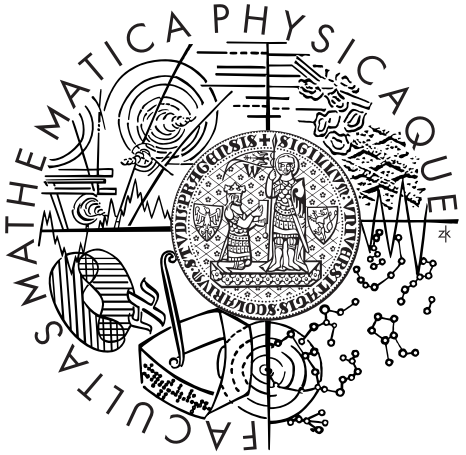
Charles University in Prague
Faculty of Mathematics and Physics

**MASTER THESIS**




Peter Hlísta

# Analysis of Real-World XML Queries

Department of Software Engineering

Supervisor of the master thesis: Doc. RNDr. Irena Holubová, Ph.D.
Study programme: Software Systems
Specialization: Software Engineering

Prague 2015

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date 4th December 2015 Peter Hlísta

Název práce: Analýza reálných XML dotazů

Autor: Peter Hlísta

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: Doc. RNDr. Irena Holubová, Ph.D.

Abstrakt: Účelem této práce bylo shromáždit a analyzovat běžně používané XQuery programy.
Ke sběru dat z internetu je nejčastěji využíván program zvaný crawler. Součástí této práce byla analýza různých crawlerů a výběr nejvhodnějšího z nich. Tento crawler byl následně upraven tak, aby nevytěžoval servery, sbíral správná data a bylo možné jeho činnost pozastavit. Před započetím sběru dat bylo nejprve nutné určit, kde bude vhodné začít a jak dlouho by celý proces měl trvat. Data jsme po stažení očistili, opravili a zvalidovali. Předmětem analýz bylo používání XQuery jazyka a jeho gramatických konstruktů (symbolů). Také jsme analyzovali XML dokumenty používané v XQuery programech a výstupy z XQuery programů.
Hlavní přínosy práce jsou v množství stažených dat (v porovnání s jinými zdroji), v stažení XML dokumentů nad nimiž se dotazují, použití Analyzeru na analyzování reálných XQuery programů a spouštění těchto reálných XQuery programů nad jejich XML dokumenty.

Klíčová slova: crawler, analýza, reální XQuery, XQConverter, Analyzer

Title: Analysis of Real-World XML Queries

Author: Peter Hlísta

Department: Department of Software Engineering

Supervisor: Doc. RNDr. Irena Holubová, Ph.D.

Abstract: The aim of this thesis was to gather and analyze the real-world XQuery programs.
The data gathering process is usually performed using the crawler. Part of the thesis was to analyze different crawlers and to choose the most suitable one. The crawler was then modified, so it would not overload servers, gather the right data and be able to pause. Before main gathering two problems had to be solved – where to start the gathering and how long it will take. After the data were gathered, they were cleaned, corrected and validated. The subject of the analysis was usage of the XQuery language and its grammar symbols. We also analyzed the XML documents used by XQuery programs and outputs from the XQuery programs.
The main contribution of this thesis is the amount of the gathered data (in comparison with other sources), as well as gathering XML documents which are being queried, using Analyzer for analyzing the real-world XQuery programs and running this real-world XQuery programs over gathered XML documents.

Keywords: crawler, analysis, real-world XQuery, XQConverter, Analyzer

I would like to thank my supervisor Doc. RNDr. Irena Holubová, Ph.D. for helpful advices, reviews of thesis working versions and grammar corrections.

I would also like to thank my brother Igor for a lot of grammar corrections, Jakub Stárka, Ph.D. for working version of Analyzer with the libraries, Jakub Kúdela for helping with CommonCrawl specific problems and others for reading this thesis and all their helpful comments.

# Contents

# Introduction

The Extensible Markup Language (XML) [58] is wide-spread nowadays. It defines how to encode documents in a format that is both human and machine readable. Although the design of XML focuses mainly on documents, it is widely used for the representation of data structures.

There are many ways to obtain data from XML documents. We can obtain them manually or use languages developed for this purpose, e.g. XML query language (XQuery) [54] or XML Path Language (XPath) [41]. In case of larger documents, or when merging more XML documents, the languages are more suitable.

The real-world documents have already been analyzed couple of times. Since XML documents are used and queried a lot, the analysis of a such queries and even queries and the documents would be beneficial. Such analyses have not been performed so far, though they can be useful for programmers of the queries evaluators for optimization, and also for designers of the new languages used for querying XML documents.

XPath expressions are mostly one line queries. XQuery queries usually have more lines and are more complex. We will be using 'XQuery program' as name for XQuery query throughout this work. Since these programs tend to be complex, it is highly probable these programs will be saved and used again.

One of the aims of this thesis is to download as many XQuery programs as possible to find out which constructs are used. The next aim is to download also XML documents used for querying. We assume that only a few XQuery programs and none XML documents will be downloaded. Another assumption is that there are no XQuery programs without query body, to be used as library for other programs. Part of XQuery is XPath expressions and we expect that the half of these XPath expressions will use predicates and in average they will consist of two steps and will use only shortened types of axis. We also assume that most of the built-in functions of XQuery 1.0 is used besides the functions to work with time and date. The most frequently used function will supposedly be function 'fn:doc'. We assume that around 10% of queries will be also possible to be written in XPath 2.0 language, which is subset of language XQuery 1.0. Supposedly clauses Where and Order by are used at least in the half of FLWOR expressions. In the case of FLWOR expressions we also assume that the clause For is used more frequently than Let. Nested FLWORs will not be very frequently used. We do not think that we are going to find out some recursive functions. Concerning operators in queries we thing that additive and multiplicative type will occur the most.

The main aim of the thesis is to download as many real-world XQuery programs as possible and to compare the mentioned hypotheses with our results and possibly download XML documents which are used in these XQuery programs.

## Outline

First part of the thesis regards to terminology and technology connected with XML documents, existing papers and works about the XQuery language and

querying over the real-world data in the other languages.

We further look at the places where we can find XQuery programs and how are we going to download them. We suppose that we will use modified crawler to download them from the Internet.

In the second part of the thesis we will take closer look at the process of getting the real-world XQuery programs and eventually also the XML documents and their analysis. Results of those analyses will be compared with the hypotheses in the last part of the thesis and we will evaluate these hypotheses.

# 1. Technologies and Terms

This thesis is focused on crawling of the Internet. First part describes basic terminology to make the further text more comprehensible.

## 1.1 Internet Technologies

The Internet is a worldwide system of interconnected computer networks. This means the computer networks are connected throughout the world.

Every website on the Internet has its uniform resource locator[1] and part of URL is domain name. This name is translated by Domain Name System[2] servers to Internet Protocol[3] address. There are 2 versions used at the present time on the Internet IPv4 and IPv6, we will use both. The DNS servers can be overloaded by request and it is called Denial of Service (DoS).

All the websites compose the World Wide Web[4], also known as the 'Web'. Here are all websites composed of web pages which we will crawl to search for the data we need.

Web pages are hypertext documents and are written in the HyperText Markup Language[5].

Web pages consist of two main parts: body and head. The body contains the web page structure and the header contains meta information. One of the meta information is meta tag for crawlers.

This meta tag task is to help the crawler identify the content of the web page. One of these meta tags is Multipurpose Internet Mail Extensions[6] type and it describes the content of the body.

To acquire a web page we must use the Hypertext Transfer Protocol[7]. This protocol uses status codes for every request. Their form is XYZ where X is a number from 1 to 5 and YZ is a two-digit number. For the purposes of this thesis the relevant codes are 2YZ – Success, 3YZ – Redirection, 4YZ – Client Error (especially 404 Not Found) and 5YZ – Server Error.

## 1.2 Search Engine

To be able to find some information on WWW in reasonable time we need search engines. They are built out of several components:

**crawler**
: Crawler is also known as **robot**, **spider** or **bot** and we will refer to it as the **crawler**. It goes through IP addresses and downloads web pages assigned to them. Usually it runs in multiple instances or threads.

[1] http://www.ietf.org/rfc/rfc1738.txt – URL
[2] http://www.ietf.org/rfc/rfc1035.txt – DNS
[3] http://tools.ietf.org/html/rfc791 – IP
[4] http://www.w3.org/ – WWW
[5] http://www.w3.org/TR/REC-html40/ – HTML
[6] http://www.ietf.org/rfc/rfc2045.txt and http://www.ietf.org/rfc/rfc2046.txt – MIME type
[7] http://tools.ietf.org/html/rfc2616 – HTTP

**indexer**
Indexer categorizes and adds the links to databases with their related keywords or terms.

**presenter**
Presenter gets the search queries from users, searches the database and provides the search results.

## 1.3 Characteristics of The Internet

The Internet is an international network of networks that consists of millions of private, public, academic, business, and government networks. As such it has some interesting characteristics. Some of them are:

**Dynamic Data**
The content of the Internet is dynamic. This is caused by many factors like pages can be added or removed from servers, many sites have dynamic content, etc. This is the reason why the data collected in different points in times mostly differ.

**Incorrect Data**
Some documents found on the Internet may contain incorrect data.

**Invisibility for Crawlers**
Web crawlers cannot naturally search the whole Internet. Here are some of the facts why it is not possible:

- For some documents there are no references from other web pages.
- Some documents can be acquired only with login information (user name and password).
- There are advices on web domains for the crawlers what files and directories should not be crawled.

**Spider Traps**
A spider trap is a set of web pages that can intentionally or unintentionally cause a crawler to make an infinite number of requests [57]. Incorrectly constructed crawlers can crash because of this. One of the examples is a dynamic page with calendar. Crawler should avoid getting into these spider traps.

## 1.4 XML Family of Languages

XQuery is a query and functional programming language that queries and transforms collections of structured and unstructured data. It was mainly developed to query over an XML documents and it has syntax for creating a new XML document. It contains the XPath (for description see bellow) expression syntax to address specific parts of an XML document. It has an SQL-like[8] 'FLWOR

[8]http://www.w3schools.com/sql/ – Structured Query Language

expression‘ for performing joins. The FLWOR expression is constructed from the five clauses after which it is named: For, Let, Where, Order by, Return.

XML documents have a hierarchical structure and it can be interpreted as a tree structure called XML tree. The root element of XML document is mandatory and is also a root of XML tree. Many statistics are defined over this XML tree like: depth, fan-in, fan out, etc.

With XML documents are related Document Type Definition (DTD) [7] which are used for the validation of an XML document. A DTD is a collection of element declarations of the form $e \rightarrow a$, where $e$ is an element name and $a$ is its content model, i.e. regular expression. This regular expression can be an empty content model or a text content or a single element name or some combination of the content models. One of the elements `s` is called start symbol. It matches over root XML element. Now we can define depth of the content model:

- For the empty content model it is 0.
- For the text content it is 1.
- For the single element name it is 1.
- For the combination of the content models it is maximum from the depth of the content models plus 1.

The depth of XML we can define as depth of the start symbol content model.

The content model is mixed when it contains definition for the combination of the content models and the text content.

The element `e` fan-in is number of different existing parents for the element `e` in the same document.

The element `e` fan-out is number of different existing children for the element `e` in the same document.

The element `e` is recursive if exist element `d` in the same document that is descendant of `e` and has the same name. Beside these XML-related languages they exist also XML path language, XML Schema definition and extensible stylesheet language transformation.

XML Path Language (XPath) [41] is a query language for selecting the nodes from an XML document. XPath is going to be used for obtaining an analysis from XML representation of XQuery program. XPath 1.0 contains fewer built-in functions, it does not contain variables or iterations, etc. XPath 2.0 is reduced XQuery 1.0. It cannot contain some of the XQuery grammar symbols. There also exist navigational XPath 2.0. This fragment excludes the use of position information, aggregation and built-in functions. Value comparisons are allowed. Navigational XPath 2.0 is used for navigating in the XML tree and testing value comparisons.

XML Schema Definition (XSD) [55, 34]specifies how to formally describe the elements in an XML document.

Extensible Stylesheet Language Transformations (XSLT) [32] is a language for transforming XML documents into other XML documents and it uses templates for transformation.

There also exists another extension to XQuery language. It is called XQuery Update Facility [43]. It brings new extension into XQuery grammar. In this thesis we will not be analyzing these documents.

## 1.5 Lexical And Syntactical Analysis

Lexical analysis is the process of converting a sequence of characters into tokens. Token is a meaningful sequence of characters specific for language.

Lexical scanner is a parser, which converts characters into tokens. Tokens are passed to syntactic parser.

Lexical error means that lexical scanner found a character on place where it should not be. This can be caused by one of 3 factors. First, there is a character missing, second, there is an incorrect character or third, there is an excessive character.

Syntactic parser creates a syntax tree from the tokens. This tree is also called an abstract syntax tree (AST) and it is a tree representation of the abstract syntactic structure of the source code. This must conform to the rules of a formal grammar or a syntactic error occurs. Formal grammar is grammar (set of rules) where the context is not given.

The Extended Backus–Naur Form (EBNF) is mentioned in thesis and it is a family of meta syntax notations, any of which can be used to express a context-free grammar.

## 1.6 XPath Versus XQuery

XPath 2.0 is a subset of XQuery 1.0, so we also want to search for XPath 2.0. Because XPath are usually used as short queries they are not often put into files.

There is a method for distinguishing one language from the another.

The XPath expression to distinguish XPath 2.0 and XQuery 1.0 is [56]:

```
/Module/@version | //ModuleDecl | //Prolog | //LetClause |
//WhereClause | //OrderByClause | //ForClause/Type |
//ForClause/@posname | //QuantifiedExpr/InClauses/InClause/Type |
//Typeswitch | //Extension | //ValidateExpr | //OrderedExpr |
//UnorderedExpr | //Constructor
```

If the resulting hit count is zero, the analyzed file is written in XPath 2.0 and also in XQuery 1.0. Otherwise it is only XQuery 1.0.

# 2. Related Work

In this chapter we are discussing existing approaches and tools for data analysis. The data includes XML data, XQuery data, etc.

## 2.1 XML Analysis

In report [51] there are described extensive analyses of the real-world XML data. These analyses resulted into the following statistical results:

- The first set, so-called **global statistics**, consisted of properties of XML data such as the number of elements of various types, the number of attributes, paths, depths and the portion of text in documents. In XSD and DTD the depths are counted for each global element used in the sample XML documents as a root element, for recursive elements the authors take into account their lowest level(s) and an infinite level for expressing the recursion.

- The second set, so-called **depth statistics**, consisted of the distribution of depths per each category because the maximum and average depths are known from global statistics. The authors claim that they cannot get similar results for corresponding XSDs, because they are too much influenced by recursion.

- The third set, so-called **level statistics**, focuses on distribution of elements, attributes, text nodes, and mixed contents per each level of XML documents.

- The fourth set, so-called **fan-out statistics**, describes the overall distribution of XML data, in other words the number of descendants of each node.

- The fifth set, so-called **fan-in statistics**, is 'inverse' to the previous fan-out statistics. In other words the number of different ascendant of each node.

- The sixth set, so-called **recursive statistics**, deals with types and complexity of recursion.

- The seventh set, so-called **mixed-content statistics**, analyzes the structure and complexity of mixed contents. The number of mixed-content elements is already known from global statistics, so these statistics are focusing on average and maximum depth of mixed content and the percentage of simple mixed-content elements.

- The eighth set, so-called **DNA statistics**, is focusing on a pattern called DNA pattern. DNA pattern contains an arbitrary amount of trivial sub-elements and just one complex sub-element, so-called degenerated branch.

- The ninth set, so-called **relational statistics**, is focusing on so-called relational pattern. This pattern is analyzed using a set of relational statistics. There are two types of relational patterns – relational and shallow relational – and also the statistics are computed separately for both.

- The last set, so-called **schema statistics**, completes the analyses with analyzing XML schema specific constructs and their real exploitation.

The authors performed whole new analysis of XML documents. They focused on large (contrary to previous works [42, 48]) number of various attributes, and performed the analysis over aptly chosen various categories.

## 2.2 XSLT Analysis

In thesis [49] the author focuses on the XSLT language. The thesis is relatively new, from 2012. The author used a simple method of getting seeds (using common web search engine Google) and from these seeds he acquired about 20 000 files. After the cleaning process there remained around 6 000 files, which were used as an input for next phases.

The author divided all the XSLT documents into well arranged caterogies and also defined a typical XSLT document. These documents were analyzed so that an XSLT benchmark test would be created according to them. This is the greatest contribution of this work.

Another strong point of the thesis is that the author showed how it is possible to download the real-world data from the Internet and divide them into categories that were analyzed.

## 2.3 Historic XML Query Languages

In paper [30] the authors compare five languages used for querying over XML data. This paper is from the year 2000 and quite obsolete because at the time the standard for the XQuery language did not exist.

The major part of the work is concentrated on the comparison of the languages as of what constructs are present in them.

The current XQuery standard contains all constructs mentioned in this work.

## 2.4 SPARQL Analysis

SPARQL is a query language for RDF data [53]. This language consists of 4 different query types: ‘SELECT‘, ‘CONSTRUCT‘, ‘ASK‘ and ‘DESCRIBE‘.

The following two papers discuss its use over real-world data.

In paper [28] the authors are analyzing real-world data. They acquired the access to two different sources of data and used them independently so they were able to compare the results.

The first data source were DBpedia [6] server logs. DBpedia is a crowd-sourced community effort to extract structured information from Wikipedia and

make this information available on the Web. These server logs consist of data usage in the course of several months.

The second one is the Semantic Web Dog Food [17] which contains information about authors and publications.

These sources provided 5 million queries from DBpedia and 2 million from the Semantic Web Dog Food.

The results of analysis show that 'SELECT' is the most frequent type of query and is used in more than 95% of cases. Other types of queries ('ASK', 'CONSTRUCT', 'DESCRIBE') are used rarely.

The authors also focused on joins. Only 4.25% of all queries contain joins. In these queries the frequencies of joins are from 1 (in 2.66% of all queries), 2 (in 0.75% of all queries) up to 10 joins per query.

The work showed how real-world users create SPARQL queries. Authors expect that their results are valuable for designers especially in the tasks of query evaluation, planification and index construction.

In paper [29] the authors analyze RDF data streams. They created C-SPARQL (Continuous SPARQL) as an extension of SPARQL that is capable of querying RDF streams.

The data used by the authors were provided by the Social Network Glue. They only had to create and implement simple tool to convert the data from XML to RDF stream. They did not consider the ability to analyze the queries because their aim was to obtain results from large number of data (potentially endless stream of data).

## 2.5 SQL Analysis

In paper [39] the authors aim at discovering and preventing SQL injection attack where SQL injection is a technique where malicious SQL statements are put into entry field (for example entry field on web page) for execution.

The authors created a tool to detect and stop SQL injections. Four steps are suggested for their tool in order to work:

**Identify Hotspots**
: First, the application code has to be scanned for places where the code enters the underlying database.

**Build SQL-query Models**
: Second, each hotspot has to build a model that will represent all the possible SQL queries that may be generated at that hotspot. In particular, an SQL-query model is a non-deterministic finite-state automaton in which the transition labels consist of SQL tokens (SQL keywords and operators), delimiters and place holders for string values.

**Instrument Application**
: Third, the tool has to be connected (runtime monitor) to the application at every hotspot.

**Runtime Monitoring**
: Finally, it checks the dynamically-generated queries against the SQL-query

model in the runtime. If a query is violating the model it is rejected and reported.

The authors consider also the efficiency and they analyze it both theoretically and practically. They claim that the overhead from using their tool is negligible and their empirical evaluations confirm this.

The analysis of SQL queries in this paper consists of comparison of incoming queries against the created statistical model in real-time.

## 2.6 XQuery Analysis

In the paper [38] the authors are focusing on transformations between two languages, XQuery and XSLT. They use different processors (programs) to run these languages.

The authors use three technologies for description of translation from XSLT to XQuery and vice versa:

- the slightly altered Extended Backus-Naur Form[1]
- the abstract syntax tree
- the attribute grammar

XSLT and XQuery have the same expressive power but they have some differences. To overcome them the authors introduce additional transformational steps. For example, an XQuery query can be nested in another query whereas XSLT has templates.

The authors also introduce rules when these steps can be skipped, thus they optimize the total translation time.

For translation of an XQuery program into an XSLT stylesheet the authors use pre-processing and post-processing steps. They also attach various optimizations of these steps, since they are demanding. In some cases these steps can be performed together.

Other steps are provided by the authors for translation of an XSLT stylesheet into an XQuery program. Since XSLT is based on choosing templates, the authors have to deal with their simulation in XQuery.

In the end the authors describe testing of queries in the original language and translated queries. They are using the XMark benchmark [22] for these tests. As expected, the execution time of translated queries is a little bit slower. The optimizations suggested by the authors accelerates the execution times of the translated query by 13%.

In the paper [27] the authors take 5 available benchmarks and analyze them. However, this paper is from 2006 so it is relatively outdated. They point out that the benchmarks use outdated XQuery dialect and they contain mistakes. This is the reason why 4 of them are not used in the research articles, according to the authors.

Next, approximately 1/3 of the queries contained errors (precisely 48 out of 163). The errors varied as follows:

[1] http://www.iso.org/iso/catalogue_detail.htm?csnumber=26153 – EBNF

- 35 queries contained static errors – errors discovered during parsing the XQuery queries.
- 11 queries contained dynamic type of errors – errors discovered during an execution of the XQuery queries.
- 2 queries contained semantic errors – i.e. the queries return a (possibly empty) sequence of items which does not correspond to the natural language description of the query answer. Both semantic errors were caused by incaution.

The authors tried to express every single test query in XPath 2.0, XPath 1.0 and their parts like Navigational XPath 2.0 and Core XPath [35]. Navigational XPath 2.0 does not use position information, aggregation and built-in functions. Core XPath does not use position information, built-in functions and comparison operators. In total, only 16 from 163 queries were genuine XQuery queries.

A well-chosen subset of tests provides the same information as the whole benchmark set (and is not time consuming). So, the particular test queries were checked whether they are redundant. Those queries which provided similar outputs were marked as redundant.

The performance similarity was defined by the authors as follows[2]:

$$q_i \equiv q_j \Leftrightarrow$$
$$\forall E, \forall D \, |Func(g_i, E, D) - Func(g_j, E, D)| \leq 15\% \cdot \frac{Func(g_i,E,D)+Func(g_j,E,D)}{2}$$

The function $Func$ counts the average time of query execution for each document $D$ using engine $E$. Query $q_i$ and $q_j$ are performance similar if for each query engine $E$ and for each document $D$, the difference in execution times of the two queries is less than 15% of the average execution time of these two queries.

Note that the authors do not give any reasons for choosing 15%.

In the end, the authors focused on micro benchmark queries. They were aware of joins being expensive operations and that there were heuristics which effectively evaluated joins in other languages. XQuery is considered to be harder to detect a join (because of its complexity), so they concentrated on the quality of implementation of joins detection. For this task they created a set of queries with different parameters:

**Syntactic patterns**

**where form**

Join where comparison of elements is done in Where Clause:

```
for $a in A, $b in B
where $a/@att1 = $b/@att2
return
($a/@att1, $b/@att2)
```

[2]formula is modified, because the authors made a mistake in it and because of its length

**pred form (predicate form)**
Join where comparison of elements is done in For Clause by filter:

```
for $a in A,
    $b in B[$a/@att1 = ./@att2]
return
($a/@att1, $b/@att2)
```

**if form**
Join where comparison of elements is done in Return by If construct:

```
for $a in A, $b in B
return
if ($a/@att1 = $b/@att2)
then ($a/@att1, $b/@att2)
else ()
```

**filter form**
Join where comparison of elements is done in Return by filter:

```
for $a in A, $b in B
return
($a/@att1,$b/@att2)
[$a/@att1 = $b/@att2]
```

These are forms for the logically equivalent ways of expressing the same value-based join. A value-based join is a join where the join condition is an equality of attribute values.

**Number of join conditions**
The authors used from 1 to 3 conditions per query.

**Boolean connectives**
The authors used conjunctions and disjunctions to combine multiple join conditions.

**Join types**
The authors considered three types of joins: simple (xs:integer), id/idref chasing (xs:ID, xs:IDREF) and self-join.

**Document-level equivalence classes**
The query set is divided into several subsets containing queries that are equivalent (have the same result). Two queries belong to the same equivalence class if they have the same intermediate results (i.e. results of the path expressions A and B) and final result.

In this paper, eleven various joins were used altogether (that means 44 different queries). By examining them the authors found out that engines were not always able to detect the join, so for specific forms and other parameters they got different execution times.

By this work the authors contributed to improving XQuery benchmarks (corrected mistakes, detected redundant queries, ...). Note that only XMark was used because other benchmarks contained errors.

In the paper [50] the authors took the benchmark queries and different database systems and compared them. The paper is aimed at a detailed comparison of six XQuery processors using available benchmark tests. These processors are three stand-alone and three XML/XQuery database systems. For simplicity the authors named both types 'engines'. They explain in detail all the problems they encountered during extensive testing of these six engines.

Five freely distributed XQuery benchmark tests were used. Each test was run n+1 times (in this study 3+1), the first run is used as a 'warm-up'.

Platform XCheck [26] was used to run given tests. It supports only one input file for one query. For this purpose the authors created a single XML file which consisted of the URIs of all documents of each multi-document benchmark. To the each query was added a preamble to obtain the sequence of documents on-the-fly.

The database systems required a running server so the authors decided to solved this problem by starting the server before running the queries and stopping it after performing of the queries. That means they put a command to start the server at the beginning and a command to shut down the server at the end of a query set. At one time only one engine was running. To preserve the same conditions for all engines they also did not optimize any of the database systems.

62 queries out of 163 had to be fixed by the authors before using them. Despite these fixes some queries were still returning syntactic or runtime errors.

The execution time was divided into five parts: query translation, query execution, result serialization, document processing and communication. However, some of the engines did not return all the values and in some cases the sum of the parts was not equal to the total sum. The authors explained this difference as the time spent on communication. In some cases the time returned by an engine has exceeded the execution time. This was the reason why the authors said the partial time could not be considered as reliable. This is also the main reason why there is no 'best' engine. So these results serve mainly for the engine developers to fix the bugs.

This paper focused on engine installation, engine setting, fixing the bugs in benchmark tests, running big numbers of test and their further presentation. The aim was achieved. The authors provided several interesting ideas – for example how to solve the problem of loading more files if it is possible to load only one file. Considering the given number of tests the authors managed to maintain obviousness.

## 2.7 Conclusion

The described papers analyze different languages. The first set analyzes XML, DTD and XML Schema documents for their validation and complexion. Surprisingly, neither DTD nor XML Schema are frequently used in real-world data.

The paper [30] provides an overview of languages we could use to query XML data before XQuery language was standardized so this paper provides relatively obsolete background.

The other two papers [28, 29] show how to perform an analysis of a query language when used with real-world data. These two papers focus on an analysis of the SPARQL language. The paper [39] which focuses on SQL queries analysis shows the creation of the statistical model of queries and their sub-sequential use in praxis using dynamic analysis.

The last three papers discuss the XQuery language. The first one describes transformation of XQuery from and to XSLT. Since it is a difficult process, most of the paper is about its heuristics. The second one evaluates XQuery benchmark tests – their modification like bug fixing and reduction of unnecessary queries. The last paper [50] is about XQuery engines and it describes a complicated process of their comparison. It offers many tests and many results to interpret. To perform the comparison it uses queries from XMark benchmark.

None of the listed papers discusses an analysis of real-world XQuery programs.

It would be interesting to gather real-world XQuery queries and detect the join form they are using (according to paper [50] about comparison of XQuery engines).

For engine developers it would be intriguing to determine which constructs of XQuery are used frequently and which only seldom (it would make the process of deciding which parts of engine should be optimized first easier).

Users would be aware of the most common mistakes made in XQuery language and therefore it would be easier to avoid them.

The aim of this thesis is to gather real-world XQuery programs, XML documents referenced in them and analyze them.

# 3. Occurrence of XQuery Programs

First, we need to obtain the input data for the analysis. We are starting the search on the W3[1] website. It maintains most of the standards, including XQuery standards, used on the web. In particular, there are Use Cases [33] and a Test Suite [21] in the XQuery language section.

## 3.1 XQuery Use Cases

Use cases were created as examples of the real-world application of XQuery. The authors sorted out 78 queries into these groups: experiences and exemplars (12), queries that preserve hierarchy (6), queries based on sequence (5), access to relational data (18), standard generalized markup language (11), string search (5), queries using namespaces (8), recursive parts explosion (1) and queries which exploit strongly typed data (12).

This is an example of a query from the group 'queries based on sequence':

In the Procedure section of Report1, what Instruments were used in the second Incision? Given Use Case query:

```
(: insert-start :)
declare variable $input-context external;
(: insert-end :)

for $s in doc("report1.xml")//section[section.title = "Procedure"]
return ($s//incision)[2]/instrument
```

Note that we are not going to include assumed output and input data because of the length of input data. A bigger example is provided in Attachment 10.7.

The fact that all the queries have input data over which they query and also relevant results to them is considered as an advantage. A disadvantage is that we cannot use these queries since they are not real-world XML queries.

## 3.2 XQuery Test Suite (XQTS)

These queries are used to test correctness of implementations of XQuery. The last version of XQTS[2] contains over 19,000 test cases/samples. This Test Suite also contains all the Use Case queries from the XQuery Use Case.

A fraction of all the queries are incorrect queries – to test, whether the implementation of XQuery language can deal with errors.

Again, for our purposes we cannot use these queries as they are not real-world XML queries.

[1] http://www.w3.org/ – The World Wide Web Consortium

[2] http://dev.w3.org/2006/xquery-test-suite/PublicPagesStagingArea/ XQTS 1.0.3

## 3.3 XQuery Benchmarks

As example of this group we describe one specific benchmark. In Section 2.6 we mentioned XMark benchmark [22] which contains 20 queries. These queries should make most of the simulation of real-world queries to reflect the real usage of XQuery. However, the usage of the language is changing so they might not reflect the reality. Again, for purposes of this thesis they are not real-world XML queries.

## 3.4 XQuery and Big Data

A frequently discussed topic nowadays is so called Big Data. Big Data is high-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation [2].

Oracle XQuery for Hadoop [14] is currently the only implementation of XQuery language which works with Big Data. The Apache Hadoop software library is a framework that allows the distributed processing of large data sets across clusters of computers using simple programming models [9].

Hadoop MapReduce is one of the modules of Hadoop. It is a system for parallel processing of large data sets. MapReduce program is composed of a Map and Reduce procedures. Map procedure provides filtering and sorting and Reduce procedure provides summary operation.

The Oracle XQuery for Hadoop is an extension for Hadoop which transforms an XQuery program to MapReduce task.

However, the queries used by this program have limitations:

- The main XQuery expression must be one or more For, Let, Where, Order by and Return (FLWOR) expressions.
- Each top-level FLWOR expression must have a For clause that iterates over an Oracle XQuery for Hadoop collection function. This For clause cannot have a positional variable.
- Each top-level FLOWR expression can have optional Let or Where clause. Order by clause is not allowed.

Unfortunately, public examples of this kind of XQuery programs are not available. In order to obtain the data we would have to request it from corporations which process Big Data with this tool, and that would be costly.

## 3.5 Common Crawl

Common Crawl [4] is an open repository of web crawled data that is universally accessible and analyzable. The corpus contains petabytes of data collected over the last 7 years. This huge data-pack contains raw web page data, extracted metadata and text extractions [4].

XQuery programs are text files and so there is a high chance that this project contains them.

The data is saved on Amazon servers and is available directly on the Internet or via AWS[3]. AWS provides a function of running virtual computers for the purpose of performing various analyses of the data saved with AWS. However, the service for running virtual computers is paid.

This repository contains the data collected during various time periods (e.g. August 2014) – so called snapshots. The data may be used as a benchmark for tests e.g. whether we have found a reasonable number of publicly free XQuery programs.

## 3.6 Rest of The Internet

And then there is the rest of the Internet. In order to be able to obtain the data from the Internet we have to create a crawler which would crawl the individual websites and search for XQuery programs. It is not a simple task since XQuery programs are the text files with specific content. Part of the data can be found using a file extension but in the rest of the cases we have to estimate whether it might be an XQuery program using heuristics.

It also takes a great amount of resources and time to crawl the whole Internet. That is why we can crawl only a part of it.

The following list sums up various places where XQuery programs might be found nowadays:

- XQuery Use Cases [33]
- XQuery Test Suite [21]
- Benchmarks like XMark benchmark [22]
- Oracle XQuery for Hadoop [14]
- The rest of the Internet
- Common Crawl [4]

[3]https://aws.amazon.com/ – Amazon Web Services

# 4. Analyzer

This thesis requires a tool for XQuery programs analysis. The tool Analyzer [44] seems to be a promising choice. This whole chapter discusses its features.

Analyzer is a framework for advanced data analysis. It was created at the Charles University in Prague as a software project in 2009. The basic version of Analyzer is using a system of plug-ins and in following years were also created many new. It is designed to be able to analyze any kind of data under the condition an adequate plug-in exists. However, the first kind of data analyzed with this program were XML documents. In particular the following attributes of XML documents were the subject of the analysis [46]:

- The size of the XML document e.g. in bytes, by the number of elements, or by the number of attributes
- The maximum depth of the XML document (the longest path from the root to a leaf)
- The distribution of various types of content models over different levels
- The recursion of elements
- The maximum and average element fan-outs
- The usage of XML Schema [55] versus DTD [7]
- The distinct element/attribute name usage
- The namespace usage

Next, the DTD data was a subject of analysis. In particular the following statistics were evaluated [46]:

- The size of the DTDs, e.g. the number of declarations of elements, attributes, notations, entities etc.
- The number of DTD specific declarations of element content (`empty`, `any`, etc.)
- The number of DTD specific declarations of attribute optionality (`#REQUIRED`, `#IMPLIED`, etc.)
- The usage of keys (`ID`, `IDREF`(s))
- The maximum, minimum and average depth of DTDs. Note that contrary to the depth of an XML document, the DTD depth is calculated using the number of rules of the grammar applied from the start symbol to the leaf (`PCDATA`).
- The average and maximum fan-ins (the number of different existing parents for a given node in a particular document) and fan-outs (the number of different existing children for a given node in a particular document)

Then the XSD data was analyzed in a similar way as XML data because each XSD is expressed in XML. Though the analysis was very similar, there were some special measurements added to the XSD analysis [46]:

- The type specification (`simpleType`, `complexType`)
- Restriction and extension of existing types
- The content model of elements (`sequence`, `choice`, etc.)
- Element groups and attribute groups (`group`, `attributeGroup`)

Analyzer also analyzed XQuery data but only on the queries from the XQuery Use Cases and the XQuery Test Suite as described in paper [46]. This paper evaluates the occurrence of core elements of the language and occurrence of XPath language axes used in the XQuery programs. It is just a demonstration that the program works (correctly), since it correctly converts all valid XQuery programs (programs without parser errors).

## 4.1 Architecture

As we can see in Figure 4.1, Analyzer framework consists of two separated levels. The first level contains so called **Shared Components** – components shared among all the projects. Analyzer can run simultaneously more projects each of them analysing various datasets. Shared components are instantiated only once in a running Analyzer application. The most important component is **Launcher**. It is responsible for executing tasks (small units of computations). Graphical User Interface (**GUI**) is a robust environment for creating projects, managing projects, running analyses and browsing reports. **Plug-ins** are meant to have the analyzing logic. Hence, for every type of analysis there must exist an appropriate plug-in.

The second level contains **Project Components**. These components are used by each project independently. In other words, it is the part which provides isolated environment for every project.

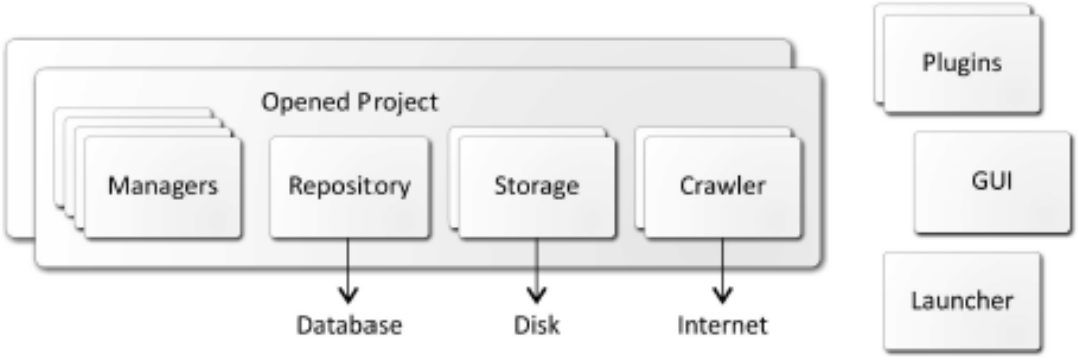



Figure 4.1: Architecture of Analyzer [46]

The project components are as follows:

**Crawler**
The crawler gathers data from the Internet.

**Storage**
The storage obtains the data (imports documents) from the file system. It also saves data (results of analysis) to the same file system.

**Repository**
Repository is a local database for each project. All calculated analytical data and the most of the configuration meta data are saved here.

**Managers**
Managers are responsible for creating, modifying and processing of the documents, collections (set of documents satisfying the same filtering criteria) and reports.

## 4.2 Implemented Crawlers

In this thesis there will be needed the real-world data obtained from the Internet. Analyzer obtains data in two different ways – from the local file system or by downloading them using a crawler.

The first version of the program did not involve a crawler and the documents were imported only from a local file system [57]. In the next version there were two crawlers implemented into the Analyzer:

**Simple Embedded Crawler**
This Crawler was created from scratch using a simple Java mechanisms. It works in the synchronous mode what means that the crawler always waits for the response before sending another request. It is used only for searching for the resources during the analysis.

The crawler is set to lower the load of the target server by using simple serialization of the requests and the number of requests is limited to one request per second.

**Egothor Embedded Crawler**
This Crawler was created as an adapter to the existing crawler from the Egothor Project [36] programmed by Leo Galamboš and collective. The behaviour of the original crawler was modified in the following way:

- The support for notifying about downloaded documents using a listener was added.
- The reaction for external signalization was added to stop crawling without the need of using the telnet.
- The support for URL injection without using the telnet was added.

Since then the Egothor Project released the new version – Egothor2 [31].

An adjusted Apache Nutch [1] crawler is also included in the Analyzer. This one was implemented to automate the data collection process and download full structures of XML documents. Additional improvements of the original Apache Nutch [1] are as follows:

**Improved address filtration**
: The crawler avoids unwanted protocols (e.g. mailto, javascript, etc.) and file formats (e.g. PDF, MP3 etc.).

**Altered document filtration**
: The crawler ignores excessively large HTML documents, assuming that they are unlikely to contain any XML-related links.

**Added whitelisting**
: Whitelisting apparently XML-related documents based on their reported MIME type and a part of contents.

**Added blacklisting**
: Blacklisting of unwanted documents. Note that due to the widespread errors in the web content, the authors found blacklisting more efficient than whitelisting.

**Altered scoring mechanism**
: The crawler makes XML-related data more favourable in the download queue.

**Added XML-based documents parsing**
: Parsing XML-based documents and locating external references in them.

## 4.3 Analyzer Plug-ins

As we have mentioned, Analyzer is an universal analysis tool. For every kind of data there is a special plug-in to analyze them. There are also plug-ins for other functionalities in Analyzer. In particular, there are 8 predefined types of plug-ins:

**detector**
: Detector recognizes the type of the processed document.

**racer**
: Racer looks for outgoing links in a given document.

**corrector**
: Corrector attempts to repair a content of a given document.

**analyzer**
: Analyzer produces the results of the analysis for a given document.

**collector**
: In order to be analyzed, all the data must be classified into collection. The analyses are then made over particular collections. The collector classifies the documents into collections of a given cluster.

**provider**
: Provider creates reports by aggregating results of the documents in a collection.

**viewer**
: Viewer enables browsing of computed results over a document.

**performer**
: Performer enables browsing of computed reports over a collection.

## 4.4 XQuery Analysis Plug-in

As we have mentioned, in this thesis we are especially interested in the plug-in for XQuery analysis.

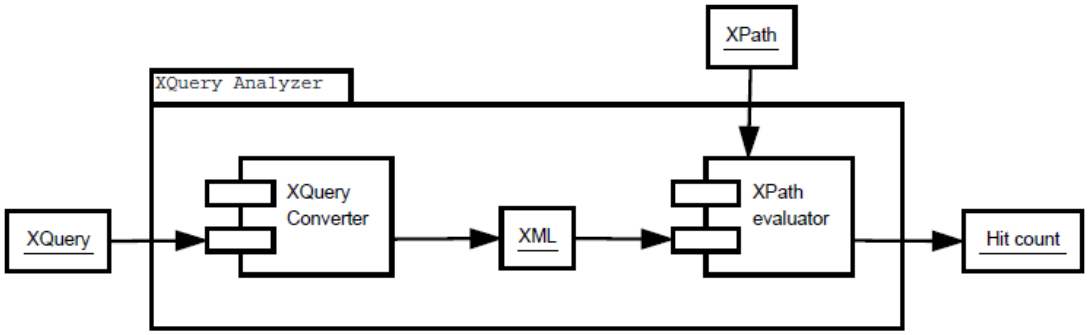



Figure 4.2: The structure of XQAnalyzer component [56]

Figure 4.2 shows the XQAnalyzer component of Analyzer created in thesis [56] which enables to analyze XQuery programs. It consists of two parts. The XQuery Converter gets an XQuery program and creates its XML representation. This representation is then sent to an XPath processor together with XPath expression. The final output is the number of XPath results.

## 4.5 XQuery Converter

XQuery Converter (XQConverter) is a part of XQAnalyzer. Not only this is included as a plug-in for Analyzer but it is also an independent command line program.

This tool creates an XML representation of an XQuery program. For illustration let us use the following simple example:

```
for $a in A
order by $a
return $a
```

From this example we get the following XQConverter output (an XML document):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Module type="main">
  <QueryBody>
    <FLWOR>
      <TupleStream>
        <ForClause varname="a">
          <BindingSequence>
            <Path initial-step="context">
              <Step>
                <Axis abbreviated="true" direction="forward"
                 kind="child">
                  <NameTest name="A"/>
                </Axis>
              </Step>
            </Path>
          </BindingSequence>
        </ForClause>
      </TupleStream>
      <OrderByClause stable="false">
        <OrderSpec>
          <VarRef name="a"/>
        </OrderSpec>
      </OrderByClause>
      <ReturnClause>
        <VarRef name="a"/>
      </ReturnClause>
    </FLWOR>
  </QueryBody>
</Module>
```

For better association of the original XQuery program to its XML representation we used colors. All the elements of XML representation are mostly self-explaining and described in Attachment 2 10.7.

The XQuery language is a complex language, hence the XQConverter cannot be a simple program which calculates words. The following example illustrates that the same word can have different meanings in different parts of the program:

```
for $for as for in "for" return <for/>
```

From this program where every `for` has a different meaning depending on the position in the program [47] XQConverter creates the following XML representation:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Module type="main">
  <QueryBody>
    <FLWOR>
      <TupleStream>
        <ForClause varname="for">
```

```
            <Type cardinality="one">
              <AtomicType name="for"/>
            </Type>
            <BindingSequence>
              <Literal quot-mark="double" type="string" value="for"/>
            </BindingSequence>
          </ForClause>
        </TupleStream>
        <ReturnClause>
          <Constructor kind="direct" type="element">
            <Name name="for"/>
          </Constructor>
        </ReturnClause>
      </FLWOR>
    </QueryBody>
</Module>
```

The first `for` is hidden in the element called ForClause. The second one is a variable and it is included in the attribute called varname. The third one is AtomicType, the fourth one is a string Literal and the last one is the Name of the returned element.

In order to be able to include the syntax of XQuery language, the output of XQConverter is respectively sophisticated (a more complex example is provided in Attachment 3 10.7).

### 4.5.1 Architecture of XQConverter

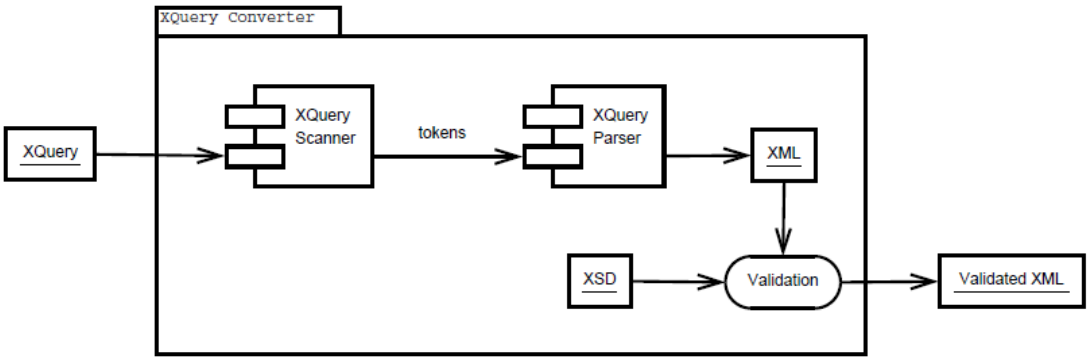



Figure 4.3: Architecture of XQConverter component [56]

As we can see in Figure 4.3 the XQConverter component consists of tree parts.

The first one includes an **XQuery lexical scanner**. Such a component is sufficient for calculating, e.g., the usage frequency of the individual words with a given meaning.

However, the XQuery language constructs and their combinations are more complex. To analyze them an **XQuery parser** is needed, too. By being connected to a **lexical scanner** it is able to recognize the structure of the programs according to the XQuery grammar [23].

The stream of the data goes into the scanner where it is transformed into lexical tokens. The tokens form the input for the parser which instead of building a syntax tree builds the XML representation of the given XQuery program.

This final XML document can be queried by a classical XPath/XQuery processor for various statistical information.

## 4.6 Summary

Analyzer is an extensive program (it uses many libraries) and it is implemented in Java version 1.6. The current Java version is 1.8, hence there is a possibility that some of the used libraries will require updates due to arisen error corrections. This can lead to highly time-consuming process without proper results.

If the errors after updating cannot be corrected, we can use an alternative console version of XQConvertor and perform analyses over the created documents using an XPath/XQuery processor.

# 5. Analysis of Web Crawlers

First of all we need to focus on data acquisition. In this chapter we will describe the term crawler, various types of crawlers, functions of the crawler and pros and cons of several specific crawlers.

## 5.1 Crawler

In general we have more options how to chose the optimal crawler for our purpose.

- It can be written from scratch. However such a decision has both positive and negative aspects. The advantage is that crawler would exactly fit our needs. On the contrary, it would take a lot of time to design the model, program it, test and debug it.
- It can be chosen from existing crawlers and edited or extended to fit our needs. This approach is less time-consuming so we can focus more on the heuristics for crawling XML data and XQuery queries and not on the basics of crawling the Internet in general. However, some modifications can be more time consuming compared to programming them from a scratch. Also a situation can rise where we would not be able to implement a feature because of a bad crawler design.

We have decided for the second approach so, in order to choose the best crawler for our purposes, we have analyzed and classified the existing crawlers.

### 5.1.1 Crawler Types

There are several criteria according to which we can classify crawlers. In particular, according to search methods there are depth first and breadth first method crawlers. The depth first searching crawlers are generally called **topical crawlers** or **topic-directed crawlers**. They are used for searching in the specific domain like **.cz**, **.net** or they are used for searching for a particular topic like the sport websites. These crawlers prefer depth first crawling of a website.

The breadth first searching crawlers are more frequent. These crawlers do not depth-crawl the website but crawl as many websites as possible. This method is suitable for parallelism and that is the reason why these crawlers often use parallel processing. Parallelism on one computer is done by multi-threading or by running more processes (running more crawlers simultaneously or running a crawler with more threads). Parallelism on more computers is acquired by a distributed web crawler.

A bottleneck for a crawler running on a single PC is CPU speed and the speed of the network connection. A distributed web crawler shifts these bottlenecks to local DNS server [57], because breadth first search generates a lot of requests and can even create Denial of Service (DoS) on DNS server. DoS arises when a computer or a network freeze, because they are overloaded by requests.

### 5.1.2 Crawler Policies

We can classify the crawlers according to their policies. The first one is **selection policy**, i. e. the way how the crawlers choose web pages to download. We divide them into the following groups:

**Restricting the followed links**
: The crawler only downloads HTML pages. All the other MIME types are ignored. It can be done using the HTTP HEAD request and afterwards MIME type examination or by exploring the file extension in the URL address.

**Path-ascending crawling**
: The crawler examines the whole URL, from which it derives other URLs which are crawled afterwards. For example, when given a seed URL of `http://www.w3.org/TR/2006/REC-xml-20060816/`, it will attempt to crawl `/TR/2006/REC-xml-20060816/`, `/TR/2006/`, `/TR/` and `/`.

**Focused crawling**
: This kind of crawler, also called **topical crawler** or **focused crawler**, attempts to download web pages that have similar content. The similarity means that the web pages contain similar words, collocations or identical URL(s).

  For example an academic-focused crawler crawls all academic resources.

The next, so-called **re-visiting policy**, relates to the fact that the Internet incorporates a large quantum of data and, in addition, a lot of data is constantly changing and new data appears. This is the reason why crawlers that are constantly searching the Internet and saving new data are necessary. To be able to effectively search over these data, an index is usually used. That means that a set of terms is chosen from a web page for local saving in a database and indexing.

We say that index becomes out-dated when specific terms change, because the web page changed. Therefore it is only rational to re-crawl the data and update the current index. Crawlers using this policy can be divided to:

**Uniform policy**
: All web pages are re-visited with the same frequency. However this policy is not used very often.

**Proportional policy**
: Web pages are re-visited according the estimated change frequency. The web pages that change more often are crawled more frequently. The challenge is to set the crawling frequency in correspondence with the estimated change frequency.

A crawler can naturally retrieve more data in a shorter time than a human searcher. Unfortunately this can overload some websites. Consequently a kind of etiquette must be followed during web crawling. A part of this problem is solved by **The Robots Exclusion Protocol** [15], also known as 'the robots.txt protocol' or 'Robots Exclusion Standard', based on the idea of giving information

about how to crawl the website from the website owner to the crawler. It is a file that typically contains the information which directories not to crawl. They can contain unimportant information about the website or they can contain so-called **spider traps**. A spider trap is a set of web pages, that makes the crawler run in the cycle and produces potentially infinite number of requests that leads to crawler crash. The file is always located in the root directory of the website. The Robots Exclusion Protocol can specify behaviour for every crawler independently.

The second type of such a strategy is so-called **politeness delay**. It is a delay between requests to the same host which reduces the load on the servers. Too many request in short time can disrupt normal website activity. This politeness delay is sometimes specified in the Robots Exclusion Protocol, too.

The last type of policy is **parallelization policy**. The crawler can run on a single PC as a single process with one search thread, However, this is very ineffective. That is why crawlers are multi-threaded and multi-processor (they can run on multiple computers). This speeds up the process of crawling of the large number of web pages (because each thread or process search on different website). However, the crawlers that run on more computers simultaneously must coordinate their work. This kind of crawler is called a **distributed web crawler**.

### 5.1.3 Filters Used by Crawler

Except for the mentioned policies, specific filters for the unwanted data are required, too. This way the crawler can go through web pages and documents that have higher probability to contain desired data.

Here are some of the filters used in crawlers:

**Whitelist**

A **Whitelist** contains rules, which define whether a given file should be downloaded/accepted. These rules mostly consider the file type determined either by file extension or MIME type. The rules can also consider what content should contain the given file.

The more complex the rules are, the longer it takes to process each file and the crawling process takes longer time. On the other hand, simple rules may either filter out required files or download/accept a large number of unwanted files.

However, in general we do not know how the downloaded files will look like, we cannot effectively set the rules in advance [57].

**Blacklist**

On the other hand **blacklist** contains rules describing which files should not be downloaded. In this case the rules are only based on observing the file extensions and the MIME type.

This approach removes only unwanted files. For example when we search for the text data we filter out all binary data.

**Address filtration**

Address filtration means that the addresses from a given list are not downloaded. Usually it is implemented as a list of URL prefixes that is checked

with the address to be downloaded. This can be extended also for particular protocols, so that for example the protocols different from HTTP are filtered out.

**Large documents**
Large documents present a specific problem. These documents can take a lot of memory during their parsing which can, in the worst-case scenario, lead to program crash. That is the reason why we define the limit (size) which separates large documents from others and filter them out. So we have to define the limit which separates large documents from the rest.

## 5.2 Existing Crawlers

At present we can choose from many existing crawlers. We will focus on the most popular ones and choose the most suitable candidate for our purposes. At the end of the chapter we provide a summary of our findings.

**Heritrix** [45]
Heritrix is an open source project with extensible functionality and archival features. It searches the web with the breadth first method and uses more threads to perform the task. The authors claim that it is designed to respect the Robots Exclusion Protocol and META robots tags[1]. It should also crawl web pages without disrupting normal website activity. Some examples of how to use Heritrix can be found in user guide on its website. It is written in Java and is still updated frequently.

This crawler already has a lot of the required features. However to use this crawler for obtaining XQuery programs, some modifications will be necessary. For example the indexing and archival parts of Heritrix will not be used, but recognizing possible XQuery programs part needs to be implemented.

**WebSPHINX** [20]
**Web**site-**S**pecific **P**rocessors for **H**TML **IN**formation e**X**traction is a Java class library and an interactive development environment for web crawlers that browses and processes web pages automatically. It consists of two parts. The Crawler Workbench is a graphical user interface that lets the user configure and control the customizable web crawler. The WebSPHINX class library has the following main features:

- The library contains multi-threaded web page retrieval in a simple application framework.
- The library supports the Robots Exclusion Protocol.
- The library uses pattern matching, including regular expressions, Unix shell wildcards, and HTML tag expressions.

However, it has some negatives. Firstly there is no mention about politeness delays and, secondly, the project was abandoned in 2002.

[1] optional tags at the beginning of web page HTML code

**JSpider** [10]

**JSpider** is a highly configurable and customizable web crawler engine. According to its author it is a multi-thread crawler designed to crawl a small number of sites. It respects the Robots Exclusion Protocol.

However, it contains some disorganized examples in the documentation and can use only one seed. The last project activity is from 2003 (without stable release).

**WebLech** [19]

The authors of WebLech created a full featured website download and mirror site[2] tool in Java which supports many features required to download websites and emulate standard web browser behaviour as much as possible. It is a multi-threaded crawler and it can be both depth-first search or breadth-first search type of the crawler. It also contains a kind of URL filtering.

On the other hand, there is neither a mention about the Robots Exclusion Protocol, nor about the politeness delays. It was abandoned in 2004.

**Arachnid** [52]

**Arachnid** is a Java-based web spider framework. It includes a simple HTML parser object that parses an input stream which contains an HTML content. Simple web spiders can be created by sub-classing of the Arachnid and by adding a code called after each web page of a website is parsed. It is a simple crawler with only one thread, no filters, no Robots Exclusion Protocol and no politeness delay. The project is updated frequently. However the authors recommend to use it only for private use on local servers, because it may put a large load on servers and even crash it.

**JoBo** [12]

**JoBo** is a simple program to download complete websites to a local computer. Basically it is a web spider. The main advantage compared to the other tools is that it can automatically fill in forms and also use cookies[3] for session handling. It also features very flexible rules to limit the number of downloads by URL, size and/or MIME type.

It is single-thread crawler and supports the Robots Exclusion Protocol and the politeness delay is implemented.

**Web-Harvest** [18]

**Web-Harvest** is an opensource web data extraction tool written in Java. It uses well established techniques for text/XML manipulation such as XQuery, Regular Expressions, etc.

It is a multi-threaded crawler but it does not support the Robots Exclusion Protocol or politeness delay. The project was left at the stage of Beta testing in 2010.

[2] A mirror site is a replica of an already existing website.

[3] An HTTP cookie is a small piece of data sent from a website and stored in a user's web browser while the user is browsing that website.

**Crawler4j** [37]

**Crawler4j** is a Java library which provides a simple interface for crawling the web. This crawler is multi-threaded with the support for the file filters, Robots Exclusion Protocol, and politeness delay. Version 3.5 is from 2013.

**Ex-Crawler** [40]

The **Ex-Crawler** project is divided into three sub-projects. The first one is the Ex-Crawler server daemon. It is a web crawler written in Java. The second part is the GUI and the third part is the web search engine. The web search engine part is written in PHP.

The **Ex-crawler** server does not support the Robots Exclusion Protocol and politeness delay. It is multi-threaded and can filter files according to their extensions. The project was abandoned during alpha testing in 2010.

**Bixo** [3]

**Bixo** is an open source web mining toolkit. It uses Cascading[4] and Hadoop projects. It is a topical crawler, uses the Robots Exclusion Protocol and politeness delay and it is a multi-threaded crawler. It is frequently updated.

**Abot C# Web Crawler** [25]

**Abot** is an open-source C# web crawler built for speed and flexibility. It is based on events so you register your functions on these events. It is also possible to create plugin for core interfaces to take control over the crawler. It is a multi-threaded crawler and is kept up-to-date.

**Scrapy** [16]

**Scrapy** is a fast high-level screen scraping and web crawling framework, used to crawl websites and extract structured data from their pages.

It does not support more threads but is possible to run more separated spiders at once for faster download. It is written in Phyton and frequently updated.

**Larbin** [13]

**Larbin** is a C++ crawler. It is multi-threaded but prefers using select[5] instead of a lot of threads for efficiency purposes. It uses standard libraries and can run on Linux. The bottleneck is speed. The faster the Internet connection is the more web pages are crawled. It was created as a part of the XYLEME project [24]. The XYLEME is XML-based learning content management solution.

Larbin was not updated since 2003.

**Egothor**

**Egothor** was partially developed by the students of the Faculty of Mathematics and Physics, Charles University in Prague. It can be configured as a standalone engine, metasearcher, peer-to-peer HUB and can be used as a library for an application that needs full-text search. Egothor version 1 is integrated into the Analyzer 4.2. The latest version is 3.1.8 from 2014.

[4] http://www.cascading.org/ – Cascading is the proven application development platform for building data applications on Hadoop.

[5] C function where execution of program waits till some data are written or read

**Apache Nutch**

**Nutch** is a highly extensible and scalable open source web crawler software project from the Apache software family[6].

It was already used in Analyzer as a crawler to download XML data 4.2. This crawler is frequently updated.

**Google Web API** [8]

**Google Web API** officially ended on November 1, 2010. It was an interface to access Google Web search through code. It also has a limitation on how many requests per day can be done. So it is not suitable as a crawler for our purpose. In Analyzer development it was used as a benchmark and test suite.

Table 5.1 contains all the discussed crawlers and their key features. The column **Language** means programming language in which it is implemented. The **Parallelism** column contains parallelization policy of crawler. The **Filters** column shows whether the crawler has a kind of filter. The **Politeness delay** refers to if crawler waits some time before accessing the same website. The **REP** column states if crawler has Robots Exclusion Protocol implemented. **Easy extensibility** represents the characteristic whether given crawler may be easily extended to required functionality. **Number of seeds** shows maximum number of seeds given to the crawler in the beginning. **Support** focuses on 3 aspects: Examples, Documentation, Updates. This column represents whether the crawler is often updated, fixed, whether it has detailed documentation, whether it has simple and illustrative examples, or possibly the combination of these characteristics.

As we can see in this table, the multi-threaded crawlers are used frequently. The crawling process is faster – they crawl more web pages in a shorter time period. Also Filtering is common ability of crawlers. They need to optimize the process – which web pages to crawl, and which to throw away. Politeness delay and REP are not so common among crawlers. Not every crawler is designed for crawling the web pages repeatedly. Without these features the crawler can crawl much more web pages during a shorter time. We can also see that a half of the analyzed crawlers are not meant to be easily extended, but they are focused on specific process or they have a complicated code. At Number of seeds it is better to start with a higher number, so that the process is faster in the beginning. It is not slowed down by politeness delay during crawling the first website. Most of these crawlers are assumed to be used in common, so they include documentation or examples and the patches to fix them are released frequently.

[6]http://www.apache.org/

| Name of crawler | Language | Parallelism | Filters | Politeness delay | REP | Easy extensibility | Number of seeds | Support |
|---|---|---|---|---|---|---|---|---|
| Abot C# Web Crawler | C# | Multi-thread | Yes | Yes | Yes | Yes | > 1 | Yes |
| Apache Nutch | Java | Multi-thread | Yes | Yes | Yes | Yes | > 1 | Yes |
| Arachnid | Java | Single-thread | No | No | No | No | > 1 | Yes |
| Bixo | Java | Multi-thread | Yes | Yes | Yes | No | > 1 | Yes |
| Crawler4j | Java | Multi-thread | Yes | Yes | Yes | Yes | > 1 | Yes |
| Egothor2 | Java | Multi-thread | No | Yes | Yes | No | > 1 | Yes |
| Ex-Crawler | Java / PHP | Multi-thread | Yes | No | No | No | > 1 | No |
| Google Web API | – | – | – | – | – | – | – | – |
| Heritrix | Java | Multi-thread | Yes | Yes | Yes | Yes | > 1 | Yes |
| JoBo | Java | Single-thread | Yes | Yes | Yes | No | > 1 | No |
| JSpider | Java | Multi-thread | Yes | Yes | Yes | Yes | 1 | Yes |
| Larbin | C++ | Multi-thread | Yes | No | Yes | No | > 1 | No |
| Scrapy | Python | More processes | Yes | Yes | Yes | No | > 1 | Yes |
| Web-Harvest | Java | Multi-thread | Yes | No | No | No | > 1 | No |
| WebLech | Java | Multi-thread | No | No | No | No | > 1 | No |
| WebSPHINX | Java | Multi-thread | Yes | No | Yes | Yes | > 1 | No |

Table 5.1: Comparison of suitable crawlers

## 5.3 Chosen Crawler

From the discussed crawlers we primarily chose crawlers with parallelization policy (multi-threaded are preferred) and can have more than one seed. This assures that the crawling process will not take a long time.

The chosen crawler also needs to be easy extensible. The Robot Exclusion Protocol and the politeness delay are required too. We do not want to be banned from some websites and be disallowed to crawl them again. Also we need to alter

searching to search for XQuery programs (specific data).

Crawler4j was chosen from the others because of its user-friendly interface, parallelization policy and also because it contains a lot of desired functionality. It is also kept up-to-date. This crawler does not have specific filters like blacklist, whitelist, etc. and is up to the user to program the specific filters. The last task is to create the component for Analyzer out of it. It is not a simple task. If successful, Analyzer will have easier access to downloaded data. If not successful, the crawler will run alone and obtained data will be imported into Analyzer from the local hard-drive.

# 6. Seeds for Crawler

Every crawler we analyzed in the previous chapter requires a seed or seeds to start the crawling process. In this chapter we are going to discus how to obtain seeds as an input for the crawler.

## 6.1 Seeds from the Web Search Engine

The crawler usually starts with URL list of web pages to visit. These URLs are called seeds.

The easiest way to obtain appropriate seeds is to use the most popular web search engine. We have selected these **web search queries** for the task:

**filetype:xq AND doc**
: This query was proposed in [57] where all types of XML data were collected. It should find all files with **.xq** file extension.

**filetype:xquery AND doc**
: This query is a derivation of the previous query, but this time the **.xquery** file extension is required.

**xquery version**
: This query is based on the fact that XQuery programs should contain 'xquery version 1.0' on the first line.

**FLWOR construct or other**
: Using XQuery language words increases a chance to gain more XQuery programs from the Internet.

We should obtain enough seeds using these **web search queries** so that our crawler can find enough programs which can be used as representative sample of XQuery programs on the Internet.

## 6.2 Google URL Result

Google, one of the biggest search engines, has several ways of getting URLs for given search query.

The most common way is via the Google website[1].

For more skilled users there is another way using the Custom Search Google API [5]. This is a programmer approach and it requires 'developer api key' and 'custom search engine id' to get started.

The last way is using Google knowledge graph. This is the only known way to get all the results of google database for specific query. Unfortunately, the graph for XQuery has not been created yet.

To extract the data from Google web page a program is needed. For example, there is a commercial program called Outwit[2]. However, despite its simple user

[1] http://www.google.com
[2] http://www.outwit.com/

interface it requires some knowledge of how to use it. So we have decided that it would be more efficient to write our own program **scraper**. It extracts URLs from the given HTML Google web page. The **scrapper** is written in Java language. We provide an HTML web page with Google URL results and it provides us with the file of results of URLs with each URL on the separated line. The URL seed we needed to extract from Google web page result is shown in following example:

```
...<h3 class="r"><a href="result_URL">title_of_result<...
```

Note that there are limitations of the Google website search. There can be only 100 results per web page and maximally 10 web pages of results with 100 results. That means maximally 1000 URL per search query.

| **Google query** | **Omitted results** | **Google results** | **Gained results** |
|---|---|---|---|
| filetype xq AND doc | NO | 80 | 80 |
| filetype xq AND doc | YES | 2,910 | 728 |
| filetype:xquery AND doc | NO | 30 | 30 |
| filetype:xquery AND doc | YES | 1,700 | 268 |
| ‘xquery version‘ | NO | 22,000 | 154 |
| ‘xquery version‘ | YES | 22,000 | 811 |
| xquery version | NO | 1,140,000 | 200 |
| xquery version | YES | 1,140,000 | 772 |
| filetype:xq | NO | 68 | 68 |
| filetype:xq | YES | 441,000 | 981 |
| filetype:xquery | NO | 61 | 61 |
| filetype:xquery | YES | 5,100 | 577 |
| for let where order by return | NO | 537,000,000 | 472 |
| **Together** | – | – | 5,202 |

Table 6.1: Google search results for queries

The table 6.1 shows us how many results (URLs) we got from the given search queries. Column **Omitted results** explains whether we used the omitted results option. Column **Google results** shows the number Google writes on the first web page of results as how many results it has in its database. For the next web page results it is in the most cases changed to a smaller number and it has just informational value. And the last column **Gained results** is the sum of the results we managed to extract from Google website for specific search query.

These data were collected 27th of June 2014.

## 6.3 Seed File

The easiest way to feed the URLs to the crawler is to put them into one file and let the crawler load them all from it. For this purpose we used Windows Powershell script to create one file with all the results, sorted and without duplicities:

```
Get-Content *.txt | Sort-Object -Unique | Set-Content input.txt
```

For better understanding the command ‘`Get-Content`‘ writes content of the given file on standard output. The command ‘`Sort-Object -Unique`‘ sorts the given string and removes duplicate ones. And the last command ‘`Set-Content`‘ writes the given content into specified file.

For crawler is better when seeds from the same website are not together so we use Unix script to randomize the file lines:

```
cat seedfile.txt | ./unix-add-random-number-at-begin-of-line \
| sort -n | awk '{print $2}' > randomlines.txt
```

Program **unix-add-random-number-at-begin-of-line** is a simple shell script:

```
while read line; do
echo $RANDOM $line
done
```

We now have the file with 4,198 unique URLs with random positions in file out of a total of 5,202 URLs.

# 7. Modification of Selected Crawler

In Chapter 5 we chose the particular crawler. Now we need to modify it so we can search for potential XQuery programs on the Internet. We will also create library out of the crawler, so we can attach it to Analyzer.

In order to be able to compile under Java 1.8, errors in Java documentation[1] had to be fixed first. We also had to manually install some libraries and let maven download other libraries. Manually installed libraries were httpclient-4.2.3.jar, je-4.0.92.jar, log4j-1.2.14.jar, tika-parsers-1.0.jar and junit-4.11.jar. We used IDE NetBeans 8.0.2 to do so.

## 7.1 Basic Setting of Crawler

The **Crawler4j** has already implemented a lot of functions and features. In the configurations we used these settings:

- Variable **crawlStorageFolder** sets the home directory for the crawler. The crawler will store all its data there.
- Variable **resumableCrawling** sets, whether the crawler will be able to resume its process right where it ended. When using this variable there is an overhead because crawler needs to store which pages are being processed. This overhead is tolerable.
- Variable **maxPagesToFetch** sets the number of pages after which the crawler stops crawling. We have tried some runs and found out that after around 3,000,000 pages the crawling process is radically slowed down because of response time of crawlers database of processed URLs. That is why we set this variable to **5,000,000** pages.
- Variable **userAgentString** sets the identity of the crawler under witch it present itself on websites. We use **crawler4j for XQuery**.
- Variable **politenessDelay** sets the milliseconds after which the crawler can access the website again. In first runs we used 1,000 ms. After looking at many robots.txt we realized that some pages uses restriction to 2 seconds. That is why we set it to **2,000** ms.
- Variable **maxTotalConnections** sets the maximum value for active connections from the crawler to the Internet at any moment in time. During testing runs we run out of descriptors[2]. We had 10 threads and it was set to 11, after this experience we set the value on unlimited descriptors and the problem never occurred again. This is why there is no need to change default value **100** connections.

[1] http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/javadoc.html – JavaDoc
[2] opened connections and also opened files

- Variable **maxDepthOfCrawling** sets the maximum depth for crawling process. By default it is unlimited, but because of spider traps it is better to set it on some value. We have taken maximum value for asynchronous crawlers used in Analyzer which is **20**.

- Variable **includeBinaryContentInCrawling** defines if pages with MIME type starting with application will be processed. Since XQuery programs identifies as 'application/xquery' [54], we used **true**.

- Value **maxDownloadSize** sets the maximum size of the downloaded and processed file. This is the previously mentioned Large document filter. We set this value on **524,288** (that represent 0,5 MB). There were two main reasons for setting this value. First, the process of parsing big files means using a lot of memory and we did not want to risk running out of memory and, second, we assume that the majority of the XQuery programs have a small file size.

We left the rest of the configurations set on default values because they are not important for our project. Since the crawler is written in Java language it runs on Java Virtual Machine[3]. JVM offers networking properties settings [11] and we tried various values for these settings:

- We found out that the setting **sun.net.spi.nameservice.nameservers** did not affect the use of the DNS server. Used DNS server was set by the operation system and not by the java virtual machine from program.

- We decided to use the value **false** for the setting **http.keepAlive**. This value means **false** means that the opened HTTP connections are closed after one request and they cannot be used for the next request to the same server.

  We used this setting as a fail-safe mechanism if setting **maxTotalConnections** would not be taken into account.

  Also if the crawler sends too many requests to different servers and runs out of file descriptors, it must either wait for a long connection timeout or go through its databases whether there are any URLs to crawl in active connections. This would be too slow in case of a big database. On the other hand, waiting for a timeout would be also ineffective. Both ways, it would slow down the data collection.

  Contrary to this option if the connection is closed right after processing of the request, creating a new connection to the same server takes less time than chosen politeness delay (no need to create a new connection to the same server is the only case when 'keep alive' would be useful).

## 7.2 Setting Crawler's Filters

There are also filters in the crawler besides the basic settings. We used all the filters mentioned in Section 5.1.3.

[3]http://docs.oracle.com/javase/specs/jvms/se7/html/index.html – JVM

**Large documents filtering**
It is set in the configurations.

**Address and protocol filtering**
It is implemented as two patterns.

Protocol filtering:

```
"(ftp|javascript|mailto|https).*"
```

This is a regular expression used for URL control – so that the beginning of the URL does not contain unwanted protocols which the used crawler is not able to handle.

Address filtering:

```
"https?\\://(facebook.com|www.like-news.us|www.w3.org).*"
```

This regular expression filters out the websites which are too extensive or were marked as a threat by an anti-virus program or all XQuery programs from them we already have.

**Whitelist filter**
It marks files supposedly being XQuery programs. It looks like this:

```
"[^\\?]+\\.xq(uery)?(\\?.*)?$"
```

This expression matches every URL that have file extension **.xq** or **.xquery**.

**Blacklist filter**
IT filters file extensions of files which the crawler will not even try to download. It is the most extensive filter and it looks like this:

```
"[^\\?]+\\." //anything but question mark at beginning
  //and then dot
+ "(css|js|ico|jar" //web files
+ "|bmp|gif|jpe?g|png|tiff?" //image files
+ "|mid|mp2|mp3|mp4|wav|wma|ogg" //audio files
+ "|avi|mov|mpe?g|ram|m4v|mkv|wmv|mod|flv|3gp" //video files
+ "|docx?|pps|pptx?|xlsx?" //office files
+ "|pdf|pdb|rtf|mobi|epub" //not wanted document files
+ "|zip|rar|z7|tar|gz" //compressed files
+ "|dll|so|bat|sh|bash" //dynamic libraries and script files
+ "|cp?p?|h|java|cs" //source code files
+ "|rm|smil|swf|exe" //other files
+ "|xml)" //we want XML documents only from XQuery program
+ "(\\?.*)?$"); //question mark and parameters at the end
                //(optional)
```

At the end of each line there is a description of the file extensions described in the given line. The special case is **.xml**. We want to download only XML documents which are referenced in the potential XQuery programs so we filter them from crawling and add them as the special seeds during process. This way they are directly downloaded and have information about their parent (potential XQuery program).

## 7.3 Fixes of the Crawler

After running the crawling process for a couple of times we noticed that the politeness delay is malfunctioning. With 10 threads and configuration set to 1,000 ms it accessed the same website 10 times during 1,000 ms. So we modified the behaviour of the implemented politeness delay. We added a dictionary as a hash map where we saved the access times for added IP addresses (IPv4 and also IPv6). The time of access was saved for all the IP addresses of the website. This was realised through DNS translation of host domain name. By this we also solved the problem of the different websites having the same IP address.

Here is the corrected code:

```
synchronized (mutex) {
    //get the last time the crawler requested something
    //from server(s) of that domain
    lastFetchTime = dictionary.getValue(webUrl.getDomain());

    //get the current time
    long now = (new Date()).getTime();

    //calculate the delay from the last access to that server(s)
    long existingDelay = (now - lastFetchTime);

    //if the politeness delay is required sleep
    if (existingDelay < config.getPolitenessDelay()) {
        //calculate for how long the sleep is needed
        long newDelay = config.getPolitenessDelay();
        newDelay -= existingDelay;

        //put the new access time to that server(s)
        //before sleep
        long newAccessTime = lastFetchTime;
        newAccessTime += config.getPolitenessDelay();
        dictionary.add(webUrl.getDomain(), newAccessTime);

        //finally sleep
        Thread.sleep(newDelay);
    }

    //before a request put the access time to the dictionary
    //(if newer time is already there, it will not overwritten)
```

```
    dictionary.add(webUrl.getDomain(), (new Date()).getTime());
}
```

When asking for the last access time of the website the dictionary makes DNS request to find all the IP addresses and it chooses the highest access time associated with this IP addresses. If no IP address is present for a given domain name, zero is returned.

When adding the last access time, the dictionary makes a DNS request to find all the IP addresses and it saves this access time for all the addresses. If a record of IP address is not present, it is created. If IP address's access time is greater, it is not overwritten.

To speed up the process, we clear the dictionary every time when it reaches 100,000 IP addresses. After the dictionary is cleared, it returns the actual time plus politeness delay multiplied by the number of request for first 'x' request. The 'x' represents the number of used threads for crawling. This ensures that the politeness delay is constantly maintained.

This solution uses 3 identical DNS requests to access one web page. Since we use our own cache DNS server this is not an issue for us. This way the cache DNS server makes only one DNS request.

Also a nearby DNS server helped us to speed up the process. The ping between the PC with the crawler and the DNS server was lower than 1 millisecond.

## 7.4 Configuration File Loading

We also added a simple loading of configurations from the file. If the file is not present, the default values are used. One line can contain either a commentary (starts with # or //) or 'variable=value' setting. We distinguish 3 types of variables:

- The value can be true or false. The respective variables are: resumableCrawling, includeHttpsPages, includeBinaryContentInCrawling, followRedirects.
- The value can be a number without '.' and ','. The variables are: maxDepthOfCrawling, maxPagesToFetch, politenessDelay, maxConnectionsPerHost, maxTotalConnections, socketTimeout, connectionTimeout, maxOutgoingLinksToFollow, maxDownloadSize, proxyPort.
- The values are taken as a whole string (must be in a correct form). The variables are: folder, userAgentString, proxyHost, proxyUsername, proxyPassword.

If a variable is not present in the configuration file the default value is used. While adding more functionality, we also added 4 more configuration parameters: numberOfThreads, xqueryCrawler, statisticsSaving, seedFile. NumberOfThreads sets an integer value of how many crawling threads will crawler use. XQueryCrawler sets if we will filter only possible XQuery program and their XML documents or return every page. StatisticsSaving sets if we will save statistics about crawling into the file after the crawling is paused or finished. And, finally, SeedFile sets a path to the file where seeds for the crawler are provided.

## 7.5 Adding Seeds from File

We often needed to insert more than just few initial seeds. That is why we decided to add this functionality, too. We also added an option to change the path to the seed file from the configuration file. The seed file contains either a comment (starts with #) or an URL on every line. We added a function that reads these URLs and if they match the filter they are added to the download queue. After each addition the crawler also downloads initial data (robots exclusion protocol), that is why we also added politeness delay functionality here.

If the configuration file does not contain seed file path, the only seeds used by the crawler are the ones inserted into it by the program code.

## 7.6 Recognizing XQuery Programs

The next functionality is to identify whether the crawled documents are XQuery programs or not.

We used filters for this. We look for MIME type in Content Type part of the HTML response header and if it matches this regular expression it goes to the next phase:

```
"(application|text)/(xml|xquery|plain|html).*$"
```

In next phase we check whether it is an HTML page. We parse it and look for a `<pre>` sections. If they contain **xquery version**, we mark them as a potential XQuery programs. We also look for files that have extension **.xq** or **.xquery** as recommends the W3C. These files are also marked as potential XQuery programs. The last place we look for the potential XQuery programs are in the text files. We search if they contain an **xquery version**. If so, we mark them as potential XQuery programs. To sum it up, we are focussing on:

- file extensions **.xquery** or **.xq**
- HTML page `<pre>` sections with **xquery version** phrase
- text files with **xquery version** phrase

In order to be able to use the crawler also for the other searches, we added an option into the configuration file. If it is **true** it checks whether downloaded web pages are potential XQuery programs and the crawler returns only these potential XQuery programs.

### 7.6.1 XML Documents Referenced in XQuery Programs

The XQuery program can contain links to various XML documents. These links are in the `doc()` function, which parameter is an absolute or a relative path to the XML documents.

First we need to parse the given links. Then we have to transform them to absolute URLs and then put them into the crawler for download.

We want these XML documents to have information about their parent XQuery program. That is why we put a reference to it when inserting these links into crawler for download.

We also want to download these XML documents priorly so there will be no URLs of the XML documents left in the download queue when we finish crawling. That is why we add priority to the links.

Heuristics will not help us a lot when correcting the defective URLs to XML documents. Every correction must be tested and we use politeness delay so every testing would take more time and we would download less potential XQuery programs. That is why we do not use heuristics for correcting XML documents URLs and we rather try to find other XQuery programs.

## 7.7 Saving the Found Data

We need to save the downloaded data transparently, and in a way they would not be overwritten. For definite differentiation of URLs crawler4j uses its own document identification number (docID) which is assigned during the crawling process. We use this docID to create an unique directory where we save all the information about a given URL. This information includes given file, parsing `<pre>` sections, downloaded XML documents and information about potentional XQuery program. This information file name is docID.txt. It containts URLs, Content Types, time and information on how many potential links on XML documents go from the saved XQuery program.

## 7.8 Saving Statistics

To monitor the crawling process we also added statistics to the crawler. This way we can monitor the progress of crawling. We also added variable saveStatistis (values **true** and **false**) to the configuration file. If there is the value **true** at the beginning the crawler looks whether there is file statistics.txt. If file is found, it loads from it the statistics from the previously paused crawling. When pausing the crawler, it writes the current statistics into this file. The form of the statistics is following: ‘statistics=value‘ as in the configuration file.

The statistics count the number of crawled web pages, the number of invalid links, the number of potential XQuery programs and the number of downloaded XML documents. Invalid links are specified as HTML status codes 404 Page not found. The statistics calculations use locking so that different threads do not increment it at the same time.

## 7.9 Returning Downloaded Pages

We added a new interface into the crawler:

```
public interface Crawler4jDownloadCallback {
    public void downloaded(Page page);
}
```

This way the crawler can be used as a library and a program using it can decide what to do with potential XQuery programs. We also added the following function into the crawler controller so any class that wants to use the crawler must use it:

```
void AddListener(Crawler4jDownloadCallback listener);
```

Also an implementation of function `downloaded` is needed. All the necessary information of web pages including content is accessible through its parameter class Page.

## 7.10 Final Architecture

In Figure 7.1 we can see UML 2 component diagram of the modified crawler. The process of crawling is controlled by the component **Controller**. It starts and sets all necessary components for crawling process. It can pause or stop crawling in the middle of crawling process and resume it again.

The component **ConfigurationLoader** loads the configuration file or creates default configurations.

The component **SeedLoader** reads the seed file and loads all the URLs into the database. It respects politeness delay. During this process it excludes URLs not matching the filters. Also it accesses the web pages and looks at robots exclusion protocol.

The component **FileSaver** looks after saving of the XQuery programs without conflicts. Also it manages that the downloaded XML documents are saved in the same directory as their parent XQuery programs.

The component **Fetcher** provides downloading of web pages with given URLs. This component contains the implementation of politeness delay, large documents filter, black filter, protocol filter, address filter and administration of connections. It also checks if the maximum of fetched URLs are not exceeded.

The component **RobotsTxtServer** contains functionality to automatically download robots exclusion protocol for websites that are about to be crawled. It evaluates if given URL can be downloaded.

The component **Frontier** saves all the URLs. Every URL is saved once under unique docID. Also it provides the fetcher with a set of URLs for crawling.

The component **Parser** obtains the downloaded web page. If the downloaded web page is a HTML page, it parses other URLs out of it.

The component **WebCrawler** provides functionality for obtaining URLs, fetching web pages from these URLs, parsing this URLs and putting new URLs from parsed web pages into database for further crawling.

The component **XQueryCrawler** is extension of WebCrawler and contains functionality for picking up potential XQuery programs. Special links to XML documents are parsed out of these programs and inserted into the database with special priority.

Interface **Crawler4jDownloadCallback** serves as event for downloaded pages. Any program using this Crawler can connect to this point and receive downloaded pages.

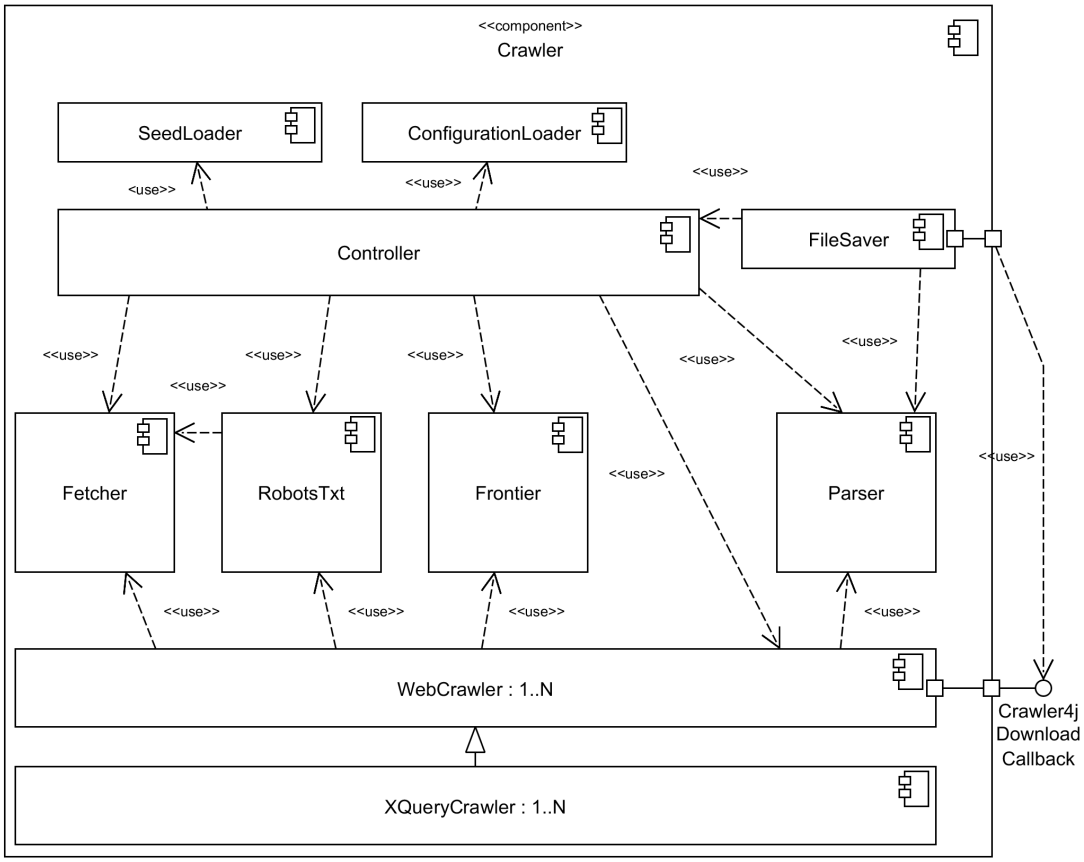



Figure 7.1: Crawler component

### 7.10.1 Making Crawler Library

We wanted to use the crawler in Analyzer so we modified the crawler and created the library out of it. However, we also created a stand-alone program with the main function for its debugging. This stand-alone program simulates the Analyzer function calls. We created an own class for this purpose. When compiling this as library jar we make this class empty. In this class we added a control whether the files exist for interaction with running program. If there is a specific file, the process of crawling is stopped (and crawler saves the last data and turns itself off) or the status is output (of how many pages where processed).

# 8. Crawler Integration into Analyzer

In the previous chapter we created a library from the crawler. In this chapter we will connect it to Analyzer.

First, we had to correct errors, so that we could compile Analyzer. We installed dependency junit4 plugin into IDE NetBeans 8.0.2 and we added all needed dependencies to Analyzer's modules with errors in IDE via 'ADD MODULE DEPENDENCY'. We also corrected the number of dependency of org.openide.util.lookup from '8.25.2' to '8.25.1'. After these errors were corrected we were able to build Analyzer and run it.

## 8.1 XQuery Crawler Module

First, we create XQuery Crawler module according to the already-created Egothor Crawler module. The following interfaces need to be implemented: `AsynchronousCrawler` and `Crawler`. Since we are going to use crawler4j, we need to implement callback for downloaded pages `Crawler4jDownloadCallback`.

The following functions are in the mentioned interfaces:

- Function `getAccessingMode` returns what kind of crawler this is: `AccessModes.ASYNCHRONOUS\_FLOODING`. This means that it is a crawler that runs in the background and returns more than one document because it crawls through the Internet.
- Function `getConfigurationSummary` returns a string with information on crawler's settings.
- Two functions `initialize` are used for initialization of the crawler. The path to the directory where crawler have its data is needed here. We have not found out how to pass second path in this initialization, so the crawler looks for the configuration file in the directory where Analyzer is running. If it does not find out configuration file, the default settings are used.
- Function `getConfiguration` returns the list of settings for this module. In our case we only set the directory, where the crawler is supposed to be running. This directory cannot be change.
- Function `exportConfiguration` takes the settings and saves a relative path from the project directory into the project file (manifest).
- Function `start` starts the crawling process. The crawling process is started only if the crawler was initialized. The return value is `true` in this case.
- Function `stop` stops the crawler and waits till all its threads are stopped. It always returns `true`.
- Function `registerDocumentListener` gets a listener which is added to the list of listeners. These listeners are alerted when any page is downloaded.

- Function `unregisterDocumentListener` removes a listener from the list of listeners.
- Function `isScheduled` returns the information whether the URL was added to the download queue.
- Function `isDownloaded` should return the information whether the URL was downloaded. We cannot distinguish reliably if the URL was already downloaded or it is in a queue to be downloaded, that is why it returns the same value as `isScheduled` function.
- Function `injectURI` passes the URL to the crawler to be downloaded.
- Function `getLatestDownloaded` gets the URL and returns the structure which contains the link to specific file, in case a file with the given URL is downloaded, otherwise it returns `null`.
- Function `getAdditionalStorages` returns the list of additional storages which are used by the crawler.
- Function `downloaded` is called every time the crawler downloads a suitable page. The page is saved to the storage with the required information and all the listeners connected to given crawler are notified.

## 8.2 Fixing Small Bugs

We were not able to create a new project when we ran Analyzer for the first time. We were getting the `Null reference exception` error. Using debugger we found out that in the project-creating scenario the values of the project in wizard windows are validated before they are saved from filled-in values. We fixed the function `validate` in `CustomWizardPanel.java`, so in the beginning it gets actual information from filled-in fields (function `storeSettings`). After this minor modification we were able to go through all the wizard dialogues for project creation. During project testing we tried to create a project with Eqothor Crawler. We got error everytime Egothor tried to initialize itself. With a debugger we found out that a bug is inside egothor library. Since it was already compiled code, we could not fix it or hack it.

After implementing our own asynchronous crawler, we successfully created a project. When we tried to create a download session, we got another error – non-existing storage during creating a download session. We found out that in the wizard for data download new instances are created but are not put into managers. Therefore the given instances did not have identification numbers in managers memory. If we, however, close the project and re-opened it (there is a pop-up window, whether we want to repair the project) and then tried to create a download session, the wizard was successfully finished.

After connecting the project to the launcher and starting up the launcher, the download process did not start. The crawler did not have a configuration file thus also no seed file. It also did not get URLs from the download session wizard. When we corrected the crawler to get the seed file from the specific folder, it started the crawling process but no downloaded pages got to Analyzer. Thus we

decided to let the crawler run separately, and then import downloaded potential XQuery programs and XML documents into Analyzer.

## 8.3 Names of Used Components

All the plugins set their names and other information from file 'Bundle.properties'. We fixed the information in the rest of the Analyzer plugins and this is how we filled them in the XQueryCrawler's module:

```
OpenIDE-Module-Name=XQuery Crawler
OpenIDE-Module-Display-Category=Asynchronous Crawler
OpenIDE-Module-Long-Description=\
    Asynchronous crawler based on Crawler4j project\
    (https://code.google.com/p/crawler4j/)
```

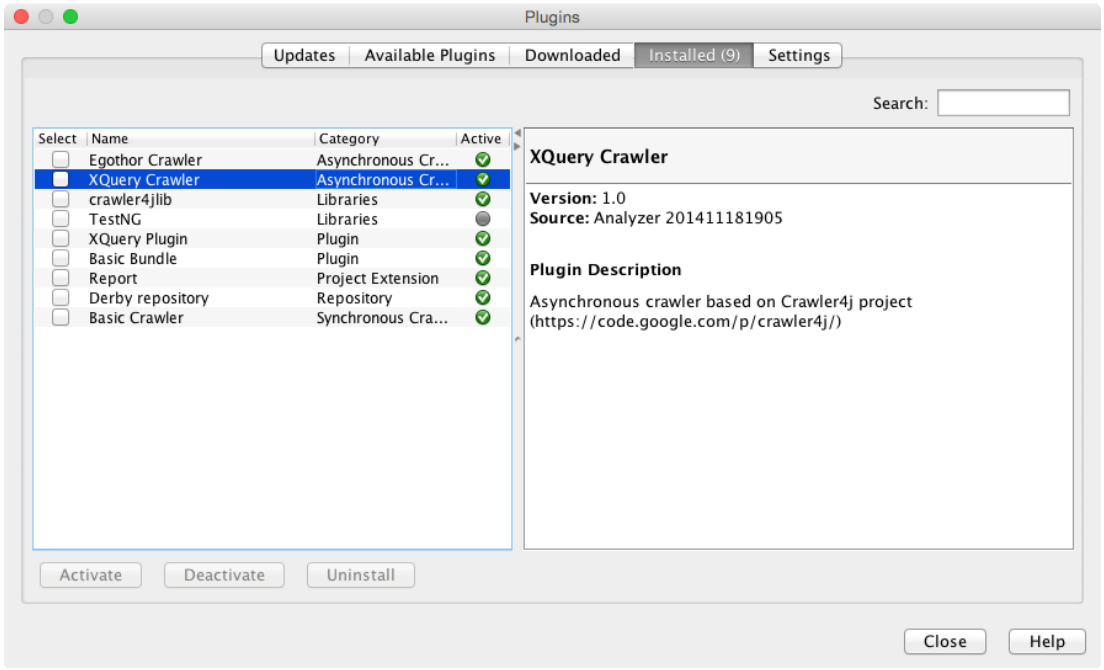



Figure 8.1: XQuery crawler in Analyzer

Figure 8.1 shows how it is presented in Analyzer.

# 9. Data Gathering

In this chapter we focus on obtaining potential XQuery programs and XML documents over which they query from the Internet. We also focus on the question whether enough potential XQuery programs are downloaded.

## 9.1 Downloading Session

The modified crawler from Section 7.10 runs on computer Mac mid-2010 with the following parameters: CPU Intel i5, RAM 12 GB, OS: OS X.

The previously mentioned limit of maximum processed web pages was set to 5,000,000. The limitation came as inspiration from thesis [57]. Their crawler processed approximately 6,000,000 pages. Out of this number 716,000 documents were downloaded, what represents approximately 12%. Only 48,262 (less that 1%) of the downloaded documents were XML-related.

We took this into account and we assumed that we will get maximally 5,000 XQuery programs (10% of their XML-related documents).

After 48 hours long downloading session our crawler saw over 8,000,000 different URLs and processed 325,954 URLs. That means that filters used in the crawler filtered over 95% of unwanted URLs. From 325,954 URLs the crawler identified and downloaded as a potential XQuery program 11,220 documents and 74 XML documents referred from them. These results exceeded our expectations, though these numbers will be lower after filtering out invalid data. 50-90% of the data is expected not to be an XQuery program or it will contain errors and thus not be valid.

### 9.1.1 Testing Sessions

At the beginning we run test downloading sessions. The purpose of these tests was to check the functionality of the crawler (both enhanced and original functionality). These runs have the maximum pages to crawl set from 10,000 to 10,000,000 pages.

These sessions were run on two different machines. The first machine parameters were: CPU: Intel dual core 2,2GHz, RAM: 2GB, OS: Windows 7. The second machine's parameters were: CPU: AMD64 RAM: 2GB OS: FreeBSD.

As a summary we got Table 9.1 that describes the testing sessions. The columns represent crawler option the maximum pages to process, the time lapse of the session, how many pages the crawler processed and how many files it downloaded.

As we can see, the limitation on 10,000 pages took longer time than 100,000 pages. The reason was that we used more seeds and many of them from the same websites (the crawling process was slowed down by the politeness delay). That is why we decided to randomize the order of loaded seeds in Chapter 6.

These sessions also set ground for values of crawler configuration, i.e. how long it will crawl and how many XQuery programs it will possibly find.

| Maximum pages to process | Time | Pages | Downloaded files |
|---:|---:|---:|---:|
| 10,000 | 5:40:02 | 10,000 | 1,827 |
| 100,000 | 4:33:24 | 100,000 | 2,144 |
| 5,000,000 | 23:42:00 | 2,975,398 | 3,351 |
| 10,000,000 | 132:00:00 | 8,238,827 | 8,501 |

Table 9.1: Testing download sessions

### 9.1.2 Download Summary

From 8,000,000 pages we got 11,220 potential XQuery programs and 74 XML documents used in these programs. During the crawling we encountered around 3,500 broken links and nearly 40,000 other fails with downloading pages.

Most from downloaded documents (9,679) where with extension **.xq** or **.xquery**, some were from HTML pages sections `<pre>` (1,524), a few were binary (14) and only 3 where in text format files.

## 9.2 Common Crawl

We decided to compare our number of potential results with Common Crawl potential results to evaluate and compare the number of found documents. That is why we downloaded Common Crawl database from August 2015. This database contains 1.81 billion web pages and offers 3 types of snapshots:

- WARC (Web ARChive) files which store the raw crawled data
- WAT (Web Archive Transformation) files which store computed metadata for the data stored in the WARC
- WET (Web archive Extracted plain Text) files which store extracted plain text from the data stored in the WARC

We chose the most extensive type, the WARC, and we downloaded approximately 34,000 files with the size of 145 TB. To download and process this size of data we used the MetaCentrum[1] where we had 24 worker PCs and 3 manager PCs with the following configuration: CPU: 2x 8-core Intel Xeon, RAM: 128 GB, OS: Linux distribution. We ran Hadoop version 2.6.0 to process the data, with two jobs described in the following subsections.

### 9.2.1 MIME Counting

URL content is mostly defined in MIME type, hence we decided to count the number of MIME types present in the database. We created the following Java programs for Hadoop:

```
Configuration conf = getConf();

Job job = Job.getInstance(conf, "WARCMimeCounter");
```

[1] https://metavo.metacentrum.cz/ – MetaCentrum Virtual Organization

```
job.setJarByClass(Main.class);
job.setNumReduceTasks(24);

Path inputPath = new Path(args[0]);
Path outputPath = new Path(args[1]);

FileInputFormat.addInputPath(job, inputPath);
FileOutputFormat.setOutputPath(job, outputPath);

job.setInputFormatClass(WARCFileInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);

job.setMapperClass(MapperImpl.class);
job.setCombinerClass(LongSumReducer.class);
job.setReducerClass(LongSumReducer.class);
```

Function `setNumReduceTasks(24)` sets the number of workers to 24 (number of worker computers in the MetaCentrum. Functions `setInputFormatClass` and `setOutputFormatClass` provide parsing of the input WARC files and outputs text files. Functions `setOutputKeyClass` and `setOutputValueClass` set the type of data to be extracted from the WARC files (for the implementation itself see below) and function `setReducerClass` sets what is done with these extracted data. In this case all data with the same key will be summed up together.

One more function is used here - `setCombinerClass` with the same parameter as `setReducerClass`. It is an optimization for the MapReduce job and it ran on the output of the map phase to minimize internal network communication between workers.

In following code is implementation of mapper functions:

```
private final Text outKey = new Text();
private final LongWritable outVal = new LongWritable(1);

private final Pattern quotePattern = Pattern.compile("[\"\']");
private final Pattern contentTypePattern = Pattern.compile(
"([Cc]ontent-[Tt]ype:[ \\t]*)([^;\\s]+)(.*)*");

public void map(Text key, ArchiveReader value, Context context)
throws IOException {
...
if (!record.getHeader().getMimetype().equals(
"application/http; msgtype=response")) continue;

//Convenience function that reads the message into a raw byte array
String recordContent = new String(IOUtils.toByteArray(record,
record.available()));
```

```
//The HTTP header gives us valuable information about
//what was received during the request
String httpHeader = recordContent.substring(0,
recordContent.indexOf("\r\n\r\n"));

Matcher contentTypeMatcher = contentTypePattern.matcher(httpHeader);
if (!contentTypeMatcher.find()) continue;

String contentType = contentTypeMatcher.group(2);
contentType = quotePattern.matcher(contentType).replaceAll("");
contentType = contentType.toLowerCase();

outKey.set(contentType);
context.write(outKey, outVal);
...
}
```

Two patterns are used here. One matches quotation marks to remove them from string and the other matches the header line of HTTP response with MIME type content to extract MIME type from it.

A WARC record contains 3 parts, but we are only interested in the last one, which contains the HTTP response itself. The first stated `if` filters out the first 2 unwanted parts. In the next step we took the header from HTTP response and we found the information with 'Content type:'. The used matcher extracts MIME type from it. Since sometimes the quotation marks are present at the beginning and at the end of MIME type, these were removed by `iquotePattern` matcher.

This job ran for 8 hours and some of the results are in Table 9.2. We can see that almost 2 billions are HTML pages. Compared to that only 9 millions are stand-alone XML documents and there is no 'application/xquery' MIME type. Either there are no XQuery programs in this Common Crawl database or they have a different MIME type. Since we found almost 3,000 different MIME types, we suppose they use different MIME type.

| **MIME type** | **Number of occurrences** |
|---|---:|
| application/xml | 1,325,368 |
| html | 11,055 |
| text/html | 1,748,429,998 |
| text/plain | 3,403,590 |
| text/xml | 7,634,915 |
| xml | 5,551 |

Table 9.2: MIME types in Common Crawl August 2015

### 9.2.2 URL Extracting

One of the methods we used to select potential XQuery programs is by using file extension. So we looked at individual URLs in Common Crawl and we selected the URLs with file extension **.xq** or **.xquery**. The following Java program was used for this extraction:

```
Configuration conf = getConf();

Job job = Job.getInstance(conf, "WARCUrlXqueryParser");
job.setJarByClass(Main.class);
job.setNumReduceTasks(24);

Path inputPath = new Path(args[0]);
Path outputPath = new Path(args[1]);

FileInputFormat.addInputPath(job, inputPath);
FileOutputFormat.setOutputPath(job, outputPath);

job.setInputFormatClass(WARCFileInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(NullWritable.class);

job.setMapperClass(MapperImpl.class);
```

Compared to the previous program the parameter `setOutputValueClass` is different. Instead of adding of the numbers we use an empty parameter so it only outputs the result. Also we run only the mapper from the MapReduce job, because we do not need to sum up results.

Implementation of mapper is following:

```
private final Text outKey = new Text();
private final NullWritable outVal = NullWritable.get();

private final Pattern urlXQueryPattern = Pattern.compile(
"(.*\\/)([^\\/]+\\.)(xq|XQ)(uery|UERY)?(\\?.*)?");

public void map(Text key, ArchiveReader value, Context context)
throws IOException {
...
String url = record.getHeader().getUrl();

//is it XQuery file?
if (!urlXQueryPattern.matcher(url).find()) continue;

//save URL
outKey.set(url);
context.write(outKey, outVal);
...
}
```

We created a pattern for matching the URLs with file extension **.xq** or **.xquery** in `urlXQueryPattern`. We went through individual WARC entries and we looked at URLs. If we found a match, we added the URL to the results.

Obtaining this list of URLs took 7 hours. It contains approximately 4,500 URLs. After manually going through them and filtering out duplicities only approximately 200 URLs were left. After checking them out, only 40 led to an XQuery program, whereas half of those led to W3C pages.

The filtered addresses were on following domains: ‘blakearchive.org‘ (book search), ‘findingaids.loc.gov‘ (searching in the library of congress), ‘data.albankaldawli.org‘, ‘data.worldbank.org‘, ‘datos.bancomundial.org‘, ‘donnees.banquemondiale.org‘ and ‘api.worldbank.org‘ (bank-sector web pages). In these cases the URL pointed to the internal website search engine and gained the output based on parameters in the URL. The search engine itself could not have been downloaded in any case. That is the reason why only 40 URLs lead to XQuery programs in the Common Crawl database from August 2015.

## 9.3 Summary

We compared the numbers of potential XQuery programs from various sources in order to find out whether we obtained enough potential XQuery programs. First, we took the number of Google results from Chapter 6 for file extension **.xq** and **.xquery**. Second, we took the number of Common Crawl results. Finally, we compared both of them with the number found by our crawler. Table 9.3 shows the comparison.

| **Source** | **Results** |
|---|---:|
| Google | 4,610 |
| CommonCrawl | 4,440 |
| our crawler | 9,679 |

Table 9.3: Comparision from various sources

We managed to obtain nearly 10,000 potential XQuery programs with extension **.xq** or **.xquery**. However, the real number of XQuery program is lower. We predict that about 1,000 will be valid XQuery programs. However, it is still better result if we compare it to thesis [57] where 50,000 XML-related documents were collected and only a small amount of these documents were XQuery programs.

We also managed to obtain only 74 XML documents referenced in these potential XQuery programs. Main reason for that is, that many references to XML documents contains not URL but file path on some local file system.

# 10. Analysis of Downloaded Data

As we have mentioned in the previous chapter, we obtained around 11,000 files using the modified crawler. In order to be able to perform the analysis, we need to clean the data to get XQuery programs and their XML documents. Then we categorize them and the programs will be ready for the analysis.

We import these programs to Analyzer and using XQConverter we try to count some statistics over them. We analyze this statistics and make conclusions.

## 10.1 Cleaning and Correcting Data

The first step of cleaning the data was to remove duplicities from the potential XQuery programs. We removed 1,884 duplicate documents from the 11,220 downloaded documents. In particular, the Following duplicities occurred:

- Identical content of the `<pre>` sections on the same page.
- Different URLs leading to the same page (the parameters after character `?` were different).

During the process of removing duplicities we checked potential XQuery programs, which had downloaded XML documents with them and did not found any duplicity at all.

In the next step we looked at how many of potential XQuery programs from various sources are valid. Table 10.1 shows the summary of the initial valid and invalid programs. The group of the invalid programs includes also many data which are not XQuery programs.

We took a closer look at the data from Binary and Text group and we found out that the data is either an XQuery program runtime dump, or a guide how to write an XQuery program, or, in the third case, a meaningless data.

| **Type of Download** | **All** | **Valid** | **Invalid** |
|---|---|---|---|
| Binary | 14 | 0 | 14 |
| File extension | 8,739 | 619 | 8,120 |
| Pre section | 550 | 223 | 327 |
| Text | 3 | 0 | 3 |
| Together | 9,336 | 842 | 8,491 |

Table 10.1: First validation results

### 10.1.1 Not XQuery Programs

Many of the downloaded data was invalid and we went through the individual errors and looked at what had caused them. 176 errors out of 8,491 were syntactic and the rest were lexical errors.

In the group of lexical errors were also incorrectly included the documents, which were not XQuery programs (over half of them). The following list shows what other data was included:

**Forum responses**
Many data from `<pre>` section came from web forums. These data have many non XQuery language parts included in them without commentary blocks. Also a lot of these cases have the program on one line in the middle of `<pre>` section and lots of unwanted lines with various content around them. Many of `<pre>` sections were actual e-mails as responses to problems with the XQuery programs.

**Licenses and Read-me files**
Some licenses or read-me files contained some of XQuery grammar words. There were approximately 20 of them.

**Theatre plays and archives**
Some of the files had the form of a theatre play or an archive file and contained the XQuery grammar symbols. There were approximately 500 of them.

**XML data**
There were also some XML documents not referenced from any downloaded XQuery program and other XML-related files.

**Short string**
Circa 10 files contained only one random string on the first line.

**Shell script**
There were around 30 shell scripts to use an XQuery program or to change a part of it in the data.

**XQuery functions description**
2 of the files were manuals of the XQuery language.

**RFC**
Around 10 RFC documents with reference to the XQuery 1.0 language were present.

**XQuery output**
In many cases there were just outputs of the XQuery programs with some description.

### 10.1.2 Lexical And Syntactical Errors

Next, we went through the files which looked like XQuery programs and had parsing errors and we tried to find out what caused the errors. We had assumed that most of the errors were caused by the human factor. The reason for this assumption is that if the program contained more identical parts, the error would be only in one of them. We assume that only one file was generated by a program, because of the comment ‘generated content‘.

The following list contains many mistakes we found and a short comment about how we think they occurred:

**Illegal character ‘:‘**

There was the start of a comment `(:` but no ending of the comment `:)`. In many cases there was no space between the variable `$variable` and the assignment symbol `:=`. The parser requires a space between a variable name and an assignment symbol.

**Illegal character ‘)‘**

There was no space between a variable and an end of function ‘)‘. We assume that the ending character was included into the variable name and we think this is what caused the error.

**Illegal character ‘&‘**

HTML symbols like ` ` start with character `&` and the parser does not understand these symbols as one character. This mistakes occurred mostly in the `<pre>` sections.

**Illegal character ‘.‘**

There were quotation marks `""` inside other quotation marks `""` without escaping and the next character after second `"` happens to be ‘.‘. This scenario also happened with other characters (which were after second `"`) and created errors `Illegal character ‘|‘` and `Syntax error`.

**Illegal character ‘1‘**

At the end of some functions there was just a number. We assume that comma, as next function parameter, or operator character like `+` was forgotten.

**Illegal character ‘v‘**

Some XQuery programs included string ‘`update value`‘ which is not in the standard XQuery language. This is a special extension of XQuery language called XQuery Update Facility [43]. That is why the parser returned the error. This scenario also happened with `Illegal character ‘r‘` for string ‘`update replace`‘ and `Syntax error` for string ‘`update insert`‘.

**Illegal character ‘r‘**

This happened on the line with string ‘`declare revalidation skip;`‘. Again this is not a part of the XQuery 1.0 language. Also the string ‘`declare sequential`‘ before the function declaration is not a part of the XQuery 1.0 language and the parser returns `Illegal character ‘s‘`.

**Illegal character ‘f‘**

This happened because of the wrong function declaration. Instead of using the keyword `declare` there was a word `define` and the parameters were `type name` instead of `name as type`.

**Syntax error**

A lot of these errors were caused by forgotten characters like right bracket and comma.

Some of them had the definitions of declarations in the wrong sequence. For exampe, the variable declaration was before the namespace declaration or the option declaration was before the module importation.

And many of them were the previously mentioned extensions of the XQuery language.

### 10.1.3 Correcting Programs

We can try to correct some of these mistakes automatically. So we created a program, that tries to repair invalid potential XQuery programs and if the result is a valid XQuery program, the changes are saved. Also this program tries to remove additional parts like HTML tag `<pre>` and replace HTML entities like `&lt;`, `&gt;`.

The following list contains corrections made by the correcting program:

- The program searches for HTML tags `<pre>` and `<code>` and removes them. Tag `<code>` is used on web forums like `stacoverflow.com`.
- The program searches for HMTL entities like `&amp;`, `&lt;`, `&gt;`, ` `, `&quot;`, `&apos;` and their numeric variations like `&#38;`, etc.. The list of the searched variations was based on the most frequent HTML entities in the `<pre>` sections.
- The program searches at the beginning of the lines if there are `>` characters or `number:` and removes them. Character `>` or more of them at beginning of a line is used in e-mail responses and `number:` is used in some websites to display the content of a file with the numbered lines.
- The program looks for places with invalid-like places and corrects the given part. Specifically it looks for places where are whitespaces in the middle of the assignment operator and it removes them, but only in case that it is not a comment section like, e.g. `(: ==== COMMENT === :)`. It also searches for whitespaces in the middle of comment symbol, e.g. `( : COMMENT :)`, and removes them. In the comments it also searches for string `(:)` and adds spaces so that the parser does not take it as the comment.

These modifications allowed us to correct 39 invalid and 190 valid documents. In case of valid documents it mostly removed HTML tags `<pre>` and replaced HTML entities.

Other errors were too difficult to create a simple correction rule, so we looked manually at other potential XQuery program. To go through all the potential XQuery programs and try to repair them was too time consuming, so we looked only at potential XQuery programs with downloaded XML documents, i.e. potentially the most interesting candidates for an analysis. 29 out of 67 were invalid and the errors were also lexical and syntactical. We managed to correct 5 programs, whereas the rest contained mistakes we were not able to correct. The following list contains what contained the files that were not corrected:

- The content of a directory
- Runtime dump of an XQuery program
- XQuery Update Facility constructs

- Only declared variables
- Combination of Spanish and English names of XQuery grammar symbols
- Some unrelated content

After cleaning and correcting the data there were 887 valid XQuery programs and 26 XML documents referenced from them.

## 10.2 Categorizing XQuery Programs

Before we proceed to the data analysis, we need to categorize the data. The first option to categorize was according to the data source, where we downloaded the data from. There would be 4 categories: binary, file extension, `<pre>` section and text. The disadvantage of this categorization is that the binary and text category does not contain any valid program and most of valid programs are in the file extension category.

The next option was to categorize the data according to the URL where the data was downloaded. This would created 9 categories: 'deri.org', 'europa.eu', 'gname.org', 'googlecode.com', 'ispras.ru', 'politicalmashup.nl', 'sourceforge.net', 'stackoverflow.com' and the others. However, we could not get any interesting data out of this categorization. Half of the categories contained diametrically different programs and the characteristics of the category 'others' were the same as the characteristics of all the data together.

The third option was to the divide programs according to their origins, whether they were written by the humans or generated by a program. However, this categorization was not possible, because we did not possessed this information. Only one file had string '`generated content`' included in a comment.

The fourth option was to run the following java program over the given data:

```
private ArrayList<ArrayList<File>> groups =
            new ArrayList<ArrayList<File>>();
private final float ENOUGHT_DIFF = 0.8f;

public void Count(File[] programs) {

    for (File program : programs) {
        if (groups.isEmpty()) {
            groups.add(new ArrayList<File>());
            groups[0].add(program);
            continue;
        }

        boolean added = false;
        for (group : groups) {
            float diff = CountDiff(program, group[0]);
            if (diff > ENOUGHT_DIFF) {
                group.add(program);
                added = true;
```

```
                break;
            }
        }

        if (!added) {
            groups.add(new ArrayList<File>());
            groups[groups.size() - 1].add(program);
        }
    }
}
```

This program takes all the file (XQuery programs) and builds group of programs accordingly. For each program it tries to compare it with group representative for each existing group and if it is similar enough it puts it in the group. Otherwise a new group is created and program is made the group representative.

Also this idea has its difficulties. On one hand we get similar programs in several categories, but there might be too many of them, or too few and the programs in them would vary too much. It also depends on what programs would be selected as a group representative. Hypothetically every program can be compared to all the other programs but the computing time would rise from linear to quadratic complexity, what makes 1,000 times longer time when there are 1,000 programs.

It strongly depends on how we define similarity of 2 XML trees, such as if we compare only the tree structure, or also name of the element, if we consider attributes, etc. And finally it depends on how we set the threshold that says that 2 trees are similar or not.

As the very last option we decided to consider following three factors:

- Where was the program downloaded from?
- Is some referred XML documents downloaded?
- Does the program contain the grammar symbol `QueryBody`, i.e. is it a library[1]?

Table 10.2 shows how the programs are categorized into 4 categories according to these 3 factors. We found out that none of the programs downloaded from the `<pre>` sections were library. We also found out that the programs downloaded from the `<pre>` sections have not downloaded any XML documents.

| **Category** | With XML | Library | `<pre>` section | File extension |
|---|---|---|---|---|
| **Number of Programs** | 20 | 46 | 260 | 561 |

Table 10.2: Categories of XQuery programs

[1]XQuery program that does not contain QueryBody grammar symbol

## 10.3 Analyzer Analyses

In this part we will describe how we used the cleaned data as an input for the Analyzer.

1. We clicked on the button 'Create new project' and created a new project using the following settings: the chosen name was default 'Project1', no crawler was selected (we already have data and we want to import them) and pattern '.*' was used (we will not download data). The rest of settings we used with default values.

2. We clicked on the button 'Create new analysis' and first we chose 'Universal Analysis'. We repeated this process for 'XQuery Plugin Analysis' with these settings:

   - We select name 'XQuery Grammar Symbols' and we inserted XPaths for all the grammar symbols of XQuery language into XPath settings.
   - We select a name 'XQuery Interesting Analysis' and inserted XPaths for various counts of statistics like recursive functions, external variables, number of nested `FLWOR`, number of XPath expresions with `Predicate`, etc.
   - We select a name 'XQuery FLWOR Counts' and inserted XPaths for various counts of `ForClause` and `LetClause` in the `FLWOR` grammar symbol into XPath settings.
   - We select a name 'XQuery FLWOR Joins' and inserted XPaths for various types of joins according to Section 2.6 into XPath settings.
   - We select a name 'XPath Steps' and inserted XPaths for various numbers of `Step` grammar symbols in the `Path` grammar symbol into XPath settings.
   - We select a name 'XPath Axis' and inserted XPaths for various types of `Axis` (also for their abbreviated forms) into XPath settings.
   - We select a name 'XPath BuiltIn Functions' and inserted XPaths for all the XPath built-in functions together with the prefix 'fn:' and also without it into XPath settings.

   All of these XPaths are listed in Attachment 4 10.7.

3. We clicked on the button 'Create import session' and chose directory with categorized XQuery programs.

4. We were warned that to start import we must connect the project to launcher so we clicked on the button 'Attach to launcher' and waited till the import was done (user log contained 'Import session finished. Number of imported files:887').

5. We clicked on the button 'Close current chain' and Analyzer started to compute the results over imported documents.

6. After the results were computed, we created five clusters using the button 'Create new cluster'. We named them 'Category 1 With XML', 'Category 2 Library', 'Category 3 Pre Sections', 'Category 4 XQuery' and 'All Categories'. For the first four categories we chose the filter 'Resource tree filter' in which we set the specific directories according to our categories. For the last cluster we did not select any filter.

7. After Analyzer finished 'classifying documents' we closed each cluster (right click on the cluster and from options select 'Close cluster').

8. The results computed over the clusters can be viewed in the part 'Available performers' in each cluster.

We found out that the results over 260 files from `<pre>` category were not computed. It was caused by the fact that their file extension was either web extensions **.htm**, **.html**, **.php**, etc. or none. Using a Unix script we changed their file extensions to **.xq** and repeated the process again:

```
ls | while read f; do mv "$f" "‘echo $f | cut -d’.’ -f1‘.xq"; done
```

This project can be found on the enclosed CD also with extracted results see Attachment 1 10.7.

## 10.4 XQuery Programs Analyses

We started with observing the frequencies of various XQuery grammar symbols, which are used frequently, and which are used hardly at all. The list of XQuery grammar symbols and their description is provided in Attachment 2 10.7.

In Table 10.3 we can see the most frequently used XQuery grammar symbols, that are not mentioned later in the analyses. In category `<pre>` section they are used just a little, which is odd, because the category consist of 30% of all analyzed programs.

From comparison grammar symbols only `GeneralComp` and `ValueComp` got into this table. Contrary, `NodeComp` was in the least used group of symbols. It occurred only 11 times in all programs.

From constructors grammar symbols only `DirectConstructor` got here.

`ComputedConstructor` occured only 340 times in all programs, what is 60-times less then `DirectConstructor`. The interesting thing is, that it occurred only 9 times in category With XML.

The least used grammar symbols of the XQuery language are `DefaultNamespaceDecl`, `OrderingModeDecl`, `EmptyOrderDecl`, `CopyNamespacesDecl`, `DefaultCollationDecl`, `BaseURIDecl`, `SchemaImport`, `ConstructionDecl`, `QuantifiedExpr (every)`, `TypeswitchExpr`, `IntersectExceptExpr`, `TreatExpr`, `CastExpr`, `NodeComp`, `ValidateExpr`, `ExtensionExpr`, `OrderedExpr` and `UnorderedExpr`. Their occurrence was only in the 1% of all valid XQuery programs.
`QuantifiedExpr (every)` is alone in this group, because `QuantifiedExpr (some)` had 2% of occurrences. We found out, that for sorting in XQuery language only `OrderByClause` from `FLWOR` grammar symbol is used.

| Grammar Symbols | With XML | Library | Pre Section | XQuery Extension | All |
|---|---|---|---|---|---|
| NamespaceDecl | 6 | 73 | 18 | 838 | 935 |
| IfExpr | 3 | 2,329 | 36 | 7,651 | 10,019 |
| CastableExpr | 0 | 382 | 0 | 1,276 | 1,658 |
| GeneralComp | 36 | 3,170 | 22 | 10,840 | 14,068 |
| ValueComp | 1 | 206 | 10 | 651 | 868 |
| Literal | 62 | 6,575 | 341 | 28,546 | 35,524 |
| DirectConstructor | 45 | 1,837 | 1,613 | 17,207 | 20,702 |

Table 10.3: XQuery grammar symbols

Interestingly the most occurrences of `TypeswitchExpr` grammar symbol was in the category `<pre>` section, so the most of its occurrences were in the examples on the Internet web pages and forums.

### 10.4.1 FLWOR Expresion

`FLWOR Expression` is the cornerstone in XQuery language. We will refer to it as `FLWOR`. It consists of optional `ForClause`, `LetClause`, `WhereClause` and `OrderByClause` and mandatory `ReturnClause`.

| FLWOR | With XML | Library | Pre Section | Xquery Extension | All |
|---|---|---|---|---|---|
| FLWOR Expression | 31 | 1,097 | 104 | 6,176 | 7,408 |
| ForClause | 32 | 555 | 72 | 3,645 | 4,304 |
| LetClause | 44 | 2,948 | 160 | 16,122 | 19,274 |
| WhereClause | 12 | 188 | 10 | 1,346 | 1,556 |
| OrderByClause | 7 | 81 | 10 | 470 | 568 |

Table 10.4: FLWOR statistics

In Table 10.4 we can see that the only category where `FLWOR` is not used much (considering the number of programs) is the `<pre>` section category. Interestingly, `FLWOR` uses more frequently `LetClause` than `ForClause`. 3,379 `FLWOR`s are without the `ForClause` and only 1,687 are without `LetClause`. We assumed that these numbers would be similar. It is also interesting that most of the `FLWOR` occurs without `WhereClause` or `OrderByClause`. These clauses filter and sort data before they are returned by `ReturnClause`.

In Table 10.5 we can see, that by far the most frequent are the `FLWOR` with only one `ForClause` in all categories. Interestingly, more than one `ForClause` occurs in only 240 cases. That means merging tables in one `FLWOR` is very rare.

Table 10.6 shows, that defined variables in `LetClause` in `FLWOR` are used often, in one program it is over 80 `LetClause` in one `FLWOR`. In the category `<pre>` section two `FLWOR`s contained over 10 `LetClause`.

In Table 10.7 we can see, that `FLWOR` are also being nested. In seven cases it is nested over 4 levels.

| For Count | With XML | Library | Pre Section | Xquery Extension | All |
|---|---|---|---|---|---|
| 1 | 17 | 521 | 68 | 3,183 | 3,789 |
| 2 | 6 | 17 | 2 | 204 | 229 |
| 3 | 1 | 0 | 0 | 6 | 7 |
| 4 | 0 | 0 | 0 | 1 | 1 |
| 5 ... 10 | 0 | 0 | 0 | 1 | 1 |
| 11 ... 20 | 0 | 0 | 0 | 2 | 2 |

Table 10.5: For counts in FLWOR

| Let Count | With XML | Library | Pre Section | Xquery Extension | All |
|---|---|---|---|---|---|
| 1 | 12 | 291 | 25 | 1,488 | 1,816 |
| 2 | 1 | 178 | 10 | 1,275 | 1,464 |
| 3 | 2 | 84 | 7 | 499 | 592 |
| 4 | 2 | 99 | 5 | 582 | 688 |
| 5 | 2 | 51 | 2 | 282 | 337 |
| 6 | 1 | 48 | 0 | 165 | 214 |
| 7 | 0 | 37 | 3 | 77 | 117 |
| 8 | 0 | 7 | 0 | 70 | 77 |
| 9 | 0 | 27 | 2 | 115 | 144 |
| 10 ... 90 | 0 | 34 | 2 | 236 | 272 |

Table 10.6: Let counts in FLWOR

| FLWOR nested | With XML | Library | Pre Section | Xquery Extension | All |
|---|---|---|---|---|---|
| 2 times | 3 | 133 | 9 | 695 | 840 |
| 3 times | 0 | 19 | 7 | 49 | 75 |
| 4 times | 1 | 0 | 0 | 31 | 32 |
| 5 and more times | 0 | 0 | 0 | 7 | 7 |

Table 10.7: FLWOR nested levels

| Join type | With XML | Library | Pre Section | Xquery Extension | All |
|---|---|---|---|---|---|
| Where | 1 | 17 | 6 | 42 | 66 |
| Predicate | 0 | 1 | 2 | 93 | 96 |
| If | 0 | 0 | 0 | 0 | 0 |
| Filter | 2 | 0 | 0 | 46 | 48 |

Table 10.8: FLWOR join 2 tables

In the paper [27] Authors thought about using various styles of writing the join of 2 tables in XQuery. In Table 10.8 we can see how often which type is used. It is interesting that the type using comparison on output using the `IfExpr` grammar symbol is not used at all.

### 10.4.2 XPath in XQuery Programs

Next, we looked at the grammar symbol `Path` which is an XPath expression used in XQuery programs. Every `Path` consists of zero to several steps. We counted the number of steps for every `Path` in Table 10.9. As we can see the number of steps through the categories behaves the same way. The most of them is with 2 steps and the highest number of steps is 10.

| **Number of Steps** | **With XML** | **Library** | **Pre Section** | **XQuery Extension** | **All** |
|---|---|---|---|---|---|
| 1 | 16 | 802 | 26 | 4,110 | 4,954 |
| 2 | 63 | 1,318 | 101 | 7,343 | 8,825 |
| 3 | 52 | 819 | 69 | 3,702 | 4,642 |
| 4 | 4 | 163 | 8 | 1,577 | 1,752 |
| 5 | 7 | 90 | 5 | 925 | 1,027 |
| 6 | 0 | 17 | 3 | 183 | 203 |
| 7 | 0 | 5 | 0 | 141 | 146 |
| 8 | 0 | 0 | 2 | 14 | 16 |
| 9 | 0 | 0 | 0 | 8 | 8 |
| 10 | 0 | 0 | 0 | 2 | 2 |

Table 10.9: Path steps

Every step consists of 3 parts. The first is the axis, one of 13 defined axes, the second is the node test and the last is the predicate. Some axes also have an abbreviated syntax. In Table 10.10 we can see used axes and their occurrences. Axes that do not have abbreviated syntax are in the column **Abbreviated** marked with '–'.

The most frequently used axis is `child`, as expected. It is interesting that the axis `following` and `preceding` are not used and contrary, the axis `following-sibling` and `preceding-sibling` are used, though they are used seldom.

The other interesting thing is that the abbreviated form of `self` axis is not used at all. Only abbreviated forms of axis are used in the category With XML. Quite interestingly, in the category `<pre>` section most of axes were in the abbreviated form. Quite interestingly full names are used rarely.

In Table 10.11, we can see that `Predicate` grammar symbol is used in one of the four `Path` grammar symbols. It is quite surprising, because we had expected that half of `Path` grammar symbols uses `Predicate` grammar symbol. As expected, XPath expression in XQuery programs queries mostly over the elements or the attributes and hardly over the text nodes.

XQuery 1.0, XPath 2.0 and XSLT 2.0 have library of built-in functions. These functions are used for various purposes in programs. They work with strings, integers, date and times, positions, document loading, etc. They have namespace `fn` but they can also be used without it.

Table 10.12 shows all collected functions with or without namespace from library of built-in functions. We have to mention that in the category XQuery programs with XML documents there were only used three of these functions: `not`, `empty` and, of course, `doc`.

| Axis | Abbreviated | Full name | All |
|---|---|---|---|
| ancestor | – | 12 | 12 |
| ancestor-or-self | – | 8 | 8 |
| attribute (@) | 5,265 | 6 | 5,271 |
| child () | 22,540 | 608 | 23,148 |
| descendant | – | 66 | 66 |
| descendant-or-self (//) | 3,457 | 0 | 3,457 |
| following | – | 0 | 0 |
| following-sibling | – | 25 | 25 |
| parent (..) | 945 | 4 | 949 |
| preceding | – | 0 | 0 |
| preceding-sibling | – | 24 | 24 |
| self (.) | 0 | 9 | 9 |
| Summary | 32,207 | 762 | 32,969 |

Table 10.10: Path axes names

| | With XML | Library | Pre Section | Xquery Extension | All |
|---|---|---|---|---|---|
| Path | 142 | 3,214 | 214 | 18,050 | 21,620 |
| Path with Predicate | 17 | 949 | 36 | 4,083 | 5,085 |
| Path query on element | 102 | 1,748 | 95 | 11,702 | 13,647 |
| Path query on attribute | 14 | 991 | 44 | 4,208 | 5,257 |
| Path query on text node | 19 | 23 | 32 | 512 | 586 |

Table 10.11: Path statistics

In 260 programs from the category `<pre>` section only 7 `doc` function calls are present and in 46 programs from the category library 184 calls are present.

We found out that in general, the functions using `number`, `boolean` and `date` and `time` are not used in the XQuery programs. The only thing which is used on a large scale are functions to work with `string`. Functions `string-length` `lower-case` and `concat` are used the most. Function `lower-case` is more than 15-times more frequent than function `upper-case`.

As we have assumed the most frequent function is `doc`.

### 10.4.3 Other Analyses

One part of the language are also variables and functions. In Table 10.13 we can see their usage frequency: how many times the variables were only declared and never used, how much external variables there are, how much recursive functions there are and how much of them are only defined.

Further we looked at the operators used in the XQuery programs. We divided them into the three groups. The first group are the additive operators: addition and subtraction. The second group are the multiplicative operators: multiply, divide, modulo and integer division. The third group are the logical operators: `and` and `or`. In Table 10.14 we can see the used operators and it shows that most of them are the logical operators. Surprisingly the most of the operators in the category `<pre>` section are the additive operators, that means on web pages,

| Name | Hits | Name | Hits |
|---|---|---|---|
| boolean | 2 | node-name | 13 |
| ceiling | 3 | normalize-space | 327 |
| concat | 178 | not | 116 |
| collection | 44 | number | 22 |
| contains | 60 | position | 167 |
| count | 149 | replace | 25 |
| data | 49 | resolve-QName | 1 |
| doc | 2,118 | resolve-uri | 1 |
| doc-available | 4 | QName | 1 |
| empty | 27 | starts-with | 72 |
| ends-with | 16 | string | 489 |
| error | 1 | string-length | 391 |
| exists | 74 | string-join | 13 |
| false | 68 | substring | 12 |
| index-of | 42 | substring-after | 17 |
| last | 18 | substring-before | 42 |
| local-name | 330 | tokenize | 11 |
| lower-case | 291 | true | 31 |
| matches | 60 | upper-case | 19 |
| name | 932 | year-from-dateTime | 1 |
| namespace-uri | 22 | zero-or-one | 1 |
| namespace-uri-for-prefix | 1 | | |

Table 10.12: Built-in functions

| Variable and Functions | With XML | Library | Pre Section | Xquery Extension | All |
|---|---|---|---|---|---|
| Variable Declaration | 5 | 488 | 1 | 1,311 | 1,805 |
| Variable Reference | 173 | 15,198 | 657 | 65,044 | 81,072 |
| External Variable | 0 | 0 | 1 | 275 | 276 |
| Not Used Variable | 0 | 142 | 0 | 92 | 234 |
| Function Declaration | 6 | 1,295 | 3 | 6,325 | 7,629 |
| Function Call | 79 | 9,175 | 472 | 52,231 | 61,957 |
| Build In Function Call | 30 | 872 | 29 | 5,330 | 6,261 |
| Recursive Function | 1 | 27 | 0 | 89 | 117 |
| Only Function Declaration | 0 | 603 | 1 | 157 | 1,104 |

Table 10.13: Variables and functions

mainly the additive operators are used.

In the next step we looked at how complicated is the output of the XQuery programs. We were looking for simple outputs – not many changes in the data processed by the XQuery program and for the complicated outputs – the data processed by the XQuery programs were transformed.

When we consider XQuery grammar symbols like `IfExpr`, `FunctionCall`, `Typeswitch`, etc. that can be used to transform the data in the complicated way we get 410 programs that have the complicated output.

| Operands | With XML | Library | Pre Section | Xquery Extension | All |
|---|---|---|---|---|---|
| OrExpr | 0 | 609 | 4 | 3,011 | 3,624 |
| AndExpr | 17 | 1157 | 1 | 3,332 | 4,507 |
| AdditiveExpr | 1 | 50 | 16 | 380 | 447 |
| MultiplicativeExpr | 0 | 22 | 6 | 158 | 186 |

Table 10.14: Operands statistics

If we try something similar with `PathExpr` and `DirectConstructor` as a simple output we get 259 programs because we think that the returns like this are unlikely to create different data.

We focused on the more accurate outputs of the XQuery programs in the following Section 10.5.

### 10.4.4 XPath 2.0 Versus XQuery 1.0

To find out which programs can be also written in XPath 2.0, we took the query from Subsection 1.6 and changed it to:

```
/*[last() and not (/Module/@version or //ModuleDecl or //Prolog or
    //LetClause or //WhereClause or //OrderByClause or
    //ForClause/Type or //ForClause/@posname or
    //QuantifiedExpr/InClauses/InClause/Type or //Typeswitch or
    //Extension or //ValidateExpr or //OrderedExpr or
    //UnorderedExpr or //Constructor)]
```

This way we filtered out the programs that do not contain any of these grammar symbols. These programs are both XPath 2.0 queries and XQuery 1.0 programs.

The number of these programs is only 16.

## 10.5 Category XQuery Programs With XML Documents

We took a closer look at 20 XQuery programs with 26 XML documents. One by one we tried to run them and eventually correct them in order to get results.

After programs and documents corrections we managed to run 13 programs out of 20. The rest of them contained the following errors: could not import schema, missing schema, undefined prefix, missing resources (twice), type error (value does not match a required type as specified by the matching rules in SequenceType Matching at element) and context item undefined (missing input XML document). Missing resources gripped us and we found out that not all the necessary XML documents had been downloaded. The XML documents existed on given pages, but the crawler did not manage to download them.

In the next step we analyzed the outputs of the 13 programs. Mostly the output was missing the root element (in five cases) in order to be qualified as a well-formed XML document. In one case the output was only text. Transformed

data from more than 1 XML documents was the output in 4 programs. In one case the output was very simple, only one element with text content. In one case the XQuery program was used as sorter of the XML document using `OrderByClause`.

One program contained a link to XML documents using whole URL and the program was able to run, whereas the XQuery processor downloaded given XML documents from the Internet.

## 10.6 XML Documents Analysis

Our crawler downloaded 74 XML documents but after filtering out the invalid XQuery programs with XML documents we had only 43 left. While going through the rest of XML documents referenced from XQuery programs we found the following errors:

- The content was the web page (it was in eight cases, from which four were duplicates of the same web page and the other four were duplicates of the other web page).
- The content was text document in nine cases.

We managed to repair one XML document, because its whole content was on one line and line numbers were included. This gives us 26 XML documents. All these documents are referred from XQuery programs found by the file extension **.xq** or **.xquery**.

We also managed to correct one XML document to well-formed by replacing special spaces between elements and attributes.

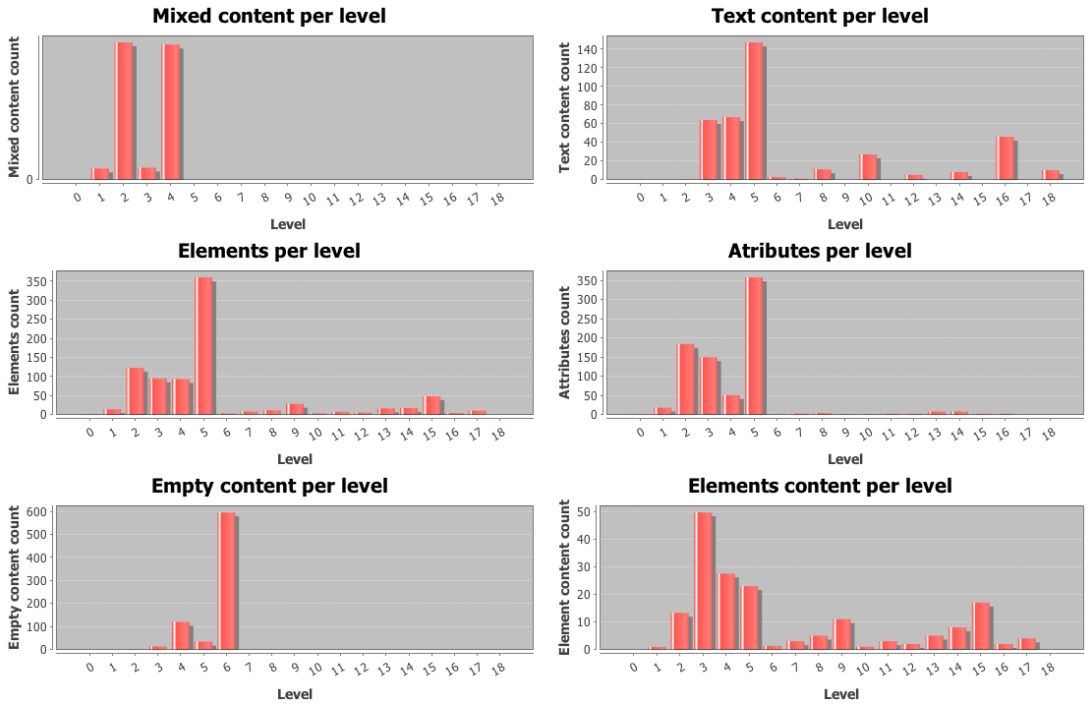



Figure 10.1: XML analysis results from Analyzer

In the next step we performed XML document analyses currently available in the Analyzer. Figure 10.1 shows the usage of elements, attributes, text nodes, etc. per level. As we can see, most of the content is present on second to fifth level. We can also see that the maximum level is 18.

Analyzer also counted that those 26 documents contained 8,951 elements and 11,401 attributes altogether, i.e. more attributes were used. In 4 XML documents the DTD was used. The smallest file has 15B and the largest has 398kB.

## 10.7 Summary

In the beginning of the chapter we focused on cleaning the data. We managed to obtain 887 valid XQuery programs out of 11,220 potential XQuery programs and we divided them into the 4 categories. These categories were based on multiple factors: where we downloaded the document (web page or URL), what type it is (library or query) and whether we managed to downloaded also related XML documents over which it queries.

In the next step we performed analyses over these categories and evaluated these analyses. We managed to obtain much interesting knowledge. Also we managed to find out the most common mistakes in XQuery programs and approximately how many XQuery programs can be found on the Internet.

Further we put our minds to downloading XML documents used in XQuery programs. We managed to download 74 of them, and, after the cleaning, we were left with 26 documents. Thanks to them we were able to look at the outputs of real-world XQuery programs. Since the file size was limited in the crawler and no large XML documents were downloaded, it would be worth to run the crawler with higher limit (i.e. filter for larger files) or with no limit on file size at all.

If we evaluate the ways of getting the XQuery programs from the various sources, then `<pre>` section source turned out to be very interesting and beneficial. After cleaning and correcting the data we were left with 260 programs, that create 29% of the all analysed programs. Figure 10.2 shows us that we mainly downloaded XQuery programs with extension **.xquery** from the Internet.

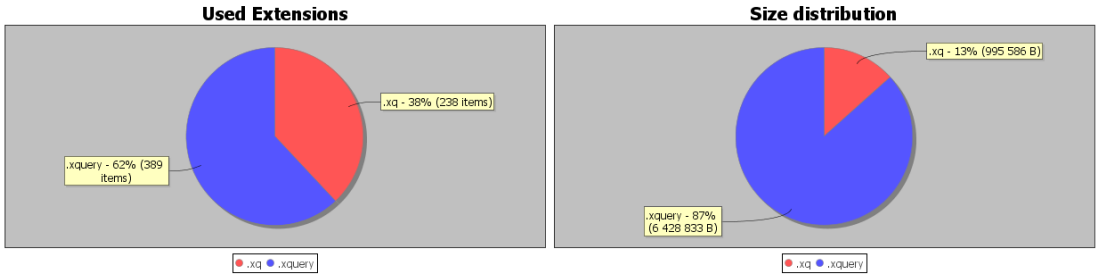



Figure 10.2: XQuery programs for file extension type of download

# Conclusion

The aim of the thesis was to gather as many real-world XQuery programs as possible and analyse them using the tool, Analyzer.

The output of the theoretical part was that no analysis of the real-world XQuery programs has been performed so far, neither has been performed analysis of XML documents used in the XQuery programs. We have found out that we have to search the Internet for the real-world XQuery programs, so we have chosen one crawler out of couple available, which we then used to find and to download the XQuery programs.

The practical part of the thesis describes the process of modifying the crawler so it would be able to distinguish potential XQuery programs and to download them. Also it would be able to parse the links to XML documents, which it also tried to download. We also added the functionality for searching for potential XQuery programs on the web pages.

During modifying the crawler functionality we repaired bug in politeness delay. Further we obtained over 4,000 URLs which were used as seeds (starting points) for the crawler and we tried to implement the crawler into Analyzer. Since Analyzer was not receiving the downloaded documents we decided to run the crawler alone.

In the result we managed to download about 11,000 potential XQuery programs and 74 XML documents referred from these potential programs. In the process crawler went through 8,000,000 URLs.

We checked up the number of potential XQuery programs using the Common-Crawl project, which regularly downloads a big part of the Internet. According to the data from July 2015 it found only 4,500 potential XQuery programs.

Downloaded data was then cleaned from the duplicities and non XQuery programs and we tried to correct a part of the invalid programs. We managed to correct 39 programs using an algorithm and 5 programs were corrected manually.

In the end we had 887 valid XQuery programs and 26 well-formed XML documents used in the 20 programs. We found out that most of the mistakes were caused by human element. The most frequent one was space on the bad place.

We divided these programs into the 4 categories according to the 3 different criteria. The categories are: programs with XML documents, libraries, presections and the last category is file extensions. The dividing criteria for the categories are: where were programs downloaded from (web page, URL, binary data or text file), data type (with query body, or without it – library) and whether we managed to download also XML documents related to these programs.

In the next part we used Analyzer tool for both XQuery programs and XML documents analyses. The output of Analyzer was processed into 12 tables and 2 figures.

In general we can say, that there are not many XQuery programs on the Internet and they do not use MIME type ‘application/xquery‘ recommended by W3C. We assumed there are no programs without query body, which serve only as libraries for the other documents, but out of 887 programs we found as much as 46 of them.

Another assumption was that half of the used XPath expressions in XQuery would contain a predicate and analyses have shown that only quarter of XPath expressions contained a predicate. Generally we can say that XPath expressions in XQuery programs use mainly shortened types of axes and have an average of 2 steps. We assumed that most of the built-in functions were used, but the analyses revealed that less than half of them was used.

Functions for working with time and date are in general hardly used and the most frequently used built-in function is `fn:doc`.

Another assumption was that only 10% of queries could be possible to be written in XPath 2.0 language, but we found out that only 16 out of 887 were possible to be written in XPath 2.0 (i.e. less than 2%). Assumption that clauses Where and Order by are used at least in the half of FLWOR expressions but we have found out they are used in less than quarter of them. We also assumed that the clause For is more frequent than the clause Let, but in reality the clause Let is 4-times more frequent. Nested FLWOR were expected to be used rarely, but it was revealed that they are used in as much as 13% of the cases. Recursive functions were expected not to occur at all, but analyses have shown that 2% of declared functions were recursive. We expected most of the operators would be additive or multiplicative type, but in result we can see, that logical operators are 13-times more frequent than additive and multiplicative types altogether.

Interesting fact is that XQuery programs do not use MIME type 'application/xquery' on the Internet but rather 'text/plain'.

Programs with XML documents were run and we were looking at the results. We managed to get the output of 13 programs. We have noticed that the outputs are of a simple character.

We also analysed XML documents using the Analyzer tool and we found out that 15% use DTD.

The contribution sum-up:

- We have shown that it is possible to download the real-world XQuery programs and XML documents which they query over.
- We have downloaded more potential XQuery programs than results from CommonCrawl data or Google website.
- We have revealed the most common mistakes which occur in the real-world XQuery programs.
- We managed to correct several mistakes in potential XQuery programs using a created program.
- We have used Analyzer tool with XQConverter plug-in for analysis of the real-world XQuery programs from thesis [56] and we have found out that it functioned not only for the XQuery Test Suite but also for the real-world XQuery programs.
- Using the Analyzer we have analyzed XML documents used in the XQuery programs.
- We summed up the results of these analyses.

# Future Work

The thesis could be expanded in several areas in the future:

- improving downloading of XML documents referred from XQuery programs (try to download them from `<pre>` sections of the same page as a program from `<pre>` sections is downloaded, canonize the link not to contain '.' or '..', add some heuristics for correcting bad URLs, etc.)
- creating a tool for correcting the invalid XQuery programs
- expanding the XQConverter with XQuery Update Facility
- implement loading the input URLs from a file in Analyzer

# Bibliography


[1] Apache Nutch v. 1.8. URL https://nutch.apache.org/. [cit. 2014-03-26].

[2] Big Data Definition. URL http://www.gartner.com/it-glossary/big-data. [cit. 2015-10-23].

[3] Bixo: A Web Mining Toolkit v. 0.9.1. URL http://openbixo.org. [cit. 2014-03-26].

[4] Common Crawl. URL https://commoncrawl.org/. [cit. 2015-10-11].

[5] Custom Search Google API. URL https://developers.google.com/custom-search/. [cit. 2014-07-16].

[6] DBpedia. URL http://wiki.dbpedia.org/. [cit. 2015-05-24].

[7] Document Type Definition. URL http://www.w3.org/TR/2004/REC-xml11-20040204/#NT-doctypedecl. [cit. 2014-07-26].

[8] Google Web Search API. URL https://developers.google.com/web-search/. [cit. 2014-03-26].

[9] The Apache Hadoop. URL http://hadoop.apache.org/. [cit. 2015-10-23].

[10] JSpider v. 0.5.0. URL http://j-spider.sourceforge.net/. [cit. 2014-03-26].

[11] Java 7 Networking Properties. URL http://docs.oracle.com/javase/7/docs/technotes/guides/net/properties.html. [cit. 2014-07-16].

[12] JoBo v. 1.4. URL http://www.matuschek.net/software/jobo/index.html. [cit. 2014-03-26].

[13] Larbin: Multi-purpose Web Crawler v. 2.6.3. URL http://larbin.sourceforge.net/index-eng.html. [cit. 2014-03-26].

[14] Oracle XQuery for Hadoop. URL https://docs.oracle.com/cd/E49465_01/doc.23/e49333/oxh.htm. [cit. 2015-10-11].

[15] The Robots Exclusion Protocol. URL http://www.robotstxt.org/orig.html. [cit. 2015-05-30].

[16] Scrapy v. 0.22. URL http://scrapy.org/. [cit. 2014-04-02].

[17] Semantic Web Dog Food. URL http://data.semanticweb.org/. [cit. 2015-05-24].

[18] Web-Harvest v. 2.0, . URL http://web-harvest.sourceforge.net/. [cit. 2014-03-26].

[19] WebLech URL Spider v. 0.0.4, . URL http://weblech.sourceforge.net/. [cit. 2014-03-26].

[20] WebSPHINX: A Personal, Customizable Web Crawler v. 0.5, . URL http://www-2.cs.cmu.edu/~rcm/websphinx/. [cit. 2014-03-26].

[21] XML Query Test Suite 1.0. URL http://dev.w3.org/2006/xquery-test-suite/PublicPagesStagingArea/. [cit. 2014-03-10].

[22] XMark – An XML Benchmark Project. URL http://www.xml-benchmark.org/. [cit. 2015-05-17].

[23] XQuery grammar. URL http://www.w3.org/TR/xquery/#nt-bnf. [cit. 2015-06-01].

[24] XYLEME PROJECT. URL http://xml.coverpages.org/xyleme.html. [cit. 2014-03-26].

[25] Abot C# Web Crawler v. 1.2.3. URL http://code.google.com/p/abot/. [cit. 2014-04-02].

[26] Xcheck: a platform for benchmarking xquery engines. In *In VLDB*, pages 1247–1250, 2006.

[27] Loredana Afanasiev and Maarten Marx. An Analysis of XQuery Benchmarks. *Inf. Syst.*, 33(2):155–181, April 2008. ISSN 0306-4379. doi: 10.1016/j.is.2007.05.002. URL http://dx.doi.org/10.1016/j.is.2007.05.002. [cit. 2015-02-15].

[28] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. An Empirical Study of Real-World SPARQL Queries. *CoRR*, abs/1103.5043, 2011. URL http://arxiv.org/abs/1103.5043. [cit. 2015-02-15].

[29] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Continuous Queries and Real-time Analysis of Social Semantic Data with C-SPARQL. In *SDoW2009*, volume 520 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009. URL http://ceur-ws.org/Vol-520/paper02.pdf. [cit. 2015-02-15].

[30] Angela Bonifati and Stefano Ceri. Comparative Analysis of Five XML Query Languages. *SIGMOD Record*, 29:2000, 2000. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.5716. [cit. 2015-02-15].

[31] W. Sherlock E. Nolan V. Sadlon M. Tamas K. Dufkova J. Podhorny Ch. Christacopoulos, L. Galambos and M. Pirchala. Egothor2 v. 3.1.5. URL http://www.egothor.org/cms/egothor2. [cit. 2014-03-26].

[32] J. Clark. XSL Transformations (XSLT) Version 1.0, November 1999. URL http://www.w3.org/TR/xslt. [cit. 2014-07-26].

[33] D. Florescu M. Marchiori D. Chamberlin, P. Fankhauser and J. Robie. XML Query Use Cases, March 2007. URL http://www.w3.org/TR/xquery-use-cases/. [cit. 2014-04-09].

[34] A. Malhotra C. M. Sperberg-McQueen D. Peterson, S. Gao and H. S. Thompson. XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes, April 2012. URL http://www.w3.org/TR/xmlschema11-2/. [cit. 2014-07-26].

[35] C. Koch G. Gottlob. Monadic queries over tree-structured data. pages 189–202. Logic in Computer Science, Los Alamitos, CA, USA, July 2002. ISBN 0-7695-1483-9. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1029828&tag=1. [cit. 2015-11-20].

[36] L. Galamboš and Others. Egothor v. 1.3. URL http://www.egothor.sf.net/. [cit. 2014-03-26].

[37] Y. Ganjisaffar. Crawler4j v. 3.5. URL http://code.google.com/p/crawler4j/. [cit. 2014-03-26].

[38] Sven Groppe, Jinghua Groppe, Niklas Klein, Ralf Bettentrupp, Stefan Böttcher, and Le Gruenwald. Transforming XSLT stylesheets into XQuery expressions and vice versa. *Computer Languages, Systems And Structures*, 37(2):76–111, 2011. ISSN 1477-8424. doi: http://dx.doi.org/10.1016/j.cl.2010.11.001. URL http://www.sciencedirect.com/science/article/pii/S1477842410000394. [cit. 2015-02-15].

[39] William G. J. Halfond and Alessandro Orso. Combining Static Analysis and Runtime Monitoring to Counter SQL-injection Attacks. In *Proceedings of the Third International Workshop on Dynamic Analysis*, WODA '05, pages 1–7, New York, NY, USA, 2005. ACM. ISBN 1-59593-126-0. doi: 10.1145/1082983.1083250. URL http://doi.acm.org/10.1145/1082983.1083250. [cit. 2015-02-15].

[40] Y. Hoppe and G. Karpouzas. Ex-Crawler v 0.1.6. URL http://ex-crawler.sourceforge.net. [cit. 2014-03-26].

[41] S. DeRose J. Clark. XML Path Language (XPath)Version 1.0, November 1999. URL http://www.w3.org/TR/xpath/. [cit. 2014-07-26].

[42] M. Kratky J. Kosek and V. Snasel. Struktura realnych XML dokumentu a metody indexovani. In ITAT 2003 Workshop on Information Technologies Applications and Theory, High Tatras, Slovakia, 2003. (in Czech).

[43] M. Dyck D. Florescu J. Melton J. Robie, D. Chamberlin and J. Simeon. XQuery Update Facility 1.0, March 2011. URL http://www.w3.org/TR/xquery-update-10/. [cit. 2014-11-26].

[44] J. Stárka M. Svoboda J. Schejbal, J. Sochna and I. Mlýnková. Analyzer - A Tool for Batch File Analysis 1.0. URL http://analyzer.kenai.com/. [cit. 2013-11-08].

[45] P. Jack and N. Levitt. Heritrix v. 3.2.0. URL http://crawler.archive.org/. [cit. 2014-03-26].

[46] Jan Sochna Jiří Schejbal Irena Mlýnková Jakub Stárka, Martin Svoboda and David Bednárek. Analyzer - A Complex System for Data Analysis.

*The Computer Journal*, 55(5):590–615, October 2011. URL http://dx.doi.org/10.1093/comjnl/bxr103. [cit. 2013-11-08].

[47] Irena Mlýnková Jiří Schejbal, Jakub Stárka. XQConverter: A System for XML Query Analysis. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6059805&tag=1. [cit. 2013-11-08].

[48] D. Barbosa L. Mignet and P. Veltri. The XML Web: a First Study. *In WWW '03, Proceedings of the 12th international conference on World Wide Web*, 2:500–510, 2003. URL http://dl.acm.org/citation.cfm?doid=775152.775223. [cit. 2015-11-20].

[49] V. Mašíček. *XSLT Benchmarking*. Master thesis, Charles University in Prague, Czech Republic, 2012. URL https://is.cuni.cz/webapps/zzp/detail/107772/?lang=en. [cit. 2015-02-15].

[50] Stefan Manegold. An Empirical Evaluation of XQuery Processors. *Inf. Syst.*, 33(2):203–220, April 2008. ISSN 0306-4379. doi: 10.1016/j.is.2007.05.004. URL http://dx.doi.org/10.1016/j.is.2007.05.004. [cit. 2015-02-15].

[51] K. Pokorny J. Mlynkova, I. Toman. Statistical Analysis of Real XML Data Collections. Technical report, Charles University, Prague, Czech Republic, June 2006. URL http://www.ksi.mff.cuni.cz/~holubova/doc/tr2006-5.pdf. [cit. 2013-11-08].

[52] R. Platt. Arachnid Web Spider Framework v. 0.4. URL http://arachnid.sourceforge.net/. [cit. 2014-03-26].

[53] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, January 2008. URL http://www.w3.org/TR/rdf-sparql-query/. [cit. 2015-11-20].

[54] M. F. Fernández D. Florescu J. Robie S. Boag, D. Chamberlin and J. Siméon. XQuery 1.0: An XML Query Language, December 2010. URL http://www.w3.org/TR/xquery/. [cit. 2014-03-10].

[55] C. M. Sperberg-McQueen S. Gao and H. S. Thompson. XML Schema Definition Language (XSD) 1.1 Part 1: Structures, April 2012. URL http://www.w3.org/TR/xmlschema11-1/. [cit. 2014-07-26].

[56] J. Schejbal. *A System for Analysis of Collections of XML Queries*. Master thesis, Charles University in Prague, Czech Republic, May 2010. URL http://www.ksi.mff.cuni.cz/~bednarek/dp/Schejbal.pdf. [cit. 2013-11-08].

[57] J. Sochna. *Collecting XML Data and Meta-Data from the Internet*. Master thesis, Charles University in Prague, Czech Republic, May 2010. URL http://www.ksi.mff.cuni.cz/~bednarek/dp/Sochna.pdf. [cit. 2013-11-08] in Czech.

[58] C. M. Sperberg-McQueen E. Maler T. Bray, J. Paoli and F. Yergeau. Extensible Markup Language (XML) 1.0, August 2006. URL http://www.w3.org/TR/2006/REC-xml-20060816/. [cit. 2013-11-08].

# List of Tables

# List of Figures

# Attachment 1

## CD Content

The enclosed CD contains:

- In the directory **chapter 3** is downloaded XQuery Test Suite version 1.0.3.
- In the directory **chapter 4** is documentations of Analyzer, XQConverter as console program and some translated XQuery programs with originals.
- In the directory **chapter 5** is theoriginal sources files of the analyzed crawlers.
- In the directory **chapter 6** is the downloaded Google web pages with results, the program to extract URLs from them and the extracted URLs in the text file.
- In the directory **chapter 7** is the project of modified crawler.
- In the directory **chapter 8** is Analyzer project with XQuery module.
- In the directory **chapter 9** is crawler testing runs outputs, two projects for Hadoop programs over CommonCrawl data and the outputs from this data minings.
- In the directory **chapter 10** is 3 java programs for correcting, clustering and validating potential XQuery programs. Also outputs from crawling the Internet are here. The XPaths used in Analyzer and the results from Analyzer analyses are also here.
- In the directory **text** is the source files, images and pdf of this thesis.

# Attachment 2

## XQuery grammar symbols [56]

These symbols are used in the XQuery grammar and their modified form is used in the XQConverter output. Particular symbols are therefore elements of the XML representation of a given XQuery program. Each symbol is provide with a short description.

**AdditiveExpr** – This symbol represents subtraction or addition operation.

**AndExpr** – This symbol represents logical expression and. It is true only if both values are true.

**BaseURIDecl** – This symbol specifies the base URI property of the static context. The base URI property is used when resolving relative URIs within a module.

**BoundarySpaceDecl** – This symbol sets the boundary-space policy in the static context, overriding any implementation-defined default.

**CastableExpr** – This symbol tests whether a given value is castable into a given target type.

**CastExpr** – This symbol converts a value to a specific datatype.

**ComputedConstructor** – This symbol allows to create nodes alternatively.

**ConstructionDecl** – This symbol sets the construction mode in the static context, overriding any implementation-defined default.

**ContextItemExpr** – This symbol evaluates to the context item, which may be either a node or an atomic value.

**CopyNamespacesDecl** – This symbol sets the value of copy-namespaces mode in the static context, overriding any implementation-defined default. Copy-namespaces mode controls the namespace bindings that are assigned when an existing element node is copied by an element constructor or document constructor.

**DefaultCollationDecl** – This symbol sets the value of the default collation in the static context, overriding any implementation-defined default.

**DefaultNamespaceDecl** – This symbol can be used in a Prolog to facilitate the use of unprefixed QNames.

**DirectConstructor** – This symbol is a form of element constructor in which the name of the constructed element is a constant.

**EmptyOrderDecl** – This symbol sets the default order for empty sequences in the static context, overriding any implementation-defined default. This declaration controls the processing of empty sequences and NaN values as ordering keys in an order by clause in a FLWOR expression.

**ExtensionExpr** – This symbol consists of one or more pragmas, followed by an expression enclosed in curly braces.

**FLWORExpr** – This symbol consists of 4 optional clauses (‘for clause‘, ‘let clause‘, ‘where clause‘ and ‘order by clause‘) and 1 required clause (‘return clause‘). The result of the FLWOR expression is an ordered sequence containing the results of these evaluations, concatenated as if by the comma operator.

**ForClause** – This symbol generates an ordered sequence of tuples of bound variables, called the tuple stream.

**ForwardAxis** – This symbol represents one of these axis ‘child‘, ‘descendant‘, ‘attribute‘, ‘self‘, ‘descendant-or-self‘, ‘following-sibling‘, ‘following‘.
**FunctionCall** – This symbol consists of a QName followed by a parenthesized list of zero or more expressions, called arguments.
**FunctionDecl** – This symbol specifies whether a function is user-defined or external. User-defined function, the function declaration includes an expression called the function body that defines how the result of the function is computed from its parameters. External functions are functions that are implemented outside the query environment.
**GeneralComp** – This symbol represents comparison using one of these symbols ‘=‘, ‘! =‘, ‘<‘, ‘<=‘, ‘>‘, ‘>=‘.
**IfExpr** – This symbol represents a condition. If the effective boolean value of the test expression is true, the value of the then-expression is returned. If the effective boolean value of the test expression is false, the value of the else-expression is returned.
**InstanceofExpr** – This symbol returns true if the value of its first operand matches the SequenceType in its second operand.
**IntersectExceptExpr** – This symbol eliminates duplicate nodes from their result sequences based on node identity. The intersect operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in both operands. The except operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in the first operand but not in the second operand.
**LetClause** – This symbol generates an ordered sequence of tuples of bound variables, called the tuple stream.
**Literal** – This symbol represents different values as ‘xs:integer‘, ‘xs:decimal‘, ‘xs:double‘, ‘xs:untypedAtomic‘, ‘xs:string‘.
**ModuleDecl** – This symbol serves to identify a module as a library module. A module declaration begins with the keyword module and contains a namespace prefix and a URILiteral.
**ModuleImport** – This symbol imports the function declarations and variable declarations from one or more library modules into the function signatures and in-scope variables of the importing module.
**MultiplicativeExpr** – This symbol represents multiplication or division or modulo operation.
**NamespaceDecl** – This symbol declares a namespace prefix and associates it with a namespace URI, adding the (prefix, URI) pair to the set of statically known namespaces.
**NodeComp** – This symbol represents comparison using one of these symbols ‘is‘, ‘<<‘, ‘>>‘.
**OptionDecl** – This symbol serves as a particular option that will be recognized by some implementations and not by others. The syntax is designed so that option declarations can be successfully parsed by all implementations.
**OrderByClause** – This symbol is used to reorder the tuple stream.
**OrderedExpr** – This symbol sets the ordering mode in the static context to ordered for a certain region in a query.
**OrderingModeDecl** – This symbol sets the ordering mode in the static context, overriding any implementation-defined default.

**OrExpr** – This symbol represents logical expression or. It is true if one of values is true.
**PathExpr** – This symbol can be used to locate nodes within trees. A path expression consists of a series of one or more steps, separated by '/' or '//', and optionally beginning with '/' or '//'.
**Prolog** – This symbol is a series of declarations and imports that define the processing environment for the module that contains the Prolog.
**QuantifiedExpr** – This symbol begins with a quantifier, which is the keyword 'some' or 'every', followed by one or more in-clauses that are used to bind variables, followed by the keyword 'satisfies' and a test expression.
**QueryBody** – This symbol consists of an expression that defines the result of the query.
**RangeExpr** – This symbol can be used to construct a sequence of consecutive integers.
**ReverseAxis** – This symbol represents one of these axises: 'parent', 'ancestor', 'preceding-sibling', 'preceding', 'ancestor-or-self'.
**SchemaImport** – This symbol imports the element declarations, attribute declarations, and type definitions from a schema into the in-scope schema definitions. For each user-defined atomic type in the schema, schema import also adds a corresponding constructor function.
**TreatExpr** – This symbol can be used to modify the static type of its operand. It does not change the dynamic type or value of its operand. Its purpose is to ensure that an expression has an expected dynamic type at evaluation time.
**TypeswitchExpr** – This symbol represent switch condition. It has an expression enclosed in parentheses, called the operand expression. This is the expression whose type is being tested. The remainder of the typeswitch expression consists of one or more case clauses and a default clause.
**UnaryExpr** – This symbol represents unary plus or unary minus.
**UnionExpr** – This symbol eliminates duplicate nodes from their result sequences based on node identity. It takes two node sequences as operands and return a sequence containing all the nodes that occur in either of the operands.
**UnorderedExpr** – This symbol sets the ordering mode in the static context to unordered for a certain region in a query.
**ValidateExpr** – This symbol can be used to validate a document node or an element node with respect to the in-scope schema definitions.
**ValueComp** – This symbol represents comparison using one of these symbols 'eq', 'ne', 'lt', 'le', 'gt', 'ge'.
**VarDecl** – This symbol adds the static type of a variable to the in-scope variables, and may also add a value for the variable to the variable values.
**VarRef** – This symbol is a QName preceded by a $-sign.
**WhereClause** – This symbol filters the tuple stream, retaining some tuples and discarding others.

# Attachment 3

## Complex XQConverter example

We picked the following Use Case XQuery program out of the XQuery Test Suite [21]. In XQuery Use Cases [33] it is located at Experiences and Exemplars as last 12th query. Query stands for: Find pairs of books that have different titles but the same set of authors (possibly in a different order).

```
(: insert-start :)
declare variable $input-context external;
(: insert-end :)

<bib>
{
    for $book1 in $input-context//book,
        $book2 in $input-context//book
    let $aut1 := for $a in $book1/author
                 order by exactly-one($a/last), exactly-one($a/first)
                 return $a
    let $aut2 := for $a in $book2/author
                 order by exactly-one($a/last), exactly-one($a/first)
                 return $a
    where $book1 << $book2
    and not($book1/title = $book2/title)
    and deep-equal($aut1, $aut2)
    return
        <book-pair>
            { $book1/title }
            { $book2/title }
        </book-pair>
}
</bib>
```

The following output was generated by the XQConverter (for better comprehension and orientation of the output, the output was separated into 8 parts, and 8 different colours were used to differentiate the parts).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Module type="main">
  <Prolog>
    <VarDecl name="input-context">
      <VarValue external="true"/>
    </VarDecl>
  </Prolog>
  <QueryBody>
    <Constructor kind="direct" type="element">
      <Name name="bib"/>
```

```
<Content>
  <String value="&#13;&#10;"/>
  <FLWOR>
    <TupleStream>
      <ForClause varname="book1">
        <BindingSequence>
          <Path initial-step="context">
            <Step>
              <VarRef name="input-context"/>
            </Step>
            <Step>
              <Axis abbreviated="true" direction="forward"
              kind="descendant-or-self">
                <KindTest kind="any-kind"/>
              </Axis>
            </Step>
            <Step>
              <Axis abbreviated="true" direction="forward"
              kind="child">
                <NameTest name="book"/>
              </Axis>
            </Step>
          </Path>
        </BindingSequence>
      </ForClause>
      <ForClause varname="book2">
        <BindingSequence>
          <Path initial-step="context">
            <Step>
              <VarRef name="input-context"/>
            </Step>
            <Step>
              <Axis abbreviated="true" direction="forward"
              kind="descendant-or-self">
                <KindTest kind="any-kind"/>
              </Axis>
            </Step>
            <Step>
              <Axis abbreviated="true" direction="forward"
              kind="child">
                <NameTest name="book"/>
              </Axis>
            </Step>
          </Path>
        </BindingSequence>
      </ForClause>
      <LetClause varname="aut1">
        <BindingSequence>
```

```
<FLWOR>
  <TupleStream>
    <ForClause varname="a">
      <BindingSequence>
        <Path initial-step="context">
          <Step>
            <VarRef name="book1"/>
          </Step>
          <Step>
            <Axis abbreviated="true"
            direction="forward" kind="child">
              <NameTest name="author"/>
            </Axis>
          </Step>
        </Path>
      </BindingSequence>
    </ForClause>
  </TupleStream>
  <OrderByClause stable="false">
    <OrderSpec>
      <FunctionCall name="exactly-one">
        <Path initial-step="context">
          <Step>
            <VarRef name="a"/>
          </Step>
          <Step>
            <Axis abbreviated="true"
            direction="forward" kind="child">
              <NameTest name="last"/>
            </Axis>
          </Step>
        </Path>
      </FunctionCall>
    </OrderSpec>
    <OrderSpec>
      <FunctionCall name="exactly-one">
        <Path initial-step="context">
          <Step>
            <VarRef name="a"/>
          </Step>
          <Step>
            <Axis abbreviated="true"
            direction="forward" kind="child">
              <NameTest name="first"/>
            </Axis>
          </Step>
        </Path>
      </FunctionCall>
```

```
        </OrderSpec>
      </OrderByClause>
      <ReturnClause>
        <VarRef name="a"/>
      </ReturnClause>
    </FLWOR>
  </BindingSequence>
</LetClause>
<LetClause varname="aut2">
  <BindingSequence>
    <FLWOR>
      <TupleStream>
        <ForClause varname="a">
          <BindingSequence>
            <Path initial-step="context">
              <Step>
                <VarRef name="book2"/>
              </Step>
              <Step>
                <Axis abbreviated="true"
                direction="forward" kind="child">
                  <NameTest name="author"/>
                </Axis>
              </Step>
            </Path>
          </BindingSequence>
        </ForClause>
      </TupleStream>
      <OrderByClause stable="false">
        <OrderSpec>
          <FunctionCall name="exactly-one">
            <Path initial-step="context">
              <Step>
                <VarRef name="a"/>
              </Step>
              <Step>
                <Axis abbreviated="true"
                direction="forward" kind="child">
                  <NameTest name="last"/>
                </Axis>
              </Step>
            </Path>
          </FunctionCall>
        </OrderSpec>
        <OrderSpec>
          <FunctionCall name="exactly-one">
            <Path initial-step="context">
              <Step>
```

```
                    <VarRef name="a"/>
                  </Step>
                  <Step>
                    <Axis abbreviated="true"
                    direction="forward" kind="child">
                      <NameTest name="first"/>
                    </Axis>
                  </Step>
                </Path>
              </FunctionCall>
            </OrderSpec>
          </OrderByClause>
          <ReturnClause>
            <VarRef name="a"/>
          </ReturnClause>
        </FLWOR>
      </BindingSequence>
    </LetClause>
</TupleStream>
<WhereClause>
  <Operator class="logical" name="and">
    <Operator class="logical" name="and">
      <Operator class="comparison" name="precedes"
      subclass="node">
        <VarRef name="book1"/>
        <VarRef name="book2"/>
      </Operator>
      <FunctionCall name="not">
        <Operator class="comparison" name="equals"
        subclass="general">
          <Path initial-step="context">
            <Step>
              <VarRef name="book1"/>
            </Step>
            <Step>
              <Axis abbreviated="true"
              direction="forward" kind="child">
                <NameTest name="title"/>
              </Axis>
            </Step>
          </Path>
          <Path initial-step="context">
            <Step>
              <VarRef name="book2"/>
            </Step>
            <Step>
              <Axis abbreviated="true"
              direction="forward" kind="child">
```

```
                    <NameTest name="title"/>
                  </Axis>
                </Step>
              </Path>
            </Operator>
          </FunctionCall>
        </Operator>
        <FunctionCall name="deep-equal">
          <VarRef name="aut1"/>
          <VarRef name="aut2"/>
        </FunctionCall>
      </Operator>
    </WhereClause>
    <ReturnClause>
      <Constructor kind="direct" type="element">
        <Name name="book-pair"/>
        <Content>
          <String value="&#13;&#10;            "/>
          <Path initial-step="context">
            <Step>
              <VarRef name="book1"/>
            </Step>
            <Step>
              <Axis abbreviated="true" direction="forward"
              kind="child">
                <NameTest name="title"/>
              </Axis>
            </Step>
          </Path>
          <String value="&#13;&#10;            "/>
          <Path initial-step="context">
            <Step>
              <VarRef name="book2"/>
            </Step>
            <Step>
              <Axis abbreviated="true" direction="forward"
              kind="child">
                <NameTest name="title"/>
              </Axis>
            </Step>
          </Path>
          <String value="&#13;&#10;        "/>
        </Content>
      </Constructor>
    </ReturnClause>
  </FLWOR>
  <String value="&#13;&#10;"/>
</Content>
```

```
    </Constructor>
  </QueryBody>
</Module>
```

# Attachment 4

## XPaths Used in Analyzer

These XPaths were used to create the analysis in program Analyzer. We list them according to the groups in which they were used with the short description of the group. Every XPath expression must be separated from others by semicolon when inserting to Analyzer so we kept this formatting here, too.

The first group 'XQuery Grammar Symbols' counts number of each XQuery grammar symbols:

```
//FunctionDecl;
//QueryBody;
//FLWOR;
//ForClause;
//LetClause;
//WhereClause;
//OrderByClause;
//QuantifiedExpr[@quantifier="some"];
//QuantifiedExpr[@quantifier="every"];
//Typeswitch;
//IfExpr;
//Operator[@class="logical" and @name="or"];
//Operator[@class="logical" and @name="and"];
//Operator[@class="range"];
//Operator[@class="additive"];
//Operator[@class="multiplicative"];
//Operator[@class="set" and @name="union"];
//Operator[@class="set" and (@name="difference" or
          @name="intersection")];
//Operator[@class="type-test" and @name="instance-of"];
//Operator[@class="type-cast" and @name="treat-as"];
//Operator[@class="type-test" and @name="castable-as"];
//Operator[@class="type-cast" and @name="cast-as"];
//Operator[@class="unary"];
//Operator[@class="comparison" and @subclass="general"];
//Operator[@class="comparison" and @subclass="value"];
//Operator[@class="comparison" and @subclass="node"];
//ValidateExpr;
//Extension;
//Path;
//Axis[@direction="forward"];
//Axis[@direction="reverse"];
//Literal;
//VarRef;
//ContextItem;
//OrderedExpr;
//UnorderedExpr;
```

```
//FunctionCall;
//Constructor[@kind="direct"];
//Constructor[@kind="computed"]
```

The second group 'XQuery Interesting Analysis' counts number of different statistics like recursive functions, external variables, number of nested `FLWOR`, number of XPath expresions with `Predicate`, etc.:

```
//FunctionDecl[@name=descendant::FunctionCall/@name];
//FunctionDecl[not (//FunctionCall/@name=@name)];
/descendant::FLWOR[not (ancestor::FLWOR) and descendant::FLWOR and
                   not (descendant::FLWOR/descendant::FLWOR)];
/descendant::FLWOR[not (ancestor::FLWOR) and
  descendant::FLWOR/descendant::FLWOR and
  not (descendant::FLWOR/descendant::FLWOR/descendant::FLWOR)];
/descendant::FLWOR[not (ancestor::FLWOR) and
  descendant::FLWOR/descendant::FLWOR/descendant::FLWOR and
  not (descendant::FLWOR/descendant::FLWOR/descendant::FLWOR/
       descendant::FLWOR)];
/descendant::FLWOR[not (ancestor::FLWOR) and
  descendant::FLWOR/descendant::FLWOR/descendant::FLWOR/
  descendant::FLWOR];
//FLWOR[not (*/ForClause)];
//FLWOR[not (*/LetClause)];
//ReturnClause[descendant::IfExpr or descendant::FunctionCall or
               descendant::Constructor[@kind="computed"] or
               descendant::Typeswitch];
//ReturnClause[child::Path or child::Constructor[@kind="direct"]];
//QueryBody[child::IfExpr or child::FunctionCall or
            child::Constructor[@kind="computed"] or
            child::Typeswitch];
//QueryBody[child::Path or child::Constructor[@kind="direct"]];
//Path[Step/Predicates];
//Path/Step[last()]/Axis[@kind!="attribute"]/NameTest;
//Path/Step[last()]/Axis[@kind="attribute"];
//Path/Step[last()]/Axis/KindTest[@kind="text"];
//VarDecl[not (@name=//VarRef/@name)];
//VarDecl/VarValue[@external="true"];
/*[last() and not (/Module/@version or //ModuleDecl or //Prolog or
                   //LetClause or //WhereClause or //OrderByClause
                   or //ForClause/Type or //ForClause/@posname or
                   //QuantifiedExpr/InClauses/InClause/Type or
                   //Typeswitch or //Extension or //ValidateExpr or
                   //OrderedExpr or //UnorderedExpr or
                   //Constructor)]
```

The third group 'XQuery FLWOR Counts' counts number of `ForClause` and `LetClause` in an one `FLWOR`:

```
//FLWOR[count(*/ForClause) = 1];
//FLWOR[count(*/ForClause) = 2];
//FLWOR[count(*/ForClause) = 3];
//FLWOR[count(*/ForClause) = 4];
//FLWOR[11 > count(*/ForClause) and count(*/ForClause) > 4];
//FLWOR[21 > count(*/ForClause) and count(*/ForClause) > 10];
//FLWOR[count(*/ForClause) > 20];
//FLWOR[count(*/LetClause) = 1];
//FLWOR[count(*/LetClause) = 2];
//FLWOR[count(*/LetClause) = 3];
//FLWOR[count(*/LetClause) = 4];
//FLWOR[count(*/LetClause) = 5];
//FLWOR[count(*/LetClause) = 6];
//FLWOR[count(*/LetClause) = 7];
//FLWOR[count(*/LetClause) = 8];
//FLWOR[count(*/LetClause) = 9];
//FLWOR[count(*/LetClause) = 10];
//FLWOR[21 > count(*/LetClause) and count(*/LetClause) > 10];
//FLWOR[31 > count(*/LetClause) and count(*/LetClause) > 20];
//FLWOR[41 > count(*/LetClause) and count(*/LetClause) > 30];
//FLWOR[51 > count(*/LetClause) and count(*/LetClause) > 40];
//FLWOR[61 > count(*/LetClause) and count(*/LetClause) > 50];
//FLWOR[71 > count(*/LetClause) and count(*/LetClause) > 60];
//FLWOR[81 > count(*/LetClause) and count(*/LetClause) > 70];
//FLWOR[91 > count(*/LetClause) and count(*/LetClause) > 80];
//FLWOR[count(*/LetClause) > 90]
```

The fourth group ‘XQuery FLWOR Joins‘ counts number of different ways of writing of the join in `FLWOR` expression:

```
//FLWOR[WhereClause/Operator/@name="equals" and
        WhereClause/Operator/descendant::VarRef/@name !=
        WhereClause/Operator/descendant::VarRef/@name and
        WhereClause/Operator/descendant::VarRef/@name =
        ReturnClause/descendant::VarRef/@name];
//FLWOR[count(descendant::ForClause)>1 and
        descendant::ForClause/descendant::Operator/@name =
        "equals" and descendant::ForClause/descendant::Operator/
                    descendant::NameTest/@name =
        ReturnClause/descendant::NameTest/@name];
//FLWOR[ReturnClause/IfExpr/TestExpression/Operator/@name
        = "equals" and count(ReturnClause/IfExpr/TestExpression/
                              descendant::NameTest/@name) > 1 and
        ReturnClause/IfExpr/TestExpression/descendant::NameTest/
        @name = ReturnClause/IfExpr/ThenExpression/
                descendant::NameTest/@name];
//FLWOR[ReturnClause/descendant::Predicates/Operator/@name =
        "equals" and count(ReturnClause/descendant::CommaOperator/
                           descendant::NameTest/@name) > 1 and
```

```
        ReturnClause/descendant::Predicates/descendant::NameTest/
        @name = ReturnClause/descendant::CommaOperator/
                descendant::NameTest/@name]
```

The fifth group ‘XPath Steps‘ counts number of `Step` grammar symbol in `Path` grammar symbol:

```
//Path[count(Step) = 1];
//Path[count(Step) = 2];
//Path[count(Step) = 3];
//Path[count(Step) = 4];
//Path[count(Step) = 5];
//Path[count(Step) = 6];
//Path[count(Step) = 7];
//Path[count(Step) = 8];
//Path[count(Step) = 9];
//Path[count(Step) = 10];
//Path[count(Step) = 11];
//Path[count(Step) = 12];
//Path[count(Step) = 13];
//Path[count(Step) = 14];
//Path[count(Step) > 14]
```

The sixth group ‘XPath Axis‘ counts number of different `Axis` grammar symbol in the abbreviated or full form:

```
//Axis[@kind="ancestor" and @abbreviated="true"];
//Axis[@kind="ancestor-or-self" and @abbreviated="true"];
//Axis[@kind="attribute" and @abbreviated="true"];
//Axis[@kind="child" and @abbreviated="true"];
//Axis[@kind="descendant" and @abbreviated="true"];
//Axis[@kind="descendant-or-self" and @abbreviated="true"];
//Axis[@kind="following" and @abbreviated="true"];
//Axis[@kind="following-siblibg" and @abbreviated="true"];
//Axis[@kind="namespace" and @abbreviated="true"];
//Axis[@kind="parent" and @abbreviated="true"];
//Axis[@kind="preceding" and @abbreviated="true"];
//Axis[@kind="preceding-sibling" and @abbreviated="true"];
//Axis[@kind="self" and @abbreviated="true"];
//Axis[@kind="ancestor" and @abbreviated="false"];
//Axis[@kind="ancestor-or-self" and @abbreviated="false"];
//Axis[@kind="attribute" and @abbreviated="false"];
//Axis[@kind="child" and @abbreviated="false"];
//Axis[@kind="descendant" and @abbreviated="false"];
//Axis[@kind="descendant-or-self" and @abbreviated="false"];
//Axis[@kind="following" and @abbreviated="false"];
//Axis[@kind="following-siblibg" and @abbreviated="false"];
//Axis[@kind="namespace" and @abbreviated="false"];
//Axis[@kind="parent" and @abbreviated="false"];
```

```
//Axis[@kind="preceding" and @abbreviated="false"];
//Axis[@kind="preceding-sibling" and @abbreviated="false"];
//Axis[@kind="self" and @abbreviated="false"]
```

The seventh group ‘XPath BuildIn Functions‘ counts number of calls of a built-in function used in XPath expression (all lines starts with `//Path/descendant::FunctionCall` and we only list predicates so it can be in the readable form):

```
[@name="node-name" or @name="fn:node-name"];
[@name="nilled" or @name="fn:nilled"];
[@name="data" or @name="fn:data"];
[@name="base-uri" or @name="fn:base-uri"];
[@name="document-uri" or @name="fn:document-uri"];
[@name="error" or @name="fn:error"];
[@name="trace" or @name="fn:trace"];
[@name="number" or @name="fn:number"];
[@name="abs" or @name="fn:abs"];
[@name="ceiling" or @name="fn:ceiling"];
[@name="floor" or @name="fn:floor"];
[@name="round" or @name="fn:round"];
[@name="round-half-to-even" or @name="fn:round-half-to-even"];
[@name="string" or @name="fn:string"];
[@name="codepoints-to-string" or @name="fn:codepoints-to-string"];
[@name="string-to-codepoints" or @name="fn:string-to-codepoints"];
[@name="codepoint-equal" or @name="fn:codepoint-equal"];
[@name="compare" or @name="fn:compare"];
[@name="concat" or @name="fn:concat"];
[@name="string-join" or @name="fn:string-join"];
[@name="substring" or @name="fn:substring"];
[@name="string-length" or @name="fn:string-length"];
[@name="normalize-space" or @name="fn:normalize-space"];
[@name="normalize-unicode" or @name="fn:normalize-unicode"];
[@name="upper-case" or @name="fn:upper-case"];
[@name="lower-case" or @name="fn:lower-case"];
[@name="translate" or @name="fn:translate"];
[@name="escape-uri" or @name="fn:escape-uri"];
[@name="contains" or @name="fn:contains"];
[@name="starts-with" or @name="fn:starts-with"];
[@name="ends-with" or @name="fn:ends-with"];
[@name="substring-before" or @name="fn:substring-before"];
[@name="substring-after" or @name="fn:substring-after"];
[@name="matches" or @name="fn:matches"];
[@name="replace" or @name="fn:replace"];
[@name="tokenize" or @name="fn:tokenize"];
[@name="resolve-uri" or @name="fn:resolve-uri"];
[@name="boolean" or @name="fn:boolean"];
[@name="not" or @name="fn:not"];
[@name="true" or @name="fn:true"];
```

```
[@name="false" or @name="fn:false"];
[@name="dateTime" or @name="fn:dateTime"];
[@name="years-from-duration" or @name="fn:years-from-duration"];
[@name="months-from-duration" or @name="fn:months-from-duration"];
[@name="days-from-duration" or @name="fn:days-from-duration"];
[@name="hours-from-duration" or @name="fn:hours-from-duration"];
[@name="minutes-from-duration" or @name="fn:minutes-from-duration"];
[@name="seconds-from-duration" or @name="fn:seconds-from-duration"];
[@name="year-from-dateTime" or @name="fn:year-from-dateTime"];
[@name="month-from-dateTime" or @name="fn:month-from-dateTime"];
[@name="day-from-dateTime" or @name="fn:day-from-dateTime"];
[@name="hours-from-dateTime" or @name="fn:hours-from-dateTime"];
[@name="minutes-from-dateTime" or @name="fn:minutes-from-dateTime"];
[@name="seconds-from-dateTime" or @name="fn:seconds-from-dateTime"];
[@name="timezone-from-dateTime" or
  @name="fn:timezone-from-dateTime"];
[@name="year-from-date" or @name="fn:year-from-date"];
[@name="month-from-date" or @name="fn:month-from-date"];
[@name="day-from-date" or @name="fn:day-from-date"];
[@name="timezone-from-date" or @name="fn:timezone-from-date"];
[@name="hours-from-time" or @name="fn:hours-from-time"];
[@name="minutes-from-time" or @name="fn:minutes-from-time"];
[@name="seconds-from-time" or @name="fn:seconds-from-time"];
[@name="timezone-from-time" or @name="fn:timezone-from-time"];
[@name="adjust-dateTime-to-timezone" or
  @name="fn:adjust-dateTime-to-timezone"];
[@name="adjust-date-to-timezone" or
  @name="fn:adjust-date-to-timezone"];
[@name="adjust-time-to-timezone" or
  @name="fn:adjust-time-to-timezone"];
[@name="QName" or @name="fn:QName"];
[@name="local-name-from-QName" or @name="fn:local-name-from-QName"];
[@name="namespace-uri-from-QName" or
  @name="fn:namespace-uri-from-QName"];
[@name="namespace-uri-for-prefix" or
  @name="fn:namespace-uri-for-prefix"];
[@name="in-scope-prefixes" or @name="fn:in-scope-prefixes"];
[@name="resolve-QName" or @name="fn:resolve-QName"];
[@name="name" or @name="fn:name"];
[@name="local-name" or @name="fn:local-name"];
[@name="namespace-uri" or @name="fn:namespace-uri"];
[@name="lang" or @name="fn:lang"];
[@name="root" or @name="fn:root"];
[@name="index-of" or @name="fn:index-of"];
[@name="remove" or @name="fn:remove"];
[@name="empty" or @name="fn:empty"];
[@name="exists" or @name="fn:exists"];
[@name="distinct-values" or @name="fn:distinct-values"];
```

```
[@name="insert-before" or @name="fn:insert-before"];
[@name="reverse" or @name="fn:reverse"];
[@name="subsequence" or @name="fn:subsequence"];
[@name="unordered" or @name="fn:unordered"];
[@name="zero-or-one" or @name="fn:zero-or-one"];
[@name="one-or-more" or @name="fn:one-or-more"];
[@name="exactly-one" or @name="fn:exactly-one"];
[@name="deep-equal" or @name="fn:deep-equal"];
[@name="count" or @name="fn:count"];
[@name="avg" or @name="fn:avg"];
[@name="max" or @name="fn:max"];
[@name="min" or @name="fn:min"];
[@name="sum" or @name="fn:sum"];
[@name="id" or @name="fn:id"];
[@name="idref" or @name="fn:idref"];
[@name="doc" or @name="fn:doc"];
[@name="doc-available" or @name="fn:doc-available"];
[@name="collection" or @name="fn:collection"];
[@name="position" or @name="fn:position"];
[@name="last" or @name="fn:last"];
[@name="current-dateTime" or @name="fn:current-dateTime"];
[@name="current-date" or @name="fn:current-date"];
[@name="current-time" or @name="fn:current-time"];
[@name="implicit-timezone" or @name="fn:implicit-timezone"];
[@name="default-collation" or @name="fn:default-collation"];
[@name="static-base-uri" or @name="fn:static-base-uri"]
```