View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace at Tartu University Library

UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Computer Science Curriculum

Kaarel Tõnisson

Mechanism for Change Detection in HTML Web Pages as XML Documents

Bachelor's Thesis (6 ECTS)

Supervisor: Peep Küngas, PhD

Tartu 2015

Mechanism for Change Detection in HTML Web Pages as XML Documents

Abstract:
Change detection of web pages is an important aspect of web monitoring. Automated web monitoring can be used for the collection of specific information, for example for detecting public announcements, news posts and changes of prices. If we store the HTML code of a page, we can compare the current and previous codes when we revisit the page, allowing us to find their changes. HTML code can be compared using ordinary text comparison, but this brings the risk of losing information about the structure of the page. HTML code is treelike in structure and it is a desirable property to preserve when finding changes. In this work we describe a mechanism that can be applied to collected HTML pages to find their changes by transforming HTML pages into XML documents and comparing the resulting XML trees. We give a general list of the components needed for this task, describe our implementation which uses NutchWAX, NekoHTML, XMLUnit, Jena and MongoDB, and show the results of applying the program to a dataset. We analyse the results of measurements collected when running our program on 1.1 million HTML pages. To our knowledge this mechanism has not been tested in previous works. We show that the mechanism is usable on real world data.

Keywords: Change detection, Difference detection, diffing, HTML, Jena, MongoDB, NekoHTML, NutchWAX, XMDiff, XML, XMLUnit

Mehhanism HTML veebilehtede muudatuste tuvastamiseks XML dokumentidena

Lühikokkuvõte:
Veebilehtede muudatuste tuvastamine on oluline osa veebi monitoorimisest. Veebi automaatset monitoorimist saab kasutada spetsiifilise informatsiooni kogumiseks, näiteks avalike teadaannete, uudiste või hinnamuutuste automaatseks märkamiseks. Kui lehe HTML-kood talletada, on võimalik seda lehte uuesti külastades uut ja eelnevat koodi võrrelda ning nendevahelised erinevused leida. HTML-koode saab võrrelda tavateksti võrdlemise meetodite abil, kuid sel juhul riskime lehe struktuuri kohta käiva informatsiooni kaotamisega. HTML-kood on struktuurilt puulaadne ja selle omaduse säilitamine muudatuste tuvastamisel on soovitav. Selles töös kirjeldame mehhanismi, millega eelnevalt kogutud HTML-koodis lehed teisendatakse XML dokumentide kujule ning võrreldakse neid XML puudena. Me kirjeldame selle ülesande täitmiseks vajalikke komponente ja oma teostust, mis kasutab NutchWAX-i, NekoHTML-i, XMLUnit-it, Jena-t ja MongoDB-d. Me analüüsime mõõtmistulemusi, mis koguti selle programmiga 1,1 miljoni HTML lehe läbimisel. Meile teadaolevatel andmetel pole sellist mehhanismi varem rakendatud. Me näitame, et mehhanism on kasutatav tegelikkuses esinevate andmete töötlemiseks.

Võtmesõnad: Muudatuste tuvastamine, diffimine, HTML, Jena, MongoDB, NekoHTML, NutchWAX, XMDiff, XML, XMLUnit

Contents

1 Introduction

The amount of content available on the World Wide Web has been rapidly growing and is showing no signs of slowing down. Hundreds of online shops, blogs, news portals and various other web pages post new content every day that a particular Internet user might find useful and important.

As the amount of content grows, it becomes increasingly difficult for a human user to keep track of the addition of new data. Manually visiting and checking dozens of web pages for updates is time-consuming, visiting hundreds or thousands is not feasible. Computer systems can help reduce the workload of humans by automatically visiting web pages and checking for changes, informing the human user whenever changes are found. However, automated change detection presents many challenges.

Change detection or *diffing* is the process of finding the changes between two objects, in our case two versions of a web page. Given two documents, diffing them results in a list of changes that when performed on the first document turn it equivalent to the second document.

Separating important and non-important changes is something an automated system cannot do alone. Not all changes are important to humans. For example, the date displayed in the corner of an online shop changing to a new day every midnight is rather unimportant information to someone interested in finding new special offers. Locating changes relative to the structure of the page is beneficial as it provides more information about the nature of the changes. Automatically generated pages often display the same information, for example the products for sale in an online shop, in a different order when the page is revisited. If the change detection system operates in a structure-aware manner, it is possible to filter out low-importance changes like reorderings. Although HTML pages are tree-like in structure, errors like unclosed tags are common enough to prevent tree change detection algorithms from working on them directly. A solution would be to first apply a cleaning algorithm to the HTML to give it a well-formed tree structure. HTML to XML cleaners such as NekoHTML [1] and HtmlTidy [2] exist that are capable of converting most HTML pages into XML documents that follow strict tree structures. Once a page has been turned into an XML document, we can apply an XML change detection algorithm to find changes that include structural info.

Timeliness and completeness are two important properties of change detection. Timeliness describes how quickly a change is detected once it occurs. Completeness describes how many of all changes that occurred were successfully detected. An ideal change detection system would detect all changes with minimal delay. In practice computational resources have limitations and some degree of loss of both will occur. An automated system can only process web pages at a certain speed. If pages are revisited often, the number of revisits that have to be processed is large, therefore changes to each page take longer to detect and the system loses timeliness. If pages are not revisited often enough, it is possible for changes to appear and disappear before the automated system notices them and the system loses completeness. Different pages change with highly varying frequencies, so revisiting all pages at equal intervals is most often suboptimal. The revisiting strategy which describes when to revisit each page has to be configured to suit to the needs of the system's users.

Many page monitoring services such as *changedetection.com* and *changealarm.com* exist that notify the user when the content of a page they subscribe to has changed, usually showing which parts of the page were altered. Other services are more specialised and exist to fulfil certain specific tasks, for instance *import.io* extracts data (for example product names and prices) from web pages and converts them into a form usable for data analysis. On subsequent uses, it also offers change detection between two data collections. These services detect changes to pages only after a user specifies the page to be monitored. As such they are not useful for cases when old versions of the page become important at a later time. If we are interested in change detection between older versions of pages, we need to archive the pages beforehand. The Internet Archive [3] is the most well-known archiver of web pages for preservation purposes. It is very likely that if we have a certain scope of web pages we are interested in, the Internet Archive does not collect all of the pages we want or does not do it at sufficient frequency, requiring us to create our own archive. Archival web crawlers such as Heritrix [4], which is also used by the Internet Archive, can be used to archive web pages according to our requirements.

Although there exist change detection systems for live pages and archiving systems that store pages, there are no ready-made systems available that would allow us to detect changes to web pages we have archived. We would also want to consider the structure of each page when detecting changes so that we could filter out less relevant changes such as reorderings. Converting the HTML page into an XML document would allow us to use XML change detection algorithms that are structure-aware. However, there are no ready-made solutions that combine HTML to XML conversion and XML change detection.

Many of the components required for solving this task already exist. Tools such as NutchWAX [5] can be used to access HTML files from archives. HTML to XML cleaners such as NekoHTML [1] and JTidy [6] perform relatively well on most HTML pages. Several XML change detection algorithms exist, some tested in scientific works (such as DeltaXML and XyDiff [7], XMDiff and MMDiff [8]), some with implementations only, but available in software packages (such as XMLUnit [9]).

In this work, our goal was to combine existing solutions into a single program that processes archived HTML pages and extracts structure-aware changes between different versions.

Our approach was to make use of existing software packages and apply them in our program to access archived HTML pages and clean them into XML documents. Once the page conforms to XML standards we can process it using an XML document change detection algorithm to obtain the changes between the two documents. Finally we output the detected changes in a format suitable for future processing.

For our implementation, we created a program using NutchWAX [5] crawler to access archived pages stored as WARC files, NekoHTML [1] to convert archived HTML pages into XML documents, XMLUnit [9] *DetailedDiff* to perform change detection of the XML documents, MongoDB [10] for storing previous versions of HTML pages for change detection and Apache Jena [11] for outputting the detected changes as RDF/XML documents. We used XMDiff [8] as a reference XML change detection algorithm to compare the performance of XMLUnit *DetailedDiff* to. We tested our program on a collection of WARC files archived from Estonian public web pages.

Our program was used to process 1.1 million pages and it extracted 28.0 million changes. The error rate (pages that failed to be processed) was 3.3% which is acceptable for the given dataset. While successful at performing its task, there are several possible improvements to the program. Using an alternative change detection algorithm or using another way of accessing archived HTML pages can improve throughput and allow quicker processing. Some extreme values of computation costs and change counts were detected in

some tests. However, their reasons remain unknown as performance dependence on page content and structure was not analysed.
The document is structured as follows. Section 2 describes related works. Section 3 describes the components used in our approach, both in general and in our implementation. Section 4 shows the experimental results of running the program on the data. Section 5 concludes with a summary of the results and possibilities of future work.

2 Related Works

Several web monitoring systems exist that offer a change detection and notification service to its users. Generally they operate on live pages and not on archived pages.
WebVigiL [12] is a change detection and notification system which offers HTML and XML change monitoring. It uses the CH-Diff algorithm for HTML and the CX-Diff algorithm for XML change detection. Page monitoring is performed by user-defined sentinels which specify page URL, keywords and page fetching frequency. WebVigiL operates a repository of different versions of pages it has fetched to reduce network traffic. When a request is made for a page, it is first looked for in the repository and only if not found there it is fetched from the website. Detected changes are displayed to the user as merged and highlighted documents or as two side-by-side documents with changes highlighted. WebCQ [13] is a change monitoring service that monitors pages chosen by the user and provides personalised change notification. WebCQ performs data extraction from certain logical structures (Table, List, Paragraph, Link, Image) from the HTML code of a page using a small HTML parser. For change detection, a sentinel compares the extracted data to a previous version cached in WebCQ. If changes exist, both versions are passed to the difference extraction module and the old version in the cache is replaced with the new one. WebCQ stores only one version of each page in its cache, it deliberately avoids archiving and therefore can detect changes only between the two latest versions of each page. Most web monitoring approaches have been based on continuous monitoring where web pages are accessed from the web directly. Liu [14] describes the WebCQ continual query system for large-scale monitoring.
WIC [15] is a general purpose algorithm for near real-time monitoring to achieve either timeliness or completeness. It allows calibrating revisit times to focus on either timeliness or completeness of results.
Multiple XML change detection algorithms exist with various methods of functioning.
Chawathe [8] describes the MMDiff and XMDiff algorithms, the names standing for "main memory" and "external memory", which describe where data is stored during the diffing process. MMDiff has quadratic memory operations, XMDiff has quadratic file system operations. Otherwise they operate in the same way, producing a minimal length list of node additions and removals needed to transform one document into the other. The algorithms work with node additions and removals only, no other operations are allowed. This makes the results of the algorithms useful reference points because the minimal number of additions and removals is a constant value between a pair of documents. On the other hand, their computation speed is much slower than that of many other algorithms.
XyDelta and DeltaXML algorithms have been tested by Cobena [16, 7]. We applied some of the metrics used there in our work, including the usage of the XMDiff algorithm as a reference for comparison and quality estimation based on change list length ratios.

Vi-DIFF [17] is a change detection algorithm for HTML pages that detects both content and structural changes based on visual perception. It applies and extends the VIPS algorithm to segment the page based on horizontal and vertical separators and constructs a "visual tree" that represents the structure of the page. That tree is used as the basis for change detection using node addition, removal and updating. Moving nodes is supported by a few approaches as well.

In this work we used MongoDB as a database for storage of previous versions of web pages. The performance of MongoDB has been analysed in some works in comparison to other database systems. One has to consider that MongoDB development is (as of time of writing) ongoing and the performance may have improved significantly compared to the versions used in those tests. A 2013 study compares MongoDB performance to SQL, showing that MongoDB is faster for insertions and simple queries, the situation relevant in our work [18]. A study by Abramova in 2014 [19] compares the execution times of MongoDB and several other NoSQL database systems run against YCSB (Yahoo! Cloud Serving Benchmark) database workloads. The results indicate that in most tests, several other database systems (such as Cassandra) have better performance than MongoDB. Another study by Abramova in 2013 [20] compares MongoDB and Cassandra at different database sizes. The study shows that as the size of the database increases, Cassandra is usually faster than MongoDB in most operations.

While there have been several approaches to HTML to XML conversion and several automatic parsers exist, there are relatively few papers on the topic of fully automated conversion. Several specialised approaches exist that depend on user interaction or set some constraints to the input HTML. HTML to XML conversion to collect data from selected online newspapers following manually specified parameters for each newspaper has been implemented in the VIPAR system [21]. HTML to XML conversion with the help of user input from visual interfaces has been done in [22]. Transformation-based learning has been applied to convert semi-structured HTML into XML [23].

3 Implementation

This section describes the components and their interactions used in our process. First the general types of components needed for the process are described, then each component of our implementation are separately covered.
The source code repository of our program is available in Appendix 1.

3.1 Process Components

In this work our aim was to detect changes of HTML pages by converting them into XML documents and finding the changes of the created XML documents. This requires the usage of certain components.
First, a main method that takes HTML input and performs our other actions on it is needed. It has to be easy to apply to a large number of HTML pages, possibly in parallel.
Second, a storage and retrieval mechanism is needed that can store at least one previous version of each HTML page. If a previous version exists, it is retrieved to be compared to the version currently being processed. If a previous version does not exist, there is nothing to compare the current version to and therefore no changes can be found.
Third, a method for converting HTML to XML is needed. HTML code may contain some number of deviations from a clean tree structure, making its usage in tree comparison problematic. If the HTML is parsed or cleaned to follow XML constraints, it can be compared using tree comparison methods more effectively. Conversion to XML is a sensible choice since the languages have many similarities. Several HTML to XML parsers are readily available, but perfect conversion can't be expected as some badly formed HTML pages can be problematic to all existing parsers.
Fourth, the XML documents must be compared and their changes extracted. As XML documents follow strict tree structures, they should be reasonably comparable using tree comparison methods. Some algorithms allow only node addition and removal while others might allow node moving or changing. Several algorithms have been tested in previous studies [16, 7].
Fifth, a method is needed to save the detected changes in a sensible format. As we wish to further process the changes in other programs, they have to be stored in an easy to use manner. A reasonable choice might be to save them as XML or JSON documents. This requires us to apply some conversion to the output of the change detection method.

3.2 Our Implementation

3.2.1 Additional Constraints on Our Implementation

Our implementation was built as one component of a larger system for web monitoring. This applied some constraints to our implementation.
In the surrounding system, HTML pages are collected into WARC files using the Heritrix [4] web crawler. WARC (Web ARChive) files are archival files which contain the content of web pages along with data about when and how it was collected [24]. Each WARC file contains some number of objects, including the HTML codes of web pages. We can assume that WARC files are given to our program in chronological order of collection and that storing only a single previous version of each page for change detection is sufficient.

Our program has to output the changes as RDF/XML documents. RDF/XML [25] documents are a type of XML documents which contain Resource Description Framework (RDF) "triples" which are subject-predicate-object expressions. The RDF/XML documents are used by another component outside of our program to display change statistics to users. We chose to use the *http://vocab.deri.ie/diff* ontology to encode the detected changes into RDF/XML triples.
Our program has to access a WARC file and for each HTML page inside, retrieve its previous version from the database of HTML pages, convert both versions into XML documents, find changes between the XML documents, and save the detected changes into RDF/XML files.

3.2.2 Components Used

For our implementation, we used the NutchWAX crawler to access WARC files and execute our required processes, MongoDB to store previous versions of HTML, NekoHTML parser to convert HTML to XML, XMLUnit to perform change detection of the XML documents and Apache Jena to create RDF/XML documents from the detected changes. The deployment diagram is shown in Figure 1. The following subsections describe the components in greater detail.

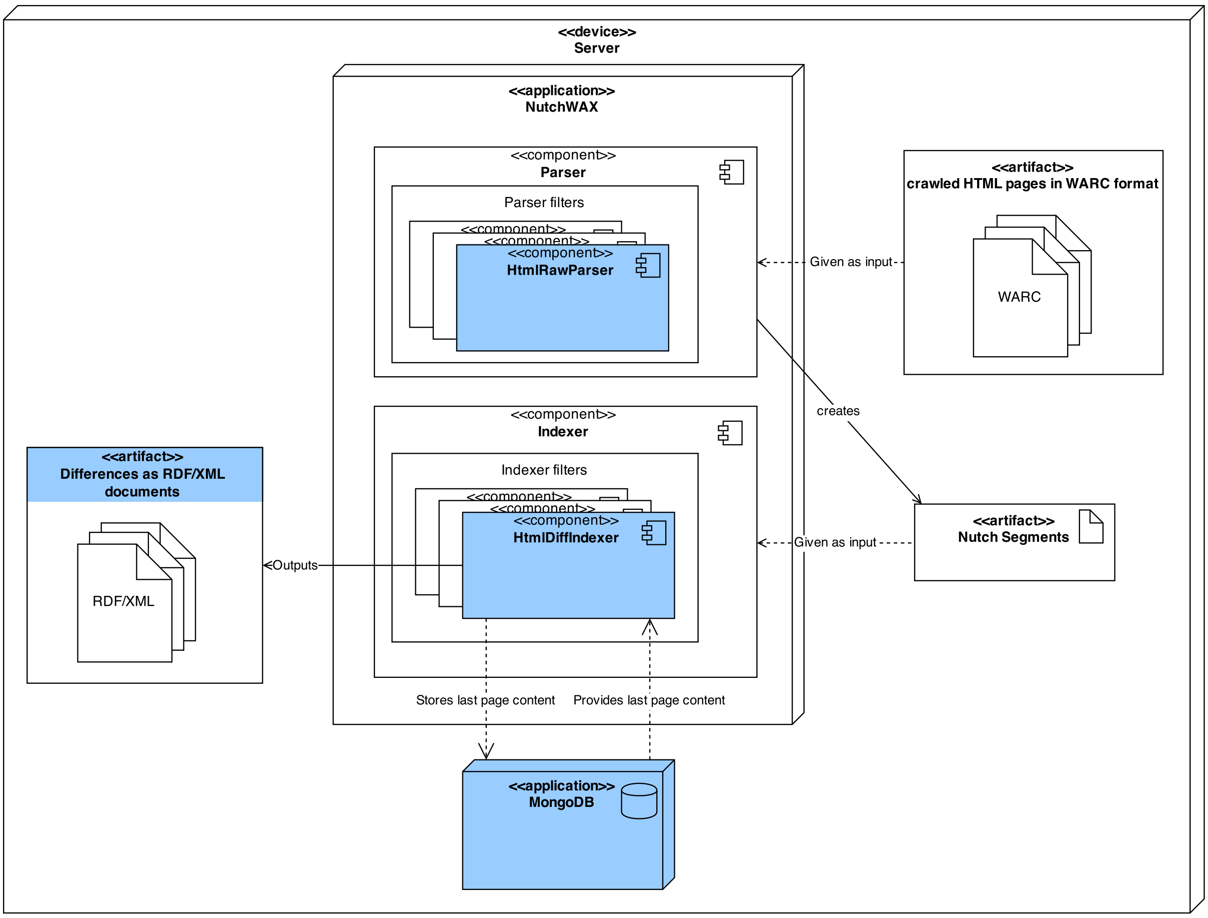


Figure 1: NutchWAX component diagram of our implementation. Components in blue are our additions to the base NutchWAX.

3.2.3 NutchWAX Crawler

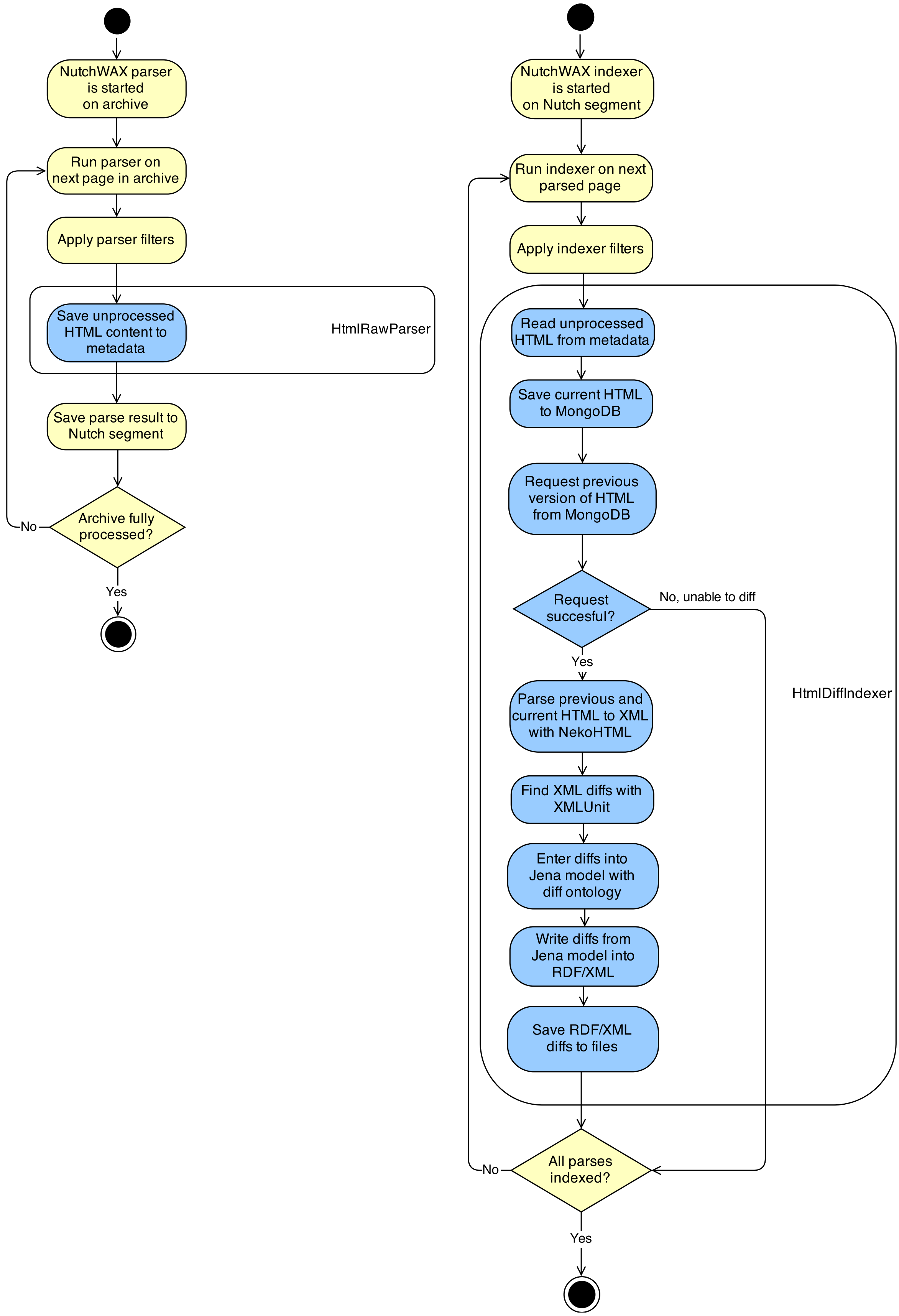
NutchWAX (Nutch Web Archive eXtensions) [5] is a modified version of the Nutch [26] web crawler. Nutch is written in Java and uses the Hadoop framework for distributed processing. While the original Nutch is designed for crawling and indexing pages from the web directly, NutchWAX takes WARC files as input instead. NutchWAX works in two steps: parsing and indexing. Parsing processes the WARC files given as input: for each object in the WARC it applies a parser function and any parse filter plugins, then saves their results into a Nutch data segment. Indexing processes the data segment created by the parser: for each object (HTML page) it applies an indexer function and any indexing filter plugins, then saves the results into an index.
Nutch (and NutchWAX) have modular configurations, allowing us to create and activate plugins so that only actions we require are performed on each HTML page. For this work, we created two plugins: *HtmlRawParser* and *HtmlDiffIndexer*. Their activity diagrams as parts of NutchWAX are shown in Figure 2.
HtmlRawParser is applied during parsing to preserve the HTML code of the pages which the parser normally discards.
HtmlDiffIndexer is applied during indexing. It performs the major part of work in our program: it retrieves the HTML code of the previous version of the page from the MongoDB database, uses NekoHTML to convert HTML to XML for both versions, uses XMLUnit to detect changes between the two XMLs, and uses Jena to convert the detected changes into RDF/XML format files encoded with *http://vocab.deri.ie/diff* ontology.
We performed additional alterations to the configuration of NutchWAX. URL filtering was set so only HTML files were allowed into parsing as we were not interested in other file types. The base output from the indexing step was disabled as we did not need to create an index, only to identify the changes.

3.2.4 MongoDB Previous HTML Versions Storage

MongoDB [10] is a document-oriented NoSQL database system. In our implementation, we used it to store the HTML code of web pages ordered by their URL address and date of collection. This allowed us to query the database for a page with a certain URL address and retrieve only the most recent version. *HtmlDiffIndexer* interacts with MongoDB with its Java driver, using it to request the previous version of the HTML code of the currently processed page and to save the new version of HTML code to the database. If a previous version does not exist, *HtmlDiffIndexer* saves the HTML code of the current version into the database, but cannot continue detecting changes as there are no two pages to compare.

3.2.5 NekoHTML HTML to XML Cleaning and XMLUnit Change Detection

CyberNeko HTML Parser [1] or NekoHTML is a Java-based HTML scanner and tag balancer. We applied NekoHTML in *HtmlDiffIndexer* to parse or HTML pages into XML documents.
XMLUnit [9] is a Java library for unit testing, including change detection of XML documents. We used XMLUnit class *DetailedDiff* to detect the changes of the XML documents created by NekoHTML. XMLUnit offers several qualifiers that can be used to detect changes in XML documents. We used *RecursiveElementNameAndTextQualifier* which promises support for comparing complex and deeply nested types. It matches an



(a) NutchWAX parser.

(b) NutchWAX indexer.

Figure 2: NutchWAX parser and indexer activity diagrams. Activities in blue are our additions to the base NutchWAX.

element to an equivalent element that can be located any number of child elements deeper in the tree.

3.2.6 Jena Model

Apache Jena [11] is an open source Java framework for Semantic Web and Linked Data applications. We used Jena to convert the changes detected by XMLUnit into RDF/XML format. For that we created a Jena model, inserted changes detected by XMLUnit into the model following the *http://vocab.deri.ie/diff* ontology, then output them from the model as RDF/XML documents.

4 Experimental Results

In this section we first describe the system hardware, software, dataset and measurements used to test the performance of our program. Then we display and analyse the results of the tests we performed.

4.1 Testing Environment

4.1.1 Hardware and Software

The experiments were run on a server. Its parameters are listed in Figure 3. The versions of software and libraries used are listed in Figure 4.

System component	**Specification**
Operating System	Debian 3.2.57-3
CPU	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
RAM	4x Kingston 4GB RAM (2x 9905471-011.A00LF, 2x 9905471-020.A00LF)
HDD	4x HGST HMS5C4040BL 4TB

Figure 3: Operating system and hardware parameters of the testing system.

Software/Library name	**Version**
Java	1.8.0_40
NutchWAX	1.3
NekoHTML	1.9.21
XMLUnit	1.6
Jena	2.11.1
MongoDB	2.6.1
diffxml	0.96

Figure 4: Software and its version used in testing.

4.1.2 Dataset

The program was run on WARC files crawled from Estonian web pages using the Heritrix [4] crawler. Due to the nature of the crawling configuration used, each WARC contained web content including HTML pages but also files of different types such as pictures or text files. We configured NutchWAX to only accept HTML type files for parsing as it is the only type we were interested in. The dataset also includes some number of "trap" cycles where the crawler keeps following automatically generated links, creating an unwanted number of pages with little relevant content. We did not attempt to remove them as that is difficult to perform automatically.
In the beginning of the experiment, the MongoDB collection of previous versions of HTML pages was emptied so that we could measure the effect of database size on MongoDB processing speed.
The number of HTML pages processed and the outcomes are listed in Figure 5. The program took 152 hours to process nearly 1.1 million pages, including repeated pages. This time does not reflect the actual performance of the program as it includes performing XMDiff change detection, which is very time-consuming and not a part of the main program.

Result	**Count**	**Percentage**
Changes were detected and extracted	543816	49.5%
No changes detected	273957	24.9%
No previous document exists	245959	22.4%
Error(could not complete process)	35881	3.3%
Total	**1099613**	100%

Figure 5: Outcome counts and percentages of page processing.

4.1.3 XMDiff XML Change Detection Comparison

In our implementation we used XMLUnit *DetailedDiff* with the *RecursiveElementNameAndTextQualifier* qualifier as the change detection method. For comparison we chose to use the XMDiff algorithm.
XMDiff is an XML document change detection algorithm that finds the minimal number of node additions and deletions [8]. It uses temporary files to perform change detection, resulting in high computation time as writing to file system is nearly always much slower than in-memory calculations. XMDiff operates only with node addition and removal, it does not use any other operations (such as moving nodes). As XMDiff finds the minimal length change list, it is useful as a reference algorithm to compare performance to as that result is constant between a pair of documents. If operations other than node addition and removal are allowed, an algortihm can beat XMDiff in change list length. XMDiff has been used as a reference for comparison in previous works studying XML document change detection performance [16, 7]. In this work we used an existing Java implementation of XMDiff in the diffxml package by Adrian Mouat [27] and edited it slightly to run in our program.

4.1.4 Recorded Measurements

To evaluate the performance of our program we added measurements logging to a number of components during the work of *HtmlDiffIndexer*. The recorded properties are listed in Figure 6.

Measurement code	**Measurement description**
doc_size	HTML document size in characters
time_DBread	time spent reading the previous version of the page from MongoDB database
time_DBwrite	time spent writing the current version of the page into mongoDB database
time_parseXML	time spent parsing the HTMLs of both versions of the page into XML documents using NekoHTML
time_diffXML	time spent detecting the changes between the XML documents using XMLUnit
time_JenaModel	time spent inserting detected changes into the Jena model
time_DiffsToFiles	time writing changes from the Jena model into files in RDF/XML format
oldDoc_nodes	XML node count of previous version of page
newDoc_nodes	XML node count of current version of page
diffcount_diffXML	length of change list found by XMLUnit
time_xmdiff	time spent detecting the changes between the XML documents using XMDiff reference method
dist_xmdiff	smallest edit distance between the XML documents found by XMDiff reference method

Figure 6: Names and descriptions of measurements taken during testing.

4.2 File Size Frequency

Figure 7 shows the distribution of the sizes of HTML pages that were processed. Figure 8 shows the cumulative distribution. 82.2% of pages are below 50 KB in size, 99.6% are below 150 KB. The largest size of a single page is 1048 KB. From 400 KB and above, page sizes that exist are sparse. In more frequent values, file sizes are unevenly distributed with several spikes of unusually frequent file sizes. This is likely due to the nature of the dataset as it can contain a large number of near-identical pages created by "traps" where the crawler keeps following links from one automatically generated page to another similar automatically generated page.

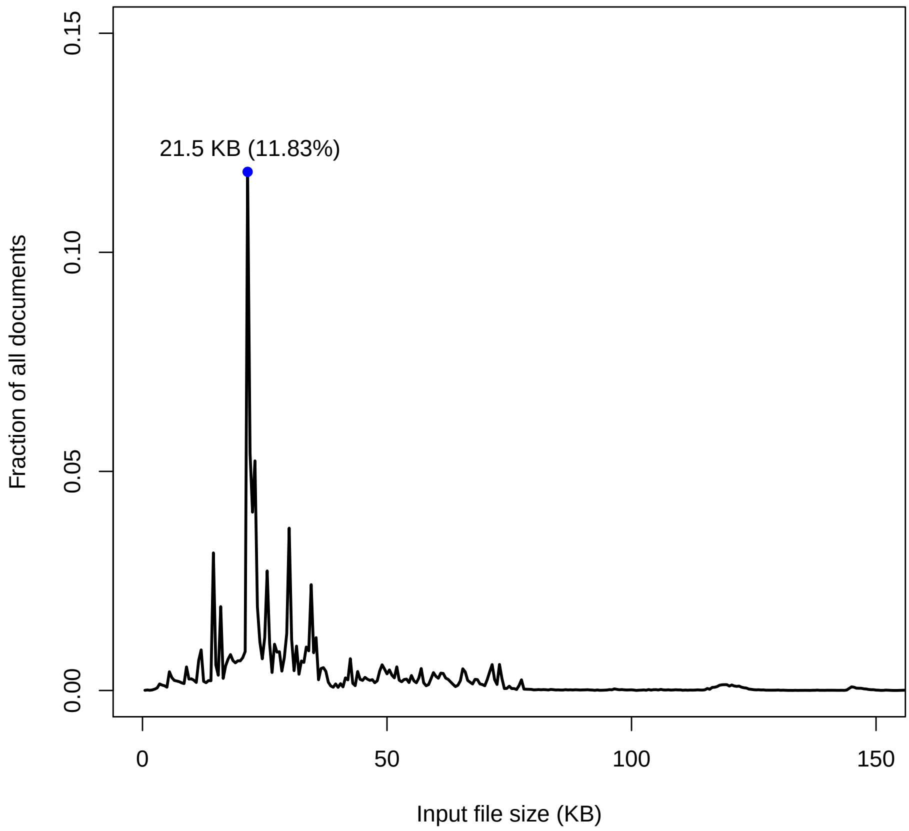


Figure 7: Distribution of HTML input file sizes. 0.5 KB resolution.

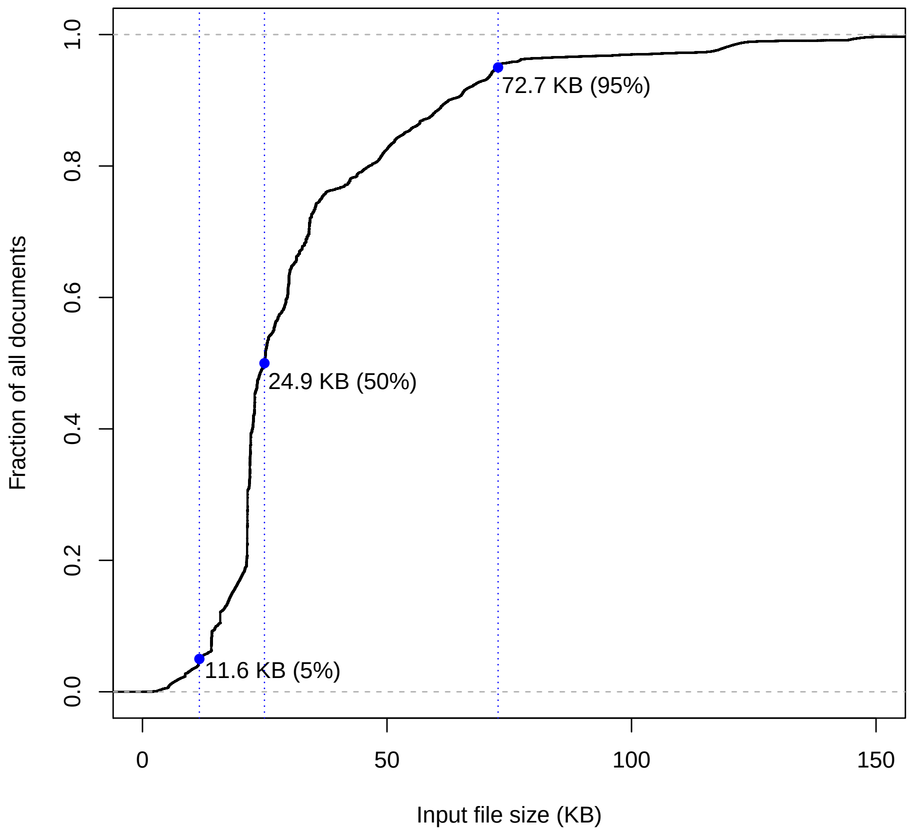


Figure 8: Cumulative distribution of HTML input file sizes. Values at 5%, 50% and 95% quantiles are shown in blue. 0.5 KB resolution.

4.3 Change Distribution by File Size

Figure 9 shows the distribution of changes by the sizes of HTML pages that were processed. Noticably high spikes of XMDiff change counts exist at 425 KB and 550 KB. XMLUnit change counts stay relatively low . In most cases, XMLUnit creates a shorter change list than XMDiff. Over all instances of successful change detection, XMLUnit has a change count mean of 49.06 and XMDiff has a change count mean of 51.46, indicating that XMLUnit may have slightly better results.

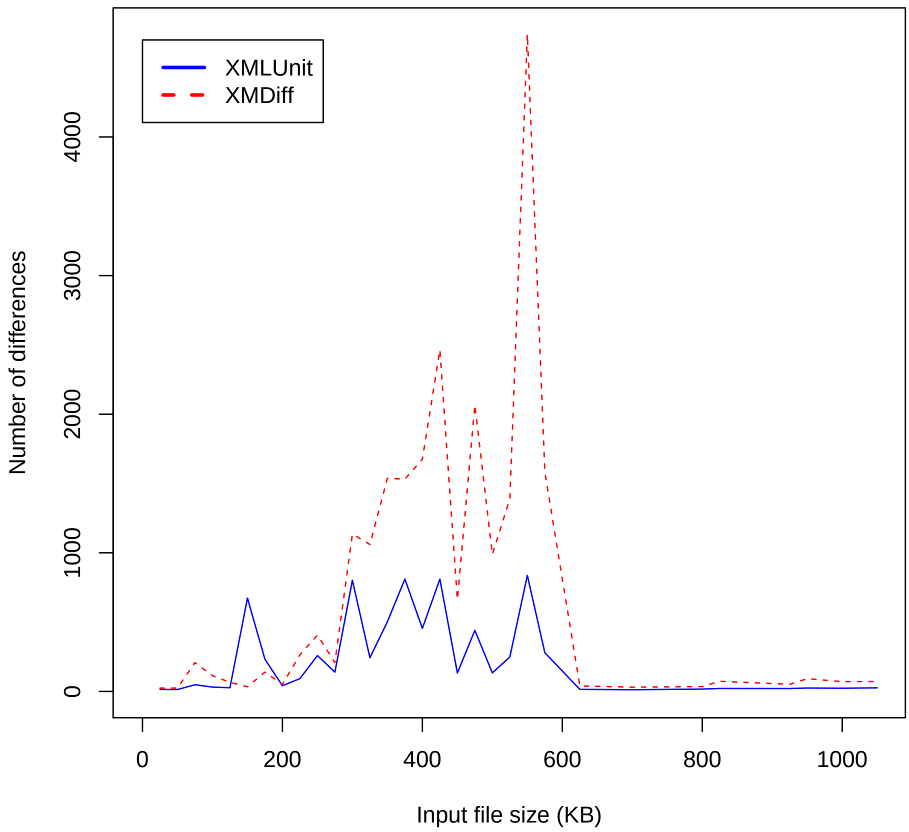


Figure 9: Change count distribution by HTML input file size. 5 KB resolution.

4.4 Quality of Changes

Change list quality is a measure of the ratios between the lengths of change lists. If we assume that both algorithms find the list of changes needed to transform one XML document into the other, the shorter list could be considered the better one. In this test we measure quality as the division between XMLUnit and XMDiff change list lengths. Quality 1 stands for an equal number of changes detected. A value above 1 means that XMLUnit has that many times more changes than XMDiff. A value below 1 means that XMLUnit has that fraction of the number of XMDiff changes.

The distribution of quality values can be seen in Figure 10. The cumulative distribution can be seen in Figure 11. In most cases, the change list lengths are equal or near-equal: in 72.9% of cases the quality is between 0.75 and 1.33. However, outliers exist in both directions where either method gives a much longer change list than the other. 1.26% of cases have their values outside the 0.25 and 4 interval. This means that while the algorithms often give similar change counts, certain pages exist where one algorithm greatly outperforms the other.

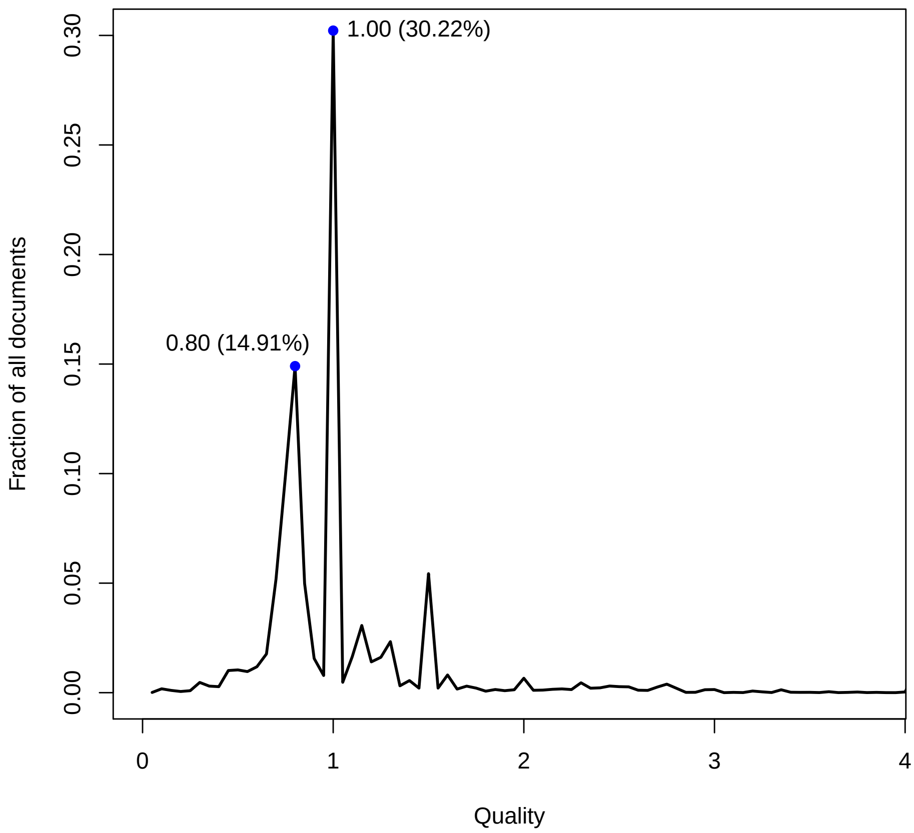


Figure 10: Distribution of XMLUnit *DetailedDiff* quality compared to XMDiff. 0.05 quality resolution.

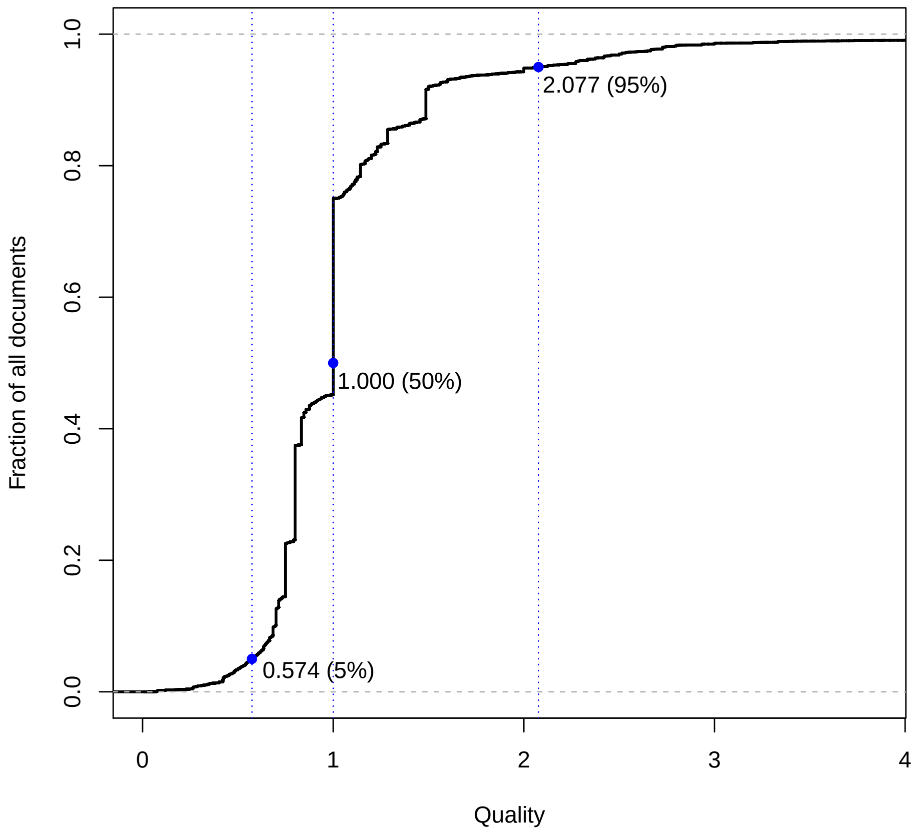


Figure 11: Cumulative distribution of XMLUnit *DetailedDiff* quality compared to XMDiff. Values at 5%, 50% and 95% quantiles are shown in blue. 0.05 quality resolution.

4.5 Computation Time Dependence on File Size

Figure 12 shows the relation between the size of input HTML code and the computation time of XMLUnit and XMDiff. XMDiff is considerably more time-consuming than the rest of the process. This is expected as a similar outcome was observed in previous studies [16, 7]. For a more relevant comparison of our components, Figure 13 shows the computation time without XMDiff and includes other components that are parts of the *HtmlDiffIndexer* plugin. We can see that file size is not noticeably correlated to the processing time of the pages. In many cases, XMLUnit change detection is the slowest part of the system. Omitted from the graph are outliers with very high computation time of Jena change insertion at certain values, which may indicate the existence of an unknown issue.

For all pages where changes were detected, the mean of the computation time sum over all components is 64.4 ms per page. This totals to 9.7 hours or only 6.4% of the program running time. When changes were not detected and therefore only a small number of components were used, the mean time cost was 23.1 ms, totalling to 3.6 hours or 2.3% of the total time. XMDiff computation time was 765.3 ms on average, resulting in 115.6 hours or 76.0% of the total computation time. The remaining 23.1 hours or 15.2% of the total was taken by separately unmeasured components, mainly NutchWAX parsing.

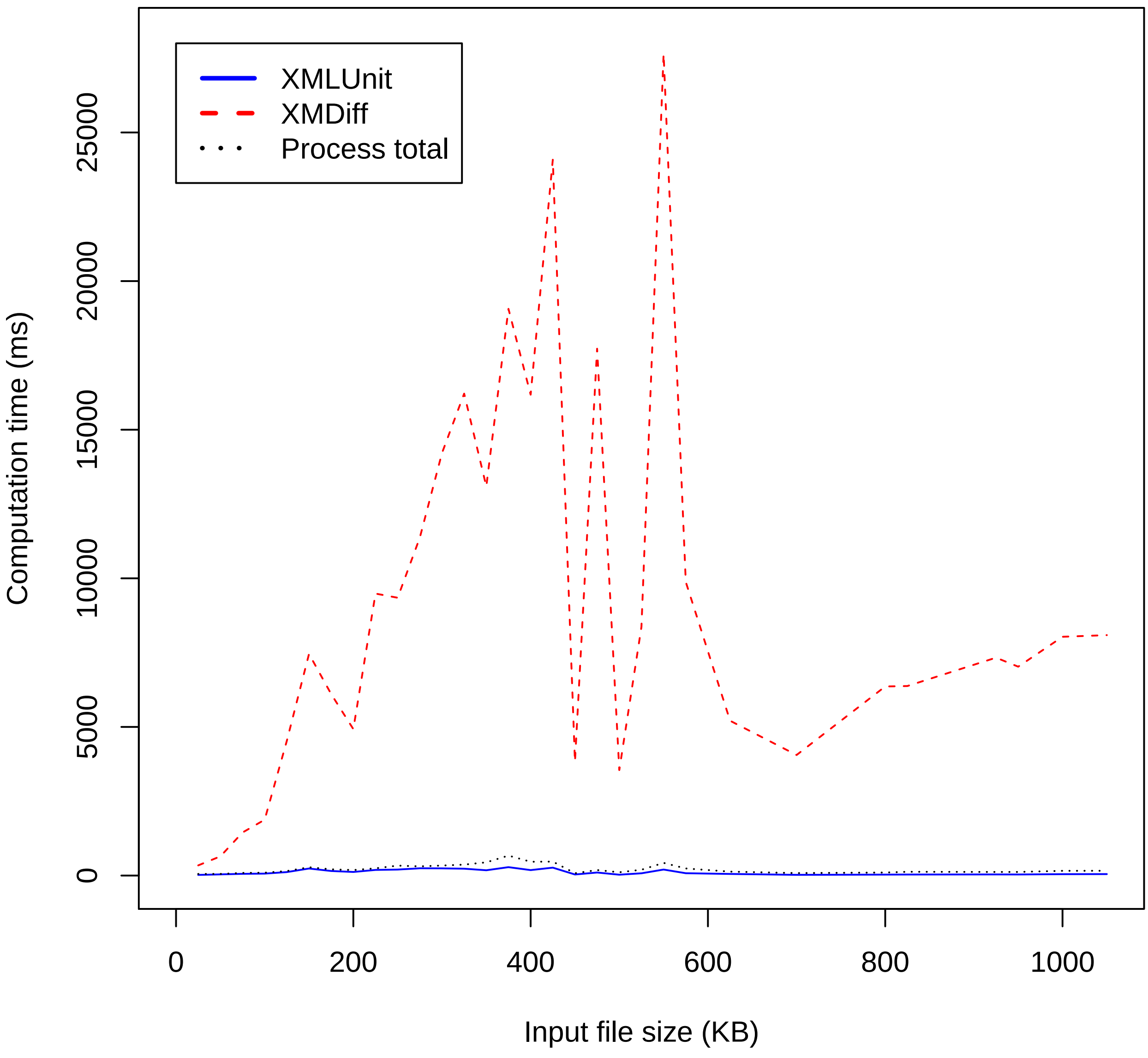


Figure 12: Computation time depending on input file size with XMDiff. Process total does not include XMDiff computation time. 20 KB resolution.

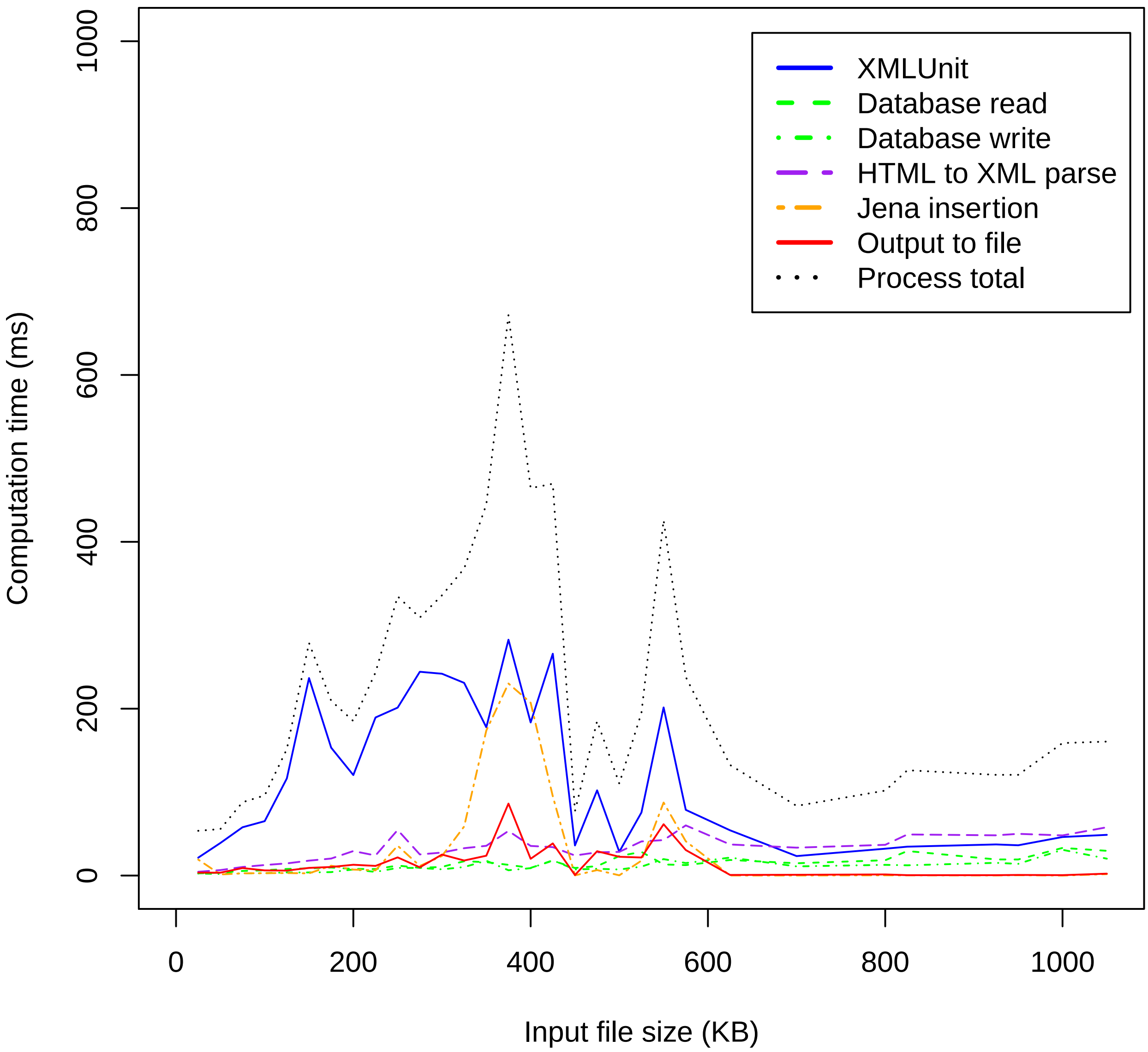


Figure 13: Computation time depending on input file size without XMDiff. 20 KB resolution.

4.6 Computation Time Dependence on Change Count

Figure 14 shows the computation time of each component depending on the number of changes detected by XMLUnit. Note that while there are 541583 (99.6%) data points with change count lower than 600, there are only 2233 (0.4%) data points with change count above 600 with many empty intervals. Thus the reliability of the lines on the right side of the graph can be considered lower as fewer points exist.
As variance is high, it is difficult to tell which components are dependent on the number of changes. One would assume that XMLUnit change detection, Jena model insertion and outputting to file would be dependent as they process the detected changes. From our results, the correlations coefficients are 0.45 for XMLUnit change detection, 0.03 for Jena model insertion and 0.36 for outputting changes to file. Therefore moderate correlation exist for XMLUnit change detection and for outputting changes to file, but not for insertion into Jena model.

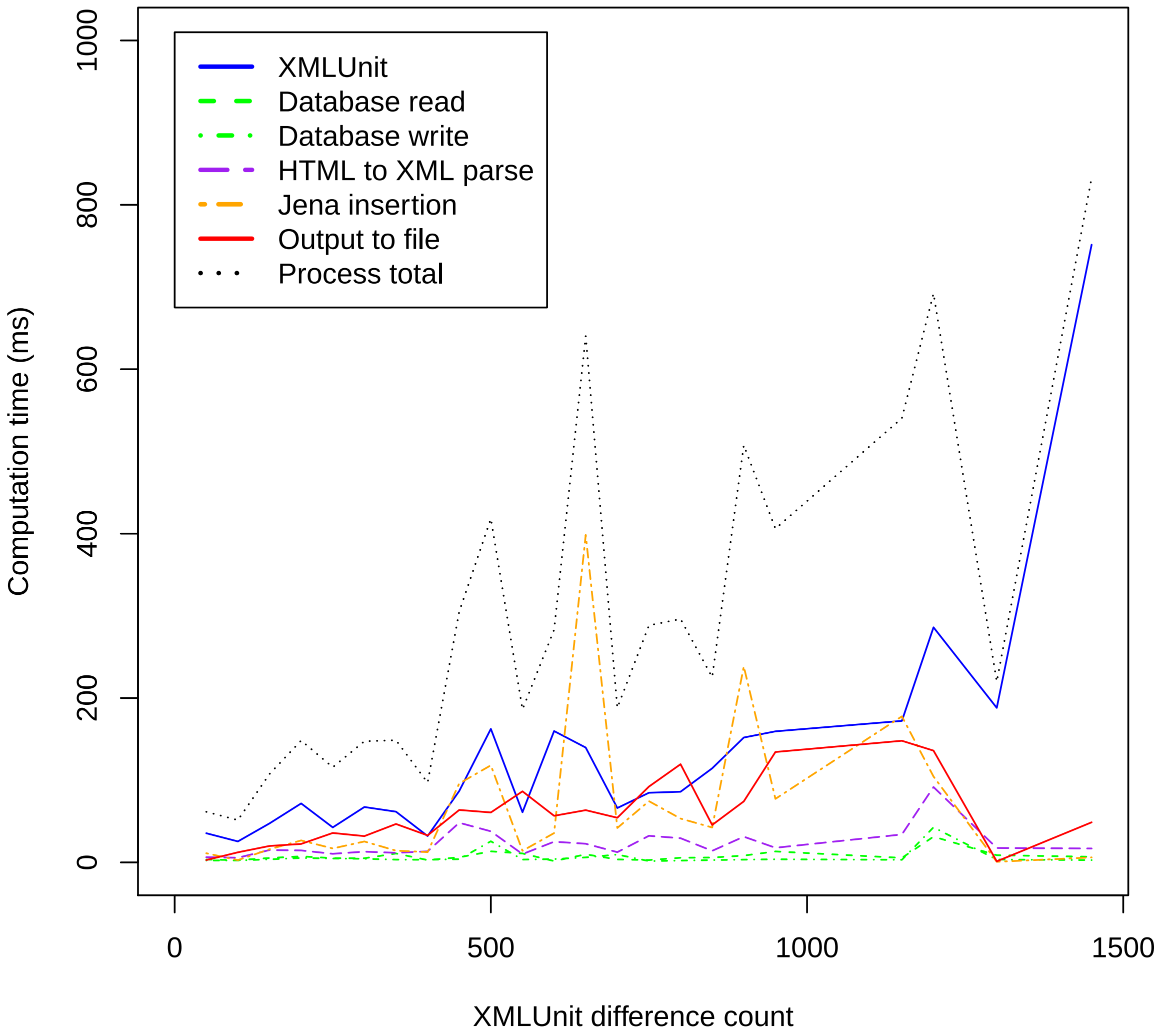


Figure 14: Computation time dependence on the number of changes detected by XMLUnit. 75 change resolution.

4.7 MongoDB Processing Speed Dependence

We tested if the speed of MongoDB read and write operations depends on the number of previous pages in the database. Previous studies have shown that the computation time of MongoDB operations may be slowed down if the amount of data becomes larger [20]. To support testing this, we started our program on an empty MongoDB collection. The results can be seen in Figure 15. Our tests did not show operation speed to be noticeably correlated to the number of documents in the database. It is worth considering that database fullness might not affect processing time at this number of documents and that testing with a larger amount of data may be necessary for dependence to become noticeable.

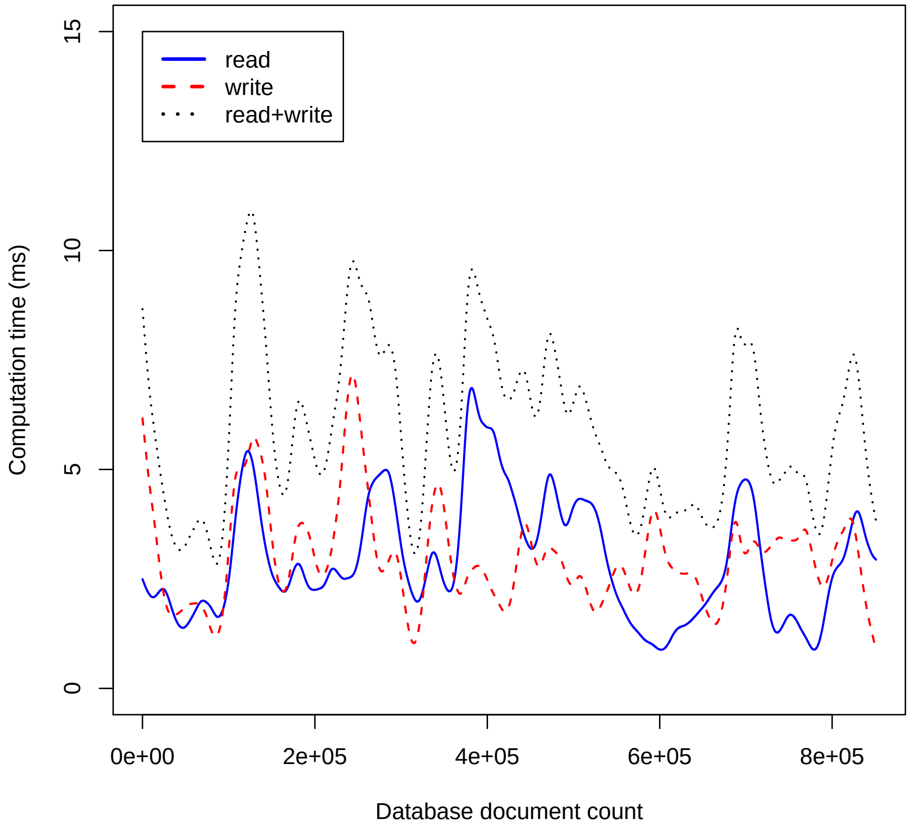


Figure 15: Computation time dependence on MongoDB database entry count.

4.8 Threats to Validity

The contents and structures of the processed pages were not analysed, therefore we cannot confirm the accuracy of our detected changes. As seen in Figure 7, the input data distribution by size has high variance in this dataset with some page sizes being much more prevalent, most likely because the dataset contains a large number of nearly identical automatically generated pages. In Figure 9 we see that there is a large variance of the number of detected changes over different file sizes for both XMLUnit and XMDiff, with

XMDiff having seemingly unreasonably large values at certain file sizes. In Figure 11 we see that while the number of changes detected by XMLUnit and XMDiff stays within 0.5 and 2.0 for more than 90% of the cases, there exist some pages with extreme values where the number of changes detected differ by several hundred times.

5 Conclusion and Future Work

In this work we proposed a mechanism for extracting changes from archived HTML pages by first converting them into XML documents and then finding the changes in the created XML documents. We described the general components one would need to perform this task and our implementation using NutchWAX, NekoHTML, XMLUnit, Jena and MongoDB. We recorded the metrics of computation time and change list lengths and analysed the results. Our program gave reasonably good results in most cases. The change detection of XMLUnit gave similar change list lengths to XMDiff in the majority of cases, although there existed outliers where the change lists were of very different lengths. However XMDiff is much slower to run than XMLunit change detection, therefore XMLUnit is more usable in practice. We observed the speed of MongoDB database operations in relation to the number of documents in the database, but could not find a correlation. It is possible that if a correlation exists, it is only revealed under a much larger database load.

There is room for future work in multiple areas related to this work. Alternatives to most components exist and testing them can give useful insight into their performance.

Alternative change detection algorithms can be used. While we used XMLUnit *DetailedDiff* for change detection, several other implementations and algorithms exist, such as XyDelta and DeltaXML [16, 7].

Alternative database systems can be tested. Some studies have shown that other NoSQL databases such as Cassandra can provide better performance than MongoDB [19, 20]. Although our tests showed database operation times to be relatively small, improvements to the speed of the overall process would still provide useful.

An alternative way of processing WARC files can be tested. While NutchWAX is configurable and plugin-friendly, it is reasonable to believe that a more specialised program could perform the task more effectively, although with less flexibility.

Testing on larger datasets may provide information not obtained in our work. We processed a dataset of 1.1 million pages. This may not have been enough to reveal the effects of database fullness on database operations.

References

[1] http://nekohtml.sourceforge.net/. NekoHTML project page.

[2] http://tidy.sourceforge.net/. HtmlTidy project page.

[3] https://archive.org/. Internet Archive.

[4] https://webarchive.jira.com/wiki/display/Heritrix/Heritrix. Heritrix project page.

[5] http://archive-access.sourceforge.net/projects/nutchwax/index.html. NutchWAX project page.

[6] http://jtidy.sourceforge.net/. JTidy Project page.

[7] G. Cobéna, T. Abdessalem, and Y. Hinnach, "A comparative study for xml change detection," 2002.

[8] S. S. Chawathe, "Comparing hierarchical data in external memory," in *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, (San Francisco, CA, USA), pp. 90–101, Morgan Kaufmann Publishers Inc., 1999.

[9] http://www.xmlunit.org/. XMLUnit project page.

[10] https://www.mongodb.org/. MongoDB homepage.

[11] https://jena.apache.org/. Apache Jena project page.

[12] J. Jacob, A. Sachde, and S. Chakravarthy, "Cx-diff: a change detection algorithm for xml content and change visualization for webvigil," *Data & Knowledge Engineering*, vol. 52, no. 2, pp. 209 – 230, 2005.

[13] L. Liu, C. Pu, and W. Tang, "Webcq-detecting and delivering information changes on the web," in *Proceedings of the Ninth International Conference on Information and Knowledge Management*, CIKM '00, (New York, NY, USA), pp. 512–519, ACM, 2000.

[14] L. Liu, W. Tang, D. Buttler, and C. Pu, "Information monitoring on the web: A scalable solution," *World Wide Web*, vol. 5, no. 4, pp. 263–304, 2002.

[15] S. Pandey, K. Dhamdhere, and C. Olston, "Wic: A general-purpose algorithm for monitoring web information sources," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pp. 360–371, VLDB Endowment, 2004.

[16] G. Cobéna, T. Abdessalem, and Y. Hinnach, "A comparative study of xml diff tools," 2004.

[17] Z. Pehlivan, M. Ben-Saad, and S. GanÃğarski, "Vi-diff: Understanding web pages changes," in *Database and Expert Systems Applications* (P. Bringas, A. Hameurlain, and G. Quirchmayr, eds.), vol. 6261 of *Lecture Notes in Computer Science*, pp. 1–15, Springer Berlin Heidelberg, 2010.

[18] Z. Parker, S. Poe, and S. V. Vrbsky, "Comparing nosql mongodb to an sql db," in *Proceedings of the 51st ACM Southeast Conference*, ACMSE '13, (New York, NY, USA), pp. 5:1–5:6, ACM, 2013.

[19] V. Abramova, J. Bernardino, and P. Furtado, "Experimental evaluation of nosql databases," in *International Journal of Database Management Systems Jun2014, Vol. 6 Issue 3*, 2014.

[20] V. Abramova and J. Bernardino, "Nosql databases: Mongodb vs cassandra," in *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13, (New York, NY, USA), pp. 14–22, ACM, 2013.

[21] T. Potok, M. Elmore, J. Reed, and N. Samatova, "An ontology-based html to xml conversion using intelligent agents," *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*, pp. 1220 – 1229, 2002.

[22] A. Sahuguet and F. Azavant, "Web ecology: Recycling html pages as xml documents using w4f," 1999.

[23] J. R. Curran and R. K. Wong, "Transformation-based learning for automatic translation from html to xml," 1999.

[24] http://www.digitalpreservation.gov/formats/fdd/fdd000236.shtml. WARC format specification.

[25] http://www.w3.org/TR/rdf-syntax-grammar/. RDF/XML W3C Recoomendation.

[26] http://nutch.apache.org/. Nutch project page.

[27] http://www.docjar.com/html/api/diffxml/pulldiff/pulldiff.java.html. XMDiff implementation in Java by Adrian Mouat.

Appendix 1. Project repository

The source code repository of our implementation is located at `https://bitbucket.org/kaareltonisson/nutchwax-with-diffing`.

Non-exclusive licence to reproduce thesis and make thesis public

I, Kaarel Tõnisson (date of birth: 6th of October 1991),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Mechanism for Change Detection in HTML Web Pages as XML Documents

supervised by Peep Küngas

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 21.05.2015