

XCheck  
A benchmark checker for XML query processors  
User manual

May 24, 2006

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	XCheck . . . . .	3
1.2	How to obtain reliable program runtimes . . . . .	4
1.3	Query elaboration times . . . . .	4
1.4	The XPath/XQuery engines already supported . . . . .	5
1.5	System requirements . . . . .	7
1.6	License use . . . . .	7
<b>2</b>	<b>Getting started</b>	<b>8</b>
2.1	Installing XCheck . . . . .	8
2.1.1	Installing the required CPAN Perl modules . . . . .	9
2.2	Configure XCheck . . . . .	10
2.3	Test the configuration . . . . .	11
<b>3</b>	<b>Running phase</b>	<b>13</b>
3.1	Build an experiment . . . . .	13
3.2	Run an experiment . . . . .	16
3.3	The update option . . . . .	19
<b>4</b>	<b>Data analysis phase</b>	<b>21</b>
4.1	Aggregation measures, statistics and plots . . . . .	21
4.2	Build a data analysis input . . . . .	24
4.3	Execute the data analysis phase . . . . .	24
<b>5</b>	<b>How to create new adapters</b>	<b>28</b>
5.1	Create a new adapters . . . . .	28
5.2	The <b>before</b> and <b>after</b> elements . . . . .	31
5.3	Regular expressions for times and errors . . . . .	32
<b>A</b>	<b>Appendix - DTD</b>	<b>35</b>
A.1	experiment.dtd . . . . .	35
A.2	outcomes.dtd . . . . .	35
A.3	analysis.dtd . . . . .	36
A.4	outcome.s1.dtd . . . . .	37
A.5	outcome.s2.dtd . . . . .	37
A.6	outcome.s3.dtd . . . . .	38
A.7	outcome.s5.dtd . . . . .	38
A.8	outcome.s6.dtd . . . . .	39
A.9	engines.dtd . . . . .	39
A.10	adapter.dtd . . . . .	40

# 1 Introduction

## 1.1 XCheck

In computing, a benchmark is a method to assess the relative performance of an object, by running a number of standard tests and trials against it. The practice of *benchmarking*<sup>1</sup> can help vendors, developers, and users to evaluate the performance of an object. Moreover, it can help researchers to spot the weaknesses of the current technology and hence to propose improved solutions. Benchmarking is particularly important when a technology is young, because in this phase many different solutions are proposed both in the academic and in the commercial world, and there is a strong need for a framework to analyze the capabilities and performance of such products as early as possible. This is the case of XML.

Running a benchmark on different applications takes a lot of time and usually generates huge amounts of raw data. Moreover, interpreting the outcomes of the evaluation is a subtle and crucially important task. The main goals of XCheck are to automatize some of the tasks involved in the practice of benchmarking *XML query processors* (or query engines) and, most importantly, to help the user to spot the weaknesses of a given processor and to compare the performance of the different engines under evaluation.

XCheck is a software for automatic execution of a benchmark on XML query processors (XPath and XQuery processors). XCheck works in two phases: *running* and *data analysis*. In the *running phase* XCheck executes the benchmark on available XML engines and stores the engine execution times that it measures itself, as well as the execution times that the engines output. Optionally, it also stores the benchmark query results. In the *data analysis phase* XCheck elaborates some statistics on the execution times gathered in the previous phase. As a result it generates performance reports (in HTML format) containing lots of plots<sup>2</sup> (in PostScript and PNG formats).

XCheck focuses on *performance benchmarks*, as opposed to *correctness benchmarks*. Thus, the benchmarks do not need to specify correct answers. Nevertheless, XCheck helps to detect incorrect answers by comparing the sizes of the query results obtained from different engines.

The input and the output of XCheck are files in open formats like XML, HTML, PostScript and Gnuplot. This means that you can integrate, modify, and why not, improve XCheck's features, in a simple way. Moreover, XCheck

---

<sup>1</sup>Not to be confused with the term *bench-marketing*, which refers to the practice by which manufacturers report only those aspects of benchmarks that show their products in the best light.

<sup>2</sup>XCheck makes plots using Gnuplot, a free software for generating plots. For more information visit <http://www.gnuplot.info>

is written in Perl and this allows it to be used on different platforms<sup>3</sup>.

[**Todo:**] This intro must be still worked out when the rest is ready. Here all XCheck features should be mentioned. By reading only this intro it should be clear what XCheck can do and what not.

## 1.2 How to obtain reliable program runtimes

Measuring reliable program runtimes is not as straightforward as it may appear. Computer's hardware and system software influence the runtime of a program. The CPU speed, the amount of main memory, the amount of cache, the operating system and the compiler all play significant roles. For Java applications, the Java virtual machine imposes another level of software between the program and the operating system that may alterate the runtimes of the applications under evaluation. In particular, the warm-up time is the time spent by an application to load and initialize its data structures when it is launched for the first time. It may influence the performance of the first runs of the program.

To avoid unreliable results, XCheck runs the same experiment  $n + 1$  times and takes the average and the standard deviation of the last  $n$  evaluation times. The bigger is the value of  $n$ , the less is the impact on the evaluation time of the warm-up time and of other costs that are external to the query evaluation. The value of  $n$  can be specified with a command line options of XCheck.

Besides collecting the elaboration times that the engines report, XCheck itself measures the query *total execution time*. This time is reported in *CPU time*<sup>4</sup>. The other times are engine dependent and can be CPU time as well as *wall clock time*<sup>5</sup>. It is important to remember this fact when comparing the performance of different engines, since, in principle, CPU time and wall clock time cannot be compared.

## 1.3 Query elaboration times

XCheck distinguishes between the following types of query elaboration times:

- *document processing time* is the time that an engine takes to *parse* the input XML documents and create the internal document representation (in *main memory* or *disk*). If an engine reports this time, then XCheck collects it and stores it under the name `doc_processing_time`.

---

<sup>3</sup>Perl builds and runs on a bewildering number of platforms. Virtually all known and current Unix derivatives are supported (perl's native platform), as are other systems like VMS, DOS, OS/2, Windows, QNX, BeOS, OS X, MPE/iX and the Amiga. For more information visit <http://www.perl.org>

<sup>4</sup>The *CPU time* (or *processor time*) is the amount of time the processor actually spends running a program.

<sup>5</sup>The *wall clock* time is the elapsed time between when a process starts to run and when it is finished. This is usually longer than the *CPU time* consumed by the process because the CPU is doing other things besides running the process, such as, running other user and operating system processes or waiting for disk or network I/O.

- *query compile time* is the time an engine takes to *parse* the query and translate it in an internal formalism of the engine, possibly with some *query rewriting* and *optimization* in between. This time can be empty if the engine doesn't perform any of these steps before executing the query. XCheck collects and reports this time under the name `query_compile_time`.
- *query execution time* is the time the engine takes to execute the query. This time is related only with the kernel of the execution and it includes only the time it takes to locate the query results without outputting them, if possible. Usually, this is the most interesting elaboration time for comparing different query evaluation techniques and implementations. XCheck reports this time under the name `query_exec_time`.
- *serialization/output time* is the time it takes an engine to construct and output the query result. XCheck reports this time under the name `serialization_time`.
- *total time* is the total time an engine takes to evaluate a query, starting with the engine invocation until the engine outputs valid results. This time XCheck measures itself (always in CPU time) and reports it under the name `total_time`.

All but the last times are engine dependent. This means that XCheck reports these times only if the engines provide them. Besides this, XCheck is not responsible for their accuracy, nor can it determine what unit do they use (CPU or wall clock).

#### 1.4 The XPath/XQuery engines already supported

XCheck is designed to communicate easily with all XPath/XQuery engines that have a command line interface. For each engine XCheck uses a specific adapter (a simple xml configuration file) that contains its running instructions. The version 0.1.5 of XCheck includes adapters for the following 9 engines:

Engines	Version	Type	Times	Ref.
SaxonB	8.7	XQuery	D, QC, QE, T	[5]
Galax	0.5.0	XQuery	T	[3]
MonetDB/XQuery	0.10.3	XQuery	D, QC, QE, S, T	[9]
Qizx/open	1.0	XQuery	QE, S, T	[11]
eXist	1.0	XQuery	T	[7]
Qexo	1.8.1 alpha	XQuery	T	[2]
Blixem	16 Jun 2005	LiXQuery <sup>6</sup>	T	[8]
XmlTaskForce	30 Sep 2004	XPath 1.0	T	[1]
Arb	?	CoreXPath <sup>7</sup>	D, QE, T	[6]

<sup>6</sup>Blixem is an engine to evaluate LiXQuery queries. LiXQuery, or Light XQuery, is a sublanguage of XQuery.

<sup>7</sup>CoreXPath is the navigational language of XPath.

where  $D$ =*document processing time*,  $QC$ =*query compile time*,  $QE$ =*query execution time*,  $S$ =*serialization/output time* and  $T$ =*total time*.

You can easily create a new adapter for your engine of choice by editing a simple "script" file in xml format. For more information about the creation of adapters read Chapter 5, "How to create new adapters".

## 1.5 System requirements

XCheck works with a Gnu/Linux operating system. In theory is possible to use XCheck with another operating systems but we have tested it only on different versions of Gnu/Linux.

XCheck is written in Perl and you need a Perl interpreter, ver. 5.8.5 or higher, installed on your computer in order to execute it. There are also some CPAN Perl modules you need to install (for more information read Section 2.1, "Installing XCheck").

If you want to generate the plots of the benchmark you also need the Gnuplot software ver. 4.0 or higher. There are not others requirements for the use of XCheck.

## 1.6 License use

XCheck is released under the GNU General Public License (GPL<sup>8</sup>).

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, visit the web site <http://www.gnu.org/copyleft/gpl.html> or write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.

---

<sup>8</sup>For more information visit <http://www.gnu.org/copyleft/gpl.html>

## 2 Getting started

### 2.1 Installing XCheck

This instructions are valid for the installation of XCheck on a Gnu/Linux operating system. In order to install XCheck you need first to install the following CPAN Perl modules:

- XML::Parser (ver. 2.34 or higher)
- XML::Checker (ver. 0.13 or higher)

You can obtain this modules on the CPAN web site <http://www.cpan.org>. If you are not familiar with the installation of the CPAN Perl modules read Section 2.1.1, "Installing the required CPAN Perl modules".

After the installation of the Perl modules you can download the `XCheck.tgz` file from the official web site of XCheck and expand it in a new directory with the following command:

```
$ tar -xzvf XCheck.tgz
```

After the execution of this command your directory will contain the following directories and files:

```
adapters (dir)
experiments (dir)
dtd (dir)
repository (dir)
XML (dir)
XCheck.pl
CLAdapter.pl
Utility.pm
Analysis.pm
engines.xml
README
License
```

The `adapters` directory contains the engine adapters, which are a set of XML file. `experiments` contains all the input and the output data (both for the running phase and the data analysis phase) of the experiments run on XCheck. Each experiment is contained in its own subdirectory under `experiments`. XCheck's installation package includes one experiment called `example`, thus inside the `experiments` directory you will find an `example` directory. All the input and some output files of XCheck are in XML format. `dtd` directory contains the dtds for these files. XCheck is using them to validate its input. The `repository` directory contains two subdirectories: `docs` and `queries`. There you should store

all the XML documents and (optionally) the queries to be used in the experiments. And finally, the XML directory contains some modules used by XCheck.

Next on the list comes `XCheck.pl`, the main part of the software. `CLAdapter.pl` is the script that processes the adapters for the command line engines and runs them. `Utility.pm` and `Analysis.pm` are modules used by `XCheck.pl`. `engines.xml` is a configuration file containing information about the engines. This is the only file you have to edit for running XCheck with the supported engines. `README` is a text file containing some brief information about XCheck's installation and usage. `License` is a text file containing the user license of XCheck.

If all has gone well so far, you now can start using XCheck. You can run XCheck with the following command:

```
$ ./XCheck.pl
```

You should see an help message like this:

```
XCheck ver. 0.1.5
A benchmark checker for XML query processors.

Usage: ./XCheck.pl [options]
Options: -run folder  execute the running phase with the
                    experiment.xml saved in folder
        -update       update the results of the experiment folder
        -n num         number of executions for each query
                    (num=3 is default)
        -s            store the query results
        -p            generate the plots of the results for
                    the running and analysis phase
        -data folder  execute the data analysis phase with
                    the analysis.xml saved in folder
        -help         print this help
```

Congratulations, you have successfully installed XCheck on your computer. Now you can configure XCheck in order to execute an experiment with the XML engines installed on your computer. Go to Section 2.2, "Configure XCheck".

### 2.1.1 Installing the required CPAN Perl modules

If you have *root* privileges on your machine you can install the CPAN modules using the following command:

```
$ perl -MCPAN -e 'install Module::Name'
```

where you need to replace `Module::Name` first with `XML::Parser` and after with `XML::Checker`.

If you don't have *root* privileges on your machine you need to install the CPAN modules locally in a private/non-standard directory. In order to do this you need to download the `XML::Parser` and the `XML::Checker` modules from CPAN's web site and expand these files in new directories with the following commands:

```
$ tar -xzf XML-Parser-2.34.tar.gz
$ tar -xzf XML-Checker-0.13.tar.gz
```

After this you can install each modules with the following commands:

```
$ cd XML-Parser-2.34
$ perl Makefile.PL LIB=/mylib PREFIX=/mylib
$ make
$ make test
$ make install
```

where `/mylib` is the path of the directory where you want to install the module. You need to repeat this sequence of commands for the `XML-Checker-0.13` directory.

At the end of these operations you need to modify the *Perl library directory* with the new path `/mylib`. In order to do this you must modify the `PERL5LIB` environment variable with the following commands.

In the *bash* shell:

```
$ PERL5LIB=/mylib/lib/perl/x.y.z:/mylib/share/perl/x.y.z
$ export PERL5LIB
```

In the *tcsh* shell:

```
$ setenv PERL5LIB /mylib/lib/perl/x.y.z:/mylib/share/perl/x.y.z
```

where `x.y.z` is the version<sup>9</sup> of your Perl interpreter (for instance 5.8.5).

For more information on the installation of CPAN modules read the CPAN Frequently Asked Questions (<http://www.cpan.org/misc/cpan-faq.html>).

## 2.2 Configure XCheck

In this section we show how to configure XCheck to enable it to communicate with the XPath/XQuery engines installed on your computer. The configuration of XCheck is very easy, you need to edit only a single xml file, `engines.xml` that is in XCheck's home directory.

This file has a structure like this:

---

<sup>9</sup>If you don't know the version of your Perl use this command: `perl -V`

```

<?xml version="1.0"?>
<!DOCTYPE engines SYSTEM "dtd/engines.dtd">
<engines>
  <engine id="SaxonB" type="xquery">
    <name>SaxonB</name>
    <version>8.6.1</version>
    <homepage>http://saxon.sourceforge.net/</homepage>
    <description>The XSLT and XQuery Processor...</description>
    <adapter>CLAdapter.pl -e saxon.xml</adapter>
    <path></path>
    <cpu_time>n</cpu_time>
  </engine>
  ...
</engines>

```

For each engine you need to write inside the `<path>` element the full path of your local engine installation. For instance, if you have installed SaxonB in the `/home/user/SaxonB` directory you will insert this path as value of the `<path>` element:

```
<path>/home/user/SaxonB</path>
```

If you don't have installed on your computer some of the engines listed in the `engines.xml` file you can leave empty their `<path>` elements.

## 2.3 Test the configuration

After the configuration phase you can test the XCheck functionality by executing an example experiment that comes with the installation package. But in order to run the example you need to have installed at least one of the XPath/XQuery engines listed on page 5.

The standard example accompanying XCheck uses the engine SaxonB. If you have this engine installed you can pursue to execute the test, otherwise you need to change the input file of the example.

In order to execute the example with a different engine you need to edit the file `experiment.xml` found in the directory `experiments/example/` and modify the element `<engine>` to contain the `id` of one of the engines present in the configuration file `engines.xml`. For instance if you want to use the engine Galax you need to modify the `experiment.xml` in this way:

```

<?xml version="1.0"?>
<!DOCTYPE benchmark SYSTEM "dtd/experiment.dtd">
<experiment>
  <name>XCheck example</name>
  <description>Example</description>
  <engines>
    <engine>Galax</engine>
  </engines>
</experiment>

```

```
</engines>
<documents>
  <document id="d0.001">
    <description>XMark document f=0.001</description>
    <file>0.001d.xml</file>
  </document>
</documents>
...
```

Now we can test the execution of XCheck with an example. Go to the home directory of XCheck and execute the following command:

```
$ ./XCheck.pl -run example
```

You should see an output like this:

```
XCheck ver. 0.1.5 (running phase)
Experiment: example.
Checking the input: xml documents... (ok)
Checking the input: queries... (ok)
Checking the input: engines... (ok)
The benchmark runs each experiment 4 times and takes
the average of the last 3
Start: Thu May 11 09:53:56 2006

Engine: SaxonB
Document: d0.001
Query: q1(ok) q2(ok) q3(ok)

Generating the html output...
Finish: Thu May 11 09:54:14 2006
Execution time of the experiment 18 (ss)
The results are stored in "experiments/example/output" directory
```

If you had some problems in the execution of this example you should read the FAQ page on XCheck's web site.

Otherwise if the execution of the example was correctly done you will find the results in the `experiments/example/output` directory. In this directory you will find two files, named `outcome.xml` and `outcome.html` containing the execution times and other information, like the query results sizes, the technical details of the computer and the operating system used etc.

## 3 Running phase

### 3.1 Build an experiment

In this chapter we will learn how to create and execute an experiment with the use of XCheck. An *experiment* consists of running a set of XML engines on a set of queries and a set of input XML documents. The user can specify the engines, the queries and the documents that participate in the experiment in an xml file, named `experiment.xml`. XCheck executes the experiment reading this xml file and running all the queries on all the documents and on all the engines in the document order of this file.

All the experiments are stored in a common directory named `experiments/`. Each experiment has its own subdirectory having a suggestive name. In order to create a new experiment you need to create a new directory under `experiments/` and save there your `experiment.xml`. Here is the input file `experiment.xml` for the accompanying example:

```
<?xml version="1.0"?>
<!DOCTYPE benchmark SYSTEM "dtd/experiment.dtd">
<experiment>
  <name>XCheck example</name>
  <description>Example</description>
  <engines>
    <engine>SaxonB</engine>
    <engine>MonetDB</engine>
  </engines>
  <documents>
    <document id="d0.001">
      <description>XMark document f=0.001</description>
      <file>0.001d.xml</file>
    </document>
    <document id="d0.002">
      <description>XMark document f=0.002</description>
      <file>0.002d.xml</file>
    </document>
  </documents>
  <queries>
    <query id="q1">
      <description>All the keywords</description>
      <syntax engine="all"><![CDATA[doc()//keyword]]></syntax>
    </query>
    <query id="q2">
      <description>The keywords in a paragraph item</description>
      <syntax engine="all"><![CDATA[doc()//item/name]]></syntax>
    </query>
  </queries>
</experiment>
```

An experiment is specified by the following elements: **name**, containing the name of the experiment; **description**, containing the description of the experiment; **engines**, containing a list of engines; **documents**, a list of XML documents; and **queries**, a list of queries.

Note that, the name of the experiment might differ from the name of its folder, e.g. "XCheck example" sits in **example** directory. When running an experiment the name of its directory is used to identify it.

You can specify the engines using their **id** attributes as reported the **engines.xml** configuration file (for instance in the reported example we use the engines **SaxonB** and **MonetDB**).

The XML documents are described by the **document** elements. For each document you must specify the **id** attribute, the **description** of the document (optional) and the file name of the document in **file** (all the xml documents must be stored in the **repository/docs** directory of XCheck). Moreover, if your document is syntactically generated you can specify how to run the document generator in order to obtain it. For instance, if you use an XMark [10] document you can use the generator element as reported below:

```
...
<document id="d0.032">
  <description>XMark document f=0.032</description>
  <file>0.032d.xml</file>
  <generator>
    <![CDATA[/mypath/xmlgen.Linux -f 0.032 -o #file]]>
  </generator>
</document>
...
```

where **/mypath** is the directory where you have installed the **xmlgen.Linux** generator. Note the use of CDATA section, **<![CDATA[ ... ]]>**, to escape any xml sensitive characters. This means everything inside a CDATA section is ignored by the XML parser.

In the **generator** element there is a special word named **#file**, this word is an XCheck *variable* that contains the file name of the document, in this case **0.032d.xml**. In other words, XCheck replaces the string **#file** with the name of the document specified in the element **file**.

The last element of the **experiment.xml** file is the element **queries**. With this element you can specify the XPath/XQuery (or other dialects) queries to be used for the experiment. For each query you must specify at least the **id** attribute and the syntax of the query or the name of the file containing the query.

If you want to insert the syntax of the query in the **experiment.xml** file you need to use the element **syntax** as shown below:

```
...
<query id="q1">
  <description>All the keywords</description>
```

```

    <syntax engine="all"><![CDATA[doc()//keyword]]></syntax>
  </query>
  ...

```

Here again you can make use of the CDATA section to escape the xml sensitive characters. You can use the attribute `engine` to specify different syntax for different engines<sup>10</sup>. In the `engine` attribute you can specify a list of engines separated by comma or the reserved word `all`. In our example we have used the string `all`, which means for every engine. You may also specify different query syntaxes, for instance, for `SaxonB` and `MonetDB`. In this case you should use two syntax elements:

```

  ...
  <query id="q1">
    <description>All the keywords</description>
    <syntax engine="SaxonB"><![CDATA[doc()/query1]]></syntax>
    <syntax engine="MonetDB"><![CDATA[doc()/query2]]></syntax>
  </query>
  ...

```

In this case `/query1` and `/query2` are syntactic variations of the same query or, in other words, they are semantically equivalent queries.

Note the use of the `doc()` string in the previous queries. This is another reserved word of XCheck and it means that for each query execution the empty function call `doc()` is replaced with the `doc("file.xml")` function call where `file.xml` is one of the xml documents specified in the `documents` element. In other words the `doc()` string is used to iterate the execution of the query on all the xml documents. You can combined the use of `doc()` and `doc("file.xml")` with a fixed `file.xml`, for instance, when a query queries two documents at once and you want to iterate first one while leaving the second one fixed.

If you are interested in the *query scalability* in your experiment you can also specify the length of the query using the attribute `length` of the `query` element, for instance:

```

  ...
  <query id="q1" length="2">
    <description>...</description>
    <syntax engine="all"><![CDATA[doc()//a/b]]></syntax>
  </query>
  ...

```

If you have very long queries the readability of `experiment.xml` could decrease. In this case you can split every query in a separated text file and use the file names in `filequery` element, instead of `syntax`. All the query files must be stored in the `repository/queries` directory of XCheck.

<sup>10</sup>This feature is useful in the cases of engines that have implemented some syntactic deviations from the W3C standards.

```

...
<query id="q1">
  <description>Return the name of all.</description>
  <filequery engine="all">q1.qxl</filequery>
</query>
...

```

As a last option of the query element, you can specify how to run a *query generator* to obtain your query, in case you have a query generator. This works as in the case of XML document generators, using the attribute `generator` of the `filequery` element. For instance:

```

...
<query id="q1">
  <description>Return the name of all.</description>
  <filequery engine="all" generator="/mygen #file">
    q1.qxl
  </filequery>
</query>
...

```

where `/mygen` is the command to generate the file `q1.qxl`. Note the use of the `#file` string, it has the same meaning as with the document generator.

### 3.2 Run an experiment

You can execute an experiment in the *running phase* of XCheck with the following command:

```
$ ./XCheck.pl -run exp1
```

where `exp1` is the name of the directory, under the directory `experiments`, where XCheck looks for the experiment specification file `experiment.xml`. XCheck reads the `experiment.xml` file and executes each query on each document and each engine in the same order as specified in the file. In other words XCheck executes the following iterations:

```

for $engine in ENGINES
  for $document in DOCUMENTS
    for $query in QUERIES
      execute($engine, $query, $document)

```

where `ENGINES`, `DOCUMENTS` and `QUERIES` are the element sequences `//engines/engine`, `//documents/document` and `//queries/query` in the document order, specified in `experiment.xml`.

During the running phase XCheck prints on the standard output some informations about the execution of the experiment. For each query XCheck uses the string `(ok)` to indicate an execution without error and the string `(!?)` to indicate

error. At the end of execution XCheck produces two files, named `outcome.xml` and `outcome.html`.

These files are stored in the directory `/experiments/exp1/output/` of XCheck.

As default option XCheck executes each query 4 times and takes the average and the standard deviation of the last 3 runs. If all four executions of a query produce errors XCheck stores the error and goes to the next query. If at least one execution gives results XCheck stores them. You can change the number of executions with the use of the option `-n num`. For instance, you can execute the queries one time (+1)<sup>11</sup> with the following command:

```
$ ./XCheck.pl -run exp1 -n 1
```

The use of the `-p` option tells XCheck to generate diverse plots presenting the experiment execution times. For instance:

```
$ ./XCheck.pl -run exp1 -p
```

After the execution of the experiment `exp1` XCheck generates several types of plots and places them in subdirectories of `/experiments/exp1/output/`. `img` subdirectory contains the plots in PNG format and `plots` contains the PostScript encoding of the same plots. To make it easy for the user to edit the plots, XCheck stores the Gnuplot codes used to generate them in `gnuplot` subdirectory. Moreover, XCheck produces several plot galleries in HTML, one for each plot type:

- `plots_queries.html`
- `plots_docs.html`
- `plots_3d.html`
- `plots_engines_queries.html`
- `plots_engines_docs.html`

This improves greatly the readability of the experiment performance results. These files are also linked in the main report file `outcome.html`, so that after the execution of the running phase the user is able to see all the files generated by XCheck viewing only `outcome.html` with an HTML browser (for instance Mozilla Firefox). All the HTML files generated by XCheck are in standard W3C HTML 4.01<sup>12</sup>.

XCheck produces five types of plots. Figure 1 contains plots of each type for our running example. **[Todo:] All types of plots should be in the figure. Each type should be described.**

---

<sup>11</sup>Every query is executed `num+1` times. One execution is always present to reduce the warm-up time.

<sup>12</sup>For more information on this standard visit <http://www.w3c.org>

Figure 1: Example of XCheck plots.

By default, XCheck doesn't store the query results for the reason of saving disk space. The user can see only the size of the results in bytes in `outcome.html`<sup>13</sup>. If you want to store the results of the queries you need to use the option `-s` of XCheck, for instance:

```
$ ./XCheck.pl -run exp1 -s
```

All the query results are stored in the directory `experiments/exp1/output/results`. The result files are also linked in `outcome.html`, in the last table of the report page.

Of course, you can use all the previous options of the running phase together, for instance:

```
$ ./XCheck.pl -run exp1 -n 2 -p -s
```

executes the experiment `exp1` with two+1 (`-n 2`) executions for each query, generating the plots (`-p`) and storing the query results (`-s`).

### 3.3 The update option

After the execution of an experiment if you want to insert new xml documents, queries or engines into the experiment without repeating the whole experiment from the beginning, you can use the `update` option. With this option XCheck keeps the old results of the experiment and executes only the new input. In this way the you don't lose time with the re-execution of the old input. For instance, if you have a big experiment with slow query execution times you can use the update option to run the experiment step by step (query by query or otherwise) and check the intermediate results. In order to execute the `update` option you must use the following command:

```
$ ./XCheck.pl -run exp1 -update
```

where `exp1` is the name of the experiment.

XCheck checks the old results of the experiment `exp1` and reads the information about the old environment used to run the experiment (CPU, amount of RAM, number of executions, ecc.). If some information about the old environment is different from the new one, or if there aren't information about the old environment, XCheck gives you a warning and asks if you want to continue anyway. Example of warning message:

```
Warning: the CPU of this computer is different from the CPU used
in the old experiment, do you want to continue anyway? (y/n)
```

---

<sup>13</sup>At the end of the `outcome.html` file is reported a table with all the sizes of the results of the queries. In this table is present a pseudo correctness test with the use of the sizes of the query results. The cells in red are the "suspicious" results. Note that this test is just indicative and should not be taken as an absolute correctness test

This is done to ensure that comparing the running times of the old and the new part of the experiment makes sense.

Another way to use the `update` option is to generate or re-generate the plots after the execution of a running phase. For instance if you have executed the running phase without the `-p` option you can use the `update` option to generate the plots for the old results of the experiment using the two options together. For instance:

```
$ ./XCheck.pl -run exp1 -update -p
```

generates the plots of the `exp1` experiment.

## 4 Data analysis phase

After the execution of an experiment XCheck can help the user to analyze the results with the use of some *aggregation measures*, *statistics* and *plots*. The input of the data analysis phase is the output of the running phase, the file `outcome.xml` to be precise.

### 4.1 Aggregation measures, statistics and plots

In the following, we define a set of *aggregation measures* used by XCheck. Let

$$t_{A,B}^k = \{t_{i,j}^k \mid i \in A, j \in B\}$$

We define:

- $\text{sum}_{A,B}^k$  as the sum of the set  $t_{A,B}^k$ ,
- $\text{avrgt}_{A,B}^k$  as the mean<sup>14</sup> of the set  $t_{A,B}^k$ ,
- $\text{medt}_{A,B}^k$  as the median<sup>15</sup> of the set  $t_{A,B}^k$ ,
- $\text{mint}_{A,B}^k$  as the minimum of the set  $t_{A,B}^k$ ,
- $\text{maxt}_{A,B}^k$  as the maximum of the set  $t_{A,B}^k$ ,
- $\text{stdevt}_{A,B}^k$  as the standard deviation<sup>16</sup> of the set  $t_{A,B}^k$ .

Given a document index  $i$  and a query index  $j$ , let us denote  $\text{size}_i$  as the size of document  $i$  and  $\text{length}_j$  as the length of query  $j$ , where the size of a document and the length of a query are defined by the user. We define the *medley relay data speed* of  $E_k$  on the document set  $A$  and the query set  $B$ , denoted by  $\text{mrds}_{A,B}^k$ , as follows:

$$\text{mrds}_{A,B}^k = \frac{|B| \cdot \sum_{i \in A} \text{size}_i}{\sum_{i \in A, j \in B} t_{i,j}^k}$$

We define the *medley relay query speed* of  $E_k$  on the document set  $A$  and the query set  $B$ , denoted by  $\text{mrqs}_{A,B}^k$ , as follows:

$$\text{mrqs}_{A,B}^k = \frac{|A| \cdot \sum_{i \in B} \text{length}_i}{\sum_{i \in A, j \in B} t_{i,j}^k}$$

---

<sup>14</sup>For  $n$  measurements  $x_1, x_2, \dots, x_n$ , the mean is defined as  $\frac{\sum_{i=1}^n x_i}{n}$

<sup>15</sup>For  $n$  measurements  $x_1, x_2, \dots, x_n$ , the median is defined as element at position  $n + 1/2$  if  $n$  is odd, or  $n/2$  if  $n$  is even, in the sorted array containing the measurements.

<sup>16</sup>For  $n$  measurements  $x_1, x_2, \dots, x_n$ , the standard deviation is defined as  $\sqrt{\frac{\sum_{i=1}^n (x_i - \mu)^2}{n-1}}$ , where  $\mu$  is the mean.

Whenever data scalability is a benchmark target, the following definition of data scalability factor becomes relevant. Let  $I = (i_1, i_2, \dots, i_k)$ , for  $k \geq 2$ , be a sequence of indexes of documents of increasing sizes. We define the *data scalability factor* of  $E_k$  on the document sequence  $I$  and the query set  $B$ , denoted by  $ds_{I,B}^k$ , as follows. If  $k = 2$ , then

$$ds_{(i_1, i_2), B}^k = \frac{\text{mrds}_{i_1, B}^k}{\text{mrds}_{i_2, B}^k}$$

If  $k > 2$ , then

$$ds_{I, B}^k = \frac{\sum_{j=1}^{k-1} ds_{(i_j, i_{j+1}), B}^k}{k-1}$$

Similarly, when query scalability is a benchmark target, the following definition of query scalability factor becomes relevant. Let  $J = (j_1, j_2, \dots, j_k)$ , for  $k \geq 2$ , be a sequence of indexes of queries of increasing lengths. We define the *query scalability factor* of  $E_k$  on the document set  $A$  and the query sequence  $J$ , denoted by  $qs_{A, J}^k$ , as follows. If  $k = 2$ , then

$$qs_{A, (j_1, j_2)}^k = \frac{\text{mrqs}_{A, j_1}^k}{\text{mrqs}_{A, j_2}^k}$$

If  $k > 2$ , then

$$qs_{A, J}^k = \frac{\sum_{i=1}^{k-1} ds_{A, (j_i, j_{i+1})}^k}{k-1}$$

Notice that, by virtue of the definition of medley relay speed, a scalability factor (either for data or query) less than 1 (respectively, equal to 1, bigger than 1) corresponds to a sub-linear (respectively, linear, super-linear) increase of the total evaluation time when moving from one instance of the problem to a bigger one.

Moreover, XCheck uses the following *similarity index* to compare the similarity of the behaviour of two engines. Let  $X = (x_1, x_2, \dots, x_n)$  be a sequence containing the results on  $n$  experiments. For each  $1 \leq i \leq n-1$ , let  $W_X^i = a_1 a_2 \dots a_{n-i}$  where, for each  $1 \leq j \leq n-i$ , we have that  $a_j = U$  if  $x_i < x_{i+j}$ ,  $a_j = D$  if  $x_i > x_{i+j}$ , and  $a_j = E$  if  $x_i = x_{i+j}$ . Let  $W_X = W_X^1 \circ W_X^2 \dots \circ W_X^{n-1}$  be the *similarity word* associated to  $X$ , where  $\circ$  is the concatenation operation. Notice that the length of  $W_X^i$  is  $n-i$  and hence the length of  $W_X$  is  $(n-1) \cdot n/2$ . Given two sequences  $X = (x_1, x_2, \dots, x_n)$  and  $Y = (y_1, y_2, \dots, y_n)$ , the *distance* between  $X$  and  $Y$  is defined as the word distance between the words  $W_X$  and  $W_Y$ , where the *word distance* between two words of the same length is the number positions of the two words that contain a different character. For instance, the word distance between UUED and EUEU is 2 (the characters in both the first and last position differ in the two words).

In the following, we describe the possible *statistics indices* that the user can ask to XCheck. Let *index* be an aggregation measure among those defined above,  $K$  be a sequence of engines,  $A$  be a sequence of documents, and  $B$  be a sequence of queries:

- S1** The user may ask to compare the engines in  $K$  according to the aggregation measure *index* computed on the document sequence  $A$  and on the query sequence  $B$ .
- S2** For each document  $d$  in the document sequence  $A$ , the user may ask to compare the engines in  $K$  according to the aggregation measure *index* computed on  $d$  and on the query sequence  $B$ . Two kinds of 2D plot are produced: the first puts the documents in  $A$  on the  $x$ -axis, the second puts the sizes of the documents in  $A$  on the  $x$ -axis. The second plot might be useful when the document sequence contains documents of increasing sizes.
- S3** For each query  $q$  in the query sequence  $B$ , the user may ask to compare the engines in  $K$  according to the aggregation measure *index* computed on  $q$  and on the document sequence  $A$ . Two kinds of 2D plot are produced: the first puts the queries in  $B$  on the  $x$ -axis, the second puts the lengths of the queries in  $B$  on the  $x$ -axis. The second plot might be useful when the query sequence contains queries of increasing lengths.
- S4** For each document  $d$  in the document sequence  $A$  and each query  $q$  in the query sequence  $B$ , the user may ask to compare the engines in  $K$  according to the aggregation measure *index* computed on  $d$  and on  $q$ . XCheck produces a 3D plot of this statistic for each engine and type of elaboration time.
- S5** For each document  $d$  in the document sequence  $A$  and each pair of engines  $i$  and  $j$  in  $K$ , let  $X$  (resp.  $Y$ ) be the sequence  $(t_{d,k}^i)_{k \in B}$  (resp.  $(t_{d,k}^j)_{k \in B}$ ). The user may ask to compute the distance between the two sequences  $X$  and  $Y$ .
- S6** For each query  $q$  in the query sequence  $B$  and each pair of engines  $i$  and  $j$  in  $K$ , let  $X$  (resp.  $Y$ ) be the sequence  $(t_{k,q}^i)_{k \in A}$  (resp.  $(t_{k,q}^j)_{k \in A}$ ). The user may ask to compute the distance between the two sequences  $X$  and  $Y$ .

In the data analysis phase the user can also generate different plots of the elaboration times with different aggregations of the data.

The user can choose to generate these plots:

- P1** Plots of times for each document.
- P2** Plots of times for each query.
- P3** Plots of times for each engine and document.
- P4** Plots of times for each engine and query.

## 4.2 Build a data analysis input

In order to execute the *data analysis phase* the user must specify the *engines*, the *documents*, the *queries*, the type of *elaboration times* and the *aggregation measures* to be used in the elaboration. Moreover the user can specify the type of statistics (S1,...,S6) and plots (P1,...,P4) that should be generated.

All these informations are reported in the `analysis.xml` file stored in the directory of the experiment. This is an example of `analysis.xml` file:

```
<?xml version="1.0"?>
<!DOCTYPE analysis SYSTEM "dtd/analysis.dtd">
<analysis>
  <input>
    <engines>Galax, Qizx, MonetDB, SaxonB</engines>
    <queries>q1,q3,q5,q7,q9</queries>
    <documents>d0.016, d0.032, d0.064</documents>
    <indices>
      sum, mean, median, min, max, stdev, mrdspeed,
      mrqspeed, dsfactor, qsfactor
    </indices>
    <times>
      doc_processing_time, query_compile_time,
      query_exec_time, serialization_time, total_time
    </times>
  </input>
  <output>
    <index>s1, s2, s3, s4, s5, s6</index>
    <plots>p1, p2, p3, p4</plots>
  </output>
</analysis>
```

In this files are reported two main elements, the `input` and the `output`. In the element `input` are reported all the `engines`, `queries`, `documents`, `indices` and `times` to be used in the elaboration of the analysis.

For the engines, documents and queries we use the *id* attribute reported in the file `outcome.xml`, containing the results of the *running phase*. For instance in the previous `analysis.xml` file we select the engines , the queries `q1,q3,q5,q7,q9` and the documents `d0.016, d0.032, d0.064`.

With this option the user can select a different aggregation of the data and generate specific analysis for a subset of the experiment.

## 4.3 Execute the data analysis phase

The user can execute the *data analysis phase* with the use of the option `-data` of XCheck. For instance if you want to execute the data analysis phase on the results of the `exp1` experiment you need to execute the following command:

```
$ ./XCheck.pl -data exp1
```

XCheck reads the `analysis.xml` file stored in the `experiments/exp1` directory and execute the *data analysis phase*. The results of the data analysis phase are reported in the file `outcome.analysis.html` under the directory `experiments/exp1/output`. This file contains all the links to the other HTML files with the statistics and plots.

XCheck will generate the following files, under the directory `output`, for each statistics:

**S1** `outcome_s1.xml` and `outcome_s1.html`

**S2** `outcome_s2.xml` and `outcome_s2.html`

**S3** `outcome_s3.xml` and `outcome_s3.html`

**S4** `outcome_s4.html`

**S5** `outcome_s5.xml` and `outcome_s5.html`

**S6** `outcome_s6.xml` and `outcome_s6.html`

and these files, under the directory `output`, for each plots:

**P1** `DA_plots_queries.html`

**P2** `DA_plots_docs.html`

**P3** `DA_plots_engines_docs.html`

**P4** `DA_plots_engines_queries.html`

The plots **P1**,...,**P4** are generated in the directory `img`, `gnuplot` and `plots`. XCheck uses the prefix `DA_` for the files generated in the *data analysis phase*, in this way these files aren't confused with the other plots generated in the *running phase*.

In Figure 2 you can see an example of **S1** statistic (`outcome_s1.html`) and in Figure 3 you can see an example of **P1** plots (`DA_plots_queries.html`).

By default XCheck doesn't generate the plots related to the statistics **S1**, **S2** and **S3**. If you want to generate these plots you need to specify the option `-p` in the command line of XCheck. For instance, using the previous command:

```
$ ./XCheck.pl -data exp1 -p
```

With this command the statistics **S1**, **S2** and **S3** will contain the plots reported in each column of each tables (in Figure 4 and 5 are reported examples of statistic **S1** and **S2** with plots).

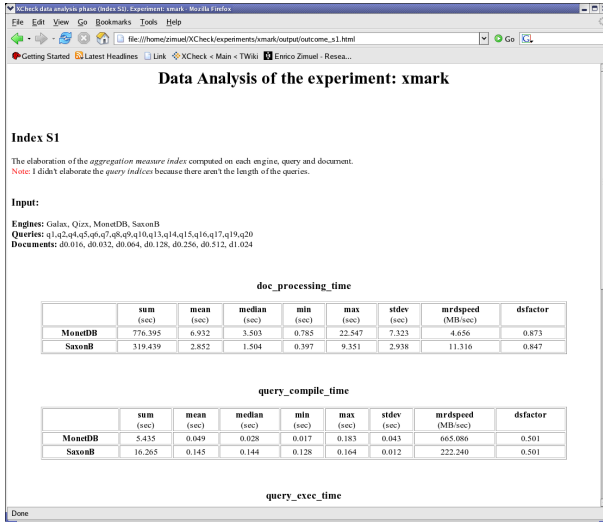


Figure 2: Example of S1 statistic

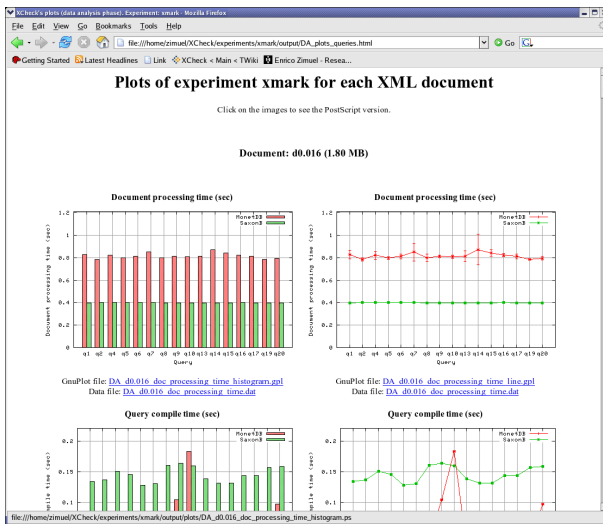


Figure 3: Example of P1 plots

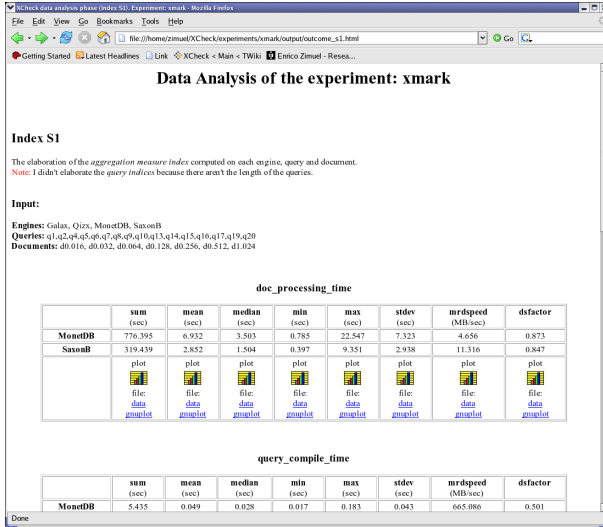


Figure 4: Example of statistic S1 with plots

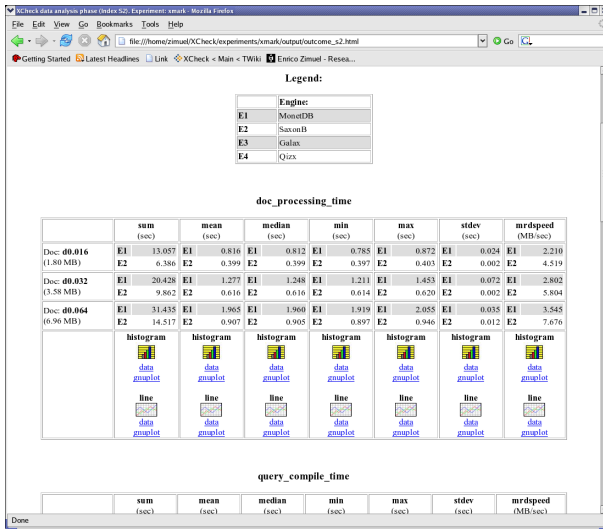


Figure 5: Example of statistic S2 with plots

## 5 How to create new adapters

In this chapter you will learn how to create adapters for new XML engines that have a command line interface. XCheck uses a simple script like XML file to communicate with the engines. This makes it easy for anybody to connect a new engine to XCheck.

### 5.1 Create a new adapters

Most of the XML engines have a command line interface to execute the queries and obtain results<sup>17</sup>. An engine adapter is an XML file containing the engine's running instructions and a formal description of its output (the query results, error messages and the elaboration times). XCheck interprets this XML file through a Perl program, named `CLAdapter.pl` and runs the engine by using the given instructions on the right input, catches and stores the output for each step of an experiment.

Imagine that you have a new XML engine named, for instance, `SuperEngine` and you want to connect it to XCheck. In order to do so you need to complete these two steps:

1. create an adapter, named for instance `superengine.xml`, and store it in the `adapters` directory of XCheck;
2. insert a new `engine` element describing the engine and its adapter in the `engines.xml` file in the main directory of XCheck.

Below is reported an example of an adapter for `SuperEngine`.

```
<?xml version="1.0"?>
<!DOCTYPE adapter SYSTEM "dtd/adapter.dtd">
<adapter>
  <engine id="SuperEngine">
    <command>
      <executable><![CDATA[
        #path/SuperEngine -q #query -o #result >& #times
      ]]></executable>
      <file_query>y</file_query>
      <fullpath_doc>y</fullpath_doc>
    </command>
  </engine>
</adapter>
```

Every adapter has the element `command` (form more information about the structure of the adapter read Appendix ??). This element specifies how to execute the engine on given input. The `executable` element contains the command to execute the engine.

---

<sup>17</sup>If an XML engine doesn't have a command line interface it should be always possible to write a simple command line interface for it.

In this example we have assumed that the running command of SuperEngine has a syntax like this:

```
$ SuperEngine -q query -o output
```

where `-q query` is the option for the file query and `-o output` is the option for the output. Moreover we assume that `SuperEngine` outputs the information about the elaboration times on the *standard output* or *standard error*. In the `executable` there are some reserved words of XCheck, they start with a hash sign `#`. These reserved words are replaced by XCheck with the proper values when the experiment it being executed. For instance, `#query` will contain the name of a query file described in the experiment that is being run. Other reserved words are: `#path` – the full path of the XML engine; `#result` – the file name of the query result; `#times` the file name of the elaboration times.

**[Todo:]** [The redirect differs for different shells.](#)

There are other two more children of `command` element: `file_query` and `fullpath_doc`. The first, `file_query`, specifies if the engine asks for the query to be saved in a separate file or served at the command line as a string parameter.<sup>18</sup> The second, `fullpath_doc`, specifies the full path or only the name of the xml documents to be used by the engine<sup>19</sup>.

With this adapter XCheck is able to execute the engine, but not to collect also the elaboration times that it possible outputs. In fact, we did not specify how to use the elaboration times generated by the engine and how to manipulate the error messages of the engine.

If you know in which format the engine outputs the elaboration times you can configure your adapter to recognize them, by adding the element `times` to the previous example.

```
<?xml version="1.0"?>
<!DOCTYPE adapter SYSTEM "dtd/adapter.dtd">
<adapter>
  <engine id="SuperEngine">
    <command>
      <executable><![CDATA[
        #path/SuperEngine -q #query -o #result >& #times
      ]]></executable>
      <file_query>y</file_query>
      <fullpath_doc>y</fullpath_doc>
    </command>
    <times>
```

---

<sup>18</sup>This option is useful to make the distinction between the two cases.

<sup>19</sup>This option is used with the XML databases (e.g. MonetDB/XQuery, eXist, etc.) which have the documents stored by their name and not the file paths.

```

    <factor_time>0.001</factor_time>
    <time id="t1">
      <line>Compilation time: (\d+) milliseconds</line>
    </time>
    <time id="t2">
      <line>Tree built in (\d+) milliseconds</line>
    </time>
    <time id="t3">
      <line>Execution time: (\d+) milliseconds</line>
    </time>
    <doc_processing_time>#t2</doc_processing_time>
    <query_compile_time>#t1</query_compile_time>
    <query_exec_time>#t3-#t2</query_exec_time>
  </times>
  <error>Error on|Fatal error</error>
</engine>
</adapter>

```

In this example the `times` element contains the following elements: `factor_time`, the multiplicative factor to transform the times given by the engine in seconds (for instance here is 0.001 that means the times given by the engine are in milliseconds); a set of `time` elements, to specify the string to be parsed from the output of the engine and finally the assignment of these values to the elaboration times `doc_processing_time`, `query_compile_time` and `query_exec_time`.

The element `time` is defined with an `id` attribute and a set of `line` elements in document order<sup>20</sup>. For instance the `t1` time contains the string "Compilation time: (\d+) milliseconds". This is the Perl *regular expression* that describes the output string of the engine for the query compile time. The pattern `(\d+)` means "any natural number" (for more information on the regular expression read Section ??, *Regular expressions for times and errors*).

The value of the time elements are assigned to the elaboration times using the following xml elements: `doc_processing_time`, `query_compile_time`, `query_exec_time`, and `serialization_time`. As you can see in the previous example the user can specify an arithmetic expression of the values of times, for instance the *query execution time* (`query_exec_time`) is the difference between `#t3` and `#t2` (`#t3 - #t2`).

Finally we have the element `error` that specifies how to intercept the error messages of the engine. For instance in the previous example is specified the string "Error on|Fatal error", which means that the engine's error message should contain at the sentences "Error on" or "Fatal error" (the regular expression `|` is equivalent to *boolean or*) in order to be recorder by XCheck as an error.

<sup>20</sup>You can specify more `line` elements and the XCheck will parse them in document order

[**Todo:**] [How do you make a difference between a legitimate error message between an engine crash?](#)

Now, in order to complete the installation of this new adapter you need to edit the `engines.xml` file of XCheck and add the informations about this new engine. For instance you need to insert the following informations:

```
<engine id="SuperEngine" type="xquery">
  <name>SuperEngine</name>
  <version>1.0</version>
  <homepage>http://www.superengine.org/</homepage>
  <description>The best XQuery engine in the world</description>
  <adapter>CLAdapter.pl -e superengine.xml</adapter>
  <path>/mypath/SuperEngine</path>
  <cpu_time>n</cpu_time>
</engine>
```

Each engine in the `engines.xml` file has `id` and `type` attributes. The `type` attribute indicates whether the engine implements XPath or XQuery (we suppose that `SuperEngine` is an XQuery engine). `name`, `version`, `homepage`, `description` are self explanatory. The `adapter` element contains the command for interpreting the engine's adapter `superengine.xml`. `path` contains the path of the engine and `cpu_time` indicates whether the engine outputs the detailed elaboration times in CPU seconds or wall clock seconds. In this example the engine outputs *wall clock time*, so `cpu_time` contains the letter `n` from "no". <sup>21</sup>.

## 5.2 The before and after elements

The communication with the XML engines can be very different from one engine to another. To be as general as possible we have inserted the possibility to execute commands before and after the execution of a query. For instance in an XML database scenario the user executes an experiment with XCheck. While executing one query the database server crashes and in order for XCheck to continue with the experiment, it has to restart the data base server. One solution for this problem is to indicate in the adapter that for each query execution, the database server is first started-up, the query is executed and after the database server is shut down.

Below is reported an example of usage of `before` and `after` elements in the adapter file:

```
...
<command>
  <before><![CDATA[
    #path/startup_DB
  ]]></before>
```

---

<sup>21</sup>For more information about the *wall clock times* and *CPU times* read the Section 5.3, "How to obtain reliable program runtimes" at pag. 4

```

<executable><![CDATA[
#path/execute -q #query -o #result >& #times
]]></executable>
<after><![CDATA[
#path/shutdown_DB
]]></after>
<file_query>y</file_query>
<fullpath_doc>y</fullpath_doc>
</command>
...

```

These elements `before` and `after` can contain the reserved word `#path`, `#query`, `#doc`, `#result`, `#times` as showed in the `executable` element.

### 5.3 Regular expressions for times and errors

We have showed a very simple example about the usage of the regular expressions for the match pattern of the output of the engine. In this section we will show another useful examples about the Perl regular expression<sup>22</sup>.

What is a regular expression? A regular expression is simply a string that describes a pattern. Patterns are in common use these days; examples are the patterns typed into a search engine to find web pages and the patterns used to list files in a directory, e.g., `ls *.txt` or `dir *.*`. In Perl, the patterns described by regular expressions are used to search strings, extract desired parts of strings, and to do search and replace operations.

We are interested in the regular expressions for the elaboration times of the engines and for the error messages of the engines. In a Perl regular expression you can use the following *metacharacters*, *repetitions* and *special characters*:

Char	Meaning
^	begin of a string
\$	end of a string
.	any character except newline
*	match 0 or more times
+	match 1 or more times
?	match 0 or 1 times; or: shortest match
	alternative
()	grouping; "storing"
[]	set of characters
{ }	repetition modifier
\	quote or special

<sup>22</sup>For more information about the *regular expression* in Perl visit the site <http://perldoc.perl.org/perlretut.html> and read the reference [4]

<b>Repetition</b>	
<i>a</i> *	zero or more <i>a</i> 's
<i>a</i> +	one or more <i>a</i> 's
<i>a</i> ?	zero or one <i>a</i> 's (i.e., optional <i>a</i> )
<i>a</i> { <i>m</i> }	exactly <i>m</i> <i>a</i> 's
<i>a</i> { <i>m</i> ,}	at least <i>m</i> <i>a</i> 's
<i>a</i> { <i>m</i> , <i>n</i> }	at least <i>m</i> but at most <i>n</i> <i>a</i> 's
repetition?	same as repetition but the shortest match is taken

<b>Special notation with \</b>	
<code>\t</code>	tab
<code>\n</code>	newline
<code>\r</code>	return (CR)
<code>\xhh</code>	character with hex. code <i>hh</i>
<code>\b</code>	"word" boundary
<code>\B</code>	not a "word" boundary

<b>Matching</b>	
<code>\w</code>	matches any single character classified as a "word" character (alphanumeric or <code>_</code> )
<code>\W</code>	matches any non-"word" character
<code>\s</code>	matches any whitespace character (space, tab, newline)
<code>\S</code>	matches any non-whitespace character
<code>\d</code>	matches any digit character, equiv. to <code>[0-9]</code>
<code>\D</code>	matches any non-digit character

In the next page we reported some examples of Perl regular expressions:

Examples	
abc	abc (that exact character sequence, but anywhere in the string)
^abc	abc at the beginning of the string
abc\$	abc at the end of the string
a b	either of a and b
^abc abc\$	the string abc at the beginning or at the end of the string
ab{2,4}c	an a followed by two, three or four b's followed by a c
ab{2,}c	an a followed by at least two b's followed by a c
ab*c	an a followed by any number (zero or more) of b's followed by a c
ab+c	an a followed by one or more b's followed by a c
ab?c	an a followed by an optional b followed by a c; that is, either abc or ac
a.c	an a followed by any single character (not newline) followed by a c
a\.c	a.c exactly
[abc]	any one of a, b and c
[Aa]bc	either of Abc and abc
[abc]+	any (nonempty) string of a's, b's and c's (such as a, abba, acbabcaaa)
[^abc]+	any (nonempty) string which does not contain any of a, b and c (such as defg)
\d\d	any two decimal digits, such as 42; same as \d{2}
\w+	a "word": a nonempty sequence of alphanumeric characters and low lines (underscores), such as foo and 12bar8 and foo_1
100\s*mk	the strings 100 and mk optionally separated by any amount of white space (spaces, tabs, newlines)
abc\b	abc when followed by a word boundary (e.g. in abc! but not in abcd)
perl\B	perl when not followed by a word boundary (e.g. in perlert but not in perl stuff)

For the elaboration times you can use the following regular expressions in order to match numbers:

- $(\d+)$  = any integer numbers
- $(\d+\.\d+)$  = any real numbers, with a dot

the parenthesis () in a Perl regular expressions means that the pattern is stored in a variable.

For the error messages of the engines you can specify the initial string of the messages, for instance if the message start with "Error:" you can simply use this string like regular expression. If the engine has different error messages you can specify more strings with the use of the or operator |.

## A Appendix - DTD

Here we reported the DTD (Document Type Definitions) of the XML files used by XCheck.

### A.1 experiment.dtd

```
<!ELEMENT experiment (name,description?,engines,documents,
                      queries)>

<!ELEMENT name      (#PCDATA)>
<!ELEMENT description (#PCDATA)>

<!ELEMENT engines    (engine+)>
<!ELEMENT engine     (#PCDATA)>

<!ELEMENT documents (document+)>
<!ELEMENT document  (description?,file,generator?)>
<!ATTLIST document id ID #REQUIRED>
<!ELEMENT file      (#PCDATA)>
<!ELEMENT generator (#PCDATA)>

<!ELEMENT queries   (query+)>
<!ELEMENT query     (description?,(syntax|filequery)+)>
<!ATTLIST query id ID #REQUIRED>
<!ATTLIST query length CDATA #IMPLIED>

<!ELEMENT syntax    (#PCDATA)>
<!ATTLIST syntax engine CDATA #REQUIRED>

<!ELEMENT filequery (#PCDATA)>
<!ATTLIST filequery engine CDATA #REQUIRED>
<!ATTLIST filequery generator CDATA #IMPLIED>
```

### A.2 outcomes.dtd

```
<!ELEMENT outcomes (engine*,info?)>

<!ELEMENT engine (document*)>
<!ATTLIST engine id ID #REQUIRED>

<!ELEMENT document (query*)>
<!ATTLIST document id CDATA #REQUIRED>
<!ATTLIST document size CDATA #REQUIRED>

<!ELEMENT query (error|(result, doc_processing_time?,
```

```

                                query_compile_time?, query_exec_time?,
                                serialization_time?, total_time))>
<!ATTLIST query                id CDATA #REQUIRED>
<!ATTLIST query                length CDATA #IMPLIED>

<!ELEMENT result               (#PCDATA)>
<!ATTLIST result               size CDATA #REQUIRED>

<!ELEMENT doc_processing_time (#PCDATA)>
<!ATTLIST doc_processing_time sd CDATA #REQUIRED>
<!ELEMENT query_compile_time (#PCDATA)>
<!ATTLIST query_compile_time sd CDATA #REQUIRED>
<!ELEMENT query_exec_time    (#PCDATA)>
<!ATTLIST query_exec_time    sd CDATA #REQUIRED>
<!ELEMENT serialization_time (#PCDATA)>
<!ATTLIST serialization_time sd CDATA #REQUIRED>
<!ELEMENT total_time         (#PCDATA)>
<!ATTLIST total_time         sd CDATA #REQUIRED>

<!ELEMENT error               (#PCDATA)>
<!ATTLIST error               type CDATA #REQUIRED>

<!ELEMENT info                (start?,finish?,duration,iterations,cpu,
                                ram,system,file_bench)>
<!ELEMENT start                (#PCDATA)>
<!ELEMENT finish               (#PCDATA)>
<!ELEMENT duration             (#PCDATA)>
<!ELEMENT iterations           (#PCDATA)>
<!ELEMENT cpu                  (#PCDATA)>
<!ELEMENT ram                   (#PCDATA)>
<!ELEMENT system                (#PCDATA)>
<!ELEMENT file_bench           (#PCDATA)>

```

### A.3 analysis.dtd

```

<!ELEMENT analysis            (input,output)>

<!ELEMENT input              (engines, queries, documents, indices, times)>
<!ELEMENT engines             (#PCDATA)>
<!ELEMENT queries            (#PCDATA)>
<!ELEMENT documents          (#PCDATA)>
<!ELEMENT indices            (#PCDATA)>
<!ELEMENT times              (#PCDATA)>

<!ELEMENT output             (index, plots?)>
<!ELEMENT index              (#PCDATA)>

```

```
<!ELEMENT plots      (#PCDATA)>
```

#### A.4 outcome\_s1.dtd

```
<!ELEMENT outcomes  (engines, queries, documents, analysis)>
```

```
<!ELEMENT engines    (#PCDATA)>
```

```
<!ELEMENT queries    (#PCDATA)>
```

```
<!ELEMENT documents  (#PCDATA)>
```

```
<!ELEMENT analysis   (engine+)>
```

```
<!ATTLIST analysis  id ID #REQUIRED>
```

```
<!ELEMENT engine     (index+)>
```

```
<!ATTLIST engine    id ID #REQUIRED>
```

```
<!ELEMENT index      (doc_processing_time?,query_compile_time?,  
query_exec_time?,serialization_time?,  
total_time?)>
```

```
<!ATTLIST index     type CDATA #REQUIRED>
```

```
<!ELEMENT doc_processing_time (#PCDATA)>
```

```
<!ELEMENT query_compile_time (#PCDATA)>
```

```
<!ELEMENT query_exec_time (#PCDATA)>
```

```
<!ELEMENT serialization_time (#PCDATA)>
```

```
<!ELEMENT total_time (#PCDATA)>
```

#### A.5 outcome\_s2.dtd

```
<!ELEMENT outcomes  (engines, queries, documents, analysis)>
```

```
<!ELEMENT engines    (#PCDATA)>
```

```
<!ELEMENT queries    (#PCDATA)>
```

```
<!ELEMENT documents  (#PCDATA)>
```

```
<!ELEMENT analysis   (engine+)>
```

```
<!ATTLIST analysis  id ID #REQUIRED>
```

```
<!ELEMENT engine     (document+)>
```

```
<!ATTLIST engine    id ID #REQUIRED>
```

```
<!ELEMENT document   (index+)>
```

```
<!ATTLIST document  id CDATA #REQUIRED>
```

```
<!ATTLIST document  size CDATA #REQUIRED>
```

```
<!ELEMENT index      (doc_processing_time?,query_compile_time?,
```

```

                query_exec_time?,serialization_time?,
                total_time?)>
<!ATTLIST index      type CDATA #REQUIRED>

<!ELEMENT doc_processing_time (#PCDATA)>
<!ELEMENT query_compile_time (#PCDATA)>
<!ELEMENT query_exec_time (#PCDATA)>
<!ELEMENT serialization_time (#PCDATA)>
<!ELEMENT total_time (#PCDATA)>

```

## A.6 outcome\_s3.dtd

```

<!ELEMENT outcomes (engines, queries, documents, analysis)>

<!ELEMENT engines (#PCDATA)>
<!ELEMENT queries (#PCDATA)>
<!ELEMENT documents (#PCDATA)>

<!ELEMENT analysis (engine+)>
<!ATTLIST analysis id ID #REQUIRED>

<!ELEMENT engine (query+)>
<!ATTLIST engine id ID #REQUIRED>

<!ELEMENT query (index+)>
<!ATTLIST query id CDATA #REQUIRED>
<!ATTLIST query length CDATA #IMPLIED>

<!ELEMENT index (doc_processing_time?,query_compile_time?,
                query_exec_time?,serialization_time?,
                total_time?)>
<!ATTLIST index      type CDATA #REQUIRED>

<!ELEMENT doc_processing_time (#PCDATA)>
<!ELEMENT query_compile_time (#PCDATA)>
<!ELEMENT query_exec_time (#PCDATA)>
<!ELEMENT serialization_time (#PCDATA)>
<!ELEMENT total_time (#PCDATA)>

```

## A.7 outcome\_s5.dtd

```

<!ELEMENT outcomes (engines, queries, documents, analysis)>

<!ELEMENT engines (#PCDATA)>
<!ELEMENT queries (#PCDATA)>
<!ELEMENT documents (#PCDATA)>

```

```

<!ELEMENT analysis (time+)>
<!ATTLIST analysis id ID #REQUIRED>

<!ELEMENT time (document+)>
<!ATTLIST time type CDATA #REQUIRED>

<!ELEMENT document (engine+)>
<!ATTLIST document id CDATA #REQUIRED>
<!ATTLIST document size CDATA #REQUIRED>

<!ELEMENT engine (#PCDATA)>
<!ATTLIST engine id CDATA #REQUIRED>
<!ATTLIST engine vs CDATA #REQUIRED>

```

## A.8 outcome\_s6.dtd

```

<!ELEMENT outcomes (engines, queries, documents, analysis)>

<!ELEMENT engines (#PCDATA)>
<!ELEMENT queries (#PCDATA)>
<!ELEMENT documents (#PCDATA)>

<!ELEMENT analysis (time+)>
<!ATTLIST analysis id ID #REQUIRED>

<!ELEMENT time (query+)>
<!ATTLIST time type CDATA #REQUIRED>

<!ELEMENT query (engine+)>
<!ATTLIST query id CDATA #REQUIRED>
<!ATTLIST query length CDATA #IMPLIED>

<!ELEMENT engine (#PCDATA)>
<!ATTLIST engine id CDATA #REQUIRED>
<!ATTLIST engine vs CDATA #REQUIRED>

```

## A.9 engines.dtd

```

<!ELEMENT engines (engine*)>
<!ELEMENT engine (name,version?,homepage?,description?,
adapter,path,cpu_time)>
<!ATTLIST engine id ID #REQUIRED>
<!ATTLIST engine type (xquery|xpath) #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT version (#PCDATA)>

```

```
<!ELEMENT homepage      (#PCDATA)>
<!ELEMENT description    (#PCDATA)>
<!ELEMENT adapter        (#PCDATA)>
<!ELEMENT path           (#PCDATA)>
<!ELEMENT cpu_time       (#PCDATA)>
```

## A.10 adapter.dtd

```
<!ELEMENT adapter (engine)>

<!ELEMENT engine      (command, times?, error?)>
<!ATTLIST engine id ID #REQUIRED>

<!ELEMENT command (before?,executable,after?,file_query,
                    fullpath_doc)>
<!ELEMENT before  (#PCDATA)>
<!ELEMENT executable (#PCDATA)>
<!ELEMENT after   (#PCDATA)>
<!ELEMENT file_query (#PCDATA)>
<!ELEMENT fullpath_doc (#PCDATA)>

<!ELEMENT times (factor_time, time+, doc_processing_time?,
                query_compile_time?, query_exec_time?,
                serialization_time?, total_time?)>
<!ELEMENT factor_time (#PCDATA)>
<!ELEMENT time (line+)>
<!ATTLIST time id ID #REQUIRED>
<!ELEMENT line (#PCDATA)>
<!ELEMENT doc_processing_time (#PCDATA)>
<!ELEMENT query_compile_time (#PCDATA)>
<!ELEMENT query_exec_time (#PCDATA)>
<!ELEMENT serialization_time (#PCDATA)>
<!ELEMENT total_time (#PCDATA)>

<!ELEMENT error (#PCDATA)>
```

## References

- [1] XML TaskForce XPath. <http://www.xmltaskforce.com>.
- [2] Qexo - The GNU Kawa implementation of XQuery. <http://www.gnu.org/software/qexo/>, 2006.
- [3] M. Fernández et al. Galax. The XQuery implementation for discriminating hackers. <http://www.galaxquery.org>, 2006.
- [4] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly, 1997.
- [5] M. H. Kay. Saxon. An XSLT and XQuery processor. <http://saxon.sourceforge.net>, 2006.
- [6] Christoph Koch. Arb. A highly scalable query engine for expressive node-selecting queries on (XML) trees. <http://www.infosys.uni-sb.de/~koch/projects/arb/>, 2006.
- [7] Wolfgang Meier. eXist. Open Source Native XML Database. <http://exist.sourceforge.net>, 2006.
- [8] University of Antwerp. Blixem. LiXQuery engine. <http://adrem.ua.ac.be/~blixem/>, 2006.
- [9] University of Munich, University of Twente, and CWI. MonetDB/XQuery. An XQuery Implementation. <http://monetdb.cwi.nl/XQuery>, 2006.
- [10] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 974–985, 2002. <http://monetdb.cwi.nl/xml/>.
- [11] Axyana software. Qizx/open. An open-source Java implementation of XQuery. <http://www.axyana.com/qizxopen>, 2006.