2012

A framework for Java-based client server connectivity for XML

Enruo Guo
Iowa State University

A framework for Java-based client server connectivity for XML

by

Enruo Guo

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Science

Iowa State University
Ames, Iowa
2012

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

JDBC, Java Database Connectivity, is a well-known and mature technology for a Java based client to connect to a relational database on a server, execute SQL commands, and process results of queries that reside on the server. Likewise, there are a few technologies such as XQJ, IBM Cognos to support such connectivity to XML. But these technologies require the whole document to be stored in memory before it can be processed. As a result, such technologies cannot handle large XML documents. Canonical Storage for XML (abbreviated CanStoreX and also Csx), is a suite of technologies that are under varying stages of development at our lab. CsxPagination paginates an XML document of any size and stores it in ready to consume pages that are loaded into main memory as needed. CsxDOM, is a Java-based DOM API for processing XML documents and CsxXQuery, built on the top of CsxDOM is for query of XML documents. Under the research reported here, we have developed CsxJBCX, Csx Java Based Connectivity for XML, built on the top of CanStoreX, CsxDOM, and CsxXQuery. It provides API to allow a client to access the XML on a server, submit an XQuery query for execution on the server, and use CsxDOM from the client to process the result residing on the server. We have also done some testing of JBCX on CyDIW (Cyclone Database Implementation Workbench) – a command-based software development environment in our lab.

CHAPTER 1. INTRODUCTION

Extensible Markup Language (XML) provides a tree-based structure for representing information. DOM API and XPath technologies recognize and deploy the native constitution of XML that make it versatile, powerful and potentially efficient. For processing, common implementations of DOM store an XML document in a tree format in the main memory by materializing parent, child, sibling relationships between nodes. This places severe limits on the size of documents as they have to be stored in main memory before they can be processed. Furthermore, things are worse because the footprint of a document in main memory inflates several folds. In order to overcome the shortcoming of DOM API, we developed our own storage technology to store, process, and query XML documents [1,2, 3].

Our interest in XML is of dual nature. First, as an enabler for our own software development. This involves expressing software, hardware, and system configuration, metadata, catalogs, and to provide language independence to some artifacts that are either standalone or being passed as parameters. Typically such files are relatively small and memory footprint of DOM API to parse and process these documents is not a pressing issue. Therefore, for ease collaboration and sharing, these XML documents are stored as plain text in the operating system. Our other interest in XML is to deal with datasets that could be potentially very large. These datasets are either already in native XML format or deploy XML for their physical representation. Large documents cannot be processed by commonly available DOM API. In order to address these issues, we have developed a storage technology, a Canonical Storage for XML, called CanStoreX, or Csx in short, which paginates an XML and stores it as a hierarchy of pages that are ready for consumption. We have also implemented CsxDOM that brings these pages in memory as they are needed. In addition we are implementing CsxXQuery, an engine to execute XQuery queries on the top of CsxDOM.

In this thesis we report CsxJBCX, Csx Java Based Connectivity for XML, a client server architecture for XML that is a counterpart of JDBC for relational databases. In JDBC the relation resulting from a client’s SQL query resides on the server, and its tuples are streamed linearly to the client for processing. Due to linear streaming, buffer management in JDBC is rather simple. Assuming that tuples are smaller than pages, in principle one buffer at the client suffices. This is not the case for XML as what will be streamed cannot always be predicted in advance. To bring the whole document to the client is not a good idea as it is a waste of time and memory to fetch potentially large segments that are not going to be needed. Instead, Csx technology provides an elegant solution to this problem. It allows client to process large document with small amount of memory. The basic idea is simple. When processing a document residing on a local machine, when a page is needed and there is page fault the page is read from the disk.

The rest of this thesis is organized as follows. In Chapter 2, we introduce our prior work on Csx technologies CsxPagination, CsxDOM, and CsxXQuery engine. We also include a brief introduction of JDBC and TCP sockets. In Chapter 3, we discuss the details of our design and implementation of CsxJBCX. In Chapter 4, we compare our system with an existing and popular system XQJ. In Chapter 5, we run benchmarking on CyDIW and report results of some experiments. We conclude in Chapter 6. A brief introduction about XMark benchmark are included in Appendix A and in the Appendix B we include the details of CyDIW experiment for benchmarking processing speed and Java heap size, and also attach a sample log file.

CHAPTER 2. PRIOR WORK

In this Chapter, we discuss previous work our group has done to support storage and access for XML. CyDIW (Cyclone Database Implementation Workbench) – a command-based software development environment in our lab will also be introduced. A brief introduction to JDBC is included. A brief overview of TCP socket technology, which we deployed to support client-server communication in CsxJBCX, will also be presented. JBCX will be discussed in the next Chapter.

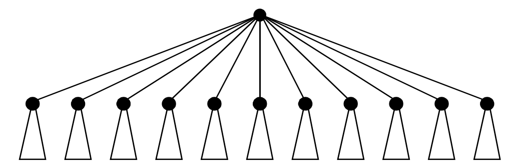
2.1. Csx technologies

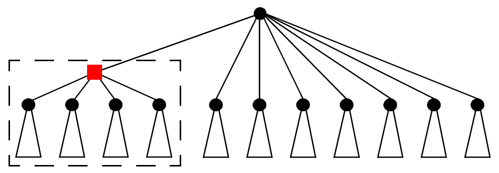
In the past a few years, our group has developed an advanced technology that provides a platform for XML document storage and access. These are under the banner of Canonical Storage for XML, CanStoreX or Csx in short. An XML document is stored in a paginated format using a CsxPagination algorithm. CsxDOM and CsxQuery are for processing and query of XML documents. These technologies are at various state of development.

2.1.1. CanStoreX

Natix partitions an XML document into small subtrees that are fitted into pages as variable length records. Therefore, it is a pagination based storage technology for XML. CanStoreX is a storage technology that paginates large XML documents and stores them as a tree of pages. Each page itself is a self-contained XML document. The physical page-based storage in CanStoreX is readily reflects the logical hierarchy of the underlying XML document. For access the document resides in a ready to use form, pages are loaded as needed.

The basic idea behind the pagination algorithm CsxPagination is shown Figure 1. Part (a) shows an XML document where root has a variable number of children. The child trees are XML elements on their own right and vary in complexity and physical size. Pagination is done recursively. The base case is when the whole document fits on a page. If this is not true, then there can be two reasons for it. First, shown in (b), is where the number of children is so large that pointers to all of them will not fit in a page. In this case, some children are grouped together and dummy parent is create to represent them, which is called _f node (f for fanout). Second, shown in (c), is when a child is too large to fit in a page. In this case, a pointer can be


(a) The base case for pagination when a document fits on a page


(b) Handling large fanout

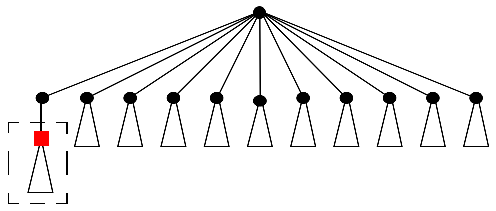

(c) Handling a large child

Figure 1. Pagination using CanStoreX

used to represent the child node, which is denoted as _c node (c for child). With the help of the two types of nodes, CanStoreX can paginate any types of XML document. However, the technology leaves the door open for future evolution where other kinds of nodes to facilitate pagination may evolve. We note that every node in the paginated document, can be addressed by a pointer (pageId, node offset), that consists of the page number containing the root of the node and the offset of the root within the page. Pagination of XML documents of up to 1 TB has been tested by CsxPagination that is scalable to larger documents

2.1.2. CsxDom

DOM (Document Object Model), as defined by W3 Consortium, is the basic technology for access of XML. It recognizes the tree-based hierarchy in XML at logical level. Broadly speaking, the implementation of DOM consists of two core interfaces Node (a subtree) and NodeList (a forest) [5]. In the current Level 3 specification one can use XPath expression to reach a Node of NodeList that makes DOM a formidable technology for access of XML. A document can also be modified. Most prevalent implementations of DOM, e.g. JDOM [6] store the whole XML document in main memory before it can be processed.

CsxDOM is an implementation of DOM that is based on Csx paginated storage. As stated above, a Node in the Csx storage can be addressed by a pointer that consists of a page number and an offset within the page on the disk. Likewise a NodeList is a list of such pointers. This offers the option that instead of materializing nodes in memory, CsxDOM can maintain pointers to nodes on the disk. This offers two-prong advantage in prudent management of memory when processing a document. First, Csx allows only the nodes that are needed to be loaded. Nodes can be very large and a given node can also be incrementally loaded depending upon what parts of the node are to be traversed. (Actually processing requires subtrees of an XML document to be traversed and nodes are also subtrees in principle the two advantages are one and the same.)

The node lists are realized as iterators. An iterator functions like a pipe where data constantly flows in a stream with regard to client's requirements. This allows pages straddled by the nodes to be brought into memory as needed. An iterator would typically contain the following methods: open() which opens the iterator and sets it up for instreaming, hasNext() which lets the client know if there is more data to be read from the iterator, getNext() which returns the next available data and close() that closes the iterator and releases the resources taken up by the iterator. Thus information could be continually read from an iterator and once an element from the iterator is processed, it is disregarded for further processing. Such iterator based stream-oriented treatment of objects is adequate and minimizes memory usage. For this reason iterators are desirable and a basic workhorse in implementation of access mechanism in databases [2,3].

In XML there is an interesting concept called axis. An axis is essentially any useful subset of nodes in an XML document together with a linearity imposed on them so that they can be efficiently processed via iterators. CsxDOM achieves the implied efficiency quite readily. W3 Consortium recommends some 13 axes to be included in implementation of DOM. Examples are children of a node, right siblings of a node, and descendants of a node linearized by their depth first traversal. Of these, all except an axis to support namespaces, are currently supported in CsxDOM. In addition, some additional iterators found useful in modularizing the code for CsxXQuery engine have also been implemented.

Besides the iterators used for returning elements based on logical hierarchy in an XML document, there exists a couple of physical native iterators which function at the storage level: DiskIterator and PageIterator. These iterators are used to instream the pages on disk sequentially to fetch the nodes contained in them and to perform page related operations such as obtaining a DOMNode given a page id and a node offset, reading the attributes associated with a specified node such as children, siblings, parent etc. Consequently, these physical iterators are invoked by almost all of the axes mentioned before and form the core of the linkage between CanStoreX and CsxDOM.

2.1.3. CsxXQuery Engine

XQuery is the query language for XML documents that is recommended by the XML Query working group of World Wide Web (W3) Consortium [7,8]. The syntax of XQuery draws a great deal from SQL used for relational database. There exist several implementations of XQuery in the industry and academia, such as XQuantum [9], Sedna [10], Berkeley DB XML [11], Qizx [15], MXQuery [16], BaseX [17], XQSharp [18], Zorba [19], MarkLogic Server [20], Timber [21]. We do not consider these engines in detail.

2.2. Cyclone database implementation workbench

Our lab has undertaken several database implementation projects. In order to organize these efforts and our needs in instruction in our courses, a common platform, called Cyclone Database Implementation Workbench (CyDIW), has been built [27]. This workbench supports implementation and intertwined usage of multiple client systems that are full-fledged command-based systems on their own right. A central GUI acts as a control center for editing and executing self contained repeatable experiments realized as batches of commands involving multiple client systems. The GUI makes CyDIW quite effective in our research projects as well as instruction. CyDIW provides useful services to client systems such as redirection of output and performance logging and reporting. CyDIW has its own page-based storage, storage and buffer managers that are shared by prototypes as client systems. As we view XML to be of infrastructural importance, CyDIW includes Csx technologies. CyDIW supports a rudimentary file system where artifacts such as relations in industry-standard binary format, indices such as B-trees, as well as Csx paginated documents can be stored. It is further enhanced by including XQuery engines as client systems that help us query and process XML documents. As an application this helps in processing XML-based log files, that can be queried to extract subsets that in turn can be reported as tables or deploy R, also is included as a client system for graphical display of experimental results.

Experiments in CyDIW can be comprehensive, self-contained, and repeatable anywhere by anyone. An experiment can create page-based storage, load data in proprietary format, view various artifacts without leaving a session, process and query data in existing database platforms (e.g. Oracle and Saxon) or newly built prototypes, view internal representations, do step by step execution, collect and collate useful information including performance logging, and report experimental results in tables or graphical format.

A client system is registered in SystemConfig.xml file, which contains information such as prefix name, class path, library path, and workspace path. Each client system needs to provide a client adapter which will be used as a bridge between CyDIW platform and the client system. When a batch of commands is sent to the main parser of CyDIW, it processes each command. With the help of clients manager, it delivers to either CyDIW execution engine or to a specific client execution engine according to the prefix of the command. Also, a client system in CyDIW can implement its own custom logging.

In developing the client server architecture CsxJBCX is registered as a client system in CyDIW with “JBCX:>” prefix. The XML documents belonging to the server reside in the storage in CyDIW. The server can be started directly or via a command in CyDIW establishing a connection to the storage. In addition to the prefix, the registration entry for JBCX in CyDIW’s SystemCofig.xml includes class path, server host name and port number. (Concept of port number is explained below under TCP Sockets.) When Server is to initialize, it will retrieve its port number from the registration setting.

The client can be run independently of CyDIW. Alternatively, it can be initialized via a command in CyDIW. (Server and Client can have CyDIW installed on their systems.) When Client is to initialize, it will retrieve both host name and port number. A parser for JBCX client and server commands is included in the adapter for JBCX. The parser verifies the validity of the command by checking if it contains all the required parameters. Custom logging has been implemented in JBCX. The custom log facility allows us to deposit any experiment stats in the XML-based log file as a legal XML element. In our experiments, we include file name, file size, memory used by Java heap, memory used by the buffer pool in the client, and time in the log file. With the help of these facilities, we can make collate and report benchmark easily.

2.3. TCP Sockets

TCP (Transaction Control Protocol) [12] provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection, see Figure 2.

A socket is one end-point of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the destination. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes: Socket and ServerSocket implement the client and the server sides of the connection, respectively.

A server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client creates a socket with server’s machine name and port number, which will be notified to server. The client also needs to identify itself to the server so sever can bind to a local port number that it will use during this connection.

Once socket on client side is notified, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port it used when it is initialized, and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client. On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

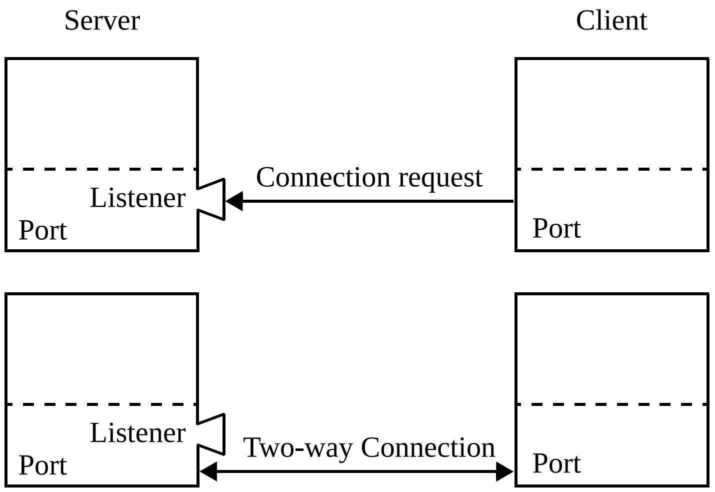


Figure 2. TCP Socket Connection between Client and Server

2.4. Java Database Connectivity (JDBC)

JDBC (Java Database Connectivity) is a Java-based API that allows a client to connect to a variety of datasources residing on servers and process them from the client. In the context of relational databases it allows a client to send SQL commands from a Java-based client for execution on the server. Thus JDBC can be seen as a hybrid of Java – an imperative general purpose query language and SQL – an algebraic language. Other hybrid languages that allow SQL statements to be executed with some general purpose programming language are also available. For example, ODBC [13] and PL/SQL [14]. PL/SQL is a hybrid of a custom-designed general-purpose programming language and SQL that eliminates the need to own a specific general purpose language.

A Driver Manager is used as a connection factory for creating JDBC connections. The API provides a mechanism for loading the relevant Java packages and registering them with the JDBC Driver Manager. JDBC is oriented towards relational databases, allowing the use of multiple driver implementations to exist and be used by an application.

To start JDBC, we need to create a Connection by calling DriverManager.getConnection()

```
Connection conn = DriverManager.getConnection (
"jdbc:somejdbcvendor:other data needed by some jdbc vendor",
   "myLogin",
   "myPassword");
```

As statement, an object is a container for an SQL statement. In JDBC, SQL statement objects are associated with connections and can be created and executed via the connections. The following shows how a statement object, a container for SQL statements, is created:

```
Statement stmt = conn.createStatement();
```

The statement to be created can be an update or a query. Perhaps the most interesting part, and certainly the most relevant part of JDBC (and other hybrid languages) that is of importance here, is that when a query is executed, the result of the query, a relation, is stored on the server; the client can pull one tuple at a time from this relation in an iterator style. Once a tuple is brought from the server to the client in its Java environment, the tuple can be processed using all the expressive power of Java. The brute expressive power offered by Java – a general purpose programming language is not available in SQL – an algebraic query language that excels in drawing complex subsets of a database with considerable ease and efficiency that cannot be paralleled by Java.

In JDBC the resulting relation is represented as a ResultSet object. Internally JDBC maintains a cursor pointing to its current row of data. The next() method moves the cursor to the next row. The next() method returns false when there are no more rows left in the ResultSet object. So it can be used to terminate the iteration through the ResultSet. The following shows how a query is executed. (For an update statement executeUpdate would be used.)

```
ResultSet rs = stmt.executeQuery("select * from Emp");
```

The following shows an outline consisting of pseudocode of how tuples are pulled from the result set and processed in the Java environment.

```
while (rs.next()) {
   store the relevant contents of the tuple rs in Java variables;
   Execute some java or SQL statements;
}
```

JDBC also allows PreparedStatement objects that are containers for templates for statements that allow SQL statements to be interspersed with “?” for latter substitution and execution. A prepared statement can be “prepared’ and executed several times. Here is an example.

```
PreparedStatement ps = conn.prepareStatement (
        “select i.*, j.* ” +
        “from Emp e, Dept d “ +
        “where e.Name = ? and d.DName = ?" );
ps.setString (1, "Alice");
ps.setInt (2, 1801);
ResultSet rs = ps.executeQuery();
```

Typically, one would substitute “?” with concrete values; however there are no such requirement – the substitutions can consist of arbitrary strings as long as the result is a legal SQL statement. This makes prepared statements highly versatile. However, from a technology point of view, it amounts to simple string substitution – and nothing more.

As stated above the most interesting part of JDBC (or any hybrid language) is that the result of a query, a relation, is stored on the database server, and it is instreamed by the user one tuple at a time. Not that such instreaming is always linear. This linearity makes JDBC very simple. It is easily supported by an iterator that serves one tuple at a time requiring one or more buffers to hold them on the client side. Typically a buffer has same size as a page.

XML based database, the counterpart of relational database, is built on Java + DOM API. DOM API is needed to recognize the tree-based structure of an XML document natively. Compared to relational databases, the processing in XML is far more complex. An application will hop from one node to another in the XML tree, possibly even cyclically. The sequence in which the XML document will be processed is utterly unpredictable in general. With traditional implementation of DOM API, where nodes reside in a highly intertwined manner, such non-linearity makes the client server architecture more difficult to realize. However, in CanStoreX this is a simple matter in principle. When a document is brought to process on the same machine where it resides, if there is page fault, the needed page is read from the local disk to the pool of buffers. In principle this is very easy to replicate in a client server architecture where the document resides on the server and it is processed by the client. The client can maintain a pool of buffers and when it is processing on page fault, the page is pulled from the server instead of from the local disk. This idea forms the basis for client server architecture in CanStoreX.

CHAPTER 3. CSXJBCX CONCEPT AND ITS IMPLEMENTATION

In this Chapter we give our view of the client server architecture for XML. Then we go on to describe its implementation of CsxJBCX. In the next Chapter we will compare CsxJBCX with XQJ.

Although relational databases require a database system to store data, XML can be stored on any computer in text format. In this respect XML is intrinsically a more ubiquitous technology compared to relational databases. XML is also far more versatile than relational databases for representation of a variety of information. XML documents are more self contained than their relational counterpart. Thus whereas in relational databases one thinks of executing a SQL query to draw a subset of the database and then process it as a client in client-server configuration, in XML there is also a need to process XML documents stored on a system directly. Of course one is also interested in processing XML documents resulting from XQuery queries. But a user needs DOM API-based technology to process XML documents in order to harness their potential. From a user’s perspective, there is no difference between XML documents, that reside on a stand alone machine, on a client, or on the server – including those XML documents that are computed on the server side on behalf of a client. In every case the client needs to assume that the document can be processed as if it is stored on the local machine where processing is taking place. In other words, once the existence of a document is known, the users should be able to process it as if the document resides on a local machine without encountering any additional seams when in a client server mode.

Because of the pagination technology, a small number of buffers in main memory are needed on the machine where document is to be processed. While processing, when CsxDOM internally encounters a page fault, the needed page is automatically loaded from the local storage or fetched from the server via a connection. The user need not be exposed to this difference. To enable this, in principle one buffer suffices on the server side. Now we proceed to describe the methodology that is deployed in CsxJBCX.

A load command is implemented so that a preexisting document on the server or one computed by a query can be connected to via one uniform mechanism. The load command associates a default number of buffers for documents residing on the server needing processing from the client. This default is set to 10. A polymorphic call also allows the number of buffers to be specified. This allows experimentation for performance. It also helps us circumvent an urgency to implement elaborate drivers such as those used in JDBC. Other than the number of buffers the user is not aware of the underlying pagination detail of CsxDOM. Thus, in order to adapt CsxDOM to CsxJBCX, no elaborate changes are needed except the ability to create a connection, and when needed send a query for execution.

A corresponding XML connectivity framework can become more difficult in technologies where the whole document that needs processing has to be loaded in memory (that too in an expanded form). In such an environment it becomes difficult for the server to detach the needed fragments (nodes) and outstream them to the client. Perhaps this is the reason that when in a client server mode, XQJ seems to prefer the user to think in terms of specific axis ahead of time that is to be processed. Once the server knows an axis, incremental serving becomes relatively easier to manage. This creates a difference between processing a document residing locally versus the one that resides on a server.

Currently, our system hasn’t implemented a prepared statement functionality yet.

After having detailed our view of XML connectivity we describe implementation of CsxJBCX on the server and client sides on top of existing Csx technologies: Csx storage, CsxDOM, and CsxXQuery. These technologies have been described in Sections 2.1, 2.2.

3.1. Server side

We assume that the storage resides on the server side and also that the documents to be processed reside on the server side. We assume that the storage has already been created and that it has some documents that can be proceed readily by the client or the documents to be processed are first computed by queries submitted to the server’s XQuery engine by the client.

An instance of ServerSocket is created to bind it to an given port number. A method called WaitforConnetion() will wait infinitely until a connection is requested from the other end. Then a new thread will be created to handle an incoming request from a client to establish a connection. Once the connection is confirmed, a client can send messages and receive responses from the server. There are a variety of messages that client could sent and they are described below under the section that provides client side details. Once the client has done its request, it will close the connection, and the thread will die. Then server continues to process other connections and wait for new connections.

Communication Handler: a class to manage connections from client, which is the core of the system. It is thread-based so that it can support multiple client systems with a thread associated with each client. The communication is through a TCP socket. So all the request and reply messages are in bytes. The bytes have to be interpreted based upon a type of request. Since there are several types of requests from clients, a unique tag is associated to identify the type of request. The handler has to parse the message tag name and handle it accordingly. Note that the client knows the type of request and when it receives the result from the server no tagging is needed to interpret the byte sequence received. Some methods defined in Communication Handler class are shown below:

· execQuery(String query) takes the requested query as an input, and calls XQuery engine to execute it. Currently the CsxXQuery engine writes an the document output of a query in plain XML and does not paginate it. But CsxDOM requires a document to be processed to be in paginated form. Therefore, pagination is invoked to convert file to.bxml file with a unique name. The execQuery() function returns the.bxml file name in bytes. One would think that the name of the document should be implicit. But there are good reasons why a unique name will be computed. We do not want to make a distinction between documents already residing on the server and those that are computed by queries. Moreover, we want a computed document to reside on the server for the sake of experimentation. A document has to be deleted explicitly; we have not implemented automatic deletions. A client needs the name of the document, whether preexisting, or computed in order to process it. The following functions have been implemented for use of the server. The needed arguments for the functions are parsed from messages received from the client.

· getRootPage(String fileName) takes the requested file name as input. After consulting the directory in the storage the id of the root page of the document is returned as a byte sequence.

· sendData(int pageId) takes the requested page id as input, and calls readPage() in Storage Manager to retrieve and return the page as a byte sequence.

· getFileSize(String fileName) takes the requested file name as input, checks the storage catalog file in the storage and return the file size in bytes.

· getDocInfo(String fileName) takes the requested file name as input, checks the storage directory and return the bytes constituting the XML-based entry in the storage directory.

3.2. Client side

When client is first initialized, it saves the host name and port number that it will use to connect to the server later. Then a startConnection() call will create a instance of Socket by using the host name and port number of the server. From then on a connection is established. Client can start communication with the server. After it finishes the communication, a closeConnection() call will be made to close the current Socket.

Client has privileges on documents in server's storage. When a query is sent to the server for execution, its output will be saved in the server's storage as a paginated.bxml file with a unique name. Then client will receive the name of the document, and start to process it. Based on the above requirements, client must have a Buffer Manager to manipulate its buffer pool, a readPage() method in Storage Manager, and CsxDOM to process the.bxml document.

To simplify our implementation, we adapt the core idea from the: StorageManagerClient in CanStoreX. In CanStoreX, when we read a page, we call readPage(pageId) which will first search the buffer pool by calling BufferManager.findPage(pageId). If the page is in there, it is read directly. Otherwise, it will read from the disk and copy it to a buffer in the buffer pool. But in client's implementation, readPage(pageId) should have a slightly different behavior, since the storage is on the server and the client doesn't have right to access it. However, reading a page should be the same when pageId is in buffer pool. The only difference is when the page is not in there. Client needs to request the page for a given page id from the server first. Server reads it from the storage and sends the page as a byte sequence, that finally client makes a copy to one available buffer determined by Buffer Manager. To process a.bxml file, i.e. to find the next node to go, CsxDOM is required. So client has a complete copy of CsxDOM. The underline readPage() method call made by the client's CsxDOM iterators has to change accordingly.

3.2.1. Messages

As stated above, the client communicates with the server by sending messages. A message is a simple XML element of the form <MessageTag>Message</MessageTag>. The Message tag uniquely identifies the type of the message and Message is a string consisting of the actual message. The message is seen as a byte sequence. On receipt the server parses the message. The server decides the action based upon the tag. Here is a list of messages and examples.

Sending a query. An element <Query> with a query is used. An example is <Query> let $auction:= doc("auctions.bxml") for $b in $auction/people/person return {$b/name/text()}</ Query>.

Request a page from the server. An element <PageID> with an integer address of a page expressed as a string is used. For example, <PageID>5032</PageID> requests Page number 5032 from the storage. Here address of a page is an integer expressed as a string. The expected return is a sequence of bytes constituting the page. In CanStoreX a page size is a storage level constant that is specified in the XML-based configuration file in the system. In our experiments we used a page size of 16 KB. Therefore, the client should get a reply consisting of a byte sequence consisting of 16KB.

Request the root of a document. An element <FileName> with name of a file is used. The expected return is a DOMNode called root. DOMNode is a data type that has been implemented by CsxDOM, which needs to be computed by server when client wants to load a file to start processing. For example, if client wants to process file with name auctions.bxml, it sends message <FileName>auctions.bxml</FileName>. The server will return the root of the auctions.bxml document as a CsxDOM node.

Request the document size. An element <FileSize> with a file name is used. The expected return is its size in storage. For example, <FileSize>auctions.bxml</FileSize> will return the size of the paginated auctions.bxml document in the storage on the server. Note that this size is different from the size of the document if it were in plain text. Although the two sizes are comparable, currently the size is not reliable in CanStoreX.

Request document information. An element <DocInfo> with the file name is used. CanStoreX directory has an XML-based entry for every file by its name. The whole XML element will be returned as a string. We caution that the format of entries in the directory in storage is likely to be enhanced in future. But it is intended that the whole element will be returned. For example, <DocInfo>auctions.bxml</DocInfo> may currently return the XML element that has document name, size and root id in the storage.

3.2.2. APIs on Client side

Here we list the APIs client can call. Note that many of the functions compose XML styled messages described above and send them to the server.

· Client (String hostName, int port) creates an instance of client class. The hostname and port are server's name and port number server is listening to, respectively. For example, if a host name is pingala and port is 8888, client can be created by Client (pingala, 8888).

· startConnection() creates a Socket object that points to a server host name and port number. For example, to start a connection after client has been initialized, simply calls client.startConnection().

· getDocumentSize (String fileName) takes file name as input. Inside the method, it sends a message with tag name <FileSize>. Return is the size of document in storage.

· getDocumentInfo (String fileName) takes file name as input. Inside the method, it sends a message with tag name <DocInfo>. Return consists of all the information about the file in storage catalog file.

· execute (String query) takes a query as input. Inside the method, it sends a message with tag name <Query>. Return is the query output file name.

· loadFile (String fileName) takes file name as input. Inside the method, it sends a message with tag name <FileName>. Return is a DOMNode type of root of the file. It is always followed by calling processOutput (), which takes the root as input. For example, if client wants to read a file with name queryOutput.bxml, it calls client.loadFile(queryOutput.bxml).

· processOutput (DOMNode fileRoot) takes a DOMNode as input and use CsxDom API to process the fileRoot. Internally, it calls readPage() method and send a message with tag name <PageID>. Page number is retrieved by CsxDOM and hidden from the client. This method is the core of our implementation.

· close() closes the current Socket to disconnect the communication.

CHAPTER 4. COMPARISON WITH EXISTING XML CONNECTIVITY SYSTEMS

There are a few existing XML data sources with XQuery engine supported. IBM Cognos [23] provides basic support for XML as a data source, and IBM Cognos Virtual View Manager is a federation engine that provides ODBC access to multiple disparate data sources, which could be in various forms. Basically, it exposes these data sources as tabular structure that Cognos can query. Another example is MonetDB/XQuery engine [24]. It is a node-based relational encoding of XQuery's data model. By defining a pre-node and post-node of each single node, each node can have a distinct coordinate x and y value which will be used in XQuery. All these XML data source with XQuery engine are not a good solution, since they are trying to duplicate the relational data base query system, but don't make use of the advantage of the relax structure of XML.

This section will provide two query examples and show how XQJ and JBCX handle them. We compare the codes side by side and discuss the similarity of the two sets of codes. Then a discussion about the disadvantage and extra work by XQJ will be shown.

4.1. XQJ

XQJ [25,26] is a counterpart of JDBC for XML. It allows a client to connect to XML data sources via a Java program, prepare and issue XQuery queries, and process the results as XML. XQJ stylistically similar to JDBC. It has JDBC-like concepts such as: Datasources, Connections, Statements and Prepared Statements. At the same time, XQJ differs from JDBC in some respects, including Data model, Typing, Static context.

In XQJ, one first establishes a XQDataSource, which is a factory for XQConnection. It encapsulates all the parameters and settings needed to create a session with a specific implementation. Every XQJ driver has its own XQDataSource implementation. It means XQDataSource supports a number of implementation-specific properties. Getter and setter could be used to retrieve these properties. XQExpression object allows a client to execute XQuery expressions. XQExpression object is created in the context of an XQConnection. XQSequence is an object to hold the result of the executeQuery() method. It represents an XQuery sequence and a cursor to iterate over the sequence. The result is a sequence of nodes on the server. To navigate through an XQSequence, a next() method is used. Initially the current position is before the first item. next() moves the current position forward and returns true if there is another item to be consumed. Once all items in the sequence have been read, next() returns false. An application can use getObject() to retrieve the current item of an XQSequence as a Java object. XQJ defines a mapping for each of the XQuery item types to a Java object value. Then we can process the object by using DOM API.

An XQuery query can return a sequence of nodes, that is akin to a NodeSet in DOM, and technically not a legal XML document by itself. This situation differs from relational databases where an SQL query always returns a single relation. Conceptually a relation is a single artifact – that internally is always a linear one directional sequence of tuples. Processing a relation residing on the server requires availability of the next() function. As the client is not required to have relational database technology at their end, the only choice is that the runtime support for next() has to be provided by the server that is mediated by middle-ware, called drivers.

The situation in XML is different. XML is a lightweight technology. Storage of XML on any computer does not necessitate sophisticated technology such as a relational database system. Processing in XML requires multidirectional hopping from one node to another in an XML tree. The sequence of hops cannot be predicted in advance. (In fact it can easily be shown as an unsolvable problem in the sense that no algorithm can exist for such prediction.) A client processes an XML document using a technology such as DOM API. The client who executes a query on the server is expected to know the contents of the result of the query and how to navigate within it. The navigation can be far more complex than a monolithic one directional linear sequential hops. The navigation is best left to the client who alone knows how to do it.

Here we bring two query examples: one retrieves a sequence of nodes (a forest, akin to a nodeset in DOM), the other retrieves a single node. In the first case the support for next() across the sequence of nodes has to be provided by XQJ yielding a DOM node that can then internally be processed by the client. In the second case only one node containing the forest will be returned and the client itself will do the corresponding navigation.

Assume an application wants to query an Oracle database, and assume the server name is AuctionServer, the port number at the server is 152 and System ID (SID) which is used to uniquely identify a particular database on a system is Sys888.

Example 1. First, we consider a simple example. In this example the XML data source on the server is auctions.xml. An XQuery query on the document to be sent from the client to the server is “for $p in document(“auctions.xml/regions/*”) return $p”. The query returns a sequence of nodes on the server, one for each region. Then the document is processed by the client. Processing requires the total of quantities of items for each region to be reported. The code is shown in Figure 3.

Example 2. This example is same as Example 1, except that the query simply includes a root tag: “for $p in document(“auctions.xm/regions/*”) return <allRegions> $p </allRegions>”. The code is shown in Figure 4.

4.2. Client code in CsxJBCX

As seen in the previous section, as a part of our Csx technologies, we have implemented our own JBCX based on CanStoreX and CsxDOM. In our case we have assumed that the client will always return a legal XML document. Therefore, all the navigation will be done by the client. Thus only Example 2 is considered here. The corresponding CsxJBCX is shown in Figure 5.

We assume that the server has been set up and is ready for the client. How this is done is covered in the next section. Next, the client is started. For this we need to specify the server host name and port number that client wants to connect. Client maintains a buffer pool that is used to process any document. This buffer pool and its size also has to be initialized when client starts. The idea to expose the buffer pool to client is to allow us to conduct experiments until the JBCX technology matures. We will show the benefits of exposing buffer size to client by discussing several experiments in Chapter 5.

Then, client starts to process the result. Again, as specified in Example 2, it wants to sum up the quantity value of each region. It uses the same DOM API method to traverse the all Regions node as the example 2 of XQJ. As seen in Figures 3 and 6, the XQJ and JBCX codes are very similar.

XQJ has some additional features, such as support to serialize the query result to a SX event stream or to a file, and support prepared queries and the use of external variables in the query. In JBCX all XML documents before and after processing are intended to reside in the storage in ready to consume paginated form as bxml files. Ordinarily, there is no need to serialize documents to store in a text form. If that is necessary, CyDIW offers a copyfile command can be used to convert a bxml document into a text based xml document. There are two main difference between XQJ and JBCX. First, in JBCX some configuration settings are visible to clients - something they may not like to be bothered with. This is intentional until the JBCX technology matures a bit.

```
// Initialization
OracleDataSource xqds = new OracleDataSource();
   xqds.setServerName("AuctionsServer");
   xqds.setPortNumber(152);
   xqds.setSID("Sys888");
// Once we have access to an XQDataSource object,  we can create XQConnection:
XQConnection xqc = xqds.getConnection();
// XQExpression objects allow a client to assemble an XQuery expression for execution
// XQExpression objects are created in the context of an XQConnection.
XQExpression xqe = xqc.createExpression();
// The result of the executeQuery() method is an XQSequence in XQJ
// Compose a query and execute it.
XQSequence xqs = xqe.executeQuery(   "for $p in document("auctions.xml/regions/*" + )
                                     "return  $p" );
// An XQSequence is not a DOM nodelist; cursor-based iteration is facilitated by XQJ
// process results
org.w3c.dom.Node oneRegion;
org.w3c.dom.Node quantity;
   int sum = 0;
   while (xqs.next()) {
      // Each time a new region is returned that is casted as a DOM node
      oneRegion = (org.w3c.dom.Node)xqs.getObject();
      org.w3c.dom Node item =oneRegion.getFirstChild();
      while (oneRegion !=null ) {
         if (item !=null){
            quantity = item.getChildNodes.item(2);
            sum += Integer.ParseInt(quantity.getNodeValue().trim());
         }
         // go to the next item node in the give region node
         item = item.getNextSibling();
         }
}
```

Figure 3. XQJ client code for Example 1

Second, JBCX potentially supports documents that are arbitrarily large. It requires very little main memory. XQJ has a severe limit on document size. As XQJ official site says: “When to process large XML document, XQJ provides some binding mode which requires more care. A value is consumed during the binding process, and it stays active and valid for all subsequent execution cycles. However, this has to be done carefully. A better implementation and improvement is needed to simplify large document process.”[25] The main difference between XQJ and csx based JBCX is that the latter would support documents

```
// initialization.  Same as Figure 3
OracleDataSource xqds = new OracleDataSource();
   xqds.setServerName("AuctionsServer");
   xqds.setPortNumber(152);
   xqds.setSID("Sys888");
XQConnection xqc = xqds.getConnection();
XQExpression xqe = xqc.createExpression();
// Compose a query and execute it.
XQSequence xqs = xqe.executeQuery(   “for $p in document(“auctions.xml/regions/*” + )
                                      “return <allRegions> $p </allRegions>” );
// process results
org.w3c.dom.Node allRegions;
org.w3c.dom.Node quantity;
   int sum = 0;
   while (xqs.next()) {
      allRegions = (org.w3c.dom.Node)xqs.getObject();
      org.w3c.dom.Node oneRegion= allRegions.getFirstChild();
      org.w3c.dom Node item = onerRegion.getFirstChild();
      while (oneRegion !=null ){
         if (item !=null){
            quantity = item.getChildNodes.item(2);
            sum += Integer.ParseInt(quantity.getNodeValue().trim());
         }
         // move item cursor to the next item node in the give region node.
         item = item.getNextSibling();
         // move region cursor to the next region node when the last item has bypassed
         if (item ==null){
            oneRegion = oneRegion.getNextSibling();
            // reset the item cursor to the item in current region node.
            if (oneRegion !=null)
                        item = oneRegion.getFirstChild();
         }
   }
```

Figure 4. XQJ client code for Example 2

```
// initialization. Syntax exposes some unnecessary low level details
// This is intentional to support ongoing development and experimentation
Client myClient = new Client (serverHost, serverPort);
StorageManagerClient smc = new StorageManagerClient(myClient, 4);
myClient.startConnection();
// Execute a query. Same query as in Figure 4
String queryResultName = myClient.execute(  "for $p in document("auctions.xml")/regions/*  +
                                            "return <allRegions> $p <allRegions>");
// process results.
DOMNode allRegions = myClient.loadFile(queryResultName);
org.w3c.dom.Node allRegions;
org.w3c.dom.Node quantity;
int sum = 0;
// identical codes as Figure 4
org.w3c.dom.Node oneRegion = allRegions.getFirstChild();
org.w3c.dom Node item = oneRegion.getFirstChild();
  while (oneRegion !=null ){
    if (item !=null){
      quantity = item.getChildNodes.item(2);
      sum += Integer.ParseInt(quantity.getNodeValue().trim());
    }
    item = item.getNextSibling();
    if (item ==null){
      oneRegion = oneRegion.getNextSibling();
      if (oneRegion !=null)
        item = oneRegion.getFirstChild();
    }
  }
```

Figure 5. JBCX client code for Example 2

of any size requiring very little main memory of client side (as well as server side). In our JBCX system, one buffer suffices on the server side. This is difficult in XQJ because the whole document that needs processing is loaded in an expanded tree form on the server side. This is not only extremely wasteful of memory; when a node is needed it is not clear what exactly should be brought to the client side, as pages form a physical boundary that are indispensable in computer systems. Thus, Csx technologies, based on pagination, are very appropriate for an all-around efficient realization of XML. However, one can argue that as languages XQJ and JBCX are very similar, the difference is mainly in the storage technology for XML.

CHAPTER 5. BENCHMARKING

In order to test efficiency of JBCX framework, we design some experiments and collect experiment data, for communication time, processing time and memory requirements. For benchmarking we have used documents of different sizes produced by XMark, the well-known benchmarking tool. XMark is described Appendix A. The benchmarking itself is done in CyDIW, mentioned in Section 2.2. Ordinarily, the server is on a separate machines, but to gain a more intimate understanding of processing time, a configuration where server is on the same machine as the client is also considered. The memory requirements are also broken into the memory used by the explicit use of buffers and heap memory used by Java internally.

For ease of experimentation, JBCX is realized as a client system of CyDIW. This offers the opportunity to develop commands for CyDIW to run suitable experiments. Experiments consist of suitable sequences of commands that also invoke services offered by CyDIW. The command-based JBCX client system is registered in CyDIW's SystemConfig.xml. The prefix "JBCX:>" is set aside for JBCX commands. On seeing a command with this prefix, CyDIW simply sends it to JBCX for execution via an adapter. The adapter consists of a command parser and support necessary interaction between the variable environments in CyDIW and JBCX. CyDIW offers two ways of logging performance: time and custom. The time based logging simply measures the time taken in execution of a command by a client system. The custom logging is implemented by the client system, in this case JBCX. This gives the client system the opportunity to break the performance into its constituent components based upon the internal runtime behavior of the client. A log entry can be returned to CyDIW. CyDIW provides facilities to record these entries in XML-based logs. Then by using any XQuery engine (e.g. Saxon) the log can be queried for preparing desired reports. The log can be displayed in form of tables by executing XQuery queries. The well known statistical package R is also available and deployed for graphical display.

5.1. CyDIW commands to support experimentation in JBCX

Here we provide a summary of the commands implemented for JBCX as a client of CyDIW. Some useful existing commands in CyDIW are also included.

Creation and use of Storage. If the storage doesn't exist on the server side, it can be created by providing specifications in an XML-based configuration file and use CyDIW's CreateStorage command. The size of storage, ranging from a few megabytes to well into terabytes can be specified. A page size can also be specified. If the storage already exists, CreateStorage command will be skipped. The number of buffers to be set aside in a CyDIW session can also be specified. Note that this does not concern us. This is a pool of buffers that can be used by multiple client systems in CyDIW. Here are the command:

```
JBCX:> CreateStorage StorageConfig.xml;
```

Binding storage to the JBCX server. The command below creates an instance of ServerSocket binding to a port, which is specified in the SystemConfig.xml file. The useStorage command is called by binding the existing storage with the server. The number of buffers to be used by the server are also specified.

```
JBCX:> InitializeServer;
JBCX:> useStorage StorageConfig.xml 4;
```

Creation and deletion of XMark documents. Server can create an XMark document by specifying its name and size. Currently no general-purpose facility for deletion of documents is available in CyDIW. However bxml documents, including XMark documents can be deleted. A directory of all the files, including those that belong to other client systems is available. Sample commands are shown below.

```
JBCX:> CreateFile auctions1.bxml 1;
JBCX:> DeleteFile auctions1.bxml;
JBCX:> ShowDirectory;
```

Starting the server. Ordinarily one assumes that server already has the needed XML documents. If not, the commands above can be used to set things up. One also assumes that the server is always in a ready state for the client. If not, the server can be started with the following command. Once the command is executed the server waits for connections by clients.

```
JBCX:> StartServer;
```

Once a server is available for connections, a client can be initialized by creating an instance of Socket which use server host name and port number specified in SystemConfig.xml file. This Socket will be notified by server. If any data is sent through this Socket, server will handle it accordingly. Also, the number of buffers in client's buffer pool should be specified. The size of each buffer is the same as the page size that is stored in server, and client use the same buffer management technology in CyDIW.

```
JBCX:> InitializeClient 4;
```

Ordinarily the detail about number of buffers is absorbed by drivers the user is not concerned about it. However, JBCX is an experimental system. We want to keep it as lightweight as possible. The pagination technology in CanStoreX helps us in this venture. We intentionally expose the number of buffers in the command to support experimentation. Once the JBCX technology matures, the setting of buffers can be mitigated more judiciously by the system.

Directory service for the client. Client can check a document's information, i.e., document size, root page by calling getDocInfo, followed by document name.

```
JBCX:> getDocInfo auctions1.bxml;
```

Query execution. To make a query, client needs to declare and initialize a variable to hold the query string, and calls ExecuteQuery. In the following the variable $$myQuery is used. Currently, for ease of parsing the variables in CyDIW are prefixed with $$.

```
JBCX:>$$myQuery | let $auction:=doc("auctions1.bxml") for $b in $auction/regions return $b;
JBCX:>ExecuteQuery $$myQuery;
```

Creation of XML-based log file. To do benchmark, we declare variables and create log file by taking advantages of CyDIW facilities. The following shows how an XML-based document for collecting benchmarks is created.

```
CyDB:> declare int $$i;
CyDB:> createLog <root> logTest.xml;
```

JBCX experiments and running them from CyDIW. Several experiments have been coded in Java. An example is processMethod_1. ProcessMethod_1 is the method that is defined in Client class. Here, we go to the region node, iterate each item node, and sum the quantity value. Users can have their self-defined method to a specific file, followed by the file name and number of buffers that is going to be used in the buffer pool. There is another version for this command where the number of buffers doesn't have to be specified. The program then will use the default buffer which is declared when client is initialized. These java programs can be run from CyDIW by using the LoadAndProcess command:

```
JBCX:> LoadAndProcess processMethod_1 auctions1.bxml 10
```

or

```
JBCX:> LoadAndProcess processMethod_1 auctions1.bxml
```

Here, the experiment is run on auctions1.bxml file and 10 buffers are used in the first example.

Performance logging. The following shows how the experiment processMethod_1 can be run and performance can be logged from CyDIW. In this example processMethod_1 is run thrice with the intention to warm-up the environment. The average of the three will be computed.

```
CyDB:> createLog <root> logTest.xml;
CyDB:> declare int $$i;
CyDB:> foreach $$i in (1,2,3) {
   JBCX:>LoadAndProcess processMethod_1 auctions1.bxml 10
   log custom>> <BenchMark> logTest.xml;
}
```

Log custom is the keyword defined by CyDIW, allowing users to deposit any experimental log entry in the XML based log file. In our JBCX subsystem adapter, we have our own self-defined custom method, where we include file name, file size, Java heap used, memory size, time. An example of custom log entry is:

```
<root>
   <BenchMark>
      <System>CsxDom</System>
      <FileName>auctions1.bxml</FileName>
      <FileSize>1</FileSize>
      <HeapUsed>4700</HeapUsed>
      <MemorySize>160</MemorySize>
      <Time>125</Time>
   </BenchMark>
</root>
```

To compute the average of each entry in the log file we use a simple XQuery supported by a query engine in CyDIW (e.g. Saxon). The average result is saved to an xml document: JBCXBenchMark.xml.

```
Saxon:>
let $e1 :=doc("CyDIW_Workspace/logTest.xml")//BenchMark[FileSize=1]
return
<avg>
   <size1>
      <HeapUsed> (avg($e1/HeapUsed/text())) </HeapUsed>
      <MemorySize> (avg($e1/MemorySize/text())) </MemorySize>
      <TimePerMB> (avg($e1/Time/text())) </TimePerMB>
   </size1>
</avg>
out >> JBCXBenchMark.xml;
CyDB:> displayFile JBCXBenchMark.xml;
```

Here the XQuery is executed by the Saxon XQuery engine. The option "out >> JBCXBenchMark.xml" available in CyDIW redirects the result of the command to a mentioned destination JBCXBenchMark.xml. The next CyDIW command displays its contents.

In order to generate a graphical reports for experimental data, R [28] is registered into CyDIW to process the data. R scripts can be executed in command batch mode in CyDIW platform. An R script is prepared for CyDIW which can read data from a XML file and generate bar plots for two dimensional data.

```
R:> CMD BATCH R_Folder/JBCXR_HS.txt;
```

OS refers to operating system that is also a client in CyDIW. To open the bar plot in pdf format:

```
OS:> CyDIW_Workspace\JBCX_HS.pdf;
```

5.2. Conducting JBCX experiments from CyDIW

After having described the commands for JBCX for use in CyDIW, we show how full fledged experiments can be conducted via CyDIW.

5.2.1. Experiment 1. Starting the server and client and processing from the client

This ia a basic experiment that shows how experiments can indeed be conducted from CyDIW. The experiments consists of two parts. In the first part the server is started from a CyDIW session. In the second part, the client is started from a new CyDIW session, a query is

```
// Start the Server;
// The server has bxml documents in storage specified in StorageConfig.xml;
// Initialize server to use the port given in SystemConfig.xml;
$JBCX:> InitializeServer;
// Create an instance of Storage Manager client and a pool of buffers;
$JBCX:> useStorage StorageConfig.xml 4;
// Start server for listening;
$JBCX:> StartServer;
```

(a) Starting the server

```
// With the server running, a client executes query and processes result.
// Start the client with a specified buffer pool;
$JBCX:> InitializeClient 4;
// Declare a query string and save it to a variable;
$JBCX:> $$Query |
let $auction:=doc("auctions1.bxml")
for $b in $auction/people/person
return <E1>{$b/name/text()}</E1>;
// Send the query to the server for execution;
$JBCX:> ExecuteQuery $$Query;
/* The name of the bxml output document will be displayed on the Output Pane. Suppose the name is
queryBXML_8.bxml. Use the LoadAndProcess command to execute the pre-developed and compiled
java program processOutput_2 to process the query output queryBXML_8.bxml with 10 buffers.
*/
$JBCX:> LoadAndProcess processOutput_2 queryBXML_8.bxml 10;
// Disconnect client from the server;
$JBCX:>Disconnect;
// The server will keep running until exiting CyDIW;
```

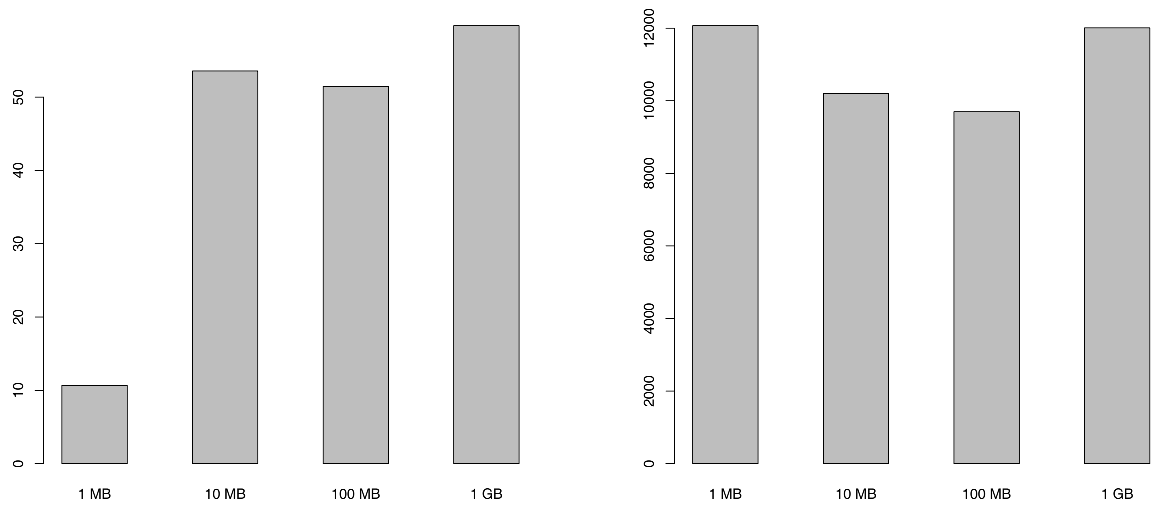
(b) Starting the client, sending a query for execution, and processing the result

Figure 6. A simple example to start server and execute query and process result on client

sent for execution on the server, and the result of the query is processed. It is also to be kept in mind that the first part of the experiment can be done remotely from any machine.

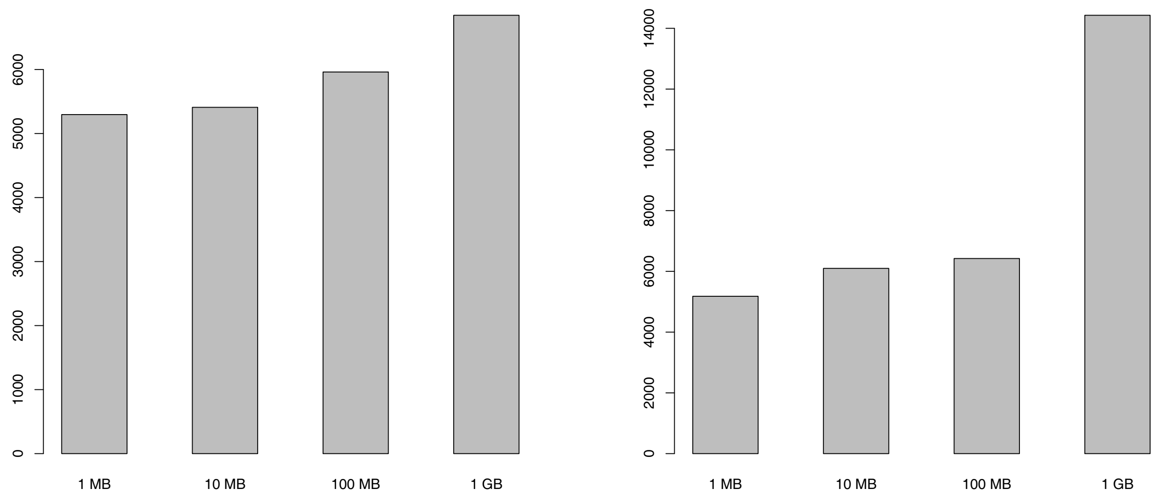
5.2.2. Experiment 2. Benchmarking processing speed and size of Java heap

In this experiment we benchmark the processing speed of a client per megabyte for different document sizes. These experiments use a specified number of buffers, 4 in our experiments, that amount to 64 KB of main memory. But JVM (Java Virtual Machine) internally uses a heap. It is difficult to control the memory used by a heap, but it is instructive to be always aware of it. These experiments are shown in an Appendix but the performance graphs are shown in Figure 7.



(i) Server and client on the same machine (ii) Server and client on different machines

(a) Processing speed in millisecond per megabyte for document of different sizes



(i) Server and client on the same machine (ii) Server and client on different machines

(b) Java heap size in kilobytes for processing documents of different sizes

Figure 7. Processing speed and memory requirements for JBCX

To conduct this experiment we created a few documents with different sizes (1MB, 10MB,100MB,1GB) on server. And a single client processed these files with a Process_Method_1, which we described in the previous chapter. A log file was created to save experimental data for files with different sizes.

We tested the experiment on two scenarios:

1. server and client on same machine

2. server and client on different machine

Since the communication is based on socket, when server and client are on same machine, it has lower network latency than where server and client are on different machine.

We measured Java heap used in KB and processing time in ms/MB, and use R to render plots. Figure 7 (a) (i) and (b) (i) are data from experiment that server and client are on same machine, whereas Figure 7 (a) (ii) and (b) (ii) show experiment results when server and client are on different machines. The required time is shown in Figure 7 (a). Figure 7 (a) (i) shows when server and client are on same machine. For document with sizes 10 MB, 100 MB and 1 GB, the required time is in the range from 50 to 60 ms/MB. However, for 1 MB document, the cost time is only 10 ms/MB. We believe it is because this document is small and is cached on the disk. We also checked document with size 4 GB and 10 GB. Their time in process is 80 ms/MB and 90 ms/MB. This trend indicates the larger the file, the longer time client needs to process it. Figure 7 (a) (ii), shows required time when server and client are on different machines. The cost time of all the documents is from 10000 ms/MB to 12000 ms/MB. The performance seems almost constant among all the document sizes. But the processing time is not in an acceptable range. The possible reason that causes the long latency and low throughput is that, the communication between server and client is based on small-sized messages. To improve the performance, we may need to allow client package several page requests at one message. Also, server should be able to response all the page requests by returning a package containing all the requested pages. Furthermore, we tried some larger file, such as 10GB, and process it with the same method: Process_Method_1. We found the performance follows the same trend as the smaller size document. It is clear that JBCX can handle very large document without any change in performance as in small document, which is encouraging.

In Figure 7 (b) (i) reports the heap size when client and server are on the same machine. For document size of 1 MB, 10 MB, 100 MB, and 1 GB, heap ranges from 5MB to 7MB. The Java heap includes start CyDIW, initialize client and create storage manager, buffer manager and allocate certain amount of buffers. We further checked the heap size used for 4 GB and 10 GB files. Both values are around 7 MB. We think the heap size is quite constant for large documents. In Figure 7 (b) (ii), when client and server are on different machines, heap size is from 5 MB to 6.5 MB on document with size from 1 MB to 100 MB. But for 1 GB document the heap size jumps to 13.7 MB. This needs to be explained in more detail and left as future work.

To get a baseline of the above experiment, we processed the documents from 1 MB to 1 GB on server side. The results shows the heap size used is from 4.5 MB to 6 MB, which doesn't show a big difference from the client-server structure. This indicates our client can have similar performance as if the data and processing were on a stand alone local machine. Total time in ms per MB is from 22 to 25 when document size range from 10 MB to 1 GB. This shows the performance can achieve 2.5 - 3 times faster when processing is on server.

5.2.3. Experiment 3. Isolation of processing time

We also did some other interesting experiments. We want to isolate the time client spent to process the document from the time server used to fetch the page from disk and send it to the client. The experiment is designed as described below and shown in Figure 8.

1. Initialize client with a large buffer size. Large means it can hold the whole file.

2. Load and process the file. Log the cost time t_1.

3. Load and process the file second time. Log the cost time t_2.

4. Compute the difference of t_1 and t_2.

We use 100 MB auctions.bxml file as an example, since 10 MB file size is not big enough, and server may have cache that could interfere the experiment. And 1 GB file is too large that will require a big amount of buffer size to hold the document. So a file of 100 MB size is a

```
/* It is assumed that the server is running. We process a 100 MB document twice with 3859 buffers. The
second time all the pages will be resident in client's buffer pool and no communication will be required.
The two times will be logged in an XML document. An XQuery will compute the difference of the two to
isolate the processing time. The resulting speed of processing will be displayed in the Output Pane.
*/
// Start the client with a pool of 3859 buffers;
$JBCX:> InitializeClient 3859;
// Create a log file for benchmarking;
$CyDB:> createLog <root> logTest.xml;
// The first run;
$JBCX:> LoadAndProcess processOutput_1 auctions100.bxml
log custom>> <BenchMark_FirstRun> logTest.xml;
// The second run;
$JBCX:> LoadAndProcess processOutput_1 auctions100.bxml
log custom>> <BenchMark_SecondRun> logTest.xml;
// Execute an XQuery to compute the time differences and show it in the Output Pane;
$Saxon:>
let $e1 :=doc("CyDIW_Workspace/logTest.xml")//BenchMark_FirstRun[FileSize=100]
let $e2 :=doc("CyDIW_Workspace/logTest.xml")//BenchMark_SecondRun[FileSize=100]
return
<Size100MB> {(($e1/Time/text())- ($e2/Time/text()) ) div 100}</Size100MB> ;
// Disconnect client from the server;
$JBCX:>Disconnect;
```

Figure 8. Isolating processing time

good compromise to measure the performance and collect experiment data. Also, consider processing file with server and client on different machines would take a huge amount of time, the experiment is chosen to take on local server. As auctions100.bxml uses 3859 pages to store region node, we initialize the client with a pool of 3859 buffers. During first processing all the needed pages end up residing in the buffer pool. During the second round the client will fetch the pages directly from the buffer pool. So t2 is the client's processing time, while t1-t2 is the network communication time plus the time server spends in reading pages from the disk.Our experiment result shows t1 is 9110 ms, t2 is 7250 ms. The difference is only 1860 ms. As we can see, client's processing time contains a large portion in the total time. There are a few reasons for this. First, network communication on local server (client and server on same machine) performs much better comparing to client-server on different machine. So the network communication time takes a very small amount of portion. Second, in order to hold the 100 MB file, we assign 3859 buffers to client's buffer pool. Although all the pages are in

```
It is assumed that the server is running with two buffers containing some specific pages. The client is
started with 1 buffer. In order to simulate processing of 100 MB document, the client makes 3859
requests for alternate pages. The server returns these pages without making any internal disk
accesses;
Start the client with a specified buffer pool
$JBCX:> InitializeClient 1;
// Create a log file;
$CyDB:> createLog <root> SocketCommunicationTime.xml;
// Execute CommunicationTest that would internally make 3859 requests;
$JBCX:> CommunicationTest 3859
log custom>> <BenchMark_Communication_3859> SocketCommunicationTime.xml;
Display the XML log file containing a single entry;
$CyDB:> displayFile CyDIW_Workspace\SocketCommunicationTime.xml;
// Disconnect client from the server;
$JBCX:>Disconnect;
```

Figure 9. Isolating network communication time

the buffer pool, to find the buffer where a page resides a linear scan is used. This explains why t2 is so large even though every page is in client's buffer pool. Although a page fault will not be encountered in our experiment, but if that were the case, finding a victim buffer would also be handled linearly. These problems need to be addressed in future.

5.2.4. Experiment 4. Isolation of network communication time

In this experiment we isolate the network communication time by designing another experiment. The experiment is designed as described below and shown in Figure 9.

Assign only one buffer on client, and more than two buffers on server. Client sends two page id request alternatively without processing. Assume the time for server to read the two pages from disk could be ignored. Once server retrieves the two pages, server can send back page content to client directly from buffer pool. However, since client has only one buffer, it has to request every time to server in order to get the page content. As an example of 100 MB auctions.bxml, it contains 3859 pages. So we set the request iteration to be 3859, and the cost time should be network communication time in 100 MB file. As our result shows, it is 1640 ms. These issues can be explored in future.

CHAPTER 6. CONCLUSION

XML is an important infrastructural technology that is instrumental in enabling other technologies. In order to harness the full potential of XML it is important that XML is implemented efficiently in terms of its footprint in main memory and runtime behavior. The main memory requirements for many implementations of DOM that store the whole document in memory place a severe limitations on the size of XML documents that can be processed. In the client server architecture this becomes even more serious as fragments of a document have to be served incrementally, but identification and separation of such fragments from the surrounding document seems a rather difficult task. For this reason pagination offers a clearer option for storing XML documents. Pages form a boundary that can be considered a necessity in any bilevel memory that consists of a faster (expensive) and a much slower (inexpensive) memory. Therefore pagination can also be viewed as a necessity and should not be viewed as a burden.

Csx technologies for XML seem promising as they are based upon storing XML in a paginated form. After having implemented Csx DOM and Csx XQuery engine, here we report implementation of Csx JBCX, the Java-based connectivity for XML documents. It seems we are able to handle documents of much larger size than XQJ. Obviously this is because Csx technologies do not require the whole document to be stored in main memory; instead they facilitate loading pages when needed.

Our implementation of Csx JBCX exposes the users to some low level details that may seem unnecessary. An example is that a user has to specify number of buffers. Also, the user needs to know the name of the document before it can be processed. This has the advantage that it allows existing documents on the server to be processed by the client. But the knowledge of the name of the document is required even when the document results from a query. Moreover, such documents are not automatically deleted after processing. Although we could circumvent these low level details, they facilitate experimentation for addressing performance issues. The low level details can be reduced or eliminated as the JBCX technology matures. In addition, in future we may also implement the prepared statement facility.

Buffer manager in client needs improvements. When a page is needed first we determine its location in the buffer pool if it is there. When it is not in the buffer pool a victim buffer has to be determined. Currently these algorithms invoke linear searches which can become expensive for a large buffer pool. Better algorithms need to be implemented. It would also be instructive to pay attention to the heap memory used by Java Virtual Machine. The Csx artifacts should ensure, as much as possible, that they make very minimal use of this memory beyond the buffer pool. This is a tricky issue as on one hand one like to use the convenience of object-oriented features offered by Java but on the other it requires one to surrender the management of memory to JVM. Intervention into memory management is much easier in languages such as C++ and C#. But then one loses the platform independence. If we continue to use Java then we may have to rely more on our own implementation of internal variables in terms of raw bytes. In any case we are mindful of this. This is why we have included experiments for monitoring the size of Java heap. We have noticed that in going from 100 MB to 1 GB document, the Java heap increases from approximately 7 MB to 14 MB. This needs attention and thoughtful resolution. We note however, that this increment seems insignificant when compared to XQJ where the corresponding memory requirements on the server side will increase from 300 MB or more to 3 GB or more. Also, what happens on the client side is more difficult to quantify.

Experiments also need to be conducted for servers located at distant locations. In addition, instead of requesting one page at a time from the server, requests for multiple pages at a time need to be considered to possibly achieve better performance. To predict the pages that are likely to be requested in near future during processing is an interesting issue. When compared to pagination in Natix [4], the Csx pages form a hierarchy that reflects the hierarchy in a base XML document. This should help in developing better predictions in JBCX.

The Csx DOM technology need to be revisited to bring it up-to-date with W3C’s Level 3 specifications. The main memory requirements need to be seriously looked into. The Csx XQuery engine is currently limited in the kinds of queries it can execute and the size of documents it can handle. This needs attention too.

APPENDIX A. THE XMARK BENCHMARK

XMark is a benchmark technology that can generate XML documents of given size. In CanStoreX implementation, it provides facilities to break XML document and store it in a paginated format. Each page itself is a self contained XML document. Once the data is stored in a paginated form as a tree, it is processed by CsxDOM, which is our version of the classical DOM API. The paginated document in CanStoreX is called .bxml document. In our experiment, we use XMark to test the capabilities and performance of our system.

The document is modelled after a database deployed by an Internet auctions site. The subtrees that form bulk of the data are item, person, open auction, closed auction and category. Items are the objects on sale or are already sold specific to different regions. Each item has a unique identifier and additional information like description, payment type etc. Persons are characterized by a unique identifier, name, e-mail address, phone number, the auctions they are interested in etc. Open auctions are auctions in progress and closed auctions are auctions that are finished. Categories are classification of items into different groups.

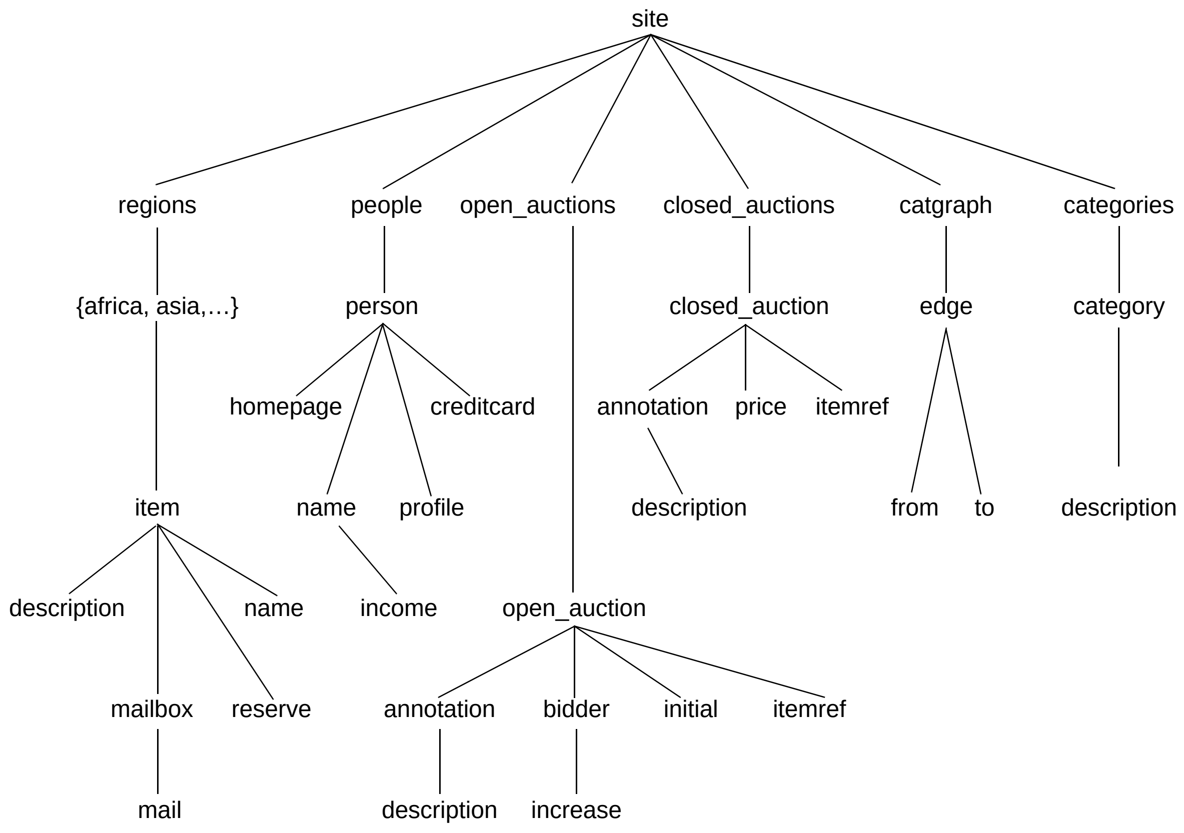


Figure 10. XMark auctions document structure

APPENDIX B. CYDIW EXPERIMENTS FOR BENCHMARKING

The experiment has been described in Section 5.2.2. In this appendix we list the code of the experiment and follow it by the listing of the XML-based log file containing the benchmark data.

6.1. The experiment

```
// Start client with 4 buffers
$JBCX:>InitializeClient 4;
// Declare variable and log file;
$CyDB:> declare int $$i;
$CyDB:> list variables;
$CyDB:> createLog <root> logTest.xml;

/*Start load and process auctions file with different sizes.
The first run is for warm-up, and the experimental data from the following 3 runs are deposited in the
log file.
*/
//Part 1: process 1MB file;
$JBCX:>LoadAndProcess processOutput_1 auctions1.bxml 4;
$CyDB:> foreach $$i in (1,2,3)
{
$JBCX:>LoadAndProcess processOutput_1 auctions1.bxml 4
 log custom>> <BenchMark> logTest.xml;
}

 //Part 2: process 10MB file;
$JBCX:>LoadAndProcess processOutput_1 auctions10.bxml 4;
$CyDB:> foreach $$i in (1,2,3)
{
$JBCX:>LoadAndProcess processOutput_1 auctions10.bxml 4
log custom>> <BenchMark> logTest.xml;
}

//Part 3: process 100MB file;
$JBCX:>LoadAndProcess processOutput_1 auctions100.bxml 4;
$CyDB:> foreach $$i in (1,2,3)
{
```

```
$JBCX:>LoadAndProcess processOutput_1 auctions100.bxml 4
log custom>> <BenchMark> logTest.xml;
}
//Part 4: process 1000MB file;
$JBCX:>LoadAndProcess processOutput_1 auctions1g.bxml 4;
$CyDB:>foreach $$i in (1,2,3)
{
$JBCX:>LoadAndProcess processOutput_1 auctions1g.bxml 4
log custom>> <BenchMark> logTest.xml;
}

//Part 5: we compute the average of each entry and send to R to make plot;
//Heap used
$Saxon:>
let $e1 :=doc("CyDIW_Workspace/logTest.xml")//BenchMark[FileSize=1]
let $e2 :=doc("CyDIW_Workspace/logTest.xml")//BenchMark[FileSize=10]
let $e3 :=doc("CyDIW_Workspace/logTest.xml")//BenchMark[FileSize=100]
let $e4 :=doc("CyDIW_Workspace/logTest.xml")//BenchMark[FileSize=1000]

return
<avg>
<HeapUsed>
<size1>{avg($e1/HeapUsed/text())} </size1>
<size10> {avg($e2/HeapUsed/text())} </size10>
<size100>{avg($e3/HeapUsed/text())} </size100>
<size1000>{avg($e4/HeapUsed/text())} </size1000>
</HeapUsed>
</avg>
out >>JBCXbenchMark_HS.xml;

$CyDB:> displayFile JBCXbenchMark_HS.xml;
$R:> CMD BATCH R_Folder/JBCXR_Heap_2.txt;
$OS:> CyDIW_Workspace\JBCX_HS_2.pdf;

//Time taken
$Saxon:>
let $e1 :=doc("CyDIW_Workspace/logTest.xml")//BenchMark[FileSize=1]
let $e2 :=doc("CyDIW_Workspace/logTest.xml")//BenchMark[FileSize=10]
let $e3 :=doc("CyDIW_Workspace/logTest.xml")//BenchMark[FileSize=100]
let $e4 :=doc("CyDIW_Workspace/logTest.xml")//BenchMark[FileSize=1000]
return
<avg>
<TimePerMB>
<size1>{avg($e1/Time/text())} </size1>
<size10> {avg($e2/Time/text()) div 10}</size10>
<size100>{avg($e3/Time/text()) div 100}</size100>
<size1000>{avg($e4/Time/text()) div 1000}</size1000>
</TimePerMB>
</avg>
out >>JBCXbenchMark_Time.xml;

$CyDB:> displayFile JBCXbenchMark_Time.xml;
$R:> CMD BATCH R_Folder/JBCXR_Time_2.txt;
$OS:> CyDIW_Workspace\JBCX_Time_2.pdf;

// Disconnect client from the server;
$JDBCX:>Disconnect;
```

6.2. Log file

This section shows a sample log file. It contains several information such as file name, file size, heap used, memory size, process time. For one document size, we run three times in order to get the average performance.

```
<root>
    <BenchMark>
      <System>CsxDom</System>
      <FileName>auctions1.bxml</FileName>
      <FileSize>1</FileSize>
      <HeapUsed>4795</HeapUsed>
      <MemorySize>64</MemorySize>
      <Time>12594</Time>
    </BenchMark>
  <!--repeat twice -->
    <BenchMark>
      <System>CsxDom</System>
      <FileName>auctions10.bxml</FileName>
```

```
        <FileSize>10</FileSize>
        <HeapUsed>6607</HeapUsed>
        <MemorySize>64</MemorySize>
        <Time>101190</Time>
    </BenchMark>
<!--repeat twice -->
    <BenchMark>
        <System>CsxDom</System>
        <FileName>auctions100.bxml</FileName>
        <FileSize>100</FileSize>
        <HeapUsed>6938</HeapUsed>
        <MemorySize>64</MemorySize>
        <Time>962093</Time>
    </BenchMark>
<!--repeat twice -->
    <BenchMark>
        <System>CsxDom</System>
        <FileName>auctions1g.bxml</FileName>
        <FileSize>1000</FileSize>
        <HeapUsed>13695</HeapUsed>
        <MemorySize>64</MemorySize>
        <Time>11728908</Time>
    </BenchMark>
<!--repeat twice -->
<!--experiment on isolation of the processing time -->
    <BenchMark_FirstRun>
        <System>CsxDom</System>
        <FileName>auctions100.bxml</FileName>
        <FileSize>100</FileSize>
        <HeapUsed>74234</HeapUsed>
        <MemorySize>61744</MemorySize>
        <Time>9110</Time>
    </BenchMark_FirstRun>
    <BenchMark_SecondRun>
        <System>CsxDom</System>
        <FileName>auctions100.bxml</FileName>
        <FileSize>100</FileSize>
        <HeapUsed>72330</HeapUsed>
        <MemorySize>61744</MemorySize>
        <Time>7250</Time>
    </BenchMark_SecondRun>
  <!--experiment on communication time by sending 3859 page requests-->
    <BenchMark_Communication_3859>
        <System>CsxDom</System>
        <FileName>NA</FileName>
        <FileSize>NA</FileSize>
        <HeapUsed>5573</HeapUsed>
        <MemorySize>16</MemorySize>
        <Time>1640</Time>
    </BenchMark_Communication_3859>
</root>
```

BIBLIOGRAPHY

[1] S. Ma, “Implementation of a Canonical Native Storage for XML, Master’s thesis, Department of Computer Science, Iowa State University, 2004.

[2] Gadia, S.K., A canonical native storage architecture for XML

[3] Krithivasan, S., 2007. Implementation of a XQuery engine for large documents in CanstoreX, Master’s Thesis, Department of Computer Science, Iowa State University, Ames, Iowa

[4] http://db.informatik.uni-mannheim.de/natixdoc/index.html

[5] http://www.w3.org/, 2012

[6] http://www.jdom.org/, 2012

[7] Boag, S. , Chamberlin, D. , Fernandez, M. F. , Florescu, D. , Robie, J. , Simeon, J. (2007). XQuery 1.0: An XML query language. Technical Report, World Wide Web Consortium, 2007. W3C recommendation 23 January 2007.

[8] Bakker, B. D. , Widarto, I. X-Hive Corporation. An Introduction to XQuery. http://www.perfec-txml.com/articles/xml/xquery.asp.

[9] CogneticSystems,Inc.XQuantum:AXMLNativeDataStore. http://www.cogneticsystems.com/

[10] Modis, I. Sedna: Native XML Database with partial support for XML Query. http:// modis.ispras.ru/sedna/index.htm.

[11] BerkeleyDB, Oracle. BerkeleyDB: An embedded XML Native Database. http://www.ora-cle.com/technology/products/berkeley-db/index.html.

[12] http://docs.oracle.com/javase/tutorial/networking/sockets/.

[13] http://msdn.microsoft.com/en-us/library/windows/desktop/ms710252(v=vs.85).aspx

[14] http://www.oracle.com/technetwork/database/features/plsql/index.html

[15] http://www.xmlmind.com/qizx/qizx.html.

[16] http://mxquery.org/, 2012.

[17] http://basex.org/, 2012

[18] http://www.xmlprime.com/xmlprime/, 2012

[19] http://www.zorba-xquery.com/, 2012

[20] http://docs.marklogic.com/, 2012

[21] http://www.eecs.umich.edu/db/timber/, 2012

[22] http://saxon.sourceforge.net/, 2012.

[23] http://www-01.ibm.com/software/analytics/cognos/, 2012

[24] http://www.monetdb.org/Documentation, 2012

[25] http://xqj.net/javadoc/, 2012.

[26] http://www.xquery.com/tutorials xqj_tutoria.

[27] X. Zhao and S.K. Gadia. *A Lightweight Workbench for Database Benchmarking, Experimentation, and Implementation*. IEEE Transactions on Knowledge and Data Engineering, Vol. 24, No. 11, pp. 1937-1949, Nov. 2012.

[28] http://www.r-project.org/, 2012.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank God Almighty for giving me a functioning body to live and learn. By sending his son Jesus Christ, I am able to live within the Holy Spirit, where all my joy, peace, faith, strength and intelligence are from.

I would like to express my deepest appreciation to my advisor, Dr. Shashi K Gadia, for his guidance, supervision, and persistent help. I also thank him for arranging financial support through this project. And a special thank to his efforts on revising the thesis. His enthusiasm for xml and database kept me constantly engaged with my research, and his personal generosity helped make my time at Ames enjoyable.

I would like to thank my committee member, Dr. Lu Ruan. Without her kindness and help, I could not have started my new career journey in computer science. Her encouragement and briefing of the academic research area guided me when I was new to this field. Also, I would like to thank Dr. Ning Fang from Chemistry Department, who is also my committee member, for his support and patience during my research and thesis.

In addition, a thank to all the members in the database group, especially Xinyuan, Sugam and Xiaofeng, for the help and time they spent with me in starting the research and finishing the thesis.

Furthermore, I take this opportunity to thank all my family members, for their trust, endless support and understandings. A special thank to my aunt, for her prayer and open a door for me to know God and believe in God.

Finally, I would like to place my sense of gratitude to everyone in the Home of Heart Fellowship, Cindy, Wei, Dafang, Senniang, Chunquan, Jing, and to all who have lent their helping hands.