Master Thesis

XQuery Benchmarking service using TPC-X

Author(s):
Singh, Anandeshwar

Publication Date:
2008

Permanent Link:
https://doi.org/10.3929/ethz-a-005684071

Rights / License:
In Copyright - Non-Commercial Use Permitted
Master thesis Report

XQuery Benchmarking service using TPC-X

Anandeshwar Singh

Systems Group
Department of Computer Science
ETH Zurich

http://www.dbis.ethz.ch/

Summer Semester 2008

Supervised By:
Dr. Peter Michael Fischer
Prof Dr. Donald Kossmann
Abstract

The Extensible Markup Language (XML) is being used extensively since its advent in Industry and in Research. The XML data is primarily used to facilitate sharing of structured data across different Information Systems, particularly via Internet. This XML data can be queried using any of the different XQuery engines available.

For past few years, many XQuery engines have been developed in industry and for research. These engines tried to cover some general expressions of XQuery first and later their implementation became more and more complex due to addition of other remaining XQuery expressions. Due to this addition and incomplete implementations, arose the need for a benchmarking service. This benchmarking service should compare the different available XQuery engines for their current XQuery expressions support and for their improvement in the implementation. In addition, this benchmarking needs to be fast and easy to do.

This thesis works on providing such a benchmarking service that could be rapidly and easily setup and produce results in more detail.
Chapter 1

Introduction

XML or Extensible Markup Language is a general-purpose specification for creating custom markup language. It is one of the ways in which data can be structured today. XML without any doubt is used extensively on internet these days for the sole purpose of sharing structured data. This XML data is queried by making use of the XML query processors (or XQuery engines). XQuery is a declarative query language, which is used to query over data that have XML structure. XQuery syntax is reviewed by W3C [1].

XML and XQuery besides being around for some years are still in its premature stage. This is evident from the fact that the syntax for XQuery keeps reviving at W3C. Many new features and functionalities have been added to XQuery expressions by W3C in near past (e.g. full-text search, etc.) The new functionalities and features try to extend the usability of XQuery. The extension of XQuery specifications implies its implementation on various XQuery engines provided by different XQuery engine developers. Many XQuery engines exist today, some tries to implement most of the features provided by W3C specifications and others try to optimize the path navigations, etc. Thus, there arises the need to compare the different XQuery engines not only for their execution time in querying but also for the length of their implementation.

The growing needs and uses of XML and XQuery gave rise to multiple XQuery engines and with this the need for benchmarking. This requirement for an XQuery benchmarking is the subject of this thesis. The main motivation behind the XQuery Benchmarking Service is the fact that no widely established and accepted benchmarks for XQuery exist, which inhibits the acceptance of XQuery and the investment into it. There are two approaches to overcome this problem. One approach is to provide a general XQuery benchmarking service. This benchmarking service for a set of data and queries can be used to test against multiple XQuery engines, so that the comparison of implementation for multiple XQuery engines can be done, and also interested parties can contribute their own queries. The second approach is to provide a benchmark that not only stresses on storage and path navigations but also on all other aspects of XQuery expression.

First approach of providing general XQuery benchmarking service deals with providing such a tool that could run multiple XQuery engines locally for a given set of queries and data. This tool should be able to run the given set of queries on the given data on multiple XQuery engines, and could note the time taken by each XQuery engine for running the queries to compare the different XQuery engines. XCheck [1] is one such match for a tool. It provides a local environment for running multiple XQuery engines and also it provides result in the form of graphical plots that compares different XQuery engines.
Another approach to provide a benchmark that stresses all aspects of XQuery expressions and not just storage or path navigation deals with providing a set of queries and documents that covers all aspect of XQuery expression. By covering all possible XQuery expressions, an XQuery engine can be tested for its implementation length. Implementation length refers to finding out, which XQuery engines do not yet implement all features, and thus the comparison for multiple XQuery engines for their implementations can be achieved.
Chapter 2

Benchmarking service requirements

An XQuery benchmarking service requires a system that tests a set of queries and documents against a set of XQuery engines, which in turn requires a local environment for running multiple XQuery engines. The system would run a set of queries over a set of data on multiple XQuery engines, so that the time taken by the engines could be used to analyze the different XQuery engines. The system should be a tool to run particular experiments submitted by the user. This experiment is a collection of queries, data and XQuery engines, which needs to be tested.

Benchmarking service requires a tool that could run multiple XQuery engines for a set of data and queries, and display the results in a more understandable way. XCheck is one of the match for such tool, and is used in this thesis. XCheck provides a local environment for running multiple XQuery engines. XCheck takes an xml file as an input and returns result in the form of html and xml file. This input xml file specifies a list of queries, documents and engines on which test needs to be done. XCheck gives result in the form of html files along with graphical plots that makes it easy to understand and compare different XQuery engines.

An XQuery benchmark that stresses all aspects of XQuery and not just storage or path navigation is another requirement. The benchmarking service should stress all aspects of XQuery to check the implementation of each XQuery engines. To achieve this, a set query that covers all possible XQuery expression is required, and TPC-X is one such benchmark. TPC-X is a set of 23 queries that covers all possible expressions in XQuery. TPC-X can be used to check the implementations of different XQuery engines. It compares the XQuery engines for their XQuery expression support. Along with TPC-X, other benchmarks like XMark and TPoX that stresses on storage and path navigation are provided to compare the different XQuery engines for their storage and path navigation support.

The benchmarking service provides data and queries from TPC-X, XMark and TPoX as mentioned above. However, another requirement for the benchmarking service is to provide such a system that can be used by the interested parties to contribute their own queries. In addition, the benchmarking service should be easy to use and can be used by anyone who is interested. To deal with this, JSP pages are used in this benchmarking service, and are published on internet for use by interested parties.

2.1 XCheck

XCheck[1] is an open-source software for automatic execution of a benchmark on XML query processors, developed at the University of Amsterdam. XCheck takes an experiment file as input and generates result in the form of HTML pages. Experiment file is an XML file, which indicates
the documents, queries and XQuery engines that need to be tested. For an XQuery engine to be executed from XCheck it requires an adapter or a wrap that indicates how to execute the engine, how to interpret the results, etc. XCheck produces the result in the form of textual data and graphical plots as well, which makes it easier to understand the experiment results.

2.2 TPC-X

TPC-X is an adapted and simplified version of the amazion.com. TPC-X is an XQuery benchmark that stresses all aspects of XQuery and not just path navigation and storage. TPC-X consists of 25 queries and a set of documents on which the queries can run. TPC-X queries cover all possible XQuery expressions, hence it makes easier to compare the different XQuery engines for the feature availability and for the quality of the specific feature implementation.

2.3 Web-based GUI

One of the basic requirements for the benchmarking service is its availability to everyone and eases of use. To overcome this requirement, web-based GUI is provided that can be deployed on a server and can be used by anyone on internet. Using the web-based GUI would not require any setup effort from the user’s side for deploying or for using the benchmark. Web-based GUI makes it easier for the user to interact with the benchmarking system. It can be used to create an experiment, view the results of the experiment, add new documents and queries, etc.
Chapter 3

Benchmarking Service

The benchmarking service is a simple web page for the user, where user can leave his choice for engines, data and queries. The benchmarking service then for each chosen engine, uploads the chosen documents and runs the chosen queries over it. If queries executed then their runtime would be presented to the user else, an error message stating the reason why the query was not executed will be raised.

This benchmarking service required development mainly in two segments. One segment being the Graphical User Interface (GUI) used by the user for running the benchmarking for a set of engines, documents and queries. Other segment being the back end that accepts user input from the GUI and runs the benchmark.

Benchmarking service mainly consists of XCheck, a set of queries and documents, GUI and demon processes. To use the benchmarking service, a user makes use of the GUI to interact with the benchmarking system. A user can create an experiment (which is a collection of data, queries and XQuery engines that needs to be tested) to run on XCheck; or upload an XML document and queries; or can view list of benchmarking experiment results by making use of the provided GUI.

Benchmarking service tries to achieve the following 5 tasks:

i. **Experiment management**: This deals with the experiment submitted by the user at its various stages. Experiment management deals with the creation of an experiment first, then it queues this experiment for execution and it executes experiment one-by-one. Experiment management also deals with the notification by e-mail to the user on submission, start and end of his experiment.

ii. **Access to experiment results**: This deals with displaying the list of benchmarking experiments. This list can be displayed completely or can be filtered by the user to return few benchmarking results. A user can filter the whole list according to a set of documents or queries or XQuery engines. Apart from filtering a general XQuery support is also provided over the whole experiment file.

iii. **Management of experiment sources**: This part deals with adding new XML documents, queries by the interest parties and maintaining it.
iv. **Web-based GUI**: Web-based GUI provides ease of use and in accessible to everybody. This part deals with the GUI that is used by the user to interact with the benchmarking system. Web-based GUI covers all aspect of user interaction with the benchmarking system starting from experiment submission to viewing benchmarking results.

v. **Back-end support**: This part deals with the queuing of the experiments, reliably executing the experiments, collecting performance numbers, etc.

### 3.1 Experiment Management

An experiment is a set of XQuery engines, documents and queries. An experiment expressed in the form of an xml file is an input to XCheck. XCheck in turn runs the queries over the documents mentioned in experiment, on different XQuery engines indicated in experiment, to get the results comparing the different XQuery engines. Experiment management broadly deals with the following benchmarking service requirements.

i. **Creation of Experiment**: As a first step to test XQuery engines for a set of queries and documents, user needs to create an experiment. The GUI illustrates the choice of XQuery engines, documents and queries that a user can select. In case user wants to use his desired queries and documents that are not available with the benchmarking service, then he first needs to add the required document and queries to the benchmarking service. After user submits his choice of queries, documents and XQuery engines on GUI, his request is first converted and expressed in a form that is understandable by XCheck. XCheck accepts an xml file named experiment.xml inside a folder that is placed inside the experiment folder of XCheck. Now to run this experiment only folder name needs to be provided to the XCheck. Hence, the user choice of experiment is converted to XML file first. This XML file is stored in a folder whose name is based on the date and time of the submission of the experiment. Therefore, if an experiment were submitted on 9th August 2008 at 1.45pm, then the name of folder in which experiment.xml file would be created will be 2008-08-09T13_45_00. Last two zeros represent seconds.

ii. **Queuing of Experiment**: There are two Demon processes running. One of these demon processes is responsible for running XCheck on the back-end. This demon process is also responsible for queuing of the experiment and for running the experiments one-by-one.

In case of no new experiment, demon process waits for 10 seconds before rechecking, until new experiment found. When the demon process finds a new experiment, it executes the experiment using XCheck. XCheck after execution of the experiment creates an output folder inside the experiment folder. This output folder acts like an indicator for the experiment execution. Demon process while searching for a new experiment, first lists all experiment folder in ascending order and scans for the first experiment folder without output folder inside it. In case an experiment found without
output folder, the demon process executes it first. Hence, naming folder based on date and time of query execution provides implicit queuing.

iii. **Execution of Experiment**: When a user submits an experiment on GUI, it converts the user selection on XQuery engines, data and queries to an XML file that is understandable by XCheck. GUI is a set of JSP pages, and one of the JSP page accepts the user selection for XQuery engines, data and queries and creates an XML file as required by XCheck. This XML file named experiment.xml is generated by making use of XQuery update facility provided by MXQuery[4]. Once experiment.xml file created and submitted, demon process executes this experiment by running XCheck on command line and goes to a wait state until the experiment finishes execution. This makes benchmarking service at a time to execute only one experiment based on the description provided in experiment.xml file.

iv. **Notification Service**: This part of benchmarking service deals with the notification to the user about the submission of the experiment, start and end of experiment execution. These notifications depend on user’s choice regarding experiment notifications. When a user submits an experiment on GUI, the GUI sends a notification to the user about the experiment submission and the demon process sends the notification regarding the start and end of the experiment execution. These notifications are simple e-mails sent to the user regarding his experiment. It notifies first about the submission of the experiment, and when the experiment starts to execute by the demon process another notification is sent to the user. At the end of the experiment execution, another notification is sent to the user along with the link to the result of the experiment. For sending e-mails JavaMAIL API[5] is used.

### 3.2 Access to Experiment Result

An XCheck experiment execution generates a set of output or result html files. This set of html file is copied by demon process to the tomcat server, from where a user can access or view the experiment results. A notification sent to the user at the end of the experiment, sends the link to the result of the experiment. All experiments those executed and those that are going to be executed, are stored on the tomcat server. Hence, a user can view his experiment results from this list as well, and also can view the experiments submitted by other users as well. The GUI lists all experiment and also provides a filter option on this list along with a general XQuery support.

i. **List of Experiments**: GUI lists all the experiments that have been submitted by the users in the past. Hence, an experiment is added to the list of all experiments, soon after its submission by the user. However, experiment results are only available once the experiment is completely executed by the demon process and the results are copied on the tomcat server by the demon process. The demon process keeps the XCheck experiment folder consistent on tomcat server.
ii. **Selection of Experiments**: The list of all experiments can be filtered by making use of the GUI. The GUI provides a selection on the entire list of experiments. The selection is based on the user selection of XQuery engines, documents and queries. E.g.: If a user selects SaxonA and Xqilla engines, store100100.xml document and top10.xq query, the all those experiments would be listed that have SaxonA and Xqilla XQuery engines, store100100.xml document and top10.xq query.

iii. **General Query over the experiments**: When a user submits an experiment on the GUI, the two XML files are created inside the folder that is executed by XCheck. This folder is exclusive for each experiment. The two XML files generated are experiment.xml and expDetail.xml. experiment.xml file is input to the XCheck for an experiment execution, and expDetail.xml stores the data related to the user who submitted the experiment. It stores the user name, email address, etc. (to view the schemas of experiment.xml file and expDetail.xml file refer appendix B). The user for entire list of experiments can query the two XML files for each experiment, by making use of the collection support provided by MXQuery. The user posts the complete query on GUI for it run over the list of all experiments, and the results obtained from running the query over MXQuery is directly displayed on the GUI.

### 3.3 Management of Experiment sources

XCheck runs an experiment that in turn is a set of XQuery engines, XML documents and XML queries. XCheck takes an experiment file as input that describes a set of XQuery engines on which experiment needs to run, and a set of XML documents and queries which needs to be run on the XQuery engines specified. Benchmarking service provides TPC-X, XMark and TPoX benchmarking documents and queries. However, interested users can upload or add their own queries and documents to this list.

i. **Adding new schema for XML documents**: Before uploading a new document on the benchmarking service, it is required to upload the schema for the document. The schema helps in checking the uploading XML document conformance with the schema, and also it prevents query belonging to one schema to run on documents having different structure.

ii. **Adding a new Document**: To add a new document to the benchmarking service, user needs to upload the document from the GUI, along with providing the details about the XML document, like schema (or dataset), name, description, etc. Only after the demon process confirms the document to be correct, it will be added to the list of documents that can be used for benchmarking.

iii. **Adding new queries**: New queries can be added to the benchmarking service by making use of the GUI. User needs to give the query details like name of the query, description, dataset, etc before adding the list of queries that can be executed by the XCheck.
iv. **Maintaining metadata of the source:** For each query, there exists an XML file that keeps the data regarding the query; and whenever a user adds a new query to the GUI, it generates the .xq file, which is an XQuery file, and an XML file that stores the details regarding the new XQuery. Same exists with the addition of the new document, however in case of XML documents, only one meta data file is maintained. Hence, when a new document is added or uploaded by the user the details regarding the same is stored in this meta-data file.

3.4 Web-based GUI

Web-based GUI is the place from where a user interacts with the benchmarking system. GUI is a set of JSP pages, which is loaded onto the tomcat server of the DBIS team of ETH Zurich. These JSP pages are accessible to everyone, and can be used for the experiment creation, or for adding new documents and queries, or for viewing the results of the experiments. The complete description of GUI is provided in chapter 5 of this report.

3.5 Back-end Support

The back-end support of the benchmarking service refers to the two demon processes running on the machines. One of the demon processes is responsible for running the experiment reliably and another demon process is responsible for copying the experiment results from XCheck folder to the tomcat server, so as to make the results accessible to the user.

i. **Demon Process 1:** Demon process 1 deals with the running of the experiment. It first waits for a new experiment to arrive, once a user submits a new experiment it is executed on XCheck by this demon process. Demon Process 1 checks for a new experiment after every 10 seconds, and if new experiment found then it sends a notification regarding the start of the experiment execution to the user and then start the experiment execution on command-line. While the experiment is getting executed, Demon Process 1 goes into a wait state until the experiment is entirely executed. In the meantime, if a new experiment arrives it needs to wait until the first experiment is entirely executed. When the first experiment gets completely executed Demon Process 1 comes out of wait state, notifies the user about the end of the experiment execution and sends a link to experiment result on the tomcat server to the user. During this time, the other demon process copies the result from the XCheck experiment folder to the tomcat server, from where it is accessible by the user.

After the first experiment is executed and notified to the user, Demon Process 1 looks for an experiment without results. It scans the experiment folder alphabetically, and executes the experiment first found without any results (this is indicated by the absence of the output folder inside the experiment folder). Since, the experiment name is derived from date and time; hence, the experiment submitted first is the first
one to be executed (like queues). In addition, reliability in executing the experiment is achieved by the fact that at a time the Demon Process executes only one experiment.

ii. **Demon Process 2**: Demon Process 2 is responsible for copying the experiment along with results onto the tomcat server from the XCheck experiment folder. Hence, as soon as an experiment is submitted or its result is generated, it is copied to the tomcat server by Demon Process 2.

The two demon processes used, adds to the scalability of the benchmarking service, as Demon Process 2 needs to be running over the machine where tomcat exists and the Demon Process 1 can run on multiple machines having XCheck with replicated documents and queries.

![Figure 3.1: Benchmarking service Architecture](image)

Above figure shows the architecture of the Benchmarking service. In the figure 3.1, boxes represent a machine and circles represent the benchmarking service deployed on each machine. Demon1 is demon process 1 and Demon2 is demon process 2. Once the user submits an experiment on the front-end machine, JSP page converts the user request for the experiment into an experiment file understandable by the XCheck. The experiment file generated is copied on one of the machines for running. Demon Process 1 executes the experiment and generates the result and graphical plots which in turn in copied by Demon Process 2 to the front-end.
3.5.1 XCheck

XCheck is an open-source software framework for automatic execution of a benchmark on the XML query processors. XCheck provides a local platform for running multiple XQuery engines. It takes an experiment XML file as input, which is a list of XQuery engines and XML documents and queries that need to be tested against the multiple XQuery engines, and generates result in the textual form and in the graphical form as well.

XCheck takes an experiment XML file as input. This XML file needs to be placed inside a folder named as the name of the experiment (e.g., if the name of the experiment is xmark, then we need to put the experiment.xml file inside the folder named xmark inside the experiment folder of XCheck). This enables XCheck to identify the name of the experiment for running and can place the output folder (consisting of textual result as well as graphical plots represented in the form of html file) inside the folder in which experiment.xml file exists.

XCheck provides a local environment for running multiple XQuery engines. For enabling an XQuery engine to run by XCheck, it requires an adapter. The adapter wraps the XQuery engine for the XCheck. An adapter for an engine is an XML file that specifies how to run the XQuery engine and how to interpret the results and error messages coming from the XQuery engine during query execution. Following engines are supported by the benchmarking service, and thus adapters for the following engines are provided:

1. SaxonB version 8.0: To run SaxonB version 8.0 XQuery engine, it requires java on the machine. SaxonB XQuery engine is executable jar file, which takes query as input. Hence, we need to copy the jar file on disk and provide the link to this jar file in engines.xml file (engines.xml file acts like meta-data for engines supported) for XCheck to execute this engine. In addition, it requires an adapter to make it run from the XCheck. This adapter provides the command to execute XQuery engine from XCheck.

2. SaxonB version 9.0: To run SaxonB version 9.0, it requires same setup as for SaxonB version 8.0. The two versions of XQuery engines provide the comparison of new engine with the old one.

3. SaxonA: SaxonA XQuery engine is also an executable jar file like SaxonB and requires same setup as any of the two versions of SaxonB.

4. MXQuery: MXQuery engine is also an executable jar file like Saxon, hence same setup needs to be followed.

5. MonetDB: For running MonetDB XQuery engine, it first requires start of the MonetDB server and only then, the queries can be sent to the MonetDB for execution. Hence, to run MonetDB from XCheck, it requires its installation on the machine and the path of its installation in engines.xml file and an adapter. The adapter for MonetDB, also consists of certain commands before and after the execution of the engine. Before and after provides the command for starting and stopping of the MonetDB engine.
MonetDB XQuery engine provides certain number of execution times that needs to be interpreted by the adapter.

6. **eXist**: eXist XQuery engine also requires server start-up and shut-down like MonetDB. Hence, same steps need to be followed for the eXist as it is for MonetDB, except eXist does not return execution times and XCheck needs to then calculate the total time for execution.

7. **Zorba**: To run Zorba on XCheck, it requires Zorba installation like eXist. However, running Zorba only requires a command along with the query file-name. Hence, the adapter for the Zorba will provide the command for executing a query, and also it will provide the interpretation for certain execution times returned by Zorba.

8. **XQuery**: XQuery engine, also requires same step-up step for running from XCheck like Zorba. However, XQuery does not return any execution times.

9. **BerkeleyDB**: BerkeleyDB XQuery engine also requires a server start-up before sending queries for execution. Hence it will follow the same step as in eXist for installation. However, in case of BerkeleyDB it requires query in some specific format, hence it requires a program to modify the provided query for execution to a form understandable by BerkeleyDB XQuery engine. (In the benchmarking service there exists an Berkeley.java code that performs this operation)

10. **Sedna**: Sedna also requires server start-up before querying and server shutdown after query execution, and it returns certain execution times like MonetDB. Hence, similar steps as MonetDB is required to run Sedna from XCheck.
Figure 3.2: XCheck metadata architecture
Benchmarking service can be used for adding new documents and queries. Hence, it requires some metadata to be maintained for documents and queries. XCheck provides its own meta-data for XQuery engines. However, it is extended for the documents and query (figure 3.2 green font colored ones are extended metadata). For each query there exists an XML file that stores the metadata for the query, and for all the documents only one metadata file is maintained. More details about the metadata and schema related with the XCheck can be found in Appendix B.

Above figure 3.3, give the abstract ER diagram for the metadata and XCheck experiment. An experiment takes entries from the engine, document and query meta-data, which is recognizable by the XCheck for running.

XCheck installation consists of multiple folders for specific tasks. For benchmarking service, some of these folders were made use of. Following lists the folders being used along with what they are used for:

1. **adapters**: This folder is used for saving all XQuery engines adapter. For each XQuery engine being supported by XCheck, it requires a corresponding adapter XML file in this folder.
2. engines: Inside this folder, jar files for those XQuery engines are copied that do not require any installation on machine, like MXQuery, SaxonA, etc

3. experiments: Inside this folder all experiments are kept. For each experiment, there exists an individual experiment folder inside this experiments folder. Inside each individual folder, there exists experiment.xml file that is understandable by XCheck for running an experiment. To run an XCheck for an experiment, only the name of the individual folder of the experiment is required to be supplied. In case of benchmarking service, web-based GUI creates the experiment.xml file and copies the experiment.xml file inside a new folder named on the basis of date and time of the experiment submission. Demon Process1, checks for an experiment alphabetically without an output folder (An experiment without output folder is indication that the experiment is not executed yet). Alphabetical scanning ensures that the experiment submitted first is executed first. Once an experiment starts execution, Demon Process1 goes into a wait state till its complete execution. This ensures that only one experiment is running at a time.

4. repository: Inside this folder all documents and queries are kept. Inside this folder there exists another folder named docs in which all documents are kept along with metadataDoc.xml, that stores the metadata of all the documents. Another folder called queries inside repository folder stores all queries and the corresponding metadata file for each query (for each query there exists an XML file with same name as query file that stores the metadata of the query). Inside queries folder admin, dataMining, productSearch and shopping folder hold all the TPC-X queries. xmark holds the XMark benchmarking queries, and tpox holds all TPoX benchmarking queries. Inside the userQuery folder all user queries uploaded from the benchmarking service are kept.
Chapter 4

TPC-X Benchmarking

4.1 Preamble

The TPC-X benchmark is a simplified and adapted simulations of the e-commerce company Amazon.com[2]. The benchmark consists of queries which represent operations on an online store that can be executed by users or administrators. The queries can be executed on different data sources which are provided by the benchmark or can be created with a data generator.

4.2 Data Structure

Generally, the data structure of TPC-X consists of products, users, items of a bookstore. Projects represent the objects, which are sold by the bookstore, like Books, DVDs and Video Games. They are classified in different categories like Books, Music & Movies and Toys & Video Games. The items stand for real items in the storage or for second hand items which are offered by users (Third Party Item).

Additionally, the data structure consists nodes to represent the connections between users and products. The Shopping Cart saves all products which a user wants to by or have bought in the past (and optionally a Third Party Item if the user wants to buy a second-hand product). A review of a product is done with the Review object and the history of a user is saved with the History object. Additionally, the user saves his wishes in the With List which can be directly converted into a Shopping Cart.

Furthermore, products can be combined in a Special Offer package with a special price.

4.2.1 Schema

An overview of the data schema can be seen in figure 4.1. The red boxes represent the data objects in the database and the blue ones the relations. Additionally, elements contain information about where they are saved, if there are nested into other elements (e.g. every user saves his History).
4.2.2 Data Generation

The data sources for the queries is automatically generated by the ToXgene data generator[3]. ToXgene is able to generate real XML data because it gets the data from the definition from the XMark and TCP-H benchmark. As a description for the generated data, ToXgene requires a XML Schema like description of the data which should be generated inclusive instruction about the amount. Currently, TPC-X provides pre-generated data source with 100 (366KB), 1000 (3.6MB), 10000 (35.9MB), 50000 (179.4MB) and 100000 (359.1MB) users and products.

![Figure 4.1: Overview of TPC-X Schema](image)
4.3 Queries

TPC-X queries cover all possible XQuery expressions. Appendix A gives the list of all TPC-X queries. The benchmark operations are divided in the following categories:

- **Admin**: Operations which are done by store administrators, like adding a new product to the catalog.
- **Data Mining**: Operations to get special information out of the data source, like the top ten buyers.
- **Searching**: Searching items in the data source.
- **Shopping**: Operations which are done by a normal user that uses the online store to buy products.

4.3.1 Updating Queries

The benchmark operations which are implemented with XQuery Update (like adding a new product). To test these queries (and to allow testing engines which cannot change the data sources persistently), we introduced the possibility to execute a test query directly after the execution of the update query. The test query is executed in the same context like the main query, ergo, it is possible to query updated variables from the main query.

4.4 Testing Environment

We chose XCheck[1] to test several XQuery engines with our benchmark. XCheck is a software for automatic execution of a benchmark on XML query processors. XCheck is able to run several queries on several engines with several data sources several times. As a result, XCheck creates an HTML and XML page with all time informations of the execution and is able to build plots to visualize the results.
Chapter 5

Using Benchmarking Service

The benchmarking service can be used for multiple purposes, starting from the creation of an experiment to uploading documents and queries. An external user can interact with benchmarking service by making use of the web-based GUI and can also install it on own machine. For installing the benchmarking service on own machine, user requires tomcat server installed first, then deploying XCheck and the benchmarking service on the machine. A tomcat server can be installed from “http://tomcat.apache.org/”.

To deploy XCheck one needs to follow the following steps:

1. Copy the XCheck-0.2.0 folder from the provided benchmarking-source directory to path from where experiment would run.
2. Install the different XQuery engines on this machine and hide those engines tag in engines.xml file inside XCheck folder that are not installed.
3. Update the engines.xml file for the engines provided (specifying paths, etc).
4. Update the adapter file for each XQuery engine. An adapter file can be found inside XCheck-0.2.0/adapters/

For deploying benchmarking service one needs to follow the following steps:

1. Compile Start.java (javac Start.java) inside the benchmarking-source directory.
2. Run Start (java Start).
3. Enter the complete path for XCheck directory (e.g., /usr/local/XCheck-0.2.0).
4. Enter the complete path to the tomcat server (e.g., /usr/local/www/var/tomcat6).
5. A new folder called xqbench will be created.
6. Copy the folder xqbench on the tomcat server, and the benchmarking service is ready to use.
7. Compile and run DemonProcess.java from the XCheck folder (set classpath to the xqbench/WEB-INF/lib folder).
8. Compile and run DemonProcess5.java from the tomcat server.
9. To set the smtp server for the email notification service unjar the mailsend.jar and update the SendMail.java file inside it.
5.1 Using benchmarking service GUI

The benchmarking service GUI can be used for multiple purposes, including experiment creation uploading documents, queries, etc. The front-end of the Benchmarking service is as shown below in figure 5.1:

![Benchmarking service front-end](image)

*Figure 5.1: Benchmarking service front-end*

From figure 5, user can navigate to JSP pages for adding documents and queries, for creating experiments and for viewing the experiment results.
Adding new document

A new document can be added to the benchmarking service by clicking on Add document option on the front-end of the benchmarking service. It will navigate to upload_docu.jsp page.

**Upload Document!**

![Form to upload XML Document]

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name of the Document:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description of the Document:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dataset:</td>
<td></td>
<td>Pre-generated data</td>
</tr>
</tbody>
</table>

*Documents having same dataset belong to same schema*

Figure 5.2: *upload_docu.jsp*

Figure 5.2 shows the upload_docu.jsp page. A user compulsorily needs to provide the name of the document, its description, the dataset (or to the schema to which it belongs to) and the document. After specifying the required details and clicking on upload, document is uploaded to the XCheck repository from where it can be used by XCheck for benchmarking.
Adding new query

A new query can be added to the benchmarking service by clicking on Add query option on the front-end of the benchmarking service. It will navigate to upload.jsp page.

![Add query!](image)

**Figure 5.3:** upload.jsp

Figure 5.3, shows the upload.jsp page, it can be used to upload a query for running on the XCheck. To upload a query, user needs to compulsorily provide the name of the query, its description, the dataset to which the query belongs, author, query and query result. If required, then any note can be added, also the categories to which the new query belongs can be chosen. Once listed everything, a new query file is created inside the XCheck repository for queries, along with an XML file that maintains the metadata of the new query.
Creating an experiment

A new experiment can be created to run on the benchmarking service by clicking on Create Experiment option on the front-end of the benchmarking service. It will navigate to choice.jsp.

**TPC-X Experiment**

### Select Engines:
- SaxonB8
- SaxonB9
- SaxoNA
- MonetDB
- eXist
- Zorba
- XQilla
- BerkeleyDB
- Sedna

### Choose Data:
- 100 Users and Product (366KB)
- 1000 Users and Product (3.6MB)
- 10000 Users and Product (35.9MB)
- 50000 Users and Product (179.4MB)
- 100000 Users and Product (359.1MB)
- XMark 10MB data
- XMark 100MB data
- TPoX 100MB data

### Select TPC-X Queries:
- Updates all Product price
- Inserting a new Product
- Deleting a Product
- Customer Segments
- Forecast

*Figure 5.4: choice.jsp*

Figure 5.4 shows the choice.jsp page, it can be used for creating a new experiment. A user first needs to choose some XQuery engines provided and then can select some XML documents from the list. Only after the selection of the documents, a user can be able to view a set of queries for selection. Only those queries would be available for selection for which the corresponding XML data is already chosen. After making selection user will be asked to fill-in his details and only after that the experiment will be submitted.
Viewing the experiments

Experiments already submitted by different users can be viewed clicking on Show Experiments option on the front-end of the benchmarking service. It will navigate to show_exp.jsp page.

**List of Experiments**

![Figure 5.5: show_exp.jsp]

<table>
<thead>
<tr>
<th>Experiment Name</th>
<th>Experiment Description</th>
<th>Experiment Result</th>
<th>Graphical Plots</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-09-15T16_25_56</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>2008-09-15T11_55_15</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>2008-09-15T09_33_33</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>2008-09-15T09_05_53</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>2008-09-15T00_05_15</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>2008-09-15T00_04_09</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>2008-09-15T23_27_48</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>2008-09-15T21_29_54</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>2008-09-15T17_46_52</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>xmark</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>2008-09-15T14_58_53</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>2008-09-15T13_18_05</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>2008-09-15T10_24_02</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>2008-09-15T08_55_10</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
<tr>
<td>2008-09-14T23_18_54</td>
<td>Description</td>
<td>Result</td>
<td>Graphical plots</td>
</tr>
</tbody>
</table>

Figure 5.5 shows show_exp.jsp page, this page lists all the experiments that have been submitted in past. This page also navigates to other different pages from where the experiment description, experiment result in textual form and in graphical form can be viewed.

Over the period of time, this list of all experiments is expected to increase and thus it requires some kind of filtering over the list of all experiments. Two filtering options are provided and a user can navigate to these pages by clicking on the Filter Experiment button and XQuery filter button, which is provided on top of the list on show_exp.jsp page. Filter Experiment will navigate to a page where set of XQuery engines, documents and queries can be chosen and it will list all those experiments in which the chosen engines, documents and queries occur. Another filtering is based on XQuery filter where user can do a general query over experiments and results.
Workflow diagram for the GUI

Figure below shows the links within the JSP pages for the benchmarking service GUI.
Chapter 6

Results

6.1 Performance Results

The graphical plot obtained after running the TPC-X benchmark for 100 users and 100 products document over all XQuery engines is following:

![Graphical plot for total execution time over multiple XQuery engines for TPC-X queries.](image)

*Figure 6.1: Graphical plot for total execution time over multiple XQuery engines for TPC-X queries.*
Last two figures (figure 6.1 and 6.2) give the total execution time for each TPC-X query. There are many more graphs generated by XCheck, based on document processing time, parsing time, etc.
The graphical plot for the XMark benchmark is as shown below for the 10MB data.

![Graphical plot for total execution time over multiple XQuery engines for XMark.](image)

**Figure 6.3:** Graphical plot for total execution time over multiple XQuery engines for XMark.
6.2 Benchmarking Service Results

The XQuery benchmarking service can be used to test a dataset and queries against a set of XQuery engines. The benchmarking service stresses all aspects of XQuery and not just storage and path navigation. The front end of the benchmarking service can be used by anyone to submit an experiment, or to upload documents and queries and then submitting an experiment. Web-based front end allows user to interact with the benchmarking service.

To use benchmarking service no setup effort is required from the user, and it is easy to use web-based front end GUI that can be used by anyone to submit an experiment. In addition, interested parties can contribute their queries and documents by making use of the provided front-end. The results produced after running an experiment enables the user to understand the comparison between different engines clearly.
Currently, benchmarking service supports the following:

XQuery engines supported:
- SaxonB
- SaxonA
- MonetDB
- Xqilla
- BerkeleyDB
- Zorba
- MXQuery
- eXist
- Sedna

Documents provided:
- TPC-X (document size 366KB, 3.6MB, 35.9MB, 179.4MB, 359.1MB)
- XMark (document size 10MB, 100MB)
- TPoX (document size 100MB)

Queries provided:
- All TPC-X benchmark queries
- XMark queries
- TPoX queries
Chapter 7

Future Work

Adding more and more XQuery engines and XML document and queries is one of the future works that can make the benchmarking service more robust. The interested parties can add XML documents and queries by making use of the GUI. However, to add more XQuery engines, it is required to have it installed on the machine running the benchmark and an XCheck adapter to run the XQuery engine from the XCheck.

Another most important future work is to make this benchmarking service public, so that many people can make use of this benchmarking service and can contribute more documents and queries to the benchmarking service. Also, making benchmarking service public would enable us to make Benchmarking service more robust by putting it in more complex situations.
Chapter A

Appendix – TPC-X Queries

1. Inserting a new Product

Query

```xml
declare variable $product :=
<book >
  <title >Bla bla </title >
  <price >84.41 </price >
  <reviews >
  </reviews >
  <items >
    <item >HG G34 </item >
    <item >HG G34 </item >
    <item >HG G34 </item >
    <item >HG G34 </item >
  </items >
  <thirdPartyItems >
  </thirdPartyItems >
  <author >
    <prename >Kouichi </prename >
    <surname >Fraysse </surname >
  </author >
  <isbn >6846518710 </isbn >
  <genre >romance </genre >
  <genre >thriller </genre >
</book >;

(: Gets number of inserted products constant :)
declare variable $nr := /store /properties /nrOfProducts ;

( (: Adds Id :) 
do insert <a id="{fn: concat ('p', $nr +1) }">/ @id first into $product
,

(: Inserts product :) 
do insert $product into /store /products
```
( Updates the number of inserted products constant :) 
do replace value of $nr with $nr + 1
)

Test Query

/ store / products /*[@id eq $product / @id ]

<table>
<thead>
<tr>
<th>Description</th>
<th>Adds a new product to the store.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>newProduct</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath update</td>
</tr>
<tr>
<td>Expected Results</td>
<td>newProduct100100.xdm</td>
</tr>
</tbody>
</table>

2. Deleting a Product

Query

import module namespace bm = "http://www.mxquery.org/bm" at "randomFunctions.xq";

declare variable $product := bm: randomProduct (/ store );

( (: deletes dependent history entries :) 
do delete / store / users / user / history / entry [ @productId eq $product / @id ]
  , (: deletes dependent ( current ) shopping cart entries :) 
do delete / store / users / user / curShoppingCart / entry [ @productId eq $product / @id ]
  , (: deletes dependent ancient shopping cart entries :) 
do delete / store / users / user / ancShoppingCarts / cart / entry [ @productId eq $product / @id ]
  , (: deletes information about dependent bought third party items :) 
do delete / store / users / user / ancShoppingCarts / cart / thirdPartyItem [ @productId eq $product / @id ]
  , (: deletes dependent wish list entries :)
do delete / store / users / user / wishLists / wishList / entry [ @productId eq $product /@id ]
,
(: deletes all special offers which contain the deleted product :)
do delete / store / specialOffers / specialOffer [ product / @productId = $product / @id ]
,
(: deletes dependent third party items inclusive the entries in the current shopping carts :)
for $thirdPartyItem in $product / thirdPartyItems / thirdPartyItem
return
do delete / store / users / user / curShoppingCart / thirdPartyEntry [ @thirdPartyId eq $thirdPartyItem /@id ]
,
(: deletes product :)
do delete $product
)

Test Query

$product

<table>
<thead>
<tr>
<th>Description</th>
<th>Deletes a random product (incl. its dependencies, e.g. entries in wish list, history, special offer, etc.) from the store.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>deleteProduct</td>
</tr>
<tr>
<td>Notes</td>
<td>Path of imported module must be adapted.</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath update module</td>
</tr>
<tr>
<td>Expected Results</td>
<td>deleteProduct100100.xdm</td>
</tr>
</tbody>
</table>
3. Gains per Month (XQueryP Version)

Query

decclare execution sequential;

decclare variable $startYear := 2006;
decclare variable $endYear := 2007;

(: sequence to save the gains for every month :)=
decclare variable $values := for $i in 1 to (( $endYear - $startYear + 1) * 12)
    return 0;
decclare variable $index;

for $cart in / store / users / user / ancShoppingCarts / cart
let $year := fn:year -from - dateTime(xs: dateTime ($cart / @created ))
let $month := fn:month -from - dateTime(xs: dateTime ($cart / @created ))
return {
    (: Adding gains from the current entry into '£values ' :) 
decclare $index := ($year - 2006) *12 + $month;
for $entry in $cart / entry
let $product := / store / products /*[@id eq $entry / @productId ]
return
    do replace $values[ $index ] with $product / price * data ($entry / @quantity ) +
        $values[ $index ]
};

set $index := 0;
for $year in $startYear to $endYear
for $month in 1 to 12
return {
    set $index := $index + 1;
    <gains year="{$year }" month="{$month }">{$values[ $index ]} </gains >
}

<table>
<thead>
<tr>
<th>Description</th>
<th>Computes the overall gains per month.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>gainsPerMonthXqueryp</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath flwor xqueryp join</td>
</tr>
<tr>
<td>Expected Results</td>
<td>gainsPerMonthXqueryp100100.xdm</td>
</tr>
</tbody>
</table>
4. Sold Together

**Query**

```xml
fn: subsequence ( 
  for $product in / store / products /*
  let $count := fn: count (/ store / users / user [ ancShoppingCarts / cart / entry / @productId = $product / @id ])
  order by $count descending
  return <product id="{ $product / @id }">
    soldTogether="{ $count }"/
  ,1 ,10)
```

<table>
<thead>
<tr>
<th>Description</th>
<th>Products which have been bought by the most different users (top 10).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>soldTogether</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath flwor ordering join</td>
</tr>
<tr>
<td>Expected Results</td>
<td>soldTogether100100.xdm</td>
</tr>
</tbody>
</table>

5. Customer Segments (XQueryP Version)

**Query**

```xml
import schema default element namespace "" at " file :/ media / ubuntudata / work / benchmark / trunk / report / data / XML_schema . xsd ";

declare execution sequential ;

declare variable $userInfos ;

 (: Computes for every user the amount of bouth books , musics , dvds , games and overall products :) 

set $userInfos :=
  for $user in / store / users / user
  return {
    declare $booksBought := 0, $musicBought := 0, $dvdsBought := 0,
$videoGamesBought := 0;
for $entry in $user / ancShoppingCarts / cart / entry
let $product := / store / products /*[@id eq $entry / @productId ]
return
typeswitch ($product )
case element (*, tBook ) return
set $booksBought := $booksBought + 1
case element (*, tMusic ) return
set $musicBought := $musicBought + 1
case element (*, tDvd ) return
set $dvdsBought := $dvdsBought + 1
case element (*, tVideoGame ) return
set $videoGamesBought := $videoGamesBought + 1
default return
();
</user id="{ $user /@id }">
<books >{$booksBought }</books >
<musics>{$musicBought }</musics >
<dvds>{$dvdsBought }</dvds >
<games>{$videoGamesBought}</games >
</all >{$booksBought + $musicBought + $dvdsBought + $videoGamesBought }}</all >
</user >

(: Sorts the users by bought books :)
let $sortedByBooks :=
for $user in $userInfos
order by $user / books descending
return <user id="{ $user /@id }">{ $user / books }</user >

(: Sorts the users by bought musics :)
let $sortedByMusics :=
for $user in $userInfos
order by $user / musics descending
return <user id="{ $user /@id }">{ $user / musics }</user >

(: Sorts the users by bought dvds :)
let $sortedByDvds :=
for $user in $userInfos
order by $user / dvds descending
return <user id="{ $user /@id }">{ $user / dvds }</user >

(: Sorts the users by bought games :)
let $sortedByGames :=
for $user in $userInfos
order by $user / games descending
return <user id="{ $user /@id }">{ $user / games }</user >
(let $sortedByAll :=
  for $user in $userInfos
    order by xs: integer ($user / all) descending
    return <user id="{$user / @id}" >{$user / all </user >
  return

(: Returns the top 10 from every category :) 
(  
  <topTenBookBuyers>{
    fn: subsequence ($sortedByBooks , 1,10)
  }<topTenBookBuyers >
  ,
  <topTenMusicBuyers>{
    fn: subsequence ($sortedByMusics , 1,10)
  }<topTenMusicBuyers >
  ,
  <topTenDvdBuyers>{
    fn: subsequence ($sortedByDvds , 1,10)
  }<topTenDvdBuyers >
  ,
  <topTenGameBuyers>{
    fn: subsequence ($sortedByGames , 1,10)
  }<topTenGameBuyers >
  ,
  <topTenBuyers>{
    fn: subsequence ($sortedByAll , 1,10)
  }<topTenBuyers >
  )

Description | Computes the top 10 book, music, dvd, game buyers and the top 10 overall buyers.  
Identifier | customerSegmentsXqueryp  
Notes | Path of imported XML schema must be adapted.  
Categories | xpath schema flwor xqueryp join  
Expected Results | customerSegmentsXqueryp100100.xdm
6. Forecast

Query

let $to := xs: dateTime ("2007-12-31T23:59:59")
let $from := fn: current - dateTime () - xs: yearMonthDuration ("P6M")

for $product in /store/products/*
let $soldCount := count (/store/users/user/ancShoppingCarts/cart[xs: dateTime (fn: data (@created)) ge $from and xs: dateTime (fn: data (@created)) le $to]/entry[
   @productId eq $product/@id])
let $stockCount := count ($product/items/item)
return
<stocked productId="{ $product/@id }">
   if ($soldCount eq 0) then
      (: returns ’inf’ when nothing was sold :) 
      "inf"
   else
      fn: round ($stockCount div ($soldCount div 6))
   </stocked>

Description
Computes how long the stock of each product will suffice with the average sold rate of the last six months.

Identifier
forecast

Categories
xpath flwor join

Expected Results
forecast100100.xdm

7. Strongly wished Products

Query

declare variable $store := /;

declare variable $countUsers := count ($store/store/users/user);
declare variable $countProducts := count ($store/store/products/*);
declare variable $weightsUsers := for $i in 1 to $countUsers return 1 div
$\text{countUsers} ;$
declare\text{variable}\ \$\text{weightsProds} := \text{for}\ \$i\ \text{in}\ 1\ \text{to}\ \text{countProducts}\ \text{return}\ 0;$

(\text{: normalization of a list of numbers :})
declare\text{function}\ \text{local} : \text{normalize}\ (\$\text{list} ) \{
\text{let} \ \$\text{sum} := \text{sum} (\$\text{list} )
\text{for} \ \$i\ \text{in}\ \$\text{list}
\text{return}
\ \$i\ \text{div}\ \$\text{sum}
\};

(\text{: Computes new product weights :)}
declare\text{function}\ \text{local} : \text{newWP}\ (\$\text{nr} ,\ \$\text{weightsUsers} ,\ \$\text{weightsProds} ) \{
\text{if}\ (\ \$\text{nr}\ \text{gt}\ 0)\ \text{then}
\text{let} \ \$\text{weightsProds}2 :=
\text{for} \ \$i\ \text{in}\ 1\ \text{to}\ \text{countProducts}
\text{let} \ \$\text{product} := \text{store} / \text{store} / \text{products} /*[\$i]
\text{return}
\ \text{sum} (\text{for}\ \$j\ \text{in}\ 1\ \text{to}\ \text{countUsers}
\text{let} \ \$\text{user} := \text{store} / \text{store} / \text{users} / \text{user} [\$j]
\text{return}
\ \text{if}\ (\ \$\text{user} / \text{wishLists} / \text{wishList} / \text{entry} / \text{@productId} = \$\text{product} / \text{@id} )\ \text{then}
\ \ \ $\text{weightsUsers} [\$i]
\ \text{else}
\ \ \ ()
\)
\text{return}
\text{local} : \text{newWU}\ (\$\text{nr} - 1,\ \$\text{weightsUsers} ,\ \text{local} : \text{normalize}\ (\$\text{weightsProds}2 ))
\text{else}
\ \$\text{weightsProds}
\};

(\text{: Computes new user weights :)}
declare\text{function}\ \text{local} : \text{newWU}\ (\$\text{nr} ,\ \$\text{weightsUsers} ,\ \$\text{weightsProds} ) \{
\text{let} \ \$\text{weightsUsers}2 :=
\text{for} \ \$i\ \text{in}\ 1\ \text{to}\ \text{countUsers}
\text{let} \ \$\text{user} := \text{store} / \text{store} / \text{users} / \text{user} [\$i]
\text{return}
\ \text{sum} (\text{for}\ \$j\ \text{in}\ 1\ \text{to}\ \text{countProducts}
\text{let} \ \$\text{product} := \text{store} / \text{store} / \text{products} /*[\$j]
\text{return}
if( $user / wishLists / wishList / entry / @productId = $product / @id ) then
    $weightsProds [$i]
else
    ()
);

local : newWP ($nr , $weightsUsers2 , $weightsProds )
};

(: Gets the 3 products with the highest weights :)
let $sortedIndex :=
    for $i at $j in local : newWP (10 , $weightsUsers , $weightsProds )
        order by $i descending
        return $j
for $i in subsequence ( $sortedIndex ,1,3)
return
    $store / store / products /*[ $i]

| Description | Computes the three products which have the biggest weights in the wish lists. This done by saving two lists. One for the user weights and one for product weights. The user weight are initialized with a default value (1 / nr of users). Then, the product weights are computed with the user weights and the entries in the wish lists (a product weight is the sum of the weights of the users which have the product in its wish list). Then the weights of the users are computed again the same way (a user weight is the sum of the weights of the product of its wish list) and the values are normalized. This algorithm is repeated recursively several times. |
| Identifier | stronglyWishedProducts |
| Categories | xpath flwor recursion |
| Expected Results | stronglyWishedProducts100100.xdm |
8. Different Buyers

**Query**

```xml
fn: subsequence ( 
  for $product in / store / products /*
  let $count := fn: count (/ store / users / user [ ancShoppingCarts / cart / entry / @productId = $product / @id ])
  order by $count descending
  return <product id="{ $product / @id }" buyers="{ $count }">
,1 ,10)
```

<table>
<thead>
<tr>
<th>Description</th>
<th>Products which have been bought by the most different users (top 10).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>differentBuyers</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath flwor ordering join</td>
</tr>
<tr>
<td>Expected Results</td>
<td>differentBuyers100100.xdm</td>
</tr>
</tbody>
</table>

9. Product Statistics

**Query**

```xml
(: Compared time intervals :)
declare variable $intervals := (< int start = "2006 -01 -01 " end = "2006 -06 -30 "/>,<int start ="2006 -07 -01 " end = "2006 -12 -31 "/>,<int start ="2007 -01 -01 " end = "2007 -06 -30 "/>,<int start ="2007 -07 -01 " end = "2007 -12 -31 "/>);

(: Iteration through all products :) 
for $product in / store / products /* 
(: Sold items of the current product :) 
let $sold := 
  for $entry in / store / users / user / ancShoppingCarts / cart / entry [ @productId eq $product / @id ]
  return <entry created ="{ $entry /../ @created }" >{ data ($entry / @quantity )}</ entry > 
return 
<product id="{ $product / @id }"> { 
```
let $sums :=
   (: Getting the quantity for each interval :) 
for $int in $intervals
   let $cur := $sold [xs: date (xs: dateTime ( @created )) ge xs: date ( $int / @start
      ) and xs: date (xs: dateTime ( @created )) le xs: date ( $int / @end )]
   return
      fn:sum ( $cur )
for $i in 1 to fn: count ( $intervals ) -1
return
   (: Printing :) 
   if ( $sums [$i] eq 0) then
      if ( $sums [$i +1] - $sums [$i] eq 0) then
         "0%"
      else if ( $sums [$i +1] - $sums [$i] gt 0) then
         "+ inf %"
      else
         "-inf %"
   else
      fn: concat (fn: round (( $sums [$i +1] - $sums [$i ])) div $sums [$i ] * 100),"
         
%")
} </ product >

<table>
<thead>
<tr>
<th>Description</th>
<th>Computes the changing of sales figures for each product.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>productStatistics</td>
</tr>
<tr>
<td>Notes</td>
<td>Optimized XQueryP version can be found under ‘productStatisticsXqueryp.xq’.</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath flwor join</td>
</tr>
<tr>
<td>Expected Results</td>
<td>productStatistics100100.xdm</td>
</tr>
</tbody>
</table>
10. Gains per Month

Query

declare variable $startYear := 2006;
declare variable $endYear := 2007;

for $year in $startYear to $endYear
for $month in 1 to 12
return
  let $prices :=
    for $entry in / store / users / user / ancShoppingCarts / cart [fn:year -from - dateTime (@created ) eq $year and fn:month -from - dateTime ( @created ) eq $month ]/ entry
  let $product := / store / products /*[@id eq $entry / @productId ]
  return
    $entry / @quantity * $product / price

return
  if (fn: count ( $prices )) then
    <gains year ="{ $year }" month ="{ $month }">{ fn: sum ( $prices ) }
  </gains >
  else
    <gains year ="{ $year }" month ="{ $month }">0</gains >

<table>
<thead>
<tr>
<th><strong>Description</strong></th>
<th>Computes the overall gains per month.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Identifier</strong></td>
<td>gainsPerMonth</td>
</tr>
<tr>
<td><strong>Notes</strong></td>
<td>Optimized XQueryP version can be found under ‘gainsPerMonthXqueryp.xq’.</td>
</tr>
<tr>
<td><strong>Categories</strong></td>
<td>xpath flwor join</td>
</tr>
<tr>
<td><strong>Expected Results</strong></td>
<td>gainsPerMonth100100.xdm</td>
</tr>
</tbody>
</table>
11. Customer Segments

Query

!(: Schema Import :)
import schema default element namespace "" at " file :/ media / ubuntudata / work / benchmark / trunk / report / data / XML_schema.xsd ";

!(: Computes for every user the amount of both books, musics, dvds, games and overall products :)
let $userInfos :=
 for $user in / store / users / user

let $booksBought := fn: sum ( 
 for $entry in $user / ancShoppingCarts / cart / entry
 let $product := / store / products /*[@id eq $entry / @productId ]
 return
 typeswitch ( $product )
 case element (*, tBook ) return
 data ( $entry / @quantity )
 default return
 ()
 )

let $musicBought := fn: sum ( 
 for $entry in $user / ancShoppingCarts / cart / entry
 let $product := / store / products /*[@id eq $entry / @productId ]
 return
 typeswitch ( $product )
 case element (*, tMusic ) return
 data ( $entry / @quantity )
 default return
 ()
 )

let $dvdsBought := fn: sum ( 
 for $entry in $user / ancShoppingCarts / cart / entry
 let $product := / store / products /*[@id eq $entry / @productId ]
 return
 typeswitch ( $product )
 case element (*, tDvd ) return
 data ( $entry / @quantity )
 default return
()
let $videoGamesBought := fn:sum(
    for $entry in $user / ancShoppingCarts / cart / entry
    let $product := / store / products /*[@id eq $entry / @productId ]
    return
typeswitch ( $product )
case element (*, tVideoGame ) return
    data ( $entry / @quantity )
default return
    ()
)

return
<user id="{ $user /@id }">
    <books>{$booksBought }</books>
    <music>{$musicBought }</music>
    <dvds>{$dvdsBought }</dvds>
    <games>{$videoGamesBought }
    <all>{$booksBought + $musicBought + $dvdsBought + $videoGamesBought }
</user >

(: Sorts the users by bought books :)
let $sortedByBooks :=
    for $user in $userInfos
        order by $user / books descending
    return <user id="{ $user /@id }">{$user / books }</user >

(: Sorts the users by bought musics :)
let $sortedByMusics :=
    for $user in $userInfos
        order by $user / musics descending
    return <user id="{ $user /@id }">{$user / musics }

(: Sorts the users by bought dvds :)
let $sortedByDvds :=
    for $user in $userInfos
        order by $user / dvds descending
    return <user id="{ $user /@id }">{$user / dvds }

(: Sorts the users by bought games :)
let $sortedByGames :=
    for $user in $userInfos
        order by $user / games descending
    return <user id="{ $user /@id }">{$user / games }

(: Sorts the users by bought products :)

49
let $sortedByAll :=
  for $user in $userInfos
    order by xs: integer ( $user /all) descending
    return <user id="{$user /@id }" >{$user / all </user >
return (: Returns the top 10 from every category :) )
	<topTenBookBuyers >{
      fn: subsequence ( $sortedByBooks , 1 ,10)
  } </topTenBookBuyers > ,
<topTenMusicBuyers >{
      fn: subsequence ( $sortedByMusics, 1 ,10)
  } </topTenMusicBuyers > ,
<topTenDvd Buyers >{
      fn: subsequence ( $sortedByDvds, 1 ,10)
  } </topTenDvd Buyers > ,
<topTenGameBuyers >{
      fn: subsequence ( $sortedByGames, 1 ,10)
  } </topTenGameBuyers > ,
<topTenBuyers >{
      fn: subsequence ( $sortedByAll , 1 ,10)
  } </topTenBuyers >
)}

<table>
<thead>
<tr>
<th>Description</th>
<th>Computes the top 10 book, music, dvd, game buyers and the top 10 overall buyers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>customerSegments</td>
</tr>
<tr>
<td>Notes</td>
<td>Path of imported XML schema must be adapted. Simplified XQueryP version can be found under ‘customerSegmentsXqueryp.xq’.</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath schema flwor join</td>
</tr>
<tr>
<td>Expected Results</td>
<td>customerSegments100100.xdm</td>
</tr>
</tbody>
</table>
12. Gains per Product Per Month

Query

```xml
declare variable $startYear := 2006;
declare variable $endYear := 2007;

for $product in /store/products/*
let $sold :=
  for $entry in /store/users/user/ancShoppingCarts/cart/entry[@productId eq $product/@id]
  return <entry created="{ $entry/../@created }">{ data ($entry/@quantity) }</entry>
return
<product id="{ $product/@id }">
  for $year in $startYear to $endYear
  for $month in 1 to 12
    return
      let $cur := $sold[fn:year-from-dateTime(@created) eq $year and fn:month-from-dateTime(@created) eq $month]
      return
        if ($cur)
          <gains year="{ $year }", month="{ $month }">
            fn:sum ($cur) * $product/price
          </gains>
        else
          ()
    )</product
```

<table>
<thead>
<tr>
<th>Description</th>
<th>Computes the overall gains per product per month.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>gainsPerProductPerMonth</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath flwor join</td>
</tr>
<tr>
<td>Expected Results</td>
<td>gainsPerProductPerMonth100100.xdm</td>
</tr>
</tbody>
</table>
13. Parameterized Offers

Query

( : Top two item ( quantity ) on the stock :) 
let $lotOnStock :=
let $prods :=
   for $product in / store / products /*
      order by count ( $product / items / item ) descending
      return $product
   return fn: subsequence ( $prods ,1 ,2)

for $product in / store / products /*
return
( : For every product ... :) 
<product id="{ $product /@id }">
   (: Offers , based on similar bought shopping carts :) 
   <sameCart >{
      let $prods :=
         for $prod in / store / products /*[ @id ne $product / @id ]
         let $count := fn: count (/ store / users / user / ancShoppingCarts / cart / entry / @productId = $product / @id and entry / @productId = $prod / @id )
         order by $count descending
         return $prod
      for $prod in fn: subsequence ( $prods ,1 ,2)
      return
         <product id="{ $prod /@id }">
   </sameCart >
   (: Offers , based on similar buyer :) 
   <samePerson >{
      let $prods :=
         for $prod in / store / products /*[ @id ne $product / @id ]
         let $count := fn: count (/ store / users / user / ancShoppingCarts / cart / entry / @productIds = $product / @id and ancShoppingCarts / cart / entry / @productId = $prod / @id )
         order by $count descending
         return $prod
      for $prod in fn: subsequence ( $prods ,1 ,2)
      return
         <product id="{ $prod /@id }">
   </samePerson >

( : Two above combined plus the two products with the biggest inventory :)
<all >
  let $prods :=
    for $prod in / store / products /*[ @id ne $product / @id ]
    let $countSameCart := fn: count (/ store / users / user / ancShoppingCarts / cart [ entry / @productId = $product / @id and entry / @productId = $prod / @id ])
    let $countSamePers := fn: count (/ store / users / user [ ancShoppingCarts / cart / entry / @productId = $product / @id and ancShoppingCarts / cart / entry / @productId = $prod / @id ])
    order by $countSameCart + $countSamePers descending
    return $prod
  let $sub := fn: subsequence ( $prods ,1 ,2)
  return
    ( 
      for $prod in $sub
        return
          <product id="{ $prod /@id }">
            for $prod in $lotOnStock
              where $prod /@id != $sub / @id
              return
                <product id="{ $prod /@id }">
      )
</all >
</ product >

<table>
<thead>
<tr>
<th>Description</th>
<th>Additional offers for each product (based on stock, similar buyers and similar bought shopping carts).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>parameterizedOffers</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath flwor ordering join</td>
</tr>
<tr>
<td>Expected Results</td>
<td>parameterizedOffers100100.xdm</td>
</tr>
</tbody>
</table>

53
14. Recommendations

Query

(: Creates statistics over bought books :) 
let $statistics :=
    for $product in / store / products /*
    return
    <product id="{ $product / @id }">
    <same >{
        let $prods :=
            for $prod in / store / products /*[@id ne $product / @id ]
        let $count := fn: count (/ store / users / user [ ancShoppingCarts / cart /
            entry / @.productId = $product / @id and ancShoppingCarts / cart / entry
            / @productId = $prod / @id ])
        order by $count descending
        return $prod
        for $prod in fn: subsequence ( $prods ,1 ,2)
        return
        <product id="{ $prod /@id }"/>
    }</ same >
    </ product >
for $user in / store / users / user

(: Five newest history entries :) 
let $sortedHistory :=
    fn: subsequence ( 
        for $entry in $user / history / entry
        order by xs: dateTime ( $entry / @time ) descending
        return $entry
        ,1 ,5)
    return 
 (: Printing :) 
< recommendation userId="{ $user /@id }">
    for $value in fn: distinct - values ( data ( $statistics [ data (@id ) = data ( $sortedHistory / @productId ]) / same / product / @id )
        return <product id="{ $value }">
</ recommendation >
Computes for every user recommendations based on its history. It searches the products which have been mostly bought by users together with the newest entries in the user’s history list.

Identifier
recommendations

Categories
xpath flwor xqueryp join

Expected Results
productStatistics100100.xdm

15. Product Statistics (XQueryP Version)

Query

```
declare execution sequential;

!(: Compared time intervals :)
declare variable $intervals := (< int start = "2006 -01 -01 " end = "2006 -06 -30 "/>,<int start = "2006 -07 -01 " end = "2006 -12 -31 "/>,<int start = "2007 -01 -01 " end = "2007 -06 -30 "/>,<int start = "2007 -07 -01 " end = "2007 -12 -31 "/>);
declare variable $intLength := fn: count ( $intervals );

for $product in / store / products /*
return <product id="{$product /@id }">
  (: Variable to save the number of sold items of the current product in each
  affected interval :) 
declare $values := for $i in 1 to $intLength return 0;

  (: Iteration through all entries to collection the quantity of sold items :) 
  for $entry in / store / users / user / ancShoppingCarts / cart / entry [ @productId = $product /@id ]
  return 
    for $i in 1 to $intLength
    return 
      if (xs: date (xs: dateTime ( $entry /../ @created )) ge xs: date ( $intervals [$i ]/ @start ) and xs: date ( $entry /../ @created )) le xs: date ( $intervals [$i ]/ @end ) then {
        do replace $values [$i] with $values [$i] + data ( $entry / @quantity );
        break
      } else
      ();
```

(: Printing of the collected data for the current product :) 
for $i$ in 1 to $intLength - 1 
return 
  if ( $values [$i] eq 0) then 
    if ( $values [$i +1] - $values [$i] eq 0) then
      "0%"
    else if ( $values [$i +1] - $values [$i] gt 0) then
      "+ inf %"
    else
      "-inf %"
  else
    fn: concat (fn: round (( $values [$i +1] - $values [$i ] ) div $values [$i] * 
                          100), "%")
} } </ product >

<table>
<thead>
<tr>
<th>Description</th>
<th>Computes the changing of sales figures for each product.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>productStatisticsXqueryp</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath flwor xqueryp</td>
</tr>
<tr>
<td>Expected Results</td>
<td>productStatisticsXqueryp100100.xdm</td>
</tr>
</tbody>
</table>
16. Top 10

Query

fn: subsequence (  
  for $product in / store / products / *  
  let $count := fn:sum (/ store / users / user / ancShoppingCarts / cart / entry [ @productId = $product / @id ]/ @quantity )  
  order by $count descending  
  return <product id="{ $product / @id }"><sold="{ $count }">/>
,1 ,10)

<table>
<thead>
<tr>
<th>Description</th>
<th>Best sellers top 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>top10</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath flwor ordering join</td>
</tr>
<tr>
<td>Expected Results</td>
<td>top10100100.xdm</td>
</tr>
</tbody>
</table>

17. Add Third Party Item

Query

import module namespace bm = " http://www.mxquery.org/bm" at "/home/dagraf/data/work/benchmark/trunk/testSuite/queries/randomFunctions.xq";

declare variable $price := " 5.95 ";
declare variable $comment := " This is a comment !";

declare variable $product := bm: randomProduct (/ store);
declare variable $seller := bm: randomUser (/ store);

declare variable $nr := / store / properties / nrOfThirdPartyItems :

(: Checks if the passed ids exist , else error :)  
if ( $product and $seller ) then
  do insert
<thirdPartyItem id="{fn: concat ('t', $nr +1 )}" seller="{ $seller /@id }">
  <price>{ data ( $price )}</price>
  <comment>{ data ( $comment )}</comment>
</thirdPartyItem>

into $product/thirdPartyItems,

, do replace value of $nr with $nr + 1

else
  fn: error ()

Test Query

$product

<table>
<thead>
<tr>
<th>Description</th>
<th>inserts a third party item.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>addThirdParty</td>
</tr>
<tr>
<td>Notes</td>
<td>Path of imported module must be adapted.</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath update module</td>
</tr>
<tr>
<td>Expected Results</td>
<td>addThirdParty100100.xdm</td>
</tr>
</tbody>
</table>

18. Remove Third Party Item

Query

(
  (: First, deleting third party items in current shopping carts :)
  do delete /store/users/user/curShoppingCart/thirdPartyEntry [ @thirdPartyItemId eq $thirdPartyItem /@id ]
  ,
  (: Deleting third party item :)
  do delete $thirdPartyItem
)
Test Query

$user / curShoppingCart

<table>
<thead>
<tr>
<th>Description</th>
<th>Removes a third party item inclusive all referencing entries.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>removeThirdParty</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath update</td>
</tr>
<tr>
<td>Expected Results</td>
<td>removeThirdParty100100.xdm</td>
</tr>
</tbody>
</table>

19. Delete Item from Cart

Query

```xml
import module namespace bm = "http://www.mxquery.org/bm" at "/home/dagraf/data/work/benchmark/trunk/testSuite/queries/randomFunctions.xq";

declare variable $user := bm: randomUser (/ store );

(: getting random entry :)
 (: 
  let £rand := fn : round (fn: count ( £user / curShoppingCart / entry ) * mxq : random ()) 
  let £entry := £user / curShoppingCart / entry [xs : integer ( £rand )] 
  :) 
  let $entry := $user / curShoppingCart / entry [1] 
  return 
  if ( not ( exists ( $entry ))) then 
    () 
  else if ( count ( $user / curShoppingCart / entry ) eq 1) then 
    do delete $user / curShoppingCart
  else 
    do delete $entry
```

Test Query

$user / curShoppingCart
<table>
<thead>
<tr>
<th>Description</th>
<th>Deletes a random entry from a random user in its current shopping cart.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>deleteItemFromCart</td>
</tr>
<tr>
<td>Notes</td>
<td>Path of imported module must be adapted.</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath update module flwor</td>
</tr>
<tr>
<td>Expected Results</td>
<td>deleteItemFromCart100100.xdm</td>
</tr>
</tbody>
</table>

20. Add User

Query

```xml
declare variable $prename := "David";
declare variable $surname := "Graf";

declare variable $nr := / store / properties / nrOfUsers ;

(  
do insert  
  <user id="{fn: concat ('u', $nr +1 ) }">  
    <prename >{ $prename }</prename >  
    <surname >{ $surname }</surname >  
    <history />  
    <ancShoppingCarts />  
    <wishLists />  
  </user >  
  into  
    / store / users  
  ,  
do replace value of $nr with $nr + 1  
)
```

Test Query

```xml
/ store / users / user [ surname eq "Graf " and prename eq "David "]
```
21. Checkout

Query

```xml
import module namespace bm = "http://www.mxquery.org/bm" at "~/home/dagraf/data/work/benchmark/trunk/testSuite/queries/randomFunctions.xq";

declare variable $user := bm: randomUser (/ store );

(: Checks if enough items are on the stock. Not necessary for third party entries because they must reference a third party item :) )
declare function local : notAvailable ( $cart ) {
  for $entry in $cart / entry
  let $product := / store / products /*[@id eq $entry / @productId]
  return
    if (fn: count ( $product / items / item ) lt xs: integer ( $entry / @quantity )) then
      fn: true ()
    else
      ()

};

if ( local : notAvailable ( $user / curShoppingCart )) then
  (: Not enough items on the Stock => error :) )
  fn: error ()
else
  ( (: Creation of an ancient shopping cart :) )
  do insert
    <cart created ="{fn: current - dateTime ()}"/>
    {
      (}
$user / curShoppingCart / entry
,
  for $thirdPartyEntry in $user / curShoppingCart / thirdPartyEntry
  let $thirdPartyItem := / store / products /*/ thirdPartyItems / thirdPartyItem [ @id eq $thirdPartyEntry / @thirdPartyItemId ]
  return
    < thirdPartyItem seller ="{ $thirdPartyItem / @seller }" productId ="{ $thirdPartyItem /../../ @id }" >
      ( $thirdPartyItem /price , $thirdPartyItem / comment )
    </ thirdPartyItem >

) into
$user / ancShoppingCarts
,
  (: Deletes all bought items :)  
for $entry in $user / curShoppingCart / entry
let $product := / store / products /*[ @id eq $entry / @productId ]
for $i in 1 to xs: integer ( $entry / @quantity )
  return
    do delete $product / items / item [$i]

,  (: Deletes the bought third party items and all referencing third party entries in other shopping carts :)  
for $entry in $user / curShoppingCart / thirdPartyEntry
  return
  (  
    do delete / store / products /*/ thirdPartyItems / thirdPartyItem [ @id eq $entry / @thirdPartyItemId ]
  ,
    do delete / store / users / user / curShoppingCart / thirdPartyEntry [  
      @thirdPartyItemId eq $entry / @thirdPartyItemId ]
  )
|  (: Deletes the bought shopping cart :)  
do delete $user / curShoppingCart

Test Query

$user
22. Add Item to Cart

Query

```xml
import module namespace bm = "http://www.mxquery.org/bm" at "/home/dagraf/data/work/benchmark/trunk/testSuite/queries/randomFunctions.xq";

declare variable $user := bm: randomUser (/ store );
declare variable $product := bm: randomProduct (/ store );
declare variable $quantity := 3;
(:
declare variable $quantity := fn : round (5 * mxq : random ());:

if ( $user / curShoppingCart / entry [ @productId eq $product /@id ]) then
  let $entry := $user / curShoppingCart / entry [ @productId eq $product /@id ]
  return
    do replace value of $entry / @quantity with $entry / @quantity + $quantity
else
  if ( $user / curShoppingCart ) then
    do insert <entry quantity =$quantity productId =$product/@id />
      as first into $user / curShoppingCart
  else
    do insert <curShoppingCart >> entry quantity =$quantity productId =$product/@id />
</ curShoppingCart > before $user / ancShoppingCarts

Test Query

$user / curShoppingCart / entry [ @productId eq $product / @id ]
```

<table>
<thead>
<tr>
<th>Description</th>
<th>Pays the current shopping cart of a randomly selected user.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>checkout</td>
</tr>
<tr>
<td>Notes</td>
<td>Path of imported module must be adapted.</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath update module flwor join</td>
</tr>
<tr>
<td>Expected Results</td>
<td>checkout100100.xdm</td>
</tr>
<tr>
<td>Description</td>
<td>Inserts a random product into the current shopping cart of a random user.</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Identifier</td>
<td>addItemToCart</td>
</tr>
<tr>
<td>Notes</td>
<td>Path of imported module must be adapted.</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath update flwor module</td>
</tr>
<tr>
<td>Expected Results</td>
<td>addItemToCart100100.xdm</td>
</tr>
</tbody>
</table>

23 Random Functions Module

**Query**

```xml
module namespace bm = "http://www.mxquery.org/bm";

(: Searches random user in the store :) declare function bm: randomUser ($store) {
  let $users := $store / users / user
  return if ( exists ( $users [73456]) ) then
    $users [73456]
  else if ( exists ( $users [35978]) ) then
    $users [35978]
  else if ( exists ( $users [8617]) ) then
    $users [8617]
  else if ( exists ( $users [227]) ) then
    $users [227]
  else if ( exists ( $users [58]) ) then
    $users [57]
  else if ( exists ( $users [1]) ) then
    $users [1]
  else
    ()

(: Version with real random values ( incl. additionally implemented random function ):
  -------------------------------
  let £rand := fn: round (fn: count ($store / store / users / user) * mxq : random () ) return
  $store / store / users / user [xs : integer ( £rand )]
```

64
Declare function `bm: randomProduct ($store)` {
  let $products := $store / products /*
  return
    if ( exists ($products [68179]) ) then
      $products [68179]
    else if ( exists ($products [35979]) ) then
      $products [35979]
    else if ( exists ($products [2073]) ) then
      $products [2073]
    else if ( exists ($products [567]) ) then
      $products [567]
    else if ( exists ($products [11]) ) then
      $products [11]
    else if ( exists ($products [1]) ) then
      $products [1]
    else
      ()
  }

Version with real random values (incl. additionally implemented random function):
--------------------------------
let $rand := fn: round (fn: count ($store / store / products /*) * mxq : random ())
return
$store / store / products /*[ xs : integer ($rand )]
};

<table>
<thead>
<tr>
<th>Description</th>
<th>Module which provides function to get a random user and a random product form the store.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>randomFunctions</td>
</tr>
<tr>
<td>Expected Results</td>
<td>addUser100100.xdm</td>
</tr>
</tbody>
</table>
24 Advanced Search

Query

```
import schema default element namespace "" at " file :/ media / ubuntudata / work / benchmark / trunk / report / data / XML_schema.xsd ";

declare variable $title := "sly , brave braids must have to";

let $product := / store / products /*[ title eq $title ]
return
typeswitch ( $product )
  case element (*, tBook ) return
    " Book found !"
  case element (*, tMusic ) return
    " Video Game found !"
  case element (*, tDvd ) return
    "Dvd found !"
  case element (*, tVideoGame ) return
    " Music found !"
  default return
    " Nothing found !"
```

<table>
<thead>
<tr>
<th>Description</th>
<th>Searches with specific values from the product description (e.g. full name).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>advancedSearch</td>
</tr>
<tr>
<td>Notes</td>
<td>Path of imported XML schema must be adapted.</td>
</tr>
<tr>
<td>Categories</td>
<td>xpath flwor schema</td>
</tr>
<tr>
<td>Expected Results</td>
<td>advancedSearch100100.xdm</td>
</tr>
</tbody>
</table>
25 Keyword Search

Query

\[\text{declare variable} \ $\text{keyword external} \ ;\]

\/[ store / products /]*[ title ftcontains ( \$\text{keyword} )]
Above figure shows the simple ER diagram for the XML benchmarking service being developed. Each experiment consists of a set of engines on which benchmarking needs to be run, documents that are required to be stored on these engines and Query that are ran on these documents using the engines chosen. Following lists the schema for experiments, engines, documents and queries.
Experiment

</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]>

Engine

</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)>

Each engine that needs to be run on XCheck are required to provide an Adapter for which the schema is:

</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?,

Document

For each document that participates in benchmark has an entry in the Document meta-data file, and this metadata has following schema:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="documentEntries">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="datasets" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="dataset" minOccurs="1" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="schema" type="xs:anyURI"></xs:element>
                    <xs:element name="referenceURI" type="xs:anyURI"></xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
Query

<xs:schema>
  <xs:simpleType name="tCat">
    <xs:restriction base="xs:string">
      <xs:enumeration value="xqueryp"/>
      <xs:enumeration value="transform"/>
      <xs:enumeration value="update"/>
      <xs:enumeration value="fulltext"/>
      <xs:enumeration value="join"/>
      <xs:enumeration value="schema"/>
      <xs:enumeration value="recursion"/>
      <xs:enumeration value="flwor"/>
      <xs:enumeration value="xpath"/>
      <xs:enumeration value="module"/>
      <xs:enumeration value="ordering"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="meta">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="id" type="xs:ID"/>
        <xs:element name="descr" type="xs:string"/>
        <xs:element name="note" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="dataset" type="xs:string"/>
        <xs:element name="created" type="xs:date"/>
        <xs:element name="author" type="xs:string"/>
        <xs:element name="categories">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="cat" type="tCat" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element name="categories"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element name="meta">
</xs:schema>
<xs:complexType>
  <xs:element name="query" type="xs:string"/>
  <xs:element name="testquery" type="xs:string" minOccurs="0" />
  <xs:element name="expectedresult" type="xs:anyURI" minOccurs="0" maxOccurs="unbounded" />
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:complexType>
</xs:element>
</xs:schema>
Bibliography