyst identifying mappings between source and target data model. When the mappings are identified, an implementation-team translates the mappings into migration programs. The business analysts specify the mappings as conceptual data models that reflect the domain semantics. The programmers must then write the physical database schemas based on the analyst’s conceptual models.

Further Yeddula, Das and Reddy (2015) describes this as an error prone process where the descriptions generally are informal sketchy textual descriptions.

2.2 XML to SQL transformation
XML is the abbreviation of eXtensible Markup Language, a way to standardize the representation of data used for storing and exchanging data. XML is a format created to allow users to define their own set of markup tags that can be adapted after the type of data that is used. The goals of XML are to create a data format that is formal and concise, straightforward, human-legible and machine-readable language (Bray et al., 2006).

Migrating data from XML to SQL requires transformation of the data. By using a parsing methodology, the user can define specific rules on how to handle the data in the transformation. These rules form a parser that connects the markup tags in XML to a table in the SQL database.

2.2.1 DOM Parsing
DOM stands for Document Object Model and is an Application programming interface(API) used for parsing XML documents. The definition is “”The DOM” is an API for accessing and manipulating documents (in particular, HTML and XML Documents).” (Dom Living Standard, 2018).

To extract information from the XML document, a tree of nodes is created in memory by parsing the entire XML document. The tree has a single root node without any parent nodes. All other nodes have a parent node and a list of child nodes. If the list of child nodes is empty, the node is known as a leaf node. A visual explanation is shown in Figure 1 where the tag A contains the information “text” and the empty self-closing tag (Brownell, 2002). After the tree has been created the DOM API navigates over the nodes to extract information stored in each node.
An advantage to DOM over SAX is that DOM permits random access of the InformationSet items because the entire XML document is stored in memory, whereas SAX only permits serial access. DOM is also able to create XML documents whereas SAX is only able to parse documents (Dom Living Standard, 2018).

### 2.2.2 SAX Parsing

Simple API for XML (SAX) is an event-based API for XML parsing (Friesen, 2016). The XML-document is parsed sequentially from start to finish with SAX. By calling a method from the parser the item is made available to the application. A visual explanation is shown in Figure 2 (Brownell, 2002). SAX is more memory efficient than DOM because it does not store the entire document to the memory, just the item that is called by the event.

Because the complete document does not need to be stored in memory before usage, SAX is appropriate for streaming applications. The applications can thus start to process the document from the beginning (Li, 2017). Another advantage prior to DOM is that SAX can handle documents of arbitrary sizes, whereas DOM needs to have available memory to store the entire document (Friesen, 2016).

### 2.2.3 VTD-XML Parsing

Virtual Token Descriptor for XML is a non-extractive document-centric parsing technique used for parsing XML documents developed by XimpleWare, (Zhang, 2008).

Tokens are a crucial part of VTD, as VTD uses tokens as a way to modify and retrieve information from XML documents. To create these tokens a program called lexical analyzer or tokenizer is used. A tokenizer converts input source code character by character, in this case the XML documents, to create tokens and lexemes. Lexeme is an identifiable sequence of characters, tokens are objects used to describe the lexeme and can contain information where the lexeme can be found in the source code.
Instead of representing the XML document with tokens as discrete string objects, VTD keeps the source text intact and uses a non-extractive tokenization, meaning that the tokens are described by offsets and lengths. Offsets and lengths is enough for describing tokens in unstructured text documents, but for XML documents more information is needed. Because of this, VTD records are used. A Record is a 64-bit integer that encodes offset, length, type and nesting depth of a token in XML. Due to the records being of the same bit length they can be stored in chunk-based buffers, resulting in records being associated with each other by a natural adjacency. Meaning that the records follows the same structure in the stored chunks as in the XML document.

One of the advantages of using VTD as a parser is its low memory usage compared to DOM. As a result of per-object allocation used in DOM, each object will have a small amount of overhead resulting in more memory needed to be allocated. Due to VTD records being integers, overhead is not a problem. VTD records are also of fixed length leading to records able to be stored in array-like memory chunks. Resulting in greatly reduced overhead per-record (The Future of XML Processing, 2018-04-20).

2.3 XML for Software Engineering

One definition of quality is the ability of a product, service, system, component, or process to meet customer or user needs, expectations, or requirements (ISO/IEC/IEEE, 2010). The ISO/IEC 25010:2011 (2011) standard provides guidelines for software product quality. There are eight quality characteristics in this ISO standard: Functional suitability, Reliability, Performance efficiency, Usability, Security, Maintainability, Portability, Compatibility.

Wagner (2013) describes three of the characteristics from the view of the developers of the software. These characteristics are maintainability, portability and compatibility. This thesis concerns the characteristic portability.

As Wagner describes portability, it is important in cases where the software needs to be moved to new or further platforms. Portability describes how strongly tied the software is to the platform. He describes the importance of portability as differing, because the software might be built for a very specific platform with no intention of movement to another platform.

To be able to evaluate each parsing methodology, a testing parameter needs to be chosen. Adaptability has been chosen from the ISO/IEC 25010:2011 standard and will be used to assess if each parsing methodology can meet the requirements of the parameter. Due to XML documents having varying structures, parsers need to be able to adapt to these differences. As specified in ISO/IEC 25010:2011 (2011), adaptability is one of the sub-characteristics of portability. The ISO standards definition of portability is: "degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments”.

As invoice XML documents can differ in information, the method needs to handle these differences. An invoice document can contain information about sender, receiver, amount and due date. If a new parameter is added to the invoice, like added GLN number, the method needs to be changed. The XML documents can also vary in form of the structure with different depths of tags, number of table entries and content.
2.4 Related Work

Earlier research comparing different XML parsers and parsing methodologies has been done. To the authors knowledge, comparison from the software engineering perspective adaptability has not been researched. The identified comparative studies regarding XML parsing are from the perspective of performance with parameters such as speed, accuracy and associated overhead costs.

One of these earlier studies that has been interesting for this work compares four different parsers, two of them being the same used in this thesis, SAX and DOM. Deshmukh and Bamnote, (2015) conducts an empirical analysis on the parsers DOM, SAX, Pull Parser and VTD for an android based application. The aim is explained as “investigate the available parsers for android platform and evaluate the computational time” (Deshmukh and Bamnote, 2015). They also propose Structured Recurrence DOM (SRDOM). Their result show that their proposed SRDOM implementation is 9 times faster than DOM in the presence of redundant structure. The parsers are compared against two different applications. In the first application a performance evaluation is conducted over Android mobile platform. Where each parser is given four different documents of various sizes. The second application is regarding XML Web Services in building and integrating real-time business systems for Banking solutions.

The conclusion presented in the study by Deshmukh and Bamnote, (2015) is that VTD outshines DOM in performance when comparing parsing time and memory usage. However, when structured recurrence is applied, the performance of SRDOM is improved significantly. The second application showed that DOM is a suitable choice for database application even though it is more memory intensive then the other parsers. This conclusion has been used as a part of the foundation for the hypothesis that is presented in this thesis.

Oliveira, Santos and Belo (2013) study about performance of Java APIs for XML processing several parsing methodologies are benchmarked. They are comparing both streaming-based API’s like SAX and StAX as well as memory-based API’s like DOM, XOM and VTD. They are measuring memory consumption and execution time for the memory-based API’s, where VTD outperforms the others. Execution time is measured for the streaming-based API’s, where SAX and StAX perform similar and better than XOM. As only memory-based API’s can manipulate elements of an XML document, a comparison between DOM and VTD is performed. In this experiment, execution time when adding, renaming and replacing elements are measured. As a conclusion they list the pros and cons for each kind of API. Although they state that VTD is generally the best model with respect to memory consumption and processing time.

Haw and Rao, (2007) have made a comparative study, benchmarking throughput and parsing time on the XML parsers, Xerces, .NET and xParse. The parsing API’s that were used are SAX, DOM, StAX and Electric XML. They describe that choosing the right parser for an organization’s respective system is crucial and critical. The use of an improper parser will lead to degradation in performance and a decrease in productivity. The inferred motivation is that their study will shed light on the alternatives to choose from.

Their result show that Xerces have better throughput for both SAX and DOM and better parsing time for DOM compared to .NET. The conclusion they draw is that the study indicates that Xerces is the best performing parser although xParse outperformed in terms of supporting large-scale of dataset efficiently. As future work they suggest benchmarking new and established API’s together in a set of benchmarking tools and performance testing the ability of XSLT.
3 Problem formulation

Migrating data from XML to SQL requires transformation of data. By using a parsing methodology, the user can define specific rules on how to handle the data in the transformation. These rules form a parser that connects the markup tags in XML to a table in the SQL database.

Financial companies may receive tabular non-recursive data in the form of XML according to XML financial standards (Treleaven, 2015). The data can be customer related information, like invoices, that needs to be parsed into an SQL-database. The data can vary in structure and because of this, choosing a parser that can adapt to different structures can be challenging.

3.1 Aim

This thesis aims to compare three parsing methodologies on how they handle transforming tabular non-recursive XML data, to be ready for insertion in a SQL database in the context of adaptability. Adaptability is a part of the ISO/IEC 25010:2011 - Systems and software engineering (ISO/IEC, 2011) and in this thesis it is defined as the change in performance when parsing XML documents with varying values for the independent variables.

3.2 Motivation

The motivation for this thesis is to create a guideline for choosing a parsing methodology that handles tabular non-recursive data transformation from XML to SQL. It is also to clarify the difference between the parsing methodologies ability to handle adaptability.

Received XML documents can be structured in diverse ways, resulting in the need to be able to handle different structures of the document. The data may also vary in number of columns and size of the information. A parser needs to handle these variations and transform the data into its correct format. Having a parsing methodology that can handle these different structures becomes important.

As XML is widely used, being able to understand the advantages and disadvantages of each parsing methodology is essential for people working with data transformation. It is crucial to know the characteristics of each parsing methodology, so that the right parsing methodology can be matched to the migration terms. That is why studies comparing different parameters between parsing methodologies are important.

Earlier research has been done on comparison of XML parsers, for example by Oliveira, Santos and Belo, (2013), Karre and Elbaum, (2002) and Haw and Rao, (2007). Although, they focus on performance and lack research about adaptability aspects. Oliveira, Santos and Belo (2013) have made research about XML parsers’ performance on memory consumption and execution time. This thesis also studies the performance in terms of execution times. The focus is however primarily on the changes to the XML documents that can indicate the parsing methodologies ability to handle adaptability.
3.3 Research questions and hypothesis

The research questions are:

1. How is the parser’s performance affected when XML document structure and data changes?
2. Which parsing approach better supports adaptability in software development?

By answering the research questions, this study can be used as a foundation for a decision when choosing the right parser methodology, regarding adaptability, for companies and researchers. Based on the research question our hypothesis is that “DOM, SAX and VTD will not perform similarly when comparing adaptability between parsers”.

3.4 Objectives

1. Set up benchmark environment and criteria
2. Build XML generator to support experimental studies
3. Evaluate SAX, DOM and VTD parsers
   a. SAX
   b. DOM
   c. VTD
4. Benchmark parsers
5. Analyze and present results

3.5 Work contribution

In Table 1 the distributed work contribution is presented.

<table>
<thead>
<tr>
<th>Objective</th>
<th>Contributor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mats &amp; Johan</td>
</tr>
<tr>
<td>2</td>
<td>Mats</td>
</tr>
<tr>
<td>3a</td>
<td>Mats</td>
</tr>
<tr>
<td>3b</td>
<td>Johan</td>
</tr>
<tr>
<td>3c</td>
<td>Johan</td>
</tr>
<tr>
<td>4</td>
<td>Mats &amp; Johan</td>
</tr>
<tr>
<td>5</td>
<td>Mats &amp; Johan</td>
</tr>
</tbody>
</table>

Table 1 Work contribution

3.6 Methodology

This subsection describes the method chosen for the study, an experiment. Alternative methods are explained and the reasoning behind selecting the three parsers. Lastly identified validity threats to the experiment are discussed on how they could affect the study.

3.6.1 Experiment

The methodological strategy is to perform an experiment. The purpose of the experiment is to evaluate the adaptability for each parsing methodology. This is achieved by gathering quantitative data from the experiments and configurations with varying amounts of columns, number of entries, numbers of characters and structure depth of the XML documents. An
experiment is fitting as control over the study and the ability to manipulate directly is desirable (Wohlin et al., 2012). The study is also about exploring relationships, how altering variables in the sample data would affect the outcome. This is one of the aspects that Wohlin et al., (2012) mentions as appropriate to investigate by conducting an experiment.

The experiment will observe the dependent variable transformation time. As an addition, the complexity of the parsing methodology will also be assessed. The independent variables are the number of columns, tables, characters and depth structure of the XML document.

The approach is to first create an XML generator and by this achieve control over the data samples. The XML documents will then be parsed and transformed to strings by each parsing methodology. The execution time will be measured from the time when calling the parsing method to when all data is added to an ArrayList. The degree of adaptability can be observed by benchmarking and measure the parsers’ performance with different configurations for the independent variable.

3.6.2 Alternative methods

As alternative methods a case study or a literature study could be used. A case study would be observational of a real-world scenario and a phenomena in its context (Wohlin et al., 2012). This means that a real-world setup of different parsing methodologies should be observed in the context of adaptability. It probably would be difficult to find isolated scenarios where the chosen parsing methodologies are used with fitting variations of the documents to observe adaptability.

A systematic literature review could also be carried out. By identifying, evaluating and interpreting available research about parsing methodologies and the software engineering aspect adaptability, knowledge can be gained to answer research questions (Kitchenham, 2004). An extensive amount of time would surely be needed to get enough domain knowledge to draw the right conclusions. Budgen and Brereton, (2006) mentions that their experiences generally are positive about the use of systematic review although there are some issues: Poor quality of search engines and the variable quality of the abstracts available for Software Engineering papers.

As a third possibility a survey could be conducted. The participants in the study must in that case be well known with the domain and have specific knowledge concerning the software engineering aspect. The chance of finding a selection sample big enough in this limited time and at this location is probably hard to achieve.

3.6.3 Selection of parsers

The selected parsers in the experiment are SAX, DOM and VTD. These parsing methodologies are chosen because they differ in concept as they are either memory-based or streaming-based. Though both DOM and VTD are memory-based they differentiate on how they access and retrieve the information from the XML document.

The differences between the parsers concept are interesting and the result may in some way be generalized to each concept. There are several different parsers, but SAX and DOM are well documented and as these two parsers use different approaches they broad the field of the study. VTD are in two of the related works, Oliveira, Santos and Belo, (2013) and Deshmukh and Bamnote, (2015), outperforming both DOM and SAX in performance. It would be interesting to investigate the parsers’ ability to adapt to changes of an XML document.
3.6.4 Validity threats

There are validity threats to the study that can be addressed as existing and some methods to avoid them.

Reliability of treatment implementation is present. The aim of the experiment is to measure the execution time between the calling of the parsing method until the data from the XML document is inserted into an ArrayList. There might be implementation depending factors that either slowdown or speed-up the process. Conformity has been strived for and the implementation have been stripped down to what seems to be on a level that only performs the needed actions.

As Wohlin et al., (2012) describes that there is a risk that the implementation may be different depending on the persons applying the treatment. Because this study needed to have the objectives split up between the authors, the parsing methodologies are implemented by different persons. Even though the implementations are carried out by different persons the authors have taken and given regular input to each other. Wohlin et al., (2012) mentions that one way to handle this threat is to strive for standard solutions and the input from both person might be a way to achieve this.

There are also external validity threats to the study, an example is the XML document could have a structure that does not reflect a real-world scenario. Also, the implementations of the parsing methodologies might not be suitable in a real-world scenario. This might lead to the result not being generalizable. This is what Wohlin et al., (2012) describes as interaction of selection and treatment. The goal of the parser implementations is to extract data from an XML document and insert the data into an ArrayList. By measuring the execution until this step and not follow with the insertions into a database, there are fewer factors that intervene.

A One-way ANOVA test shows the variance between and within the groups. If there is more variance between groups than within groups, then this would be evidence against believing that they were the same (Bingham and Fry, 2010). By performing an ANOVA test and evaluate the confidence intervals, a degree of assurance that the results are generalizable can be achieved.

Another threat to external validity are interaction of setting and treatment. Wohlin et al., (2012) describes this as the effect of not having an experimental setting that is representative. The experiment has been carried out on a consumer laptop, an Apple MacBook Pro. As the delicate time measured easily can be skewed by other processes running on the computer, this is a validity threat. An experimental setting that have very little impact on the experiment have been strived for. To achieve this, intervening processes have been shut down and each measurement have been run separately.

To ensure that the measurements are valid, each document is averaged from 10 runs. The mean of these 10 executions are used as the outcome for the current configuration. This approach can reduce the impact of possible outliers and improves the level of certainty for the measures. With the use of an ANOVA test, the significance of the measured value can be further achieved.
4 Implementation

This section explains how the experiment was designed and the approach to conduct the experiment. The data collection for the experiment is detailed on how it was generated and configured for the experiments. The experiment environment used a MacBookPro11,5 with an intel core i7 running at 2.8Ghz, L3-cache: 6MB, 16Gb of DDR3 1600Mhz RAM, and macOS Sierra version 10.12.6 as operating system. The parsers are built with Java JDK 8.1.

4.1 Experiment Design

We conducted an experiment to evaluate if the parsers’ performance changes according to different configurations of XML documents. The experiment is carried out to measure the dependent variable, execution time. The execution time is defined as the time it takes to instantiate the parser, parse the file and extract the information to an ArrayList, with the tag name and its corresponding data as a pair.

The approach is to conduct four experiments, each containing configuration changes like number of columns, depth of the XML document, number of characters used as data and amount of entries. By conducting an experiment on the parsers’ ability to handle changes to these attributes, the parsers’ strengths and weaknesses may be observed.

For each configuration 10 document are created, each document is parsed 10 times. The mean time for parsing each configuration is then computed. The mean time of the parsers will then be compared against each other to evaluate which one is the fastest to parse the data from the XML document into an ArrayList.

The experiment is executed in the following order.

Measuring execution time with increasing amount of:

1. Columns
2. Entries of tables
3. Characters
4. Depth of document

Each parser was implemented with the only purpose to extract the data on the deepest level of the XML structure.

4.1.1 Independent variables

The dependent variable, execution time, is measured by getting the difference in system time, from the start of calling the parsing method to the end of insertions to the ArrayList. Each XML document is parsed 10 times and the mean time of the 10 executions are logged. To be able to gather knowledge about the parsers’ ability to handle adaptability, ten different document structures with an increasing amount of the independent variable is executed.

The independent variables, used to investigate the parsers property adaptability, are the increasing number of columns, tables, character and depth.
4.2 Experiment implementation

To make the measurement of execution time for each parser as equal as possible, all parsers read the same XML documents and store the data the same way. An ArrayList was chosen for storing the parsed data, the ArrayList contains the Java object Pair<K, V>. K standing for Key, stores the name of the column in the XML document. V meaning Value, stores the data for that column.

4.2.1 SAX parser

The implementation of SAX is designed to read the XML document and as it reaches the given depth an event is triggered. The event is to extract the node name and the containing information. The name and information are added to an ArrayList as a pair. When the parser reaches the end of the document the parsing is considered done.

Each document is iterated 10 times. The start time is defined, and instances of the SAX parsing objects are created. The file and the UserHandler object, that handles the events, are passed as arguments to the parsing method. After the parsing method the stop time is defined and average times for the document and configuration are computed.

The UserHandler of SAX extends the DefaultHandler for events. The first method to execute is startDocument that is called at the first tag of the document. The static ArrayList pairs is cleared to remove any pairs from an eventual document that has been executed before the actual execution.

At each opening tag the method startElement, Figure 3, is called. The class variable counter is incremented at each call to keep track of the depth of the current element. If the depth equals the defined depth of the document, where the information is located, the stringBuilder is set to 0. This is carried out to clear any remaining information from the previous element.

```java
@override
class UserHandler {
    static ArrayList<Pair<String, String>> pairs;

    @Override
    public void startElement(String uri, String localName, String qName, Attributes attributes) {
        counter++;
        if (counter == main.depth) {
            stringBuilder.setLength(0);
        }
    }
}
```

Figure 3 SAX code, startElement

After the execution of startElement, the method characters, Figure 4, is called. If at the correct depth, the information in the element is appended to the stringBuilder.

```java
@Override
public void characters(char[] buffer, int start, int length) {
    if (counter == main.depth) {
        stringBuilder.append(new String(buffer, start, length));
    }
}
```

Figure 4 SAX code, characters

At each closing tag the method endElement, Figure 5, is called. If the element is at the defined depth a pair, containing the element name as key and the information in the element as value, is created. At the end of the method the variable counter is decremented. This pair is then inserted into the ArrayList. When all elements in the document have been parsed the execution of the userHandler is complete.
The methods startElement, characters and endElement are defined without operation in DefaultHandler and therefore overridden, specified with the annotation @Override, by the UserHandler.

### 4.2.2 DOM parser

The DOM implementation starts with parsing the XML document into a Document object, Figure 6. Before the document object is created, the timer for measuring parsing time is started and the current time is saved to the variable startTime.

```java
long startTime = System.nanoTime();

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document xmlDoc = builder.parse(file);
```

**Figure 6 DOM code, Parsing**

A NodeList is created containing all elements, Figure 7. The for-loop iterates until an element is found that has the same name as the entry tag. The application will then know that it found the closing tag of the entry. The column names are stored in a List containing strings.

```java
NodeList nl = xmlDoc.getElementsByTagName("*");
List<String> xmlColumnNamesList = new ArrayList<>();
for (int i = 0; i < nl.getLength(); i++) {
    if (i > (depth - 1)) {
        if (nl.item(i).getNodeName() == xmlColumnNamesList.get(depth - 1)) {
            break;
        }
    }
    xmlColumnNamesList.add(nl.item(i).getNodeName());
}
```

**Figure 7 DOM code, Get Column Names**

Xpath is used to navigate through the parsed XML document, Figure 8. By using the pathway variable, the user can tell the application on what depth to retrieve information from. A for-loop is then used to get the data from each column on that depth. A Pair object is created with the names from the xmlColumnNamesList and the data from the getTextContent function, Figure 9. After this is done the timer is stopped and saved to the variable stopTime. By subtracting startTime from stopTime the time spend from start to stop can be calculated.

```java
XPath xpath = XPathFactory.newInstance().newXPath();
String pathway = "";
for (int i = 0; i < depth; i++) {
    pathway += "/" + xmlColumnNamesList.get(i);
}
NodeList res = (NodeList) xpath.evaluate(pathway, xmlDoc, XPathConstants.NODESET);
```

**Figure 9 DOM code, Get Pathway**

```java
@Override
public void endElement(String uri, String localName, String qName) {
    if (counter == main.depth) {
        pair = new Pair<>(qName, stringBuilder.toString());
        main.pairs.add(pair);
    }
    counter--;
}
```

**Figure 5 SAX code, endElement**
for (int i = 0; i < res.getLength(); i++) {
    Node node = res.item(i);
    for (int m = depth; m < xmlColumnNamesList.size(); m++) {
        pairs.add(new Pair<>(xmlColumnNamesList.get(m), getTextContent(node, xmlColumnNamesList.get(m))));
    }
}

long stopTime = System.nanoTime();

Figure 8 DOM code, Navigation Using Xpath

static private String getTextContent(Node parentNode, String childName) {
    NodeList nlist = parentNode.getChildNodes();
    for (int i = 0; i < nlist.getLength(); i++) {
        Node n = nlist.item(i);
        String name = n.getNodeName();
        if (name != null && name.equals(childName))
            return n.getTextContent();
    }
    return "";
}

Figure 9 DOM code, getTextContent()

4.2.3 VTD parser

The VTD implementation starts by reading the XML document to memory, the content is stored in a byte array. Before this the timer for measuring parsing time is started and the current time is saved to the variable startTime. The byte array is then assigned to a VTDGen object using the member method setDoc(), Figure 10. The VTDGen object can then be used to generate VTD buffers and hierarchical info by parsing the object with VTDGen::parse().

long startTime = System.nanoTime();

VTDGen vg = new VTDGen();
vg.setDoc(b);
vg.parse(true);

Figure 10 VTD code, Parse file

To navigate across the hierarchy of the parsed document, a VTDNav object is created, Figure 11. VTDNav has two views of the VTD tokens, a flat and hierarchical view. To navigate through the tokens a cursor is moved to tell the application what token the application should point to. Instead of moving the cursor manually, the application uses AutoPilot, the cursor is then moved with depth-first document order node iteration. With calling ap.selectElement("*") the AutoPilot is told to match with all strings(selecting every element) when ap.iterate() is called.

VTDNav vn = vg.getNav();
AutoPilot ap = new AutoPilot(vn);
ap.selectElement("*");

Figure 11 VTD code, Navigation

To get the data for each column, ap.iterate() is called, Figure 12. The while-loop will iterate through all tokens and when the depth of a token is the same as the user defined depth as, the application will get the text for that column. A Pair object is then created with the data for that column together with the name of the column and added to the ArrayList pairs. After this is done the timer is stopped and saved to the variable stopTime. By subtracting startTime from stopTime the time spent from start to stop can be calculated.
```
while (ap.iterate()) {
    if (vn.getCurrentDepth() == depth) {
        int t = vn.getText();
        if (t != -1) {
            pairs.add(new Pair<>(vn.toString(vn.getCurrentIndex()),
                vn.toNormalizedString(t)));
        }
    }
}
long stopTime = System.nanoTime();
```

**Figure 12** VTD code, Get Data

### 4.2.4 XML document generator

A generator that could generate XML data with different depths, columns amount, entries and amount of characters between tags could not be found if not bought for money. Instead a generator was built to generate the correct and controlled data. The generator is built on Java with JDK 10. In the program, declared variables are used to set the configuration of the documents. The output documents are named after the configuration to assure that the correct sample data are used in the parser. In Figure 13 the creation of the XML depth is shown. The defined number of elements with randomized names are started here.

```
XMLOutputFactory xMLOutputFactory = XMLOutputFactory.newInstance();
XMLStreamWriter xMLOutputFactory = xMLOutputFactory.createXMLStreamWriter(new FileOutputStream(filename));
xMLOutputFactory.writeStartDocument();
for (int i = 1; i < depth; i++) {
    randomElement = (rand.nextInt(dictionary.size()));
    xMLOutputFactory.writeStartElement(dictionary.get(randomElement));
}
```

**Figure 13** XML Generator code, Creating depth of XML document

At the end of the last created depth-tag the entry-tag is made. Nested under the entry-tag a specified number of column-tags are created. A string of characters is added to the tag. After all the column-tags are made the closing tag to the entry is created. This process is iterated as many times as it is defined in the entries variable. This step is shown in Figure 14. The chosen number of character gets randomized with the use of Apache Commons library `RandomStringUtils`.

```
for (int i = 1; i <= entries; i++) {
    xMLOutputFactory.writeStartElement(dictionary.get(randomElement));
    for (int j = 1; j <= columns; j++) {
        randomString = org.apache.commons.lang3.RandomStringUtils.random(characters, true, false);
        xMLOutputFactory.writeCharacters(randomString);
    }   xMLOutputFactory.writeEndElement();
}
```

**Figure 14** XML Generator code, Creating entry and columns

After the creation of entry-tags the document gets its closing-tags and the document are closed. A sample of the output documents has been checked with an online validation tool to see if they are well formed (Free Online XML Validator (Well formed)).
4.2.5 Data Collection

The data for the experiment is generated by the XML generator and is split up into four sections, Figure 15. Each section contains ten folders with each folder containing 10 XML documents. The folders in each section differ by the configuration that is measured, Table 2. The ten documents in each folder has the same structure but vary regarding name on tags and characters used as data.

![Figure 15 Generated Data](image)

The structure of an XML document can be seen in Figure 16. In this figure the depth is defined as 3 in the XML generator. Number of columns are 8 with 5 characters per column, contained in 1 entry.

![Figure 16 Description of XML document structure](image)

Different XML documents have been generated for each experiment. Depending on what will be measured the XML documents will have different structures. Table 2 shows what data that has been used for each of the experiments. For example, row 1 indicate an experiment with a varying number of columns but fixed depth, text and entries.

<table>
<thead>
<tr>
<th>Experiment</th>
<th># Columns</th>
<th># Depth</th>
<th># Characters</th>
<th># Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Columns</td>
<td>10,20...90,100</td>
<td>3</td>
<td>5</td>
<td>3000</td>
</tr>
<tr>
<td>2. Depth</td>
<td>8</td>
<td>10,20...90,100</td>
<td>5</td>
<td>3000</td>
</tr>
<tr>
<td>3. Characters</td>
<td>8</td>
<td>3</td>
<td>100,200...900,1000</td>
<td>3000</td>
</tr>
<tr>
<td>4. Entries</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>3000,6000...27000,30000</td>
</tr>
</tbody>
</table>

Table 2 Experiment data description
5 Results

This section starts with evaluating the results obtained from the experiments. An ANOVA test is then presented together with an analysis discussing the result. In Table 2 there is an explanation to how the document was structured for each experiment.

5.1 Evaluation

The first experiment is to evaluate how the execution time changes when the number of columns increases.

The result of the first experiment shown as a line chart in Figure 17. The chart shows that the DOM takes longer time to parse the documents than both SAX and VTD, an increase from 0.029 to 0.295 seconds. VTD takes longer time to parse than SAX, an increase from 0.013 to 0.104 seconds. SAX starts at 0.011 and ends at 0.093 seconds. There is a difference in the steepness of the curves, where DOM is the steepest curve and SAX is the least steep. As the number columns increases the performance gap also increases.

![Figure 17 Number of columns](image)

The second experiment is to evaluate how the execution time changes when the number of entries increases.

In Figure 18 the chart with increasing number of entries is shown. The general result is similar to the one in Experiment 1, where DOM takes the longest time, VTD takes less time and SAX takes least time to execute. There is a difference in the steepness but not as distinct as in Experiment 1 with columns.
The third experiment is to evaluate the execution time when increasing characters.

The results of Experiment 3 are shown in Figure 19. Contrary to the earlier experiments, VTD now performs worse than DOM. SAX stills performs the best but meets DOM between 900 and 1000 characters. The trendlines also shows that VTD and DOM intersect at 100 characters.
The fourth and last experiment is to evaluate the execution time when increasing the depth. In Figure 20 the chart with the results are shown. Just as in Experiment 1 and 2, SAX and VTD outperform DOM. The trendlines show that SAX and VTD have similar trend with an almost flat line. The trendline of DOM though have a slight increase.

5.2 Analysis

A One-way ANOVA test shows the variance among and between the groups to assert the obtained simulation results in the experiment.

The descriptive statistics will give the confidence interval. The standard error estimates where the true mean for the entire population will be.

The ANOVA test shows that Experiment 1, 2 and 3, Figure 21, Figure 22 and Figure 23, have a confirmed difference between the groups with a confidence level of 95%. However, the test show that Experiment 4, Figure 24, fails to reject the null hypothesis, that there is no difference between the groups. This may be attributed to the way input data was generated in this particular latter experiment, where depth was not associated with breadth XML entries at each depth-level. Such extended experimental setup is suggested as part of a potential future work.
Experiment 1, 2 and 3 shows the means variation across SAX, DOM and VTD parsing times. While a 95% confidence is obtained for the difference of mean values, the intersection between confidence intervals is non-empty which suggests that no conclusion can be made as of which parsing technique is contribution to this difference of means. Thus, a post-hoc analysis could be performed, such as the Tukey test. The raw mean data used for the ANOVA test is made available in Table 3.

<table>
<thead>
<tr>
<th>Depth (No)</th>
<th>Execution (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SAX</td>
</tr>
<tr>
<td>10</td>
<td>0,010514991</td>
</tr>
<tr>
<td>20</td>
<td>0,009953178</td>
</tr>
<tr>
<td>30</td>
<td>0,010363716</td>
</tr>
<tr>
<td>40</td>
<td>0,010167772</td>
</tr>
<tr>
<td>50</td>
<td>0,010275723</td>
</tr>
<tr>
<td>60</td>
<td>0,010057089</td>
</tr>
<tr>
<td>70</td>
<td>0,010079496</td>
</tr>
<tr>
<td>80</td>
<td>0,010196344</td>
</tr>
<tr>
<td>90</td>
<td>0,010429793</td>
</tr>
<tr>
<td>100</td>
<td>0,011163977</td>
</tr>
</tbody>
</table>

Table 3 Raw data, Experiment 4 - Depth
5.3 Discussion

The amount of characters used in the configurations for the fourth experiment, number of characters, was increased from 10…100 to 3000…30000. The result from that experiment did not show a clear trend in processing time and could not be analysed correctly. Therefore, the result could not be generalized. The dataset, 10…100, was too small to be measured, leading to difficulties in making a conclusion of the differences between parsers. The change of performance between the parsers can be a result of how parsers transform the information from the element into a string. When VTD parses an XML document it creates a byte array containing all the data. To transform this data into a string to store the data in the ArrayList, VTD uses toString methods to transform the data into strings. This transformation of large number of characters seems to be the bottleneck of the parsing time. DOM creates an object of the parsed file from beginning and by doing this, the need of transforming the data into string for each element is not needed. Because it was already parsed into string when DOM parsed the entire file into an object. Resulting in large amount of characters not having such a big effect on parsing time for DOM.

Experiment 1 and 2 follow the assumed anticipations. Because DOM needs to build a document object and then traverse the tree, this should result in overhead time when compared to SAX and VTD. The overhead of when VTD builds the record is so small, that even as a memory-based approach, it still performs well. Due to the depth and breadth structure of the XML document, and the simple instructions, SAX can go through the document fast as it only extracts information at the deepest element.

As the depth experiment is based on a structure, with breadth and information to extract only on the furthest depth, a flat trendline is probably what to expect. The slight increase in DOM probably has to do with its impact on execution time as the file gets bigger.

The effect the chosen method has on the study is that concrete measures were gathered. The differences regarding adaptability would probably be the same, with similar trends but with depth as an exception because of the simple structure of the XML document. Although the results are bound to the given implementations and those are strictly experimental. With a case study, the results probably would be quite equal, with DOM performing the worst and SAX and VTD better. VTD would probably get a better result than SAX, as a real-world implementation would be far more complex and thus take longer time to execute. If the method systematic literature review were to be chosen, then VTD probably would perform the best. This is based on the information gathered by reading the articles by Oliveira, Santos and Belo, (2013) and Deshmukh and Bamnote, (2015), describing how VTD outperforms other approaches. Although, to the authors knowledge, no previous study of adaptability has been made and the general impression would probably lead to this conclusion. A survey could probably go either way as it is based on the chosen samples impressions and experiences. There might be a chance that the samples are substantially influenced by their personal favourite and lack of experience with other parsers. If a solution with a given parser solves a task or issue at a company, the chance of them trying to implement new or other parsers is highly unlikely.

When comparing between the parsers with the definition given in the background section on adaptability. SAX has the best adaptability, though the implementation of SAX took the longest time to implement and required the most complex code, the use of multiple classes and functions in different classes. After SAX in the ranking of adaptability is VTD. VTD has a
good parsing time and low complexity on the code when comparing to SAX. VTD only has 17 sequential lines of code in the same method compared to SAX’s 36 that were spread out through different classes and functions. If taking in to consideration the time to implement the solution, when comparing adaptability of VTD and SAX, VTD would probably be the winner. Because of its ease of handling and low parsing time, leading to an efficient and effective parsing solution. DOM is somewhere in the middle of both complexity and parsing time. DOM has the worst parsing time in most of the experiments, but had a lower complexity than SAX. Resulting in a solution that is not the best nor the worst in any of the aspects.

The use of an ArrayList to store parsed data needs to be considered due to being a part of the parsing time measured. Various storing structures works in different ways, leading to the time taken to store an object can vary. An ArrayList is resizable and the size is not set from the beginning and will expand the more objects that are added, storage allocation will be done dynamically during the execution. Compared to a List with a fixed size from the start of the execution, storage allocation will not be done dynamically and could result in a lower execution time due to less time spent on storage allocation.

The effect of using an ArrayList in this experiment is not that crucial, due to all parsers using the same storage type in their implementation. If the parsers had used different storing structures, the impact could have been bigger because the time spent on memory allocation and adding objects to the storage could differ. By using the same storing structure in all implementations, the time taken on memory allocation will be the same for all parsers.
6 Conclusion

Comparing parsers is not something new and has been done before from a wide range of different aspects. Measuring the performance of parsers has been done before but studies lack the viewpoint of adaptability. Information that is sent can change in structure from time to time and these changes need to be handled. The parsers’ ability to handle these changes are described as the property “adaptability”. This study aims to compare three parsing methodologies on how they handle transforming tabular non-recursive XML data, to be ready for insertion in an SQL database in the context of adaptability.

The results from this study show that the parsers differ in performance, where DOM overall is the slowest and SAX and VTD perform somewhat equal. Although there are differences in the performance between the parsers depending on what changes are made to the XML document. The ANOVA tests shown in the result section, shows that there is a difference between the groups in Experiment 1, 2 and 3 with a confidence level of 95%. Experiment 4 on the other hand does not show a difference between the groups.

6.1 Future Work

In this thesis there are only columns on a specific depth of the XML document, taking the study a step further would be to have XML documents with columns on each depth. Leading to the implementation of the parsers needing to be more generalizable, it would be interesting to see how this would affect parsing time and the ease of handling.

A continuation of this thesis would be to study more software engineering aspects, such as reusability. XML documents can have different structures, comparing parsing methodologies for reusability could lead to guidelines for choosing a parsing methodology that could reuse the same solution for different tasks.

6.2 Ethical aspects

One of the positive social aspects found in the thesis is how parsing time can affect a company and its customers. If the right parser is chosen, it can reduce the processing time to parse considerable number of documents. This could lead to the company saving money by reducing computing time, the customers of the company may also experience faster responsivity in the company’s applications. However, if the parsing takes too long, applications handling for example bidding could result in items being sold to the wrong person. If a bid is given before the bidding time is over but the parser is overflown with requests, the highest bid may not be read before the bidding time is over. Leading to a lower bid being the winner of the item because the parser could not handle the amount of request sent to it.

Another ethical aspect of the thesis is the creation of code and the generated data. By generating data, there is not a problem with data being confidential because it was generated specifically for the experiments in this thesis. The code created is also written from scratch by the authors, resulting in that no problem with subjects outside of the study having control over the code.
References


