2007

Implementation of a XQuery engine for large documents in CanstoreX

Srikanth Krithivasan
Iowa State University

Implementation of a XQuery engine for large documents in CanstoreX

by

Srikanth Krithivasan

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

2007

DEDICATION

I would like to dedicate this thesis to my family and friends without whose support this would have never been possible. I would like to specially dedicate this thesis to my parents and to my brother for their loving care through my various phases of life. I would also like to thank my friends for making my stay at Iowa State University a pleasant and memorable one and for their moral support and assistance during the writing of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to all those who have helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Shashi K. Gadia for his guidance, patience, support, inspired style of teaching and encouraging me to do research and come this far. I strongly believe that without his guidance and motivation which he offered during my research and the writing of this thesis, this would have never been possible. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Leslie Miller and Dr. Sree Nilakanta and also for allowing me to implement a specific module of my research as part of their course project. I would additionally like to thank Dr. Samik Basu, Dr. Pavan Aduri for their amazing style of teaching.

My acknowledgement list extends to all the people whom I met at ISU who have made my stay pleasant and memorable. My thanks to my roommates Visu and Vasanth for accepting to stay with me during our first year, bearing all the troubles that I gave them and still deciding to be my roommates for a second successive term. I would also like to thank Satya, Puvi, Atul, Hari, Richard, Santosh, Harish, Parichey, Faraz, Kavitha, Kamna for their camaraderie and for helping me during my troubled times. My lab-mates Jia, Seo-Young, Matt deserve my gratitude for assisting me with my project and for providing an ambient atmosphere for me to work.

ABSTRACT

XML is a markup language used for storing documents which contains structured information. Its flexibility helps in storing, processing and querying diverse and complex documents with any structure. While theoretically, XML could be used to handle any documents, the currently available parsers require large amounts of main-memory resulting into severe restriction on the size of XML documents. As a result, some technologies have been developed to break the XML documents in to smaller chunks and allow the parsers to load only a specific portion of the document when needed.

Two major but diagonally opposite approaches for storing an xml document on the disk have emerged. The first breaks an xml document into parent child pairs and stores them into relational storage [2, 3, 4, 5, 6]. The second approach builds a native storage for xml that attempts to directly capture xml hierarchy [7, 8, 9, 10]. Canonical Storage for XML (CanStoreX) is a native storage technology being developed by our group at Iowa State University that has been tested for pagination of xml documents up to 100 Gigabytes in size [29]. CanStoreX requires that every page is a self-contained xml document on its own right. Thus the pages themselves form an xml-like hierarchy.

XML can be used to encode a variety of data. Examples are system configuration, metadata, documents such as books, relational data, and object-oriented data. An array of technologies has developed to process xml documents. Our major interest in xml lies in the view that an xml document can be considered a database which can then be queried. There exists several query engines for xml [15, 16, 20]. Kweelt is an excellent early platform that supports the Quilt query language [31]. Quilt [18] is a preliminary query language which has subsequently been extended to XQuery, a query language that has been standardized by the W3 Consortium [17].

Quilt, the query language that Kweelt supports, is superseded by XQuery. Earlier, this issue has largely been addressed in [32]. The original Kweelt uses DOM parser; therefore it can only handle small documents. The main focus of this thesis is to deploy CanStoreX to query documents of the size of gigabytes. The resulting platform has been extensively tested.

CHAPTER 1. INTRODUCTION

This chapter provides a brief introduction to the various technologies and applications used in the development of the XQuery application. It discusses about the current XML applications prevalent in the industry, throws light on the characteristics of the same and emphasizes on the need for a new application to process huge documents. A few technologies described in the thesis are XML, the XQuery language, the Quilt language, the Kweelt platform for Quilt, the CanStoreX architecture for the storage of XML documents and the current implementation of the Kweelt platform.

1.1 Introduction to XML

XML stands for eXtensible Markup Language and is used for the digital representation of documents [1]. XML was developed more to describe data in contrast to HTML which was developed to display data. Thus, HTML is used to represent how data is presented to the end-user while XML is used to represent the contents of the data itself. Whereas HTML uses system-defined tags, in XML tags are user-defined. XML documents can range from being completely arbitrary without any predefined structure to those that conform to predefined structure. Two popular technologies to describe the structure of XML documents and to validate them are Document Definition Type (DTD) and XML Schema Document (XSD).

As with any other markup language, an application for processing an xml document utilizes a parser to serve it logical units of the xml document as needed. The two most widely used parsers are DOM (Document Object Model), a tree based parser and SAX (Simple API for XML), an event based parser. The SAX parser [36] scans the xml document serially from beginning to end and while logical units in the xml document are matched, some predefined actions are taken by the application. Although SAX parser is efficient it is not suited for dynamic conditional navigation of an xml document. DOM [35] caters to the richness of an XML document well but it requires that the whole document is loaded as a fully expressed tree in main memory. DOM requires main memory that is 5 to 10 times of the document itself. Therefore, DOM is not scalable and that is an obstacle for applications that utilize DOM API. The main memory issue of DOM has been addressed by Shihe Ma [28]. He developed a pagination algorithm for CanStoreX with the requirement that every page is a self contained xml document. The algorithm was implemented in Java. This allowed Ma to paginate documents up to 1 Gigabyte in size. In Mas representation, the xml contents of pages were stored as plain text and for parsing of pages DOM parser was invoked. Unfortunately, peculiar memory in Java became an obstacle in pushing the scalability of the pagination algorithm beyond the gigabyte range. Daniel Patanroi addressed the scalability issue by developing a binary format for the xml-pages [29]. The binary version of CanStoreX has been tested for documents up to 100 gigabytes in size ant it is expected to work for terabyte size documents. As stated above, the main focus of this thesis is to utilize CanStoreX to develop a query engine for the xml query language XQuery.

1.2 Introduction to XQuery

XQuery is the proposed query language for querying XML documents that is being recommended by the XML Query working group of World Wide Web (W3C) consortium [17, 21, 22]. XQuery uses XPath expressions to address the various nodes in a document. The syntax of XQuery is similar to that of SQL used for traditional database languages. The grammar of XQuery is publicly available [17] which has been used to develop a parser by Satyadev Nandakumar [32]. The parser currently supports most of the read operations associated with XQuery. This proved to be very much helpful in the development of XQuery application.

1.3 XQuery Implementations

There exists quite a few implementations of XQuery in the industry and academia with diverse storage technologies of XML documents. We consider only those implementations along with either a native or a non-native storage [7] for XML documents since we believe these would overcome the limitations of main-memory processing and would scale to documents of the size of terabytes. A few implementations [11, 12, 13, 14, 24] exist in the industry and a few [5, 8, 10] in the academia. Though the internal implementation of storage of XML documents is not of importance to us, these implementations could be used for a comparison of the throughput of our application.

1.4 Introduction to Kweelt/Quilt

Kweelt is a platform which offers an implementation of a query language of XML called Quilt. Kweelt is open source (GPL), completely written in Java and is easily extensible [31]. Kweelt provides an abstract representation of the nodes, tags and other entities involved with a XML document so that these could be implemented in accordance with the implementor's choice. It is further supported by xerces parser which contains both DOM and SAX implementation required to parse documents.

Quilt is a XML query language proposed by Chamberlin, Florescu and Robbie [18]. The XQuery language is built upon the syntax of Quilt and hence bears several resemblances to the Quilt semantics. The parser application for Quilt has been implemented in the Kweelt platform and has been tested to successfully query XML documents. Though the language is currently not used and has been replaced by XQuery, a prior implementation turned out to be very helpful in comprehending the nuances of XQuery and developing the query application.

1.5 Introduction to memory concepts

Any application requires a well-maintained memory organization which is critical for the performance of the application. Memory organization is ideally assumed to be bi-level though there could exist several levels too. In a bi-level memory organization, we have the main memory at the upper level which is used for processing information. This memory is ideally faster and consequently more expensive. At the lower level, we have the disk (typically a magnetic disk or tape) which is used as the secondary storage. This memory is slower and cheaper too. Most of the contemporary systems deploy caching of data at the disk level and at the main-memory level leading to several layers in the memory organization.

Since the disk memory is cheaper, information is primarily stored on the disk and transferred to the main memory as and when required. This gives rise to the concept of a page used to transfer data between the disk and main memory. The page is the physical unit of data transfer and information is exchanged in sets of pages. The disk space manager is used to maintain data on the disk in various pages and return the requested page. Since a requested page is ideally expected to be requested again in the future, the page contents are ideally stored in the main memory itself for a certain period of time. This gives rise to the concept of a buffer which is used to store the contents of a page in the main memory. A buffer manager is used to monitor the contents of the buffer and update buffers with page contents when required. Any data storage system would contain an implementation of the disk space manager and the buffer manager to read and write data from its storage.

1.6 Introduction to CanStoreX

CanStoreX was originally developed to counteract the main memory limitation of DOM parser and to allow parsing of large documents. Natix and CanStoreX, a CANonical STORagE for Xml documents, are examples of native XML Database Management System (XBMS) that break an XML document into smaller chunks each of which is a self-contained XML document in its own right. In CanStoreX, the chunks are pages in computer systems that are units of access between main memory and the disk. These pages are interlinked to each other through special nodes called f-nodes and c-nodes. These nodes preserve the structure and content of the original XML document so that the document could be retrieved completely through these nodes without any loss in content and in structure. CanStoreX offers two flavors of storage: textual page implementation, where the pages are stored in plain-text and binary page implementation, where the textual information is encoded onto binary and stored on the storage. The binary version is more scalable than the textual version due to its efficient storage ad retrieval mechanisms and hence has been adopted as the standard for parsing. The binary version in turn offers two strategies for storing pages: fixed node size strategy where the nodes have a fixed length and variable node size strategy where the nodes have variable size depending on the information to be paginated. CanStoreX has been able to successfully parse documents up-to 100 GB in size proving that it is scalable for huge documents and ideal for query processing.

1.7 Introduction to XQuery Implementation

The thesis deals with describing the XQuery implementation on the binary version of CanStoreX. Though the work resembles more of an integration of the various utilities, the core of the development involved understanding of the various architectures and building new modules to allow interaction of the utilities and properly fusing the result of one module onto another. A preliminary version of XQuery has been developed and tested on documents up-to 100 GB in size, verified for accuracy and benchmarked for throughput performance.

The rest of the thesis is organized as follows. Chapter 2 provides a brief overview of the existing Quilt implementation and Chapter 3 discusses the limitations on processing huge documents. Chapter 4 elucidates the new implementation emphasizing the ability to query large documents and other performance enhancements. Chapter 5 provides a few sample queries of the XQuery language. This is followed by a discussion on the sorting technique implemented for XML nodes exploiting the CanStoreX architecture in Chapter 6. The experimental setup and the results are discussed in Chapter 7. Finally, the thesis is concluded with the current status and the suggested future works on the model in Chapter 8.

CHAPTER 2. BACKGROUND

This chapter presents a brief overview of the existing state of XQuery and Quilt and discusses about the current implementation of Quilt on the Kweelt platform. This includes a general description of the functionalities of XQuery and Quilt language. The chapter further discusses on the challenges that exist to implement XQuery on the Kweelt platform. Besides, the chapter also discusses the implementation of CanStoreX storage and the mechanism of data retrieval.

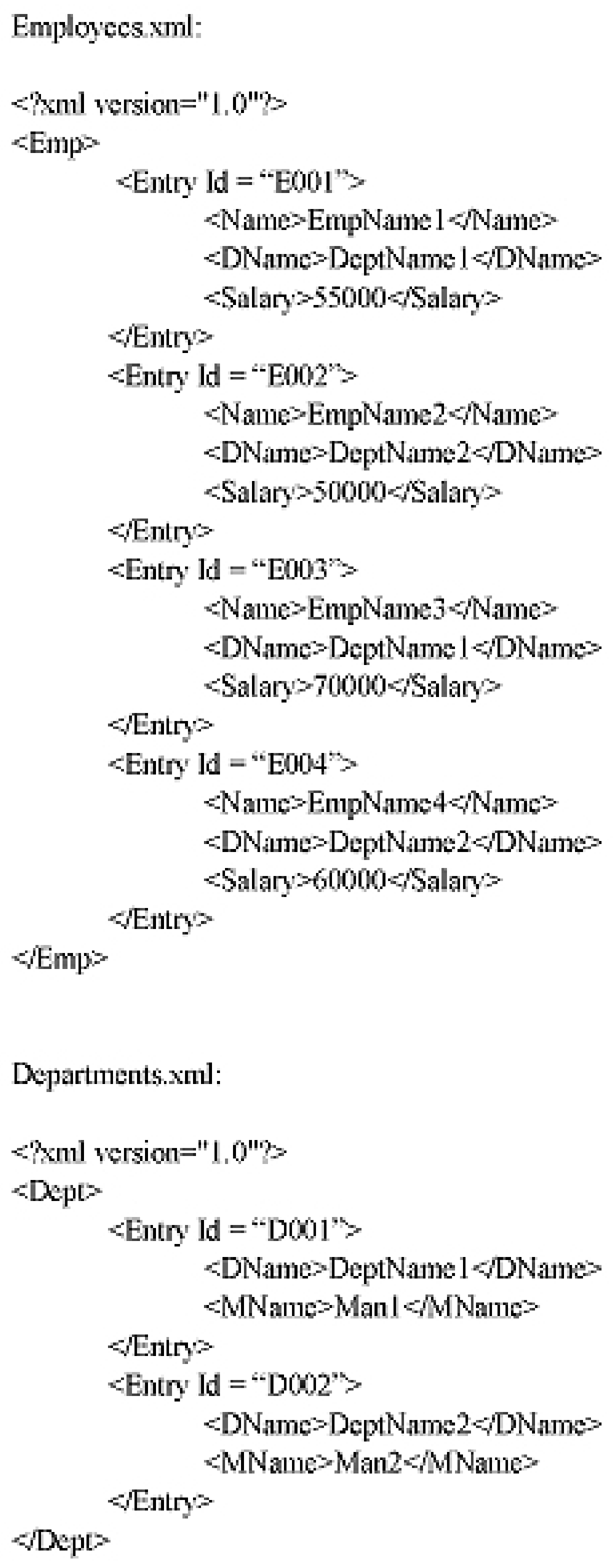
2.1 XQuery

XML arrived as a supplement for the then highly popular HTML since there arose a need to store and exchange data in some convenient format which could be recognized in a browser and by systems with diverse configurations. While XML was initially used only to transfer data between applications, apparently people came to realize that any information could be structured in a XML document regardless of how diverse and complex it was which would also help in extracting specific portions of the document. Thus specific query and retrieval methods were devised which would extract desired entities from the XML document. XQuery is one such language which has been recommended by W3C to develop query patterns on XML documents. It is currently being developed by the W3C XML Query Working Group and is derived from a XML Query language called Quilt which in turn is borrowed from several other languages such as XPath, XQL, XML-QL, SQL and OQL. XQuery is ultimately considered to provide a platform-independent, powerful and easy means to retrieve information from XML as what SQL did to traditional database systems [22].

2.1.1 The XQuery Language

Since XQuery is designed to work on XML documents composed of hierarchical tree-based structure, the data model is based on XPath expressions and defines every node to contain a well-defined label besides a parent node, sibling nodes on either side and a set of children nodes. Each of these nodes would again be recursively defined to contain the above mentioned characteristics. The advantage with this data model lies with the fact that any document or a collection of documents could be composed into one single conceptual structure and be processed uniformly [21, 23].

XQuery is a functional language consisting of quite a few operators and expressions. The following sections provide a glimpse into the syntactic and semantic structures of the language. The examples mentioned below work on the sample XML documents ”Employees.xml” and ”Departments.xml” specified in Figure 5.1.



Employees.xml:

```
<?xml version="1.0"?>
<Emp>
         <Entry Id = "E001">
                  <Name>EmpName1</Name>
                  <DName>DeptName1</DName>
                  <Salary>55000</Salary>
        </Entry>
        <Entry Id = "E002">
                  <Name>EmpName2</Name>
                  <DName>DeptName2</DName>
                  <Salary>50000</Salary>
        </Entry>
        <Entry Id = "E003">
                  <Name>EmpName3</Name>
                  <DName>DeptName1</DName>
                  <Salary>70000</Salary>
        </Entry>
        <Entry Id = "E004">
                  <Name>EmpName4</Name>
                  <DName>DeptName2</DName>
                  <Salary>60000</Salary>
        </Entry>
</Emp>
```

Departments.xml:

```
<?xml version="1.0"?>
<Dept>
        <Entry Id = "D001">
                  <DName>DeptName1</DName>
                  <MName>Man1</MName>
        </Entry>
        <Entry Id = "D002">
                  <DName>DeptName2</DName>
                  <MName>Man2</MName>
        </Entry>
</Dept>
```

Figure 2.1   XML Documents

2.1.2 Path Expressions

XPath [19, 27] is the language which is the XML standard for specifying paths in a XML document. A couple of sample path expressions are specified below.

Find the salaries of employees:

```
document("Employees.xml")//Entry/Salary
```

Find the Names of Employees working with DeptName1 and getting paid more than 60000.

```
document("Employees.xml")//Entry[DName = 'DeptName1' AND Salary >= 60000]
```

XPath contains two kinds of query patterns, expressions and predicates. Expressions are specified following a node and specify the pattern the query needs to look for in the document. A "/" pattern suggests that the user is looking only for the children of the current node and the query does not need to look at the document any further while a "//" suggests that the user is looking for descendants of the current node. Predicates act on a specific node and validate whether the node satisfies a specified criterion or criteria. The 'DName' and 'Salary' attributes specified in the above mentioned query are predicates which test whether the nodes qualify both these conditions.

2.1.3 FLWR Expressions

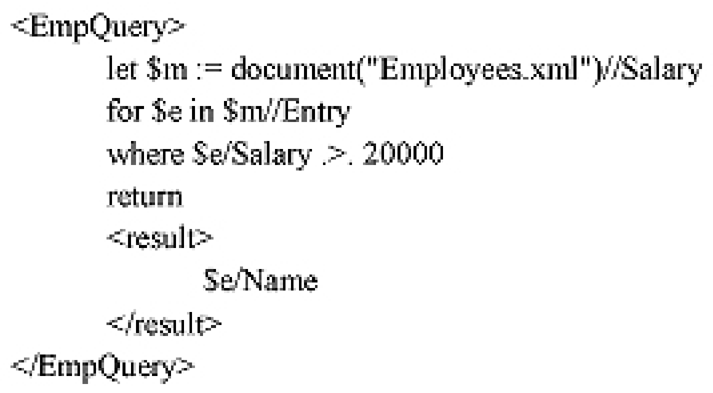
FLWR expressions constitute the core of XQuery functionality and is analogous to Select-From-Where Clause of the SQL language. A FLWR expression consists of the following clauses:

2.1.3.1 FOR-Clause

The FOR clause binds a variable to a collection of nodes so that the variable could iterate over the collection for processing the elements one at a time. The FOR clause is useful for processing individual elements.

2.1.3.2 LET-Clause

The LET clause is used to bind a variable to a collection of nodes so that the variable validates the collection a whole rather than individual elements. It is useful for aggregate operations and for comparing / evaluating sets. A sample FLWR expression is illustrated in Figure 2.2.



```
<EmpQuery>
        let $m := document("Employees.xml")//Salary
        for $e in $m//Entry
        where $e/Salary .>. 20000
        return
        <result>
                $e/Name
        </result>
</EmpQuery>
```

Figure 2.2 Sample FLWR Expression

2.1.3.3 WHERE-Clause

The WHERE clause is used in conjunction with FOR/LET clause to evaluate predicates in order to qualify or disqualify the nodes and collections of nodes under consideration. It can be viewed as a filter that returns the filtered list of nodes and collections of nodes that qualify.

2.1.3.4 RETURN-Clause

The RETURN clause is used to return elements either to the outermost query or to the output stream. This clause is used to build element constructors and is executed for each occurrence of a FOR/LET clause.

2.1.4 Operators/ Functions

XQuery provides a host of operators and functions that act on individual elements or collections and return the desired information. Like SQL, XQuery also allows the aggregate functions include sum, count, avg, max and min. Besides, it provides the regular arithmetic, relational, boolean and logical operators, conditional statements such as if-then-else, quantified expressions such as some, every, satisfies etc.

2.1.5 XQuery Grammar

The current version of XQuery's grammar is available at [17] which contains the Extended Backus-Norm Form (EBNF) of the same. The current version includes updates to XML documents as proposed in a recent proposal and hence the grammar is modified to accommodate the same. However, the current application uses only a subset of the grammar which includes the basic FLWR expressions, path expressions and a few operators and functions. As mentioned before, the extension of XQuery grammar to the Kweelt platform was undertaken and successfully completed by Nandakumar [32]. The grammar had then been implemented for a textual version of CanStoreX though using the DiskDom implemented earlier by Ma [28].

2.2 Quilt

Quilt is a query language to process XML documents derived from several other languages such as XML-QL, XPath, XQL, YATL and XSQL. Quilt was developed by Jonathan Robie, Don Chamberlin and Daniela Florescu [18] and was basically built to combine the rich flavors available in the above specified languages to design a small, implementable language to meet the requirements specified in the W3C XML working group's XML Query requirements. Quilt relies on the structure of XML document and could process queries based on simple node predicates, combining two heterogeneous documents, references, parent/child relationships, attributes etc. XQuery was developed around Quilt.

2.3 Kweelt platform

Kweelt [31] is a framework to query XML data specifically designed to provide an evaluation engine to support the Quilt language. It offers multiple back ends such as Oracle Parser, Sun Parser and comes with a built-in DOM, SAX and Wizdom combine into one Xerces package.

Further, Kweelt allows users to develop their own implementations of the Node and Node-Factory classes using the parsers thereby promoting more diverse XML storage technologies. It is open-source licensed with the GPL, completely written in Java and is quite extensible. While Kweelt is specifically built to support Quilt, it does not offer all the features provided by the Quilt language. Besides, Kweelt provides new features such as Typed referencing and dereferencing nodes in the same document (IDREFS), in-line XML, Java external functions etc.

2.3.1 Kweelt Architecture

The core modules of the Kweelt platform reside in the package xacute.quilt specifically the query parser and the query evaluator. The classes in this package extend interfaces and constants defined in xacute.common. The basic entities are represented by the Node and NodeList classes. Node refers to a single element in the XML document while NodeList refers to a collection of such nodes rooted at specific node. These are instantiated through a NodeFactory class. Kweelt provides a generic implementation of these elements and allows the user to develop his own architecture and add to it. The basic implementations are available in xacute.impl package. One implementation which Kweelt provides is that of xacute.impl.dom interface which contains a DOM based implementation and xacute.impl.xdom containing the Xerces DOM based implementation.

The Kweelt evaluation engine executes a host of expressions, the primary one being Quilt-Expression which handles the whole expression that is being passed on to the application. Several distinct expression engines include FLWRExpression handling only the FLWR expressions, FilterExpression handling the filters specified with each clause, AttributeExpression handling the attributes associated with a node etc. The results of these expressions are wrapped onto a Value object which defines a generic implementation of the result. The class is being extended by several classes to return node-specific information such as ValueString, ValueNum, ValueBool and ValueNodeList.

A context is associated with every Quilt variable to determine its scope and life-time during the evaluation of a query. Every variable is thus bound to a Context, a binding to indicate if the clause is a FOR/LET clause, the root node to evaluate the variable and any other predicate or filter information. The evaluation of the contexts are being taken care of by the EvalContext class which evaluates each context and returns a specific class of the Value node.

The creation of the nodes and the node lists is completely left to the user to develop his/her own implementation. The corresponding interfaces are accessed through an instance of the NodeFactory class which the user creates.

2.3.2 Extending the framework

The basic idea to extend the Kweelt framework is to create an instance of the parser, to parse the specified query onto a QuiltQuery object and evaluate the object. The information about the nodes and node lists would be initially determined through a NodeFactory object provided by the EvalContext. To accomplish this, the evaluator would require a handler to stream the query output. Once the handler and the evaluation context are obtained, the rest of the processing involves evaluating the context on the various bind variables and streaming the results to the handler associated with the context.

Building a new NodeFactory involves the following steps: parsing the XML document and storing it in the required format, instantiating nodes and node lists depending on the usage and offering the primitives associated with them. A Node, for instance, need to have the primitives to read the label associated with it, its children, descendants, parent, siblings, ancestors information. These need to be implemented as well for a new framework to be built upon.

2.4 CanStoreX

Besides the query parser and the evaluation engine, one other significant feature in the application is the actual storage of XML documents. It is very much apparent that the entire application could not be run just with the main-memory in hand since the documents can be very large in size. As had been discussed before, even parsing these documents could not proceed beyond documents of 10s of MB in size due to excessive main-memory consumption. CanStoreX addresses the issue of memory requirement.

2.4.1 CanStoreX Architecture

As stated before, CanStoreX [28, 29] is a native storage for XML. CanStoreX requires each page to be a legal XML document in its own right. The pages are linked to each other through a hierarchical structure thereby maintaining the relationship between the nodes so that the entire document could be reproduced or navigated without any loss in content or in structure. CanStoreX uses auxiliary nodes called f-node and c-node which are used to link pages. The f-node is used to group a sequence of siblings having the same parent. A subtree rooted at the f-node is stored on a single page. The c-node contains a pointer to a child page where a subtree rooted at a f-node resides. Pagination refers to the process of parsing through the XML document and splitting it into several pages storing each of them onto the storage space. The end result of pagination would be a page Id pointing to a page containing the root node of the document. A sample XML document and the binary page storage structure corresponding to the document are shown in Figure 2.3 and Figure 2.4.

```
<A>
    <B>
        <C>...</C>
        <D>...</D>
        <E>...</E>
    </B>
    <F>
        <G>...</G>
    </F>
    <H></H>
</A>
```

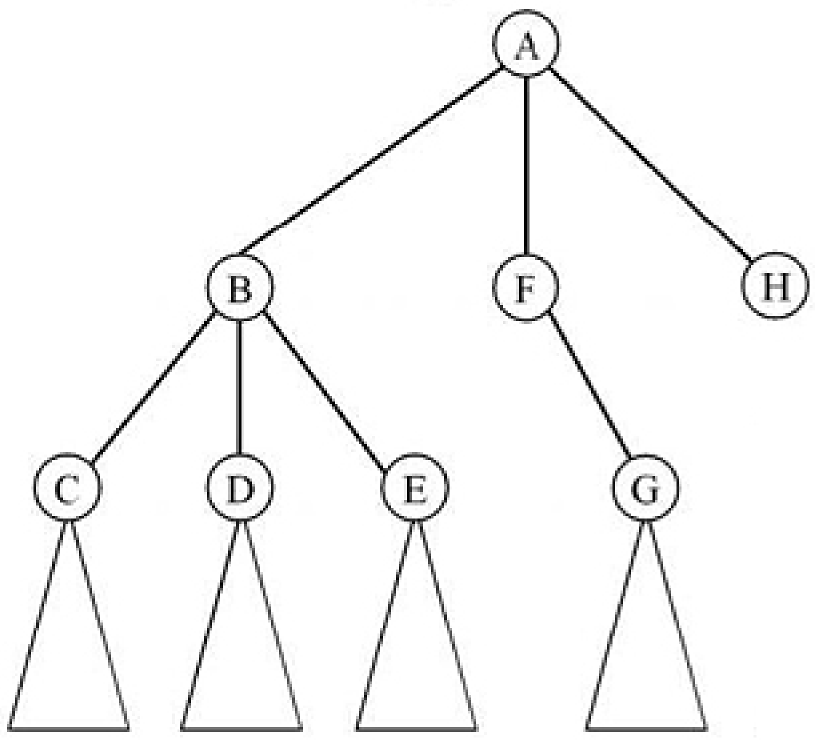


Figure 2.3 Sample XML Document

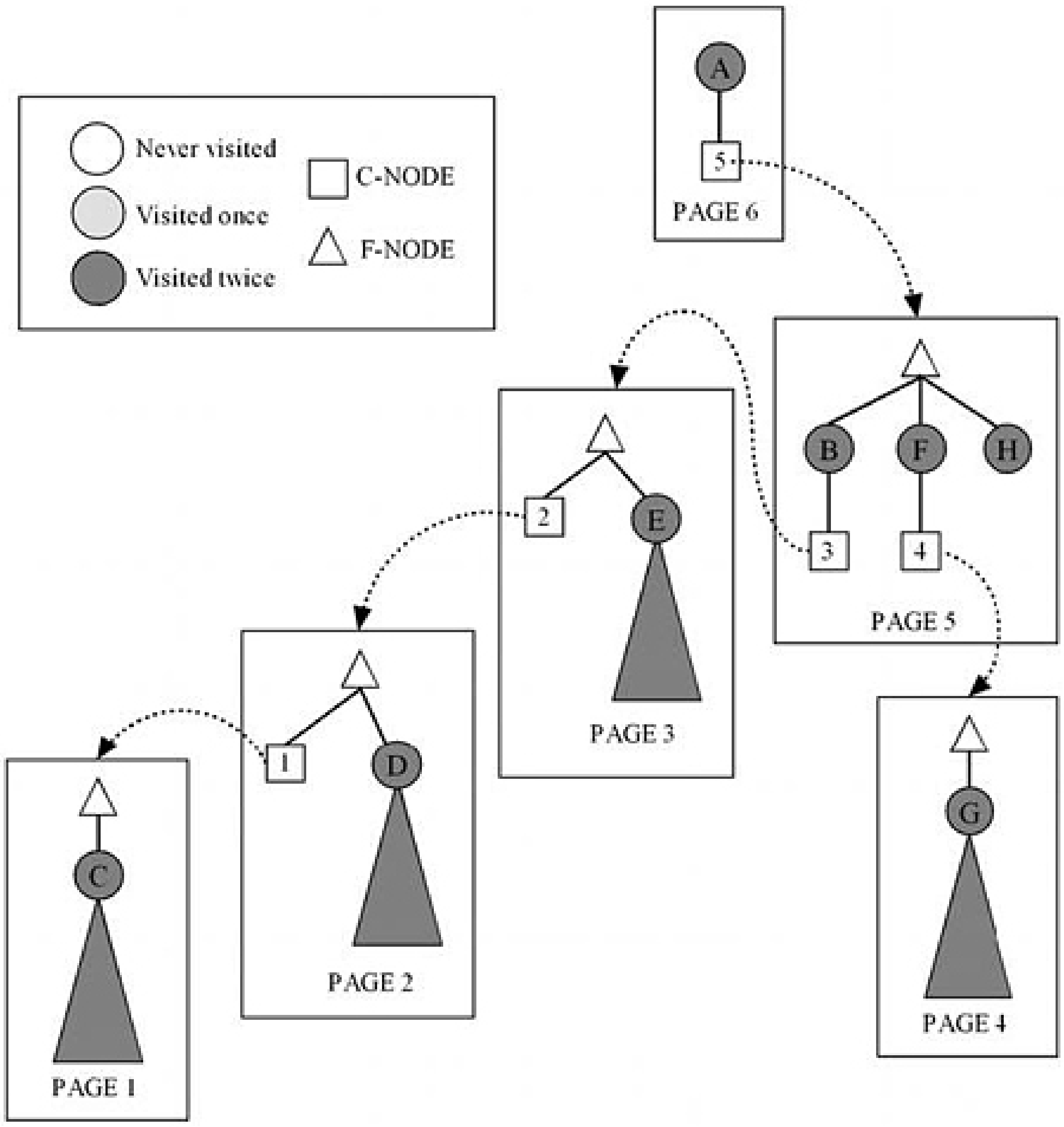


Figure 2.4   Binary Storage structure pertaining to the XML Document

CanStoreX offers two flavors of storage: a textual-page based implementation where the pages are stored in plain-text on the disk. This technique suffers from the drawback that one has to rely on some utility like DOM to parse pages on the fly. This is particularly troublesome in runtime environment in Java where pages become binary objects allocated and maintained on the heap in main memory. Mechanisms such as pinning and user-defined caching mechanisms are not available. To counteract this problem, a binary version of CanStoreX had been developed which stores the page in binary format on the disk. The advantages with this format are that the binary page does not require a parser to be loaded onto the memory. The pages are organized hierarchically in a tree-like structure that are readily navigated, thereby eliminating dependency on parsers such as DOM. CanStoreX controls the main memory usage through buffering that is is subjected to user-defined buffer replacement policies. This is expected to work well for a document that is essentially a tree of pages.

With the binary page implementation in place, CanStoreX seems to offer the complete freedom to parse documents of size in the range of terabytes. Currently, CanStoreX had been tested for documents up to 100 GB in size and had been found to successfully parse and recreate the documents.

2.5 Prior Work

This section deals with the previous work that has gone in building the XQuery application. The storage system for the application was initially developed by Ma [28]. He developed a textual version of a Canonical Storage System for XML documents where a typical XML document is split into smaller XML documents and stored in the form of plain-text on the pages. The system suffered from the drawback that the pages had to be brought into main memory and had to be converted into DOM pages on the fly which occupied an enormous space in the memory. Further usage of DOM made CanStoreX dependant on a third party DOM parser. Daniel Patanroi [29] developed a binary version of CanStoreX where the pages are stored in binary version along with the hierarchy information which removed the dependence on a DOM parser and enabled the application to scale very well.

Once the storage system has been built, the development of XQuery engine was initially undertaken was Satyadev [32]. He developed a parser for the XQuery language using the

Kweelt framework which already supported the Quilt language. A query engine to evaluate the queries was initially developed by Robert [33] in which he designed a DOM interface for the CanStoreX architecture. This was further improved by Srikanth and Matt [34] to build a full-fledged implementation of the XQuery evaluation engine.

2.6 Building the Application

With the existence of XQuery, Kweelt, Quilt and CanstoreX performing their expected functionalities, the desired end-product would be to combine them all and create one application that uses the CanStoreX storage, parses the specified XQuery and evaluates the same on the Kweelt platform and return the results. One main problem with these is that the applications function in their own way and needs to be integrated with each other to allow information to pass through them. As a preliminary step, the CanStoreX functionality needs to be integrated with the Kweelt platform and specific primitives have to be developed to instruct the Kweelt execution engine to use CanStoreX rather than the default main-memory storage. Further, the results of XQuery parsing are currently wrapped onto Kweelt objects which need to be converted to CanStoreX pages to process them. The thesis discusses all the challenges and the implementation issues that existed in integrating these applications and the various functionalities introduced in developing a single application that effectively co-ordinates with these utilities and provides a XQuery engine.

CHAPTER 3. NEED FOR A NEW ARCHITECTURE

Kweelt has been implemented very thoughtfully. It is highly modular and easily extensible. The main issues are that it supports Quilt on one hand and utilizes SAX and DOM as the underlying parsers. The Quilt has been superseded by the W3 standard XQuery for query of XML documents. As stated before, this problem has already been addressed in [32]. CanStoreX eliminates the need for SAX to load XML documents at query time as they are preloaded in the CanStoreX storage in a paginated binary form. CanStoreX also eliminated the need for DOM and has its own implementation DiskDom that allows efficient utilization of main memory. This section describes how the current implementation of Kweelt affects the performance and also provides means to overcome these limitations.

3.1 Main-Memory Usage

The current implementation of Kweelt loads the entire document onto the main memory and builds a DOM tree out of the same to evaluate the query on the document and return the results. Main memory several times the size of the document is required for query processing which develops into a bottleneck from querying huge documents. The main-memory is always restricted and should in theory do not depend on the size of the document. A need arises to provide a storage structure which stores the XML document and provides only a portion of the same to the main memory during processing.

3.2 Storage of Intermediate Results In the Main Memory

During the evaluation of a query, several intermediate nodes are obtained as a result of processing and these need to be stored for further processing. Currently, Kweelt stores all these intermediate nodes in the main memory itself occupying a vital part of the storage which could have been used for further processing. With a complex query being processed, there could be several such intermediate nodes lying idle in the main memory. Further, the space occupied by these nodes are not properly deallocated since the garbage collection is left to the Java Virtual Machine. As a result, the performance deteriorates quickly and the overall throughput is affected. These intermediate nodes could be stored on external storage and accessed when needed thereby providing better usage of the main memory.

3.3 Creation of In-Memory NodeLists

Another implementation issue with the Kweelt code is the usage of NodeLists when a collection of nodes needs to be processed. During the evaluation of certain queries, a hierarchy of nodes (such as ancestors / descendants / children) need to be evaluated and Kweelt loads all such nodes into the main memory into a NodeList structure. This structure is retained in the memory until the final execution of query although the nodes inside the list only need to be accessed one at a time. This list is apparently superfluous and could be replaced with a dynamic structure which reads one node at a time from the storage space with the specified hierarchy. The existence of such node lists add to the excessive main-memory consumption and needs to be replaced for the application to scale better.

3.4 Recursive Function Calls

A lot of recursive functions exist in the Kweelt source code given that it operates on a XML document with a tree based structure. Though recursive calls are easier to code while processing tree information, they consume quite a lot of memory by creating several function stacks. These could be replaced by non-recursive or iterative functions which maintain a user stack and control the flow of information along the stack. Though coding an iterative function is a bit tricky and generally involves a lot of effort, it greatly aids in reducing the memory usage and is useful for debugging purposes too.

The Kweelt implementation provides a very good platform for new developers to implement their own storage for XML documents and to build parsing applications for XQuery; nevertheless it contains a few bottlenecks which prevent users from using their applications to scale to very large documents. CanStoreX helps to solve the storage structure of XML documents while the problem of excessive main-memory consumption still exists. The following sections discuss on the solutions implemented to overcome the main memory problem and provide XQuery its capability to query large documents.

CHAPTER 4. ENHANCEMENTS IN THE NEW MODEL

The new implementation of XQuery would focus on a couple of issues. One would be to integrate CanStoreX with the current version of Kweelt and use CanStoreX as the default storage structure for XML documents. We limit ourselves to using only the binary version of CanStoreX for reasons mentioned before. The second area of focus is to remove the main-memory limitation of Kweelt by adding several new primitives to access information directly from the storage. This section would elucidate further on these new extensions.

4.1 Integrating CanStoreX

The CanStoreX architecture is introduced into the application by creating a new package xacute.impl.csx which would contain the definition for the NodeFactory class pertaining to CanStoreX. The factory would be responsible for creating nodes and node lists which comply with the storage structure. A typical CanStoreX node would inherit all its features from org.w3c.dom.Node due to the industry wide usage of the DOM structure. Hence the interface of the node is unchanged while the actual implementation pertains to that of CanStoreX. For example, a node would still possess firstChild, nextSibling, prevSibling etc. in accordance with the DOM terminology while the implementation of these primitives would be based upon the CanStoreX architecture that allows these nodes to be possibly scattered on different pages. This interface ideally abstracts the details of storage structure from the client and would represent a node to be a DOM Node to the outside world. A primitive version of DOMNode was developed by Robert Stark [33] which we built upon to provide a full-fledged functionality.

A CanStoreX node list would look identical to a normal NodeList except for the fact that instead of the node, the node list stores a pointer to the node position on the disk. This is one huge advantage accomplished with CanStoreX. Since the structure of CanStoreX defines each node to be in a page with a particular offset, the pair (PageId,NodeOffset) could be used to determine the position of a node on the storage. DOMNodeList exploits this information and stores the pairs in the node list and when required, the actual node is read based on the pointer information and evaluated against the predicates. Considering that there could be millions of such nodes and storing even the pointer information is space consuming, these pairs are eventually written onto the disk in a specified format explained later to be read from the storage whenever they are required.

4.2 Conversion of NodeLists to Iterators

With a robust storage structure in place, we now turn our attention to limiting the main-memory usage. The initial improvement would be to convert the node lists to specific iterators. An iterator functions like a pipe where data constantly flows with regard to client's requirements. An iterator would typically contain the following methods: open which opens the iterator and sets it up for reading, 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 read from the iterator is processed, it is disregarded for further processing. Such iterator based stream-oriented treatment of objects is adequate and therefore desirable in databases [34].

There exists different kinds of node lists owing to the existence of different types of hierarchy in a xml document such as list of children, ancestors, descendants, siblings etc. To cater to these needs, different types of iterators have been designed and developed to retrieve nodes based on its traversal position and are discussed below.

4.2.1 AncestorOrSelfNodeIterator

This iterator is used to return the current node and its ancestors (parent, grandparent, great-grandparent nodes etc.) in a sequential fashion. The iterator would initially return the current node and would move a level up every time the next node is requested until it either reaches the root node or the user explicitly invokes the close function.

4.2.2 AncestorNodeIterator

This iterator inherits its functionality from the AncestorOrSelfNodeIterator and returns all the ancestors of the current node except the node itself. The iterator would initially return the parent of the current node and would move a level up every time the next node is requested until it either reaches the root node or the user explicitly invokes the close function.

4.2.3 ChildNodeIterator

This iterator is used to return all the children of the current node. The iterator would return the first child of the node initially followed by the right siblings of the first child until it either reaches a node with no right sibling or the user explicitly invokes the close function.

4.2.4 DescendantSelfNodeIterator

This iterator is used to return the current node and all of its descendants (any node which is an immediate child or has an ancestor which is a child of the current node, ideally any node which could be reached through a simple path from the current node). The iterator performs its traversal in a pre-order fashion where it returns the node it has currently visited before it traverses to a new node. The nodes are returned in the exact order in which they are visited. The iterator is implemented in a non-recursive way using stacks to control the traversal information and to determine the next point of hop from a current node position. It also ensures that the entire subtree is visited and the nodes are visited only once thereby avoiding repetitive traversals. A simple traversal of the iterator would return the nodes in the order of their left-to-right association with their siblings i.e. a node would be visited before a descendant of its right siblings and after the descendants of its left siblings.

4.2.5 DescendantNodeIterator

The iterator inherits its functionality from the DescendantSelfNodeIterator with the only exception being that the current node is not returned and the traversal starts with the first child of the current node.

4.2.6 DOMNodeListIterator

The iterator is used to group a set of nodes which are completely unrelated to each other i.e. do not have a specific hierarchy. These nodes could be produced during the evaluation of a predicate or an expression and needs to be stored as intermediate results for further processing. The nodes are stored in the DOMNodeList which contains pointers to the original node on disk. The iterator would return the nodes in the exact sequence in which they appear in the DOMNodeList.

4.2.7 ParentNodeIterator

The iterator is used to return only the parent node associated with the current node. Since the tree-based structure of XML allows only one parent to be connected to a node, the iterator returns only the parent node and exits.

4.2.8 SelfNodeIterator

The iterator is used to return just the current node. The iterator is defined for compatibility purposes and merely does the task of returning the current node and exiting.

4.2.9 AttributeNodeIterator

The iterator is used to return all the attributes associated with the current node. It returns the attribute in the order in which they exist in the original XML document.

4.2.10 PrecedingSiblingNodeIterator

The iterator is used to return all the left siblings of the current node. It returns the immediate left sibling of the node initially followed by the siblings to the left of them until it either reaches a node with no left sibling or the user explicitly invokes the close function.

4.2.11 FollowingSiblingNodeIterator

The iterator is used to return all the right siblings of the current node. It returns the immediate right sibling of the node initially followed by the siblings to the right of them until it either reaches a node with no right sibling or the user explicitly invokes the close function.

4.2.12 NestedDOMNodeIterator

This is a variant of the normal iterators in that it does not return elements based on a specific hierarchy by itself. The iterator is used to group a set of iterators by nesting one iterator with another, for example, an AncestorNodeIterator is setup as the base iterator and a ChildNode iterator is set to be the child iterator of that. This would essentially imply that for all ancestors of the current node, return the children of each of the ancestors following the hierarchy specified i.e. it would return all the children of the parent of the node (the immediate ancestor) before returning the children of the ancestors of the parent node. One important observation to be considered is that the iterator should never return the ancestors of the current node; instead it should only return their children even though it processes them. The flexibility of XQuery allows such expressions to be defined and a generic implementation is required to handle these. The implementation is defined to theoretically allow any number of nesting levels supporting any kind of iterators.

4.2.13 SequentialDOMNodeIterator

The iterator is similar to NestedDOMNodeIterator that it is used to group a set of iterators in a specified sequence. The iterator executes the first iterator in the sequence returning all the nodes in the specified order and when the first iterator is exhausted, it moves to the second iterator, returns all its nodes and so on. The iterator is merely used to combine several other iterators discussed and return elements in the sequence of their enclosing iterator.

4.2.14 StepDOMNodeIterator

The iterator again contains a base iterator and a child iterator and executes in a way similar to NestedDOMNodeIterator. Besides the base and child iterators, it also contains predicates or filter expressions that need to be evaluated against each node and returns only those nodes in the sequence which satisfy the predicates. If a node fails to satisfy a given expression, the iterator moves forward to read the next node. This iterator would be very helpful in evaluating WHERE clauses which contains predicate filters.

4.2.15 PrecedingNodeIterator

This iterator is used to return all the preceding nodes that are not ancestors of the current node. It involves three iterators inter-twined in a complicated fashion. It initially creates an AncestorNodeIterator to read all the ancestors of the current node. It then sets up a NestedDOMNodeIterator with the AncestorNodeIterator as the base iterator and PrecedingSiblingNodeIterator as the child iterator; the preceding siblings of all the ancestor nodes are returned. It further sets up another NestedDOMNodeIterator with the previous iterator as the base iterator and DescendantSelfNodeIterator as the child iterator; the descendants of all the previous iterator are returned. The iterator essentially performs the following function: for all the ancestors of the current node, it reads the descendants of each of the previous siblings of each ancestor.

4.2.16 FollowingNodeIterator

This iterator is used to return all the following nodes that are not descendants of the current node. It again involves three iterators inter-twined in a complicated fashion. It initially creates an AncestorNodeIterator to read all the ancestors of the current node. It then sets up a NestedDOMNodeIterator with the AncestorNodeIterator as the base iterator and FollowingSiblingNodeIterator as the child iterator; the following siblings of all the ancestor nodes are returned. It further sets up another NestedDOMNodeIterator with the previous iterator as the base iterator and DescendantSelfNodeIterator as the child iterator; the descendants of all the previous iterator are returned. The iterator essentially performs the following function: for all the ancestors of the current node, it reads the descendants of each of the following siblings of each ancestor.

4.2.17 ValueResultIterator

This iterator is used to return the results associated with a query expression. The results could either be directly streamed from the storage or could be stored as intermediate results on the disk at some other location. The iterator returns the result nodes one at a time along with any tag information associated with them.

4.2.18 Native Iterators

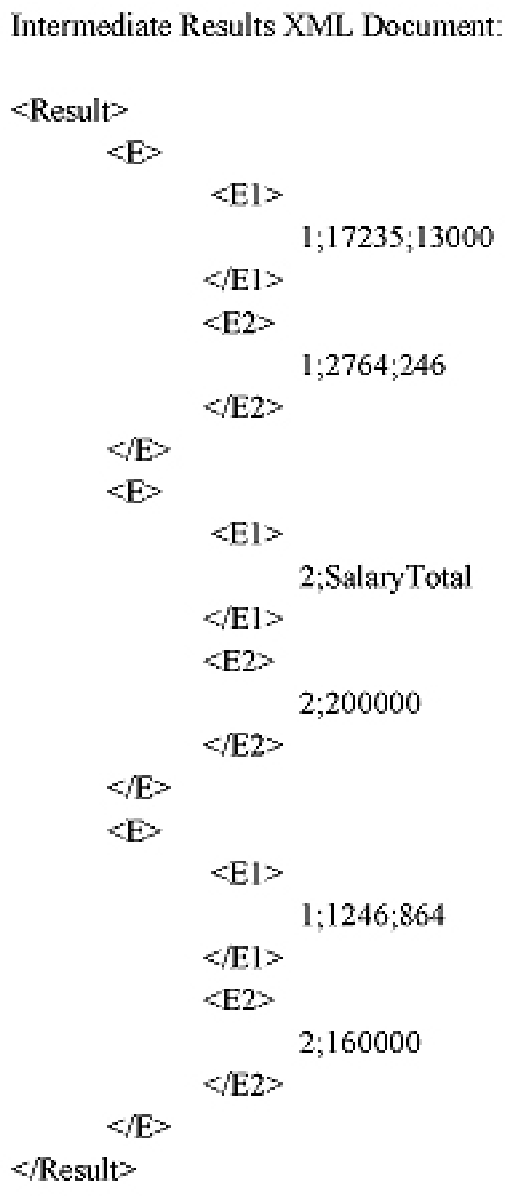
Besides the iterators used for returning elements based on a hierarchy, there exists a couple of native iterators which function at the storage level: DiskIterator and PageIterator. These iterators are used to read the pages on disk sequentially with the nodes that they contain and to perform page related operations such as obtaining a DOMNode given a pageId and a node-Offset, reading the attributes associated with a specified node such as children,siblings,parent etc. These iterators are used consequentially in almost all of the iterators discussed before and form the core of the linkage between CanStoreX and DOM.

4.3 Processing Intermediate Results

The iterators remove NodeLists from the Kweelt implementation which enhances the performance of the application to a great extent. However, when the queries involve lot of sub-querying, the intermediate results are still stored in the main-memory which would turn out to be a show-stopper in the event of sub query returning huge results. To overcome this issue, the intermediate results are stored onto the same storage space in different sets of pages. These pages are linked to each other; hence only the starting page number is required to iterate through these pages. Since the result could contain different items, a storage format for the intermediate results has been developed and implemented. This section would discuss about the format of the intermediate data in detail.

4.3.1 Intermediate Results Storage Format

The intermediate results are ideally the end-products of nested queries which need to be evaluated / compared with the result of the outer query. To facilitate this comparison, the nodes are stored onto the disk and an iterator is set up on top of the storage to read these nodes one at a time. The resulting nodes could either be nodes in the xml document or could be strings or numbers which happen to be the result of processing (such as the result of aggregate functions). To differentiate between the two types, a tag is added to each node to determine if the node qualifies to be a DOM node or a literal string. The resulting node structure would be (1,pageId,nodeOffset) or (2,LiteralValue). The first data refers to the type and if it is "1", the node is considered to be a DOM node and the pageId and nodeOffset information are retrieved. If it is "2", the node is a literal string and is read accordingly. Since the intermediate results could contain other tags imposed by the query, these tags are also embedded in the storage format. A typical intermediate result page corresponds to the resulting XML document specified in Figure 4.1.



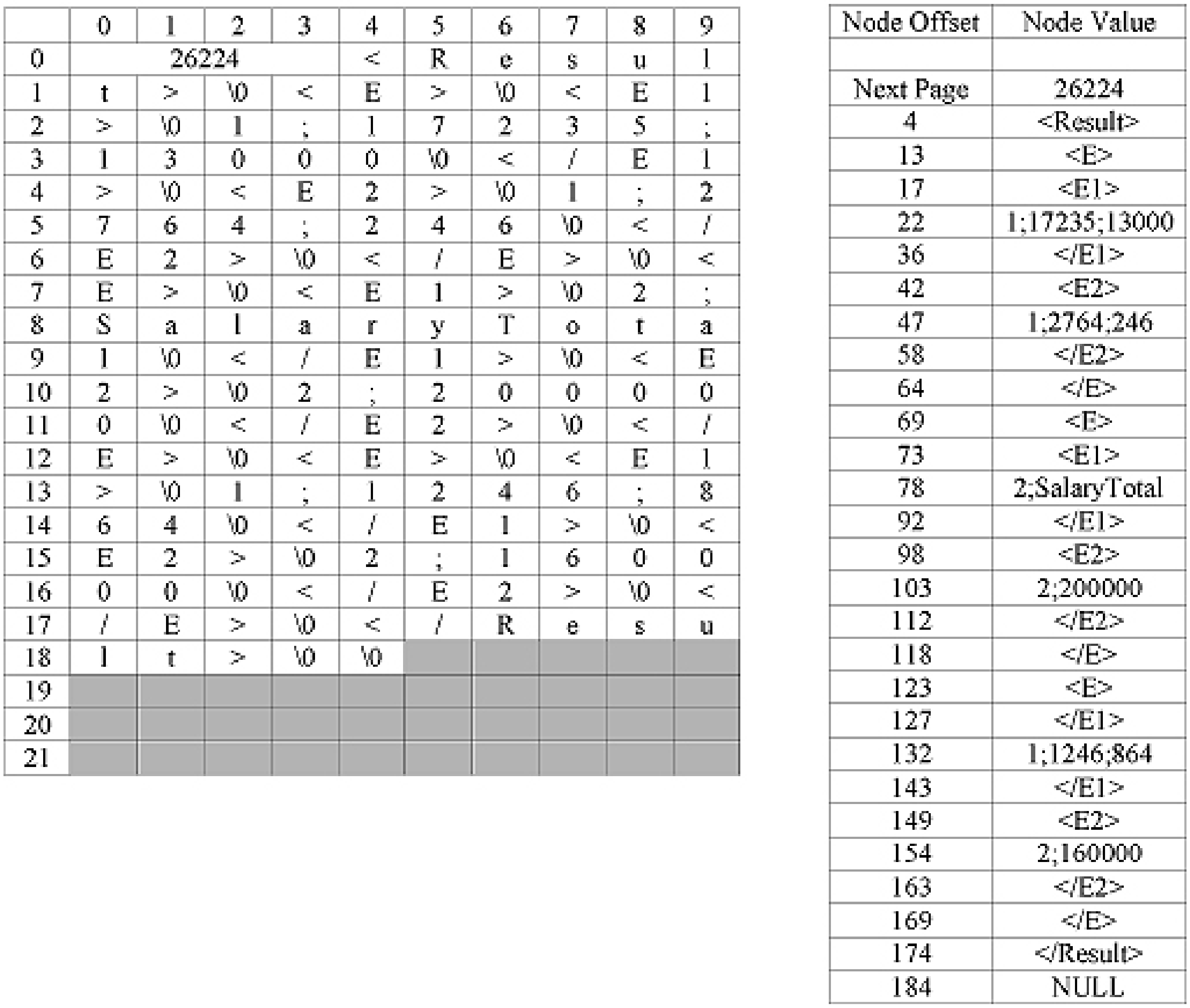
Intermediate Results XML Document:

```
<Result>
        <E>
                <E1>
                        1;17235;13000
                </E1>
                <E2>
                        1;2764;246
                </E2>
        </E>
        <E>
                <E1>
                        2;SalaryTotal
                </E1>
                <E2>
                        2;200000
                </E2>
        </E>
        <E>
                <E1>
                        1;1246;864
                </E1>
                <E2>
                        2;160000
                </E2>
        </E>
</Result>
```

Figure 4.1 Intermediate Results as a XML Document

The E, E1 and E2 tags are specified in the XQuery, ";" symbol is used to delimit the information present on the storage and the starting tag determines the type of node. This way of representing nodes is more generic and any intermediate information could be captured and stored in this fashion.

A page is composed of several such nodes and node information. Ideally, it would turn out that a single page would not be sufficient to store all the nodes. To counteract this, several pages are being used to store the node structure and these pages are inter-linked to each other. The first 4 bytes of every page are specially reserved to store the next page Id that the page points to with the exception being the last page which would store the pageId of "0". The allocation and deallocation of these pages are completely handled by the buffer manager; it allocates a page whenever it is requested and deallocates it when the user requests it to do so after processing. The flexibility of storing the intermediate results onto the storage structure allows the application to handle documents of any size. Further since the storage is native to the application and the pages are duly deallocated, no wastage of space occurs too. A sample page with the node information is specified in Figure 4.2.



	0	1	2	3	4	5	6	7	8	9
0	26224				<	R	e	s	u	l
1	t	>	\0	<	E	>	\0	<	E	1
2	>	\0	1	;	1	7	2	3	5	;
3	1	3	0	0	0	\0	<	/	E	1
4	>	\0	<	E	2	>	\0	1	;	2
5	7	6	4	;	2	4	6	\0	<	/
6	E	2	>	\0	<	/	E	>	\0	<
7	E	>	\0	<	E	1	>	\0	2	;
8	S	a	l	a	r	y	T	o	t	a
9	l	\0	<	/	E	1	>	\0	<	E
10	2	>	\0	2	;	2	0	0	0	0
11	0	\0	<	/	E	2	>	\0	<	/
12	E	>	\0	<	E	>	\0	<	E	1
13	>	\0	1	;	1	2	4	6	;	8
14	6	4	\0	<	/	E	1	>	\0	<
15	E	2	>	\0	2	;	1	6	0	0
16	0	0	\0	<	/	E	2	>	\0	<
17	/	E	>	\0	<	/	R	e	s	u
18	l	t	>	\0	\0					
19										
20										
21										

Node Offset	Node Value
Next Page	26224
4	<Result>
13	<E>
17	<E1>
22	1;17235;13000
36	</E1>
42	<E2>
47	1;2764;246
58	</E2>
64	</E>
69	<E>
73	<E1>
78	2;SalaryTotal
92	</E1>
98	<E2>
103	2;200000
112	</E2>
118	</E>
123	<E>
127	</E1>
132	1;1246;864
143	</E1>
149	<E2>
154	2;160000
163	</E2>
169	</E>
174	</Result>
184	NULL

Figure 4.2 Intermediate Results in a Page format

4.3.2 Result Iterator

An iterator has been developed to read the results from the disk and pass it on to the application. The iterator inherits all its features from the Value class and is hence referred to as

ValueResultIterator. The inheritance feature allows the application to pass any value objects to the query evaluation engine and the corresponding instance is then casted for further processing. Thus ValueResultIterator is comparable with other Value objects such as ValueString, ValueNum, ValueNodeList etc. and is primarily used for manipulation of temporary results.

4.4 Conversion of recursive methods to iterative ones

The Kweelt implementation possesses quite a lot of recursive methods to perform traversal and computation on tree structures. These methods consume a lot of main-memory by building function stacks and diminish the efficiency of the application. This limitation has been removed by converting all such recursive methods to iterative ones where a stack is used to store the temporary levels of recursion. The information on the stack is retrieved whenever it is required and processed upon accordingly. The conversion has been quite helpful in BindingTree class where several bind variables are evaluated recursively. The iterative code ensured that the variables are evaluated in the specified order and any intermediate information are properly maintained in the stack. The enhancement allowed the application to process documents beyond 10G in size which used to terminate with a "OutOfMemoryError" before. A few other places where the iterative version served its purpose are (1) Iterators to read nodes based on a specified hierarchy such as DescendantNodeIterator, FollowingNodeIterator, PrecedingNodeIterator etc. (2) De-pagination of the document rooted at a specific node (3) Evaluation Methods for the bind variables.

4.5 Re-implementing the basic Kweelt functionality

The preliminary version of Kweelt was originally developed assuming smaller documents and complete usage of main-memory due to which the application stopped working even for documents up to 10 MB or beyond in size. A complete redesign of the functionality was required to enable the application to process very large documents by involving the use of disk than main-memory. As a result, the primitive methods are all re-written to process information with a limited amount of memory and store the temporary results onto the disk.

The primary change involved was with the evaluation of FOR and LET clauses to cope up with multiple bind variables (the query could contain several FOR/LET clauses some separate while a few inter-twined with each other). Further, the QuiltExpression and several other classes were modified to store the temporary results onto the disk and return a ValueResultIterator object. The principles of object-orientation enabled the method interfaces to remain intact while the implementation was modified to suit our purpose. A major accomplishment of the new implementation is that the original Kweelt interfaces still remain intact and unchanged while only the functionality of these modules had been revamped. This would allow any other user to plug-in his own modifications with only the knowledge of Kweelt interfaces. The re-implementation is expected to make Kweelt scalable in terms of size and structure and allows the flexibility to query documents with a minimal set of requirements for main-memory and without any limitations for the document size.

4.6 LET Clause Evaluation

The new implementation of Kweelt extends the default implementation for both the FOR and LET clauses; however the LET clause offers functionalities such as performing aggregate operations on the document which are absent with a FOR clause. These operations pertain to COUNT,SUM,AVG,MAX,MIN for each of the bind variables. The default implementation used to perform these operations only on demand i.e. only when the user requests the application to perform a specific aggregate function. The new implementation exploits this window of opportunity by computing all the functions at a single execution. At a later point of time, if and when the client requests for particular aggregate functions are encountered, the value is read directly from the table and returned to the client. The stream-based computation of aggregate functions is rather inexpensive and this strategy avoids repetitive traversal of the document and saves a lot of processing time and resources. In some queries a let variable is only used for aggregation. This can be detected by a look-ahead mechanism during compilation. If this is the case, there is no need to save the list of nodes bound to the let variable.

With the above specified enhancements incorporated into the model, the application has been tested for a varied number of queries which are structurally and functionally different. The next section focuses on the type of queries that are being developed to test the application and evaluate its performance throughput. A more comprehensive scrutiny of the code revealed that quite a few enhancements do exist which could improve the efficiency much higher.

CHAPTER 5. XQUERY SPECIFICATION

The chapter deals with the various XQuery specifications that have been tried and tested with the query engine. Since the original implementation of Kweelt was confined to Quilt,it works only for a minimal subset of the XQuery language. Kweelt does not support quite a few advanced features such as functions, complex operators etc. This section would describe in detail the features that are supported by the new engine along with sample queries. To illustrate the queries, consider an organizational setting where there exists two documents Employees.xml and Departments.xml containing the information about an organization's employees and departments respectively as specified in Figure 5.1.

Employees.xml:

```
<?xml version="1.0"?>
<Emp>
         <Entry Id = “E001”>
                 <Name>EmpName1</Name>
                 <DName>DeptName1</DName>
                 <Salary>55000</Salary>
        </Entry>
        <Entry Id = “E002”>
                 <Name>EmpName2</Name>
                 <DName>DeptName2</DName>
                 <Salary>50000</Salary>
        </Entry>
        <Entry Id = “E003”>
                 <Name>EmpName3</Name>
                 <DName>DeptName1</DName>
                 <Salary>70000</Salary>
        </Entry>
        <Entry Id = “E004”>
                 <Name>EmpName4</Name>
                 <DName>DeptName2</DName>
                 <Salary>60000</Salary>
        </Entry>
</Emp>
```

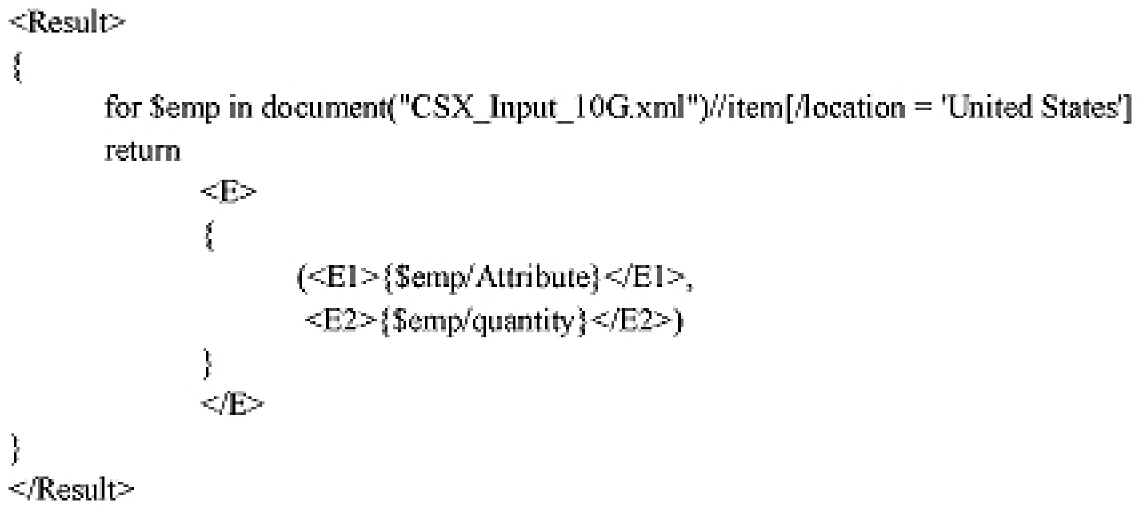
Departments.xml:

```
<?xml version="1.0"?>
<Dept>
        <Entry Id = “D001”>
                 <DName>DeptName1</DName>
                 <MName>Man1</MName>
        </Entry>
        <Entry Id = “D002”>
                 <DName>DeptName2</DName>
                 <MName>Man2</MName>
        </Entry>
</Dept>
```

Figure 5.1 Sample XML Documents

5.1 Simple form of FOR

The most primitive of all XQuery operations is the "FOR" clause. The clause is very similar to the "FROM" clause used in SQL and is used to iterate over nodes in a specific XML document and along a specific path. The clause supports expressions from the XPath language including predicates defined in the path. Predicates are filters that constitute a XPath expression which are used to evaluate nodes for specific criterion or a set of criteria and return only the filtered nodes. The FOR clause is followed by a return clause, optionally via a where clause. A simple FOR expression without a where clause is illustrated in Figure 5.2.



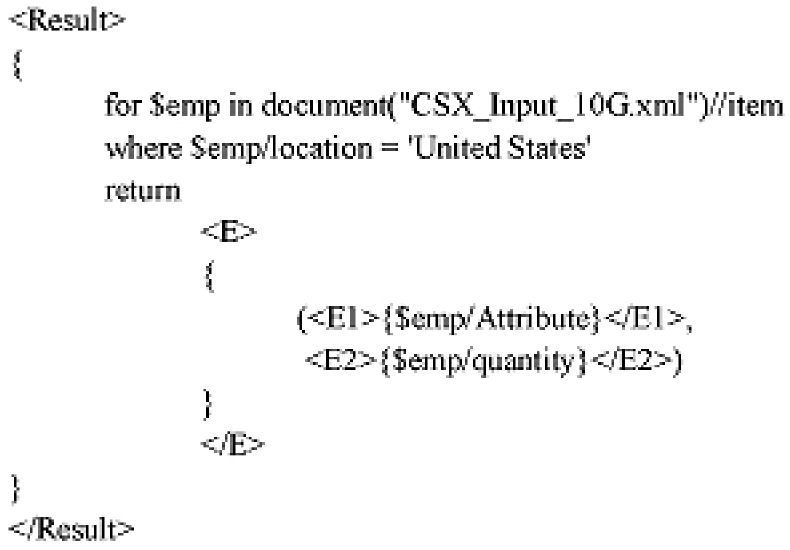
```
<Result>
{
        for $emp in document("CSX_Input_10G.xml")//item[/location = 'United States']
        return
                <E>
                {
                        (<E1>{$emp/Attribute}</E1>,
                         <E2>{$emp/quantity}</E2>)
                }
                </E>
}
</Result>
```

Figure 5.2 FOR Query

In the above mentioned example, the "/" and "//" are the XPath expressions specifying "children" and "descendant" nodes respectively. The "[/location = 'United States']" refers to the predicate filter returning nodes which satisfy the "location" constraint. The return clause contains tags E, E1 and E2 used to wrap the resulting nodes.

5.2 FOR together with WHERE

The ”FOR” expression, as mentioned before, could be used in conjunction with the ”WHERE” clause to filter nodes based on a specified constraint(s). The functionality of such an expression would be identical to that of a ”FOR” clause used with a predicate with the ”where” clause substituting for the predicate filter. The usage of ”for” with a ”where” is analogous to that of ”from” with ”where” in a traditional SQL syntax. A sample query with FOR-WHERE appears in Figure 5.3.



```
<Result>
{
        for $emp in document("CSX_Input_10G.xml")//item
        where $emp/location = 'United States'
        return
                <E>
                {
                        (<E1>{$emp/Attribute}</E1>,
                         <E2>{$emp/quantity}</E2>)
                }
                </E>
}
</Result>
```

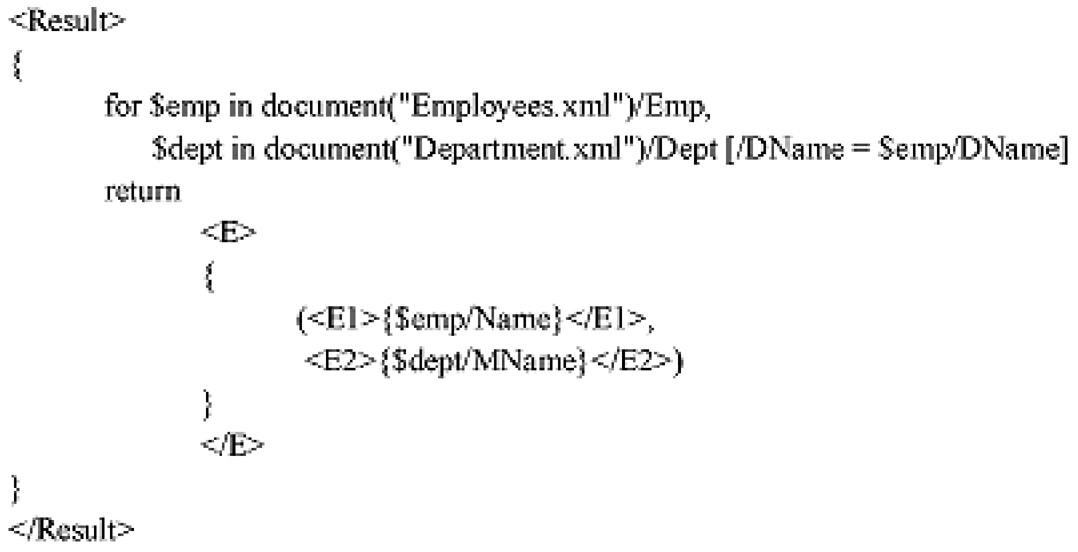
Figure 5.3 FOR-WHERE Query

The query is the same as the previous query written with a predicate filter and would return results identical to those of the previous one.

5.3 FOR-FOR Clause

The ”FOR-FOR” expression is used to combine multiple XML documents and return results that spans over several of the documents. The expression facilitates the join of documents that need to be processed for some common information. The query syntax is similar to a ”for” statement with the only exception being that the documents could be specified either with multiple ”for” commands or with a single ”for” command with the documents delimited by a comma. The processing time of this operation increases quadratically as the documents increase linearly in size since it involves comparing several documents at a time. The current implementation provides a brute-force technique to evaluate the query with each node of the outer document being compared with all nodes of the inner document every time.

There would arise a need to determine the list of employees along with their supervisor information. The query specified in Figure 5.4 answers such a requirement.

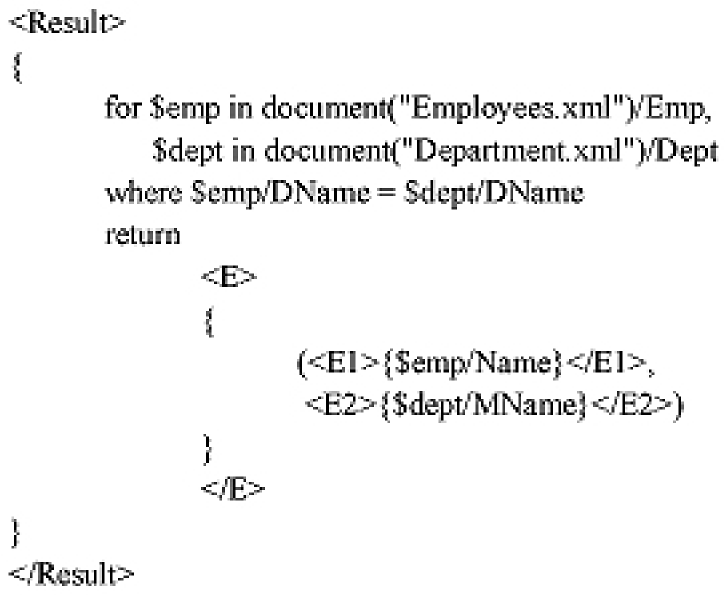


```
<Result>
{
        for $emp in document("Employees.xml")/Emp,
            $dept in document("Department.xml")/Dept [/DName = $emp/DName]
        return
                <E>
                {
                        (<E1>{$emp/Name}</E1>,
                         <E2>{$dept/MName}</E2>)
                }
                </E>
}
</Result>
```

Figure 5.4 FOR-FOR Query

5.4 FOR-FOR-WHERE Clause

As the case is with ”FOR-WHERE”, a ”FOR-FOR-WHERE” expression is semantically similar to that of a ”FOR-FOR” expression with a predicate filter. The query mentioned in Figure 5.5 on the XML documents ”Employees.xml” and ”Departments.xml” would return the same results as that were returned by the previous query.



```
<Result>
{
        for $emp in document("Employees.xml")/Emp,
            $dept in document("Department.xml")/Dept
        where $emp/DName = $dept/DName
        return
                <E>
                {
                        (<E1>{$emp/Name}</E1>,
                         <E2>{$dept/MName}</E2>)
                }
                </E>
}
</Result>
```

Figure 5.5 FOR-FOR-WHERE Query

5.5 LET Clause

A "LET" expression is used to operate on a collection of nodes rather than individual nodes. It facilitates set theoretic operations, quantifiers, and computes aggregate information on the collection such as count, sum, avg, max and min. The clause does not have a direct analogy with the SQL language which makes it a powerful and a vital expression of the XQuery language. The construct closest to let expression in XQuery in SQL is the group by clause. In SQL group by comes after the where clause that eliminates tuples that may not be of interest. There can be only one group by in a query block where as let can be used multiple subsets based upon independent criteria. In SQL tuples and groups are filtered by where, and having clauses, respectively, whereas in XQuery a single where clause takes care of several types of nodes as well as forests (collections of trees each rooted at a specific node). There could also exist multiple let variables each referring to its own forest. To cite an example, two forests could be compared based on their size (total set of nodes) or based on their depth.

An example to elucidate the "LET" clause is provided in Figure 5.6:

```
<Result>
{
        let $e1 := document("CSX_Input_10G.xml")//item
        return
                <E>
                {
                        (<E1>{max($e1/price)}</E1>,
                         <E2>{sum($e1/quantity)}</E2>)
                }
                </E>
}
</Result>
```

Figure 5.6 LET Query

Similar to a ”FOR-FOR”, let variables could also be used in conjunction along with predicate filters and also with a WHERE clause to check the forest for specified criteria as shown in Figure 5.7.

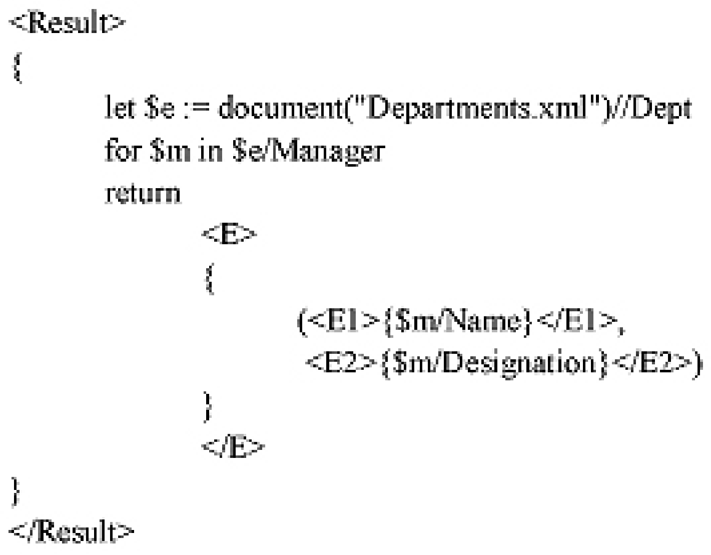
```
let $e1 := document("Employees.xml")//Emp [/location = 'United States']
return
<Result>
{
        <E> $e1/DName,
        let $e2 := document("Employees.xml")//Emp [/DName = $e1/DName && /Salary >. 80000]
        where count($e2) .>=. 20
        return
                <E>
                {
                        (<E1>{ avg($e1/Salary)}</E1>,
                         <E2>{ sum($e1/Salary)}</E2>)
                }
                </E>
}
</Result>
```

Figure 5.7 LET-LET Query

The above example evaluates two forests (1) a forest containing the employees information who work in United States and (2) a forest containing the employees who work in the specified department and earn more than 80000. The second forest is evaluated for the total number of nodes and only when it is greater than or equal to 20, the average of salaries and the total salaries of all employees belonging to the first forest is displayed. In other words, the query could be phrased in English as "For every department whose employee work in the United States, evaluate if the department has more than 20 employees who earn more than 80000 and return the average and total salaries of all such departments considering all employees who work in the department irrespective of their salary". Such a query becomes quite complicated in SQL requiring more than one query block.

5.6 FOR-LET Clause

A "LET" Expression returns a forest of nodes which is evaluated for set properties. There exists some scenarios when the forest is required to be traversed one node at a time to evaluate each node or to perform node specific computations. To counteract such situations, a LET clause is allowed to be combined with a FOR clause to iterate over the nodes. This functionality possess the distinct advantage that the forest could be verified either for group semantics or each node of the forest could be verified for individual element semantics.The clause is frequently used in queries to exploit this functionality and to offer the capability to query forest information. A sample FOR-LET query is given in Figure 5.8.

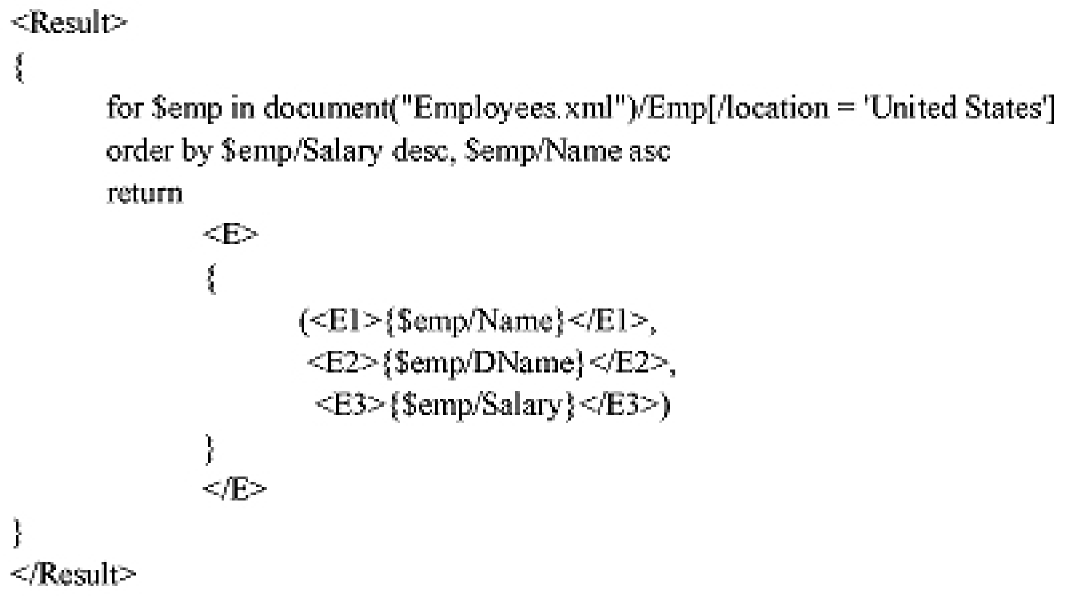


```
<Result>
{
        let $e := document("Departments.xml")//Dept
        for $m in $e/Manager
        return
                <E>
                {
                        (<E1>{$m/Name}</E1>,
                         <E2>{$m/Designation}</E2>)
                }
                </E>
}
</Result>
```

Figure 5.8   FOR-LET Query

5.7 ORDER BY Clause

The "ORDER BY" clause is used to sort the result nodes in a specific order. The order of sorting is specified by the keywords "ASC" and "DESC" with the default being ASC. Multiple attributes could be specified in the sort clause in which case the sorting happens with the attributes in the order prescribed. Further, the type of sorting could vary with each and every attribute in which case the sorter adheres to the required standards. For example, a sort clause could contain "ORDER BY Salary desc, first name asc, DOB desc". The clause specifies the sorter to sort data based on the salary in the decreasing order. For employees with the same salary, the sorter is expected to sort them based on their first names in the ascending order and for people with the same first name, sort the results from older to younger. A simple example of SORT clause is mentioned in Figure 5.9.



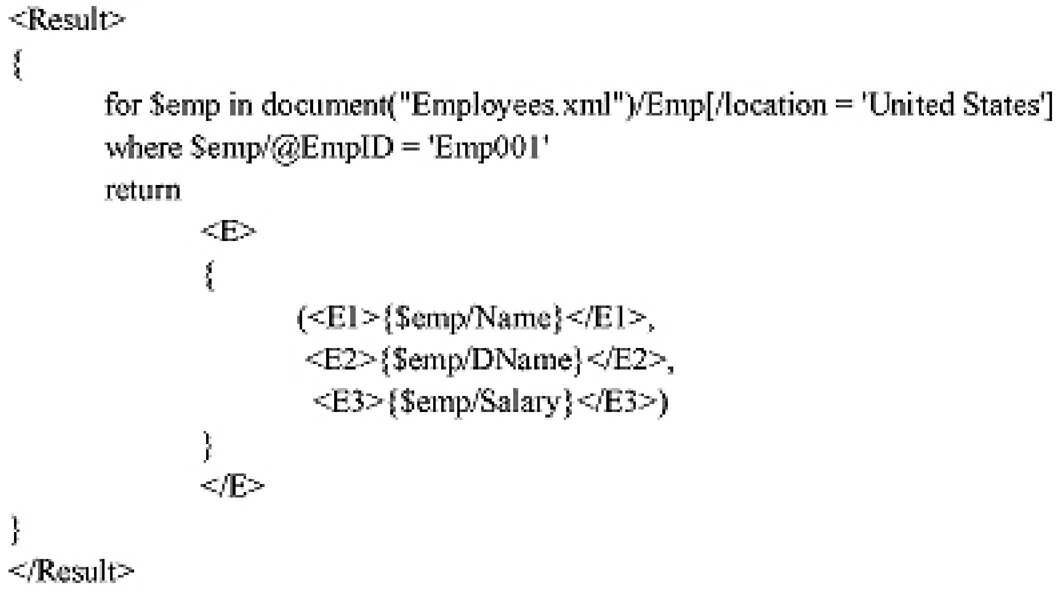
```
<Result>
{
        for $emp in document("Employees.xml")/Emp[/location = 'United States']
        order by $emp/Salary desc, $emp/Name asc
        return
                <E>
                {
                        (<E1>{$emp/Name}</E1>,
                         <E2>{$emp/DName}</E2>,
                          <E3>{$emp/Salary}</E3>)
                }
                </E>
}
</Result>
```

Figure 5.9 SORT Query

The default implementation of Kweelt provided an internal sorting algorithm in which the nodes are sorted internally in the main-memory. Apparently, the application did not scale beyond documents of 10 MB in size due to the main-memory limitation. The new implementation features an external sorting strategy involving the storage space and sort-and-merge algorithm which helped the application to scale up to documents of 10 GB in size. The following chapter discusses the sort algorithm and its implications in detail.

5.8 Attribute Clause

The "@" symbol is used to indicate attributes of a node and any expression involving the "@" symbol would evaluate the queries on the attributes pertaining to the node rather than the node or its children. This could be used anywhere in the query ranging from the bind variables to the predicate filters to the return clause. A simple example for the attribute clause is specified in Figure 5.10.



```
<Result>
{
        for $emp in document("Employees.xml")/Emp[/location = 'United States']
        where $emp/@EmpID = 'Emp001'
        return
                <E>
                {
                        (<E1>{$emp/Name}</E1>,
                         <E2>{$emp/DName}</E2>,
                          <E3>{$emp/Salary}</E3>)
                }
                </E>
}
</Result>
```

Figure 5.10 Sample Query involving Attributes

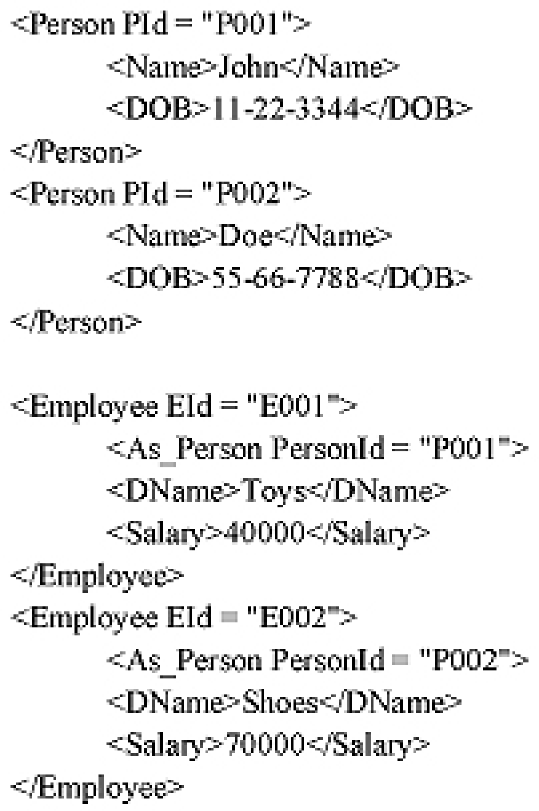
5.9 Object-Oriented Clauses

Besides the regular FLWR expressions, the XQuery application also supports quite a few object-oriented clauses which are used to reference objects based on their Id attribute. Object-oriented clauses are required to maintain references and hierarchies between nodes and to store the common attributes pertaining to a set of nodes in its parent node. A simple example for reference would be an organization database wherein several employees work with the same department. The department information need not be repeated for every employee but rather be stored in a parent node which is linked (pointed) to by each of the employee node. This saves a lot of storage space but still maintains the employee-department linkage and could be followed through to retrieve the information pertaining to every employee. This referencing mechanism could also be used to build hierarchies. The following sections describe the object-oriented clauses that are being supported by the application and their query syntax.

5.9.1 Typed References Clause

The Typed Reference Clause (or more commonly referred to as the ”arrow operator”) is introduced in Quilt and is used to dereference a pointer pointed by an IDREF attribute to the corresponding element node. The clause is predominantly used in conjunction with the attribute clause since the attribute normally contains the Id of every node. An expression specified by the arrow operator such as ”EmpId–Person@PId” indicates that given an EmpId, search for Person objects with the PId attribute matching the EmpId and return the Person Node. This expression is ideally used for referencing and inheritance purposes.

To elucidate the concept further, consider an object-oriented scenario consisting of Person Nodes with the Attributes PId and the children nodes to be Name, DOB. Consider a company setting containing employee nodes with EmpId as the attribute and PId, DName, Salary as the children nodes. An XML representation of the same is depicted in Figure 5.11:



```
<Person PId = "P001">
        <Name>John</Name>
        <DOB>11-22-3344</DOB>
</Person>
<Person PId = "P002">
        <Name>Doe</Name>
        <DOB>55-66-7788</DOB>
</Person>

<Employee EId = "E001">
        <As_Person PersonId = "P001">
        <DName>Toys</DName>
        <Salary>40000</Salary>
</Employee>
<Employee EId = "E002">
        <As_Person PersonId = "P002">
        <DName>Shoes</DName>
        <Salary>70000</Salary>
</Employee>
```

Figure 5.11 Sample Object-Oriented XML Document

A path expression to extract the DOB of an employee given an employee Id would be:

```
/Employee[/@EId = "E001"]/As_Person/@PersonId -> {Person@PId}/DOB
```

The basic advantage of such an approach would be that multiple nodes could reference attributes from a single node and the information is not required to be repeated every time. This would save a lot of space both in terms of size and structure of the document and would provide a central resource to add, delete and edit parent information. The parent-child relationship is maintained through the arrow operator and this could in theory extend to any number of levels required. With an appropriate database design in place, it also helps in making user queries more natural avoiding spurious joins.

5.10 XMark Queries

The XML documents used in the application are generated from a tool called XMark. XMark [37] is a benchmarking technique which creates a synthetic XML document. XMark provides a scalable document database and a comprehensive set of queries. The XML documents generated model an Internet auction website, a typical e-commerce application. The main entities are person, open auctions, closed auctions, item and category. The hierarchical schema of the documents is depicted in Figure 5.12.

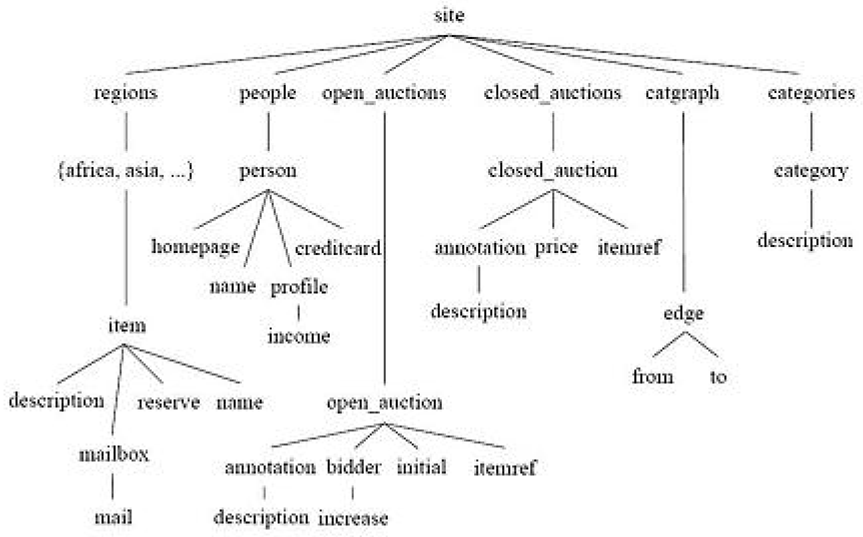


Figure 5.12 Hierarchical Schema for XMark XML Document

XMark has come up with a set of 20 XQueries to be evaluated on the documents. These queries explore through the various possible paths in the documents and are quite useful to benchmark the query performance and throughput. The application is tested for accuracy and efficiency by running these queries on documents pertaining to different sizes ranging from 100 KB to 100 GB in size. These queries help evaluate the throughput of the query evaluation engine and compare the performance of the application to other XQuery applications that exist. It should be noted that currently no indexes and optimization have been used. These can be studied and implemented in future to significantly enhance the throughput of queries.

CHAPTER 6. SORTING TECHNIQUE

Sorting refers to the process of ordering nodes in a specified order as desired by the user. Sorting is very useful in quite a few applications such as a library management system where the books are arranged by specific criteria, student enrollment system where the student records are ordered by their last name etc. In the context of database, sorting is used by many external applications and is also used internally to improve the performance of the database and query evaluation engine. For example to make a join faster, temporary sorted copies of the operands, that can be large xml documents, could be created. Sorting is extensively used in creating indices e.g: B+ trees and hashes on existing documents. The feature is implemented in traditional database systems through the concept of a (record key - record pointer) pair where the record keys are sorted through a B-tree or a heap and the record pointers specify the location of the node on the disk. An advantage with the traditional database system is that the records have fixed length and hence it is convenient to store them in pages and during sorting two records can be swapped even when they reside in different pages. Swapping is an important operation in sorting and it becomes problematic when records vary in length leading to fragmentation. In XML nodes can occupy several pages and moving it from one block to another can be expensive. Furthermore, a sorting algorithm can move a node multiple times leading to spurious disk accesses repeatedly.

Fortunately, CanStoreX allows us to solve this issue with its storage structure. As had been mentioned before, CanStoreX stores documents in pages and have a (pageId, nodeOffset) pair associated with each node which could be used as a pointer to the node. With this structure in place, we could consider XML documents to be similar to fixed-size documents and apply the sorting algorithm. The sorter would now accept a (record pageId, record nodeOffset, record key) triplet and would order the pointers based on the record key. The section would elaborate more on this structure and the sorting technique that is applied to sort XML nodes.

6.1 Current Implementation

The default version of Kweelt uses an internal sorting algorithm using node lists and a Map containing these lists. The complete sorting technique is performed only using the main-memory which apparently leads to the main-memory exhaustion issue. NodeLists are used to store the nodes along with their hierarchy and every node list is compared node-to-node with the other lists to determine the precedence and a new NodeList is created every time to store the ordered nodes. The technique uses a lot of temporary storage in addition to storing the entire document in the main-memory which is not feasible and extensible for larger documents.

6.2 Proposed Sorting Technique

The new technique proposed for sorting is ideally required to consume less of main-memory and use the storage on the disk for the temporary results. The NodeLists are replaced by the Iterators mentioned before to read the node and its corresponding hierarchy. The sorter would now involve both the main-memory and the storage to perform the sorting technique and hence would involve an external sorting algorithm. The implementation details and the complexity of the algorithm are discussed below.

6.2.1 External Sorting Algorithm

External sorting is essentially done in two phases: creation of runs and merging. A run is a certified sorted sequence of nodes. The initial phase is accomplished by reading sets of nodes onto the main memory and sorting them using an internal sorting mechanism. The sorted segments (runs) are now written onto the storage on to a different location. The latter phase merges these sorted runs to form larger runs until a single certified run is obtained which would essentially have sorted all the nodes. The following section provides an example to elucidate on the sorting mechanism. To avoid loss of generality, the example assumes documents of fixed size which could be replaced by the key-pointer triplets of the nodes.

Consider a sorter required to sort 1000 records which are stored on the disk in 10 pages each containing 100 records. There are 3 buffers available to be used. With this setup, the sorting phases are explained below:

6.2.1.1 Creation of Runs

The initial phase involves reading pages that could fit in the buffer, sorting them individually through an internal sorter and writing them back onto the disk. Since there are 3 buffers that are available to us, every time a set of 3 pages are read from the disk, sorted and written onto the disk in a different location. The end of this phase would provide us with 4 runs each containing 3 pages sorted in order except the last run which would contain only one page. The buffer configuration and the storage structure after the creation phase are illustrated in Figure 6.1.

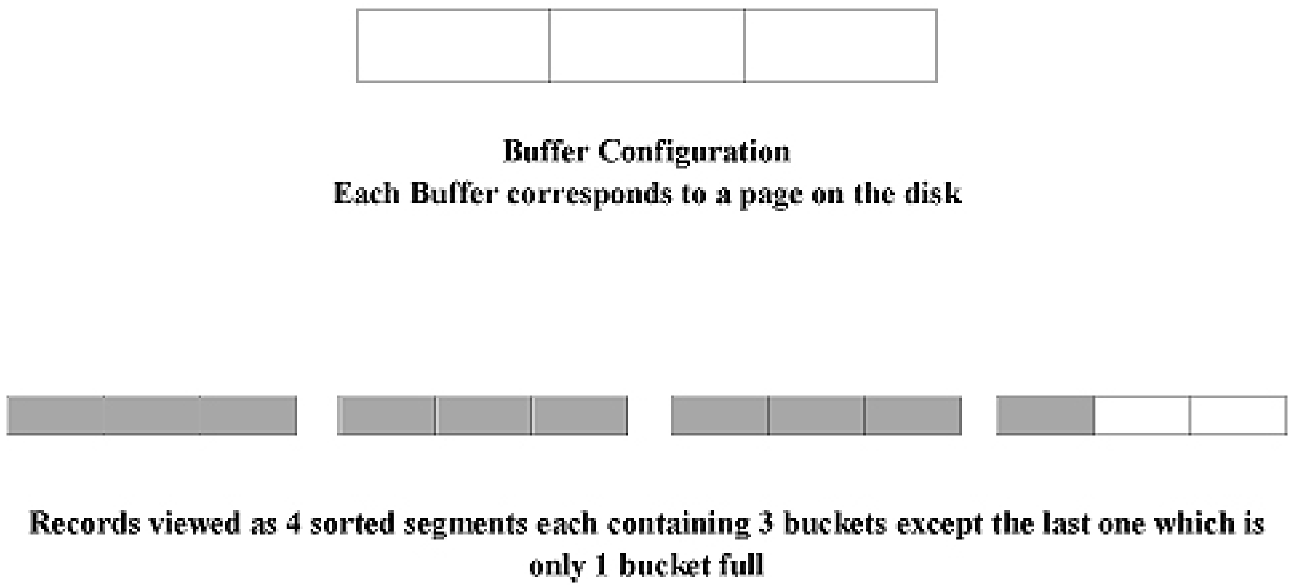

Buffer Configuration
Each Buffer corresponds to a page on the disk
Records viewed as 4 sorted segments each containing 3 buckets except the last one which is only 1 bucket full

Figure 6.1 Sorting Configuration

6.2.1.2 Merging

Merging involves reading multiple runs at a time and merging them to form one larger run. The process is repeated for all the runs until a single run is obtained in which case the given set of nodes are sorted. To illustrate our example, the 3 buffers are configured as specified in figure to set up two input streams and one output stream. This configuration would ideally be used to merge two runs at a time. The records are sequentially read from each of the input stream, compared for precedence and the appropriate record is written onto the output buffer. The next record from the corresponding input stream is read and the process is repeated until either of the input stream runs out of records in which case all the records from the other stream is written directly onto the output buffer. The records are written continuously onto the output buffer, which when full, writes the records onto the storage and maintains a reference to the same. The merging technique is depicted in the Figure 6.2.

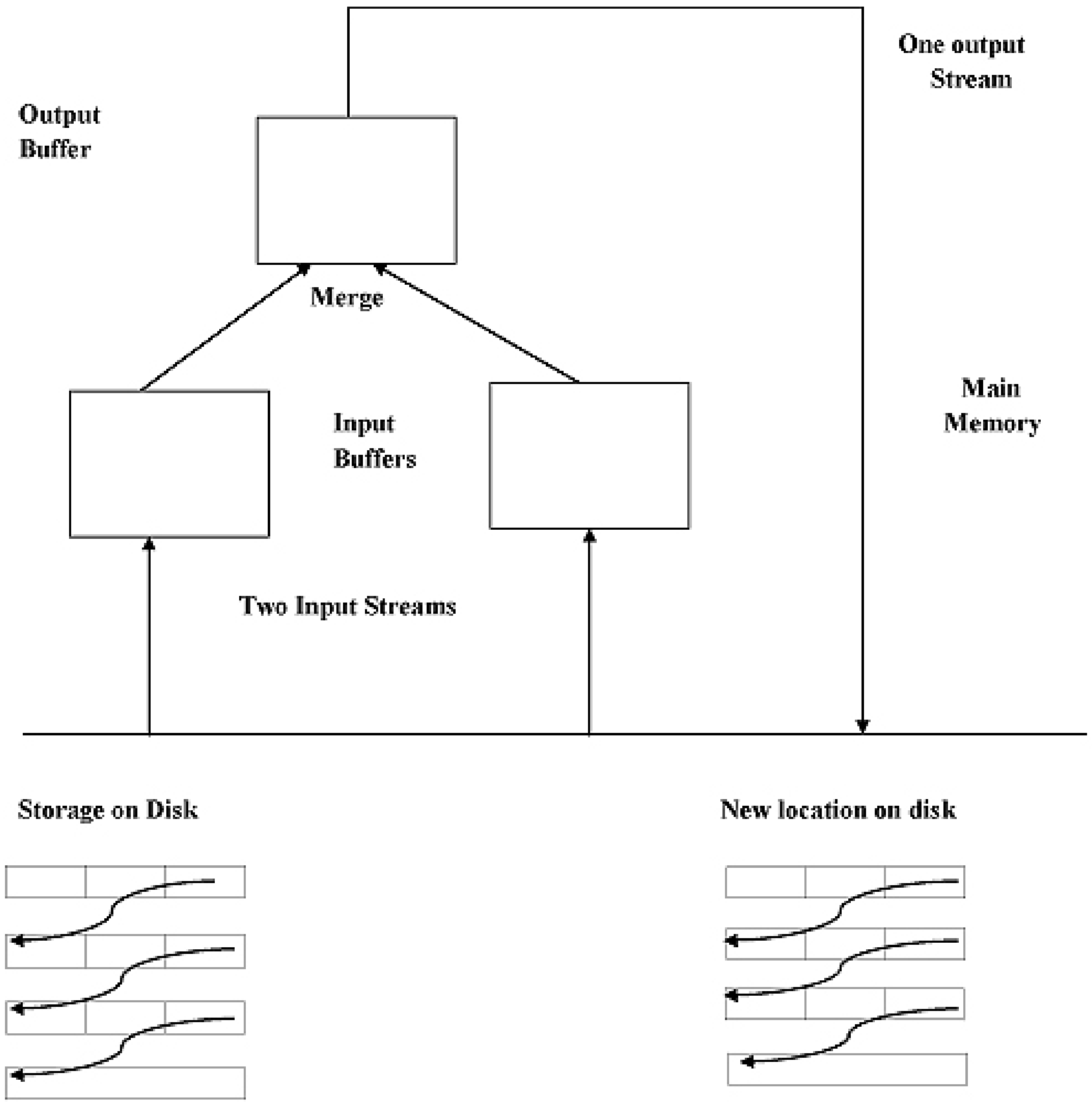


Figure 6.2 Merge Configuration

At the end of the first pass of merging, there would be 2 runs with the first one containing 6 records and the second one consisting of 4 records. The process is repeated one more pass to obtain one single run of 10 records at which point the nodes are sorted as shown in the Figure 6.3. A pass consists of reading a set of pages, sorting them and writing them back onto the disk. It leads to a reduced number of runs with each run pointing to a larger set of records that had been sorted. The number of passes needed to sort a set of records is a good indicator of the performance of the algorithm. The current implementation uses a 2-way merging and this could be replaced by a k-way merging to better improve the sorting performance.

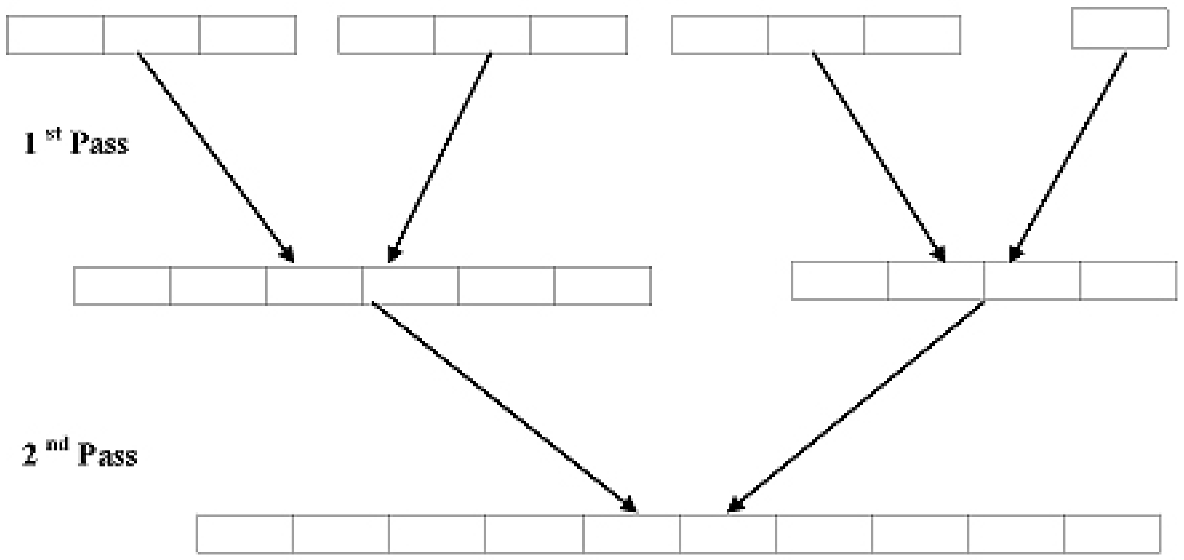


Figure 6.3 Merge Technique

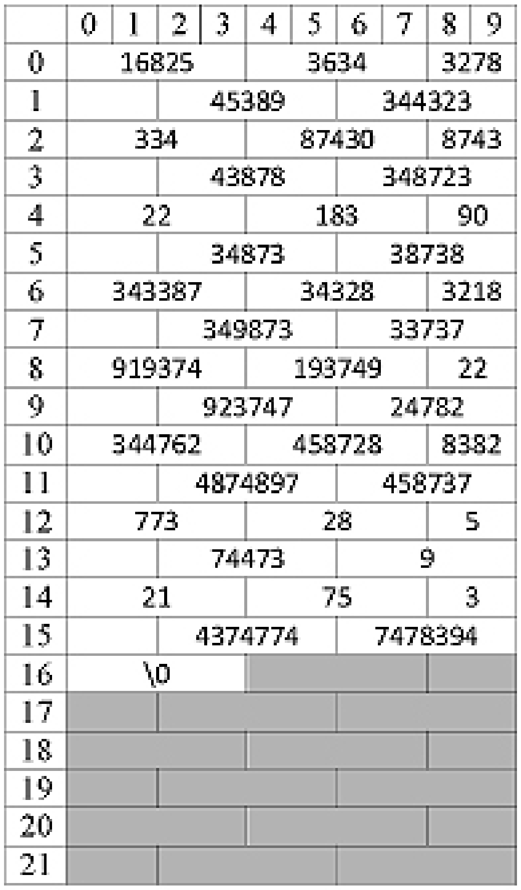
6.3 Implementation Specifications

The sorting technique mentioned above is implemented with the help of Iterators, buffers and linked list of pages. Further, since the runs need to be stored on the disk, a special storage format is developed to store the runs and refer them during the merging process. This section would discuss the implementation details of the algorithm.

6.3.1 Creation of Linked List of Pages

A certified run in the sorting mechanism is characterized by a set of pages that are previously sorted. The number of pages in a run would vary from one run to another. Further, due to the varying length of the keys, a run involving two sets of pages is not expected to produce a larger run containing exactly the sum of the pages. The new run could, in some instance, contain pages that are more than the total number of source pages combined. This requirement hinders the flexibility to use a fixed size data structure to store the pages of a run. To counteract this, a linked list of pages had been implemented with the algorithm having knowledge of only the page Id of the starting page. An Iterator is provided to iterate over the pages from the starting page till the last page returning one record at a time. Every page stores the Id of the next page in a special location through which the page traversal happens.

The number of such linked lists could, in theory be large for documents involving lots of records and storing the list in the main-memory would not be feasible and scalable to very large documents. To avoid this issue, the list of pages are stored in a separate page the Id of which would be provided to the application. Further, the list of pages could be large enough to span multiple pages, hence these are written onto multiple pages linked to each other with the same strategy as mentioned above. In the end, the sorter would be provided with a pageId containing a set of page Ids each of which points to the starting page of a linked list of pages. A sample page containing the pageIds is illustrated in Figure 6.4.



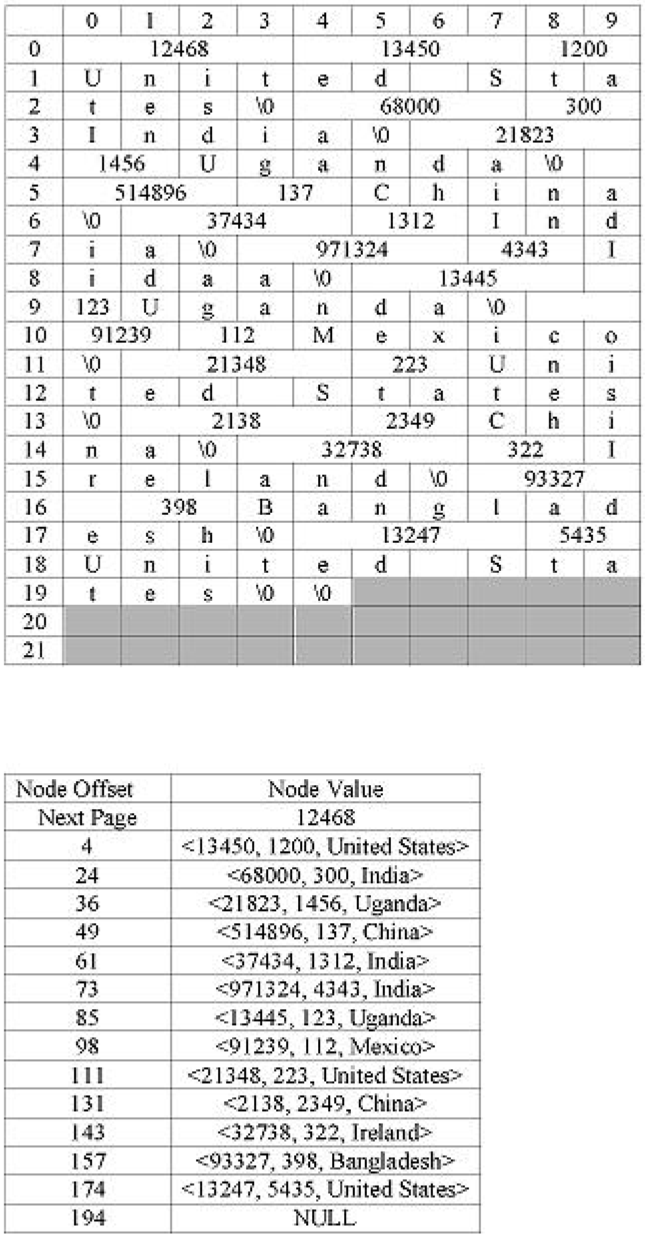
	0	1	2	3	4	5	6	7	8	9
0	16825				3634				3278	
1			45389				344323			
2	334				87430				8743	
3			43878				348723			
4	22				183				90	
5			34873				38738			
6	343387				34328				3218	
7			349873				33737			
8	919374				193749				22	
9			923747				24782			
10	344762				458728				8382	
11			4874897				458737			
12	773				28				5	
13			74473				9			
14	21				75				3	
15			4374774				7478394			
16	\0				▨				▨	
17	▨		▨				▨			
18	▨				▨				▨	
19	▨		▨				▨			
20	▨				▨				▨	
21	▨		▨				▨			

Figure 6.4 Sample Page Format

Ideally the structure would initially comprise of number of linked lists each pertaining to one certified run. This number would decrease approximately by half after every pass while the length of each linked list increases by two folds indicating that there now exists a larger certified run with more sorted records. At the end of the process, there would exist one linked list containing all records in a sorted fashion.

6.3.2 Storage Structure

The sorting technique involves the temporary results to be written onto the disk. Since the records involved could span more than a single page, a special storage format needs to be developed to store these on the storage and retrieve them in the specified order. The information to be stored involves the record key and the record pointers, essentially a (record-page-id,record-node-offset,record-key) triplet. While the record-page-id and the record-node-offset are fixed-size in length, the record-key is in principle variable-size. To handle this requirement, the pageId and nodeOffset are initially stored in the page followed by the record key stored as a string terminated with a special character. The termination character is checked for every time to determine the end of the record key. The initial few bytes are specifically reserved for the Id of the next page with the last page containing a 0. A typical storage page with the record structure is depicted in Figure 6.5. The record key used is the ”location” attribute while the pageId and nodeOffset are converted from integer to bytes and stored in the page instead of storing them as literal strings.



	0	1	2	3	4	5	6	7	8	9
0	12468				13450				1200	
1	U	n	i	t	e	d		S	t	a
2	t	e	s	\0	68000				300	
3	I	n	d	i	a	\0	21823			
4	1456		U	g	a	n	d	a	\0	
5	514896			137		C	h	i	n	a
6	\0	37434				1312		I	n	d
7	i	a	\0	971324				4343		I
8	i	d	a	a	\0	13445				
9	123	U	g	a	n	d	a	\0		
10	91239		112		M	e	x	i	c	o
11	\0	21348				223		U	n	i
12	t	e	d		S	t	a	t	e	s
13	\0	2138				2349		C	h	i
14	n	a	\0	32738				322		I
15	r	e	l	a	n	d	\0	93327		
16		398		B	a	n	g	l	a	d
17	e	s	h	\0	13247				5435	
18	U	n	i	t	e	d		S	t	a
19	t	e	s	\0	\0	▨	▨	▨	▨	▨
20	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
21	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨

Node Offset	Node Value
Next Page	12468
4	<13450, 1200, United States>
24	<68000, 300, India>
36	<21823, 1456, Uganda>
49	<514896, 137, China>
61	<37434, 1312, India>
73	<971324, 4343, India>
85	<13445, 123, Uganda>
98	<91239, 112, Mexico>
111	<21348, 223, United States>
131	<2138, 2349, China>
143	<32738, 322, Ireland>
157	<93327, 398, Bangladesh>
174	<13247, 5435, United States>
194	NULL

Figure 6.5 Sort Results Storage Page Format

The technique mentioned above is completely implemented and tested for accuracy and performance. The issues of main-memory consumption and internal sorting overhead are removed from the application making it functional to perform sorting on documents of any size. The mechanism consumes quite a few pages from the storage but the pages are deallocated once the algorithm is completed and the results are returned back to the client. For accuracy purposes, in the process of sorting, each of the runs is individually verified to ensure that there are no intermediate false positives.

CHAPTER 7. EXPERIMENTAL RESULTS INTERPRETATION

This section describes the experiments that had been conducted on the XQuery application and the implications of the same. The chapter begins with a description of the computer system used for the experimental purposes. It further introduces a couple of performance metrics used to evaluate the application and then elucidates the results of the experiments.

7.1 Computer System Benchmark

The machine set up for the experiments possesses the following configuration:

7.1.1 CPU

The processor used was a AMD Athlon 64 X2 dual core processor 3800+ with a speed of 2.01 GHz, 64+64 KB L1 cache and 512 KB L2 cache.

7.1.2 RAM

The main memory used had a capacity of 1 GB.

7.1.3 Hard-disks

A total of 3 hard-disks were used to run the queries. The first hard-disk is a Serial ATA disk with a capacity of 80 GB and with manufacturers rating of 7200 RPM, 300 MB/s data transfer rate and 16 MB internal cache.This was primarily used to store the operating system files and the source code of the application along with the XMark generated XML files.

The second hard-disk is a Serial ATA disk with a capacity of 465 GB and with manufacturer's rating of 7200 RPM, 300 MB/s data transfer rate and 16 MB internal cache. This was used as the storage for the application to store the XML documents in the paginated binary format.

The third hard-disk is a Serial ATA disk with a capacity of 465 GB and with manufacturer's rating of 7200 RPM, 300 MB/s data transfer rate and 16 MB internal cache. This was used as the output storage to store the results of XQuery in a binary format. The structure of the output storage resembles that of the application storage and the similarity is maintained to determine the speed of data transfer across the disks.

All the disks were formatted with NTFS using 16KB allocation unit.

7.1.4 Operating System

Windows XP with service pack 2 was used as the operating system.

7.2 Application Benchmark

The application had quite a few benchmarks, in contrast to the system, which were set to be in synchronous with the characteristics of the computer system.

7.2.1 Pagination PageSize

The PageSize was set to be 16 KB in size to be in sync with the allocation unit of the operating system.

7.2.2 Storage Files

The files on the disk representing the CanStoreX storage were RandomAccessFiles created using Java. These files grow dynamically as and when information is appended to them which are constrained by the characteristics of the operating system. To avoid this, the files are initially filled with null values ensuring that the pagination process does not result in the files being expanded.

7.2.3 XML FileSize

The XML documents were generated from a tool called XMark. The tool generates a file with the specified size while the characteristics of the document such as fan-out and depth are internally taken care of by the utility. The documents generated were of size 100 KB, 1 MB, 10 MB, 100 MB, 1 GB, 10 GB, 50 GB and 100 GB. The smallest XMark could generate was around 45 KB.

7.2.4 Pagination Strategy

The pagination algorithm had two strategies implemented: fixed-size nodes and variable-size nodes.

7.3 Performance Metrics

A couple of performance metrics had been devised to evaluate the performance of the application and to interpret the results in a quantitative fashion. The section provides a brief definition of the metrics.

7.3.1 Running Time

The running time is defined to be the total time in seconds that the application requires to produce the expected result. This includes the time spent in parsing the query, initializing the query engine, reading from the storage, writing/reading the intermediate results and writing the results onto the output storage.

7.3.2 Throughput

The throughput is defined to be the amount of data processed per second and is measured in mega bytes per second. This metric is a more realistic one and is used to compare the performance of application on documents of different size and on queries of varying levels of complexity. For queries involving a single document, the metric is directly measured by dividing the size of the document by the time taken to evaluate the query. For queries involving multiple documents (for example 2 documents), the metric is evaluated as follows: For each node processed in the outer document, the inner document would be traversed completely once. Hence the total size of documents would be estimated as the number of nodes in the outer document multiplied by the size of the inner document. This total size is then divided by the total time spent in processing to obtain the throughput.

7.4 Result Interpretations

Appendix A depicts the results of the experiments conducted on documents with a linear increase in size. The XQuery expressions could be ideally classified into two groups one with a "/" path expression involving only a specific portion of the document and the other with a "//" path expression involving a traversal of the entire document. The throughputs of the application for "/" expression are evidently quite higher than that of "//" expressions when run on the same document due to the aforementioned behavior.

Figures 7.1 - 7.4 provide a graphical representation of the throughput of the query engine on documents of various sizes and on queries of varying levels of complexity. Figure 7.1 depicts the performance of the application on simple queries which scan only a small portion of the XML document while Figure 7.2 reflects the performance on queries which involve parsing the entire document. The efficiency of the new sorting algorithm is also captured in Figure 7.3 and Figure 7.4 represents the performance on a few XMark queries that had been executed.

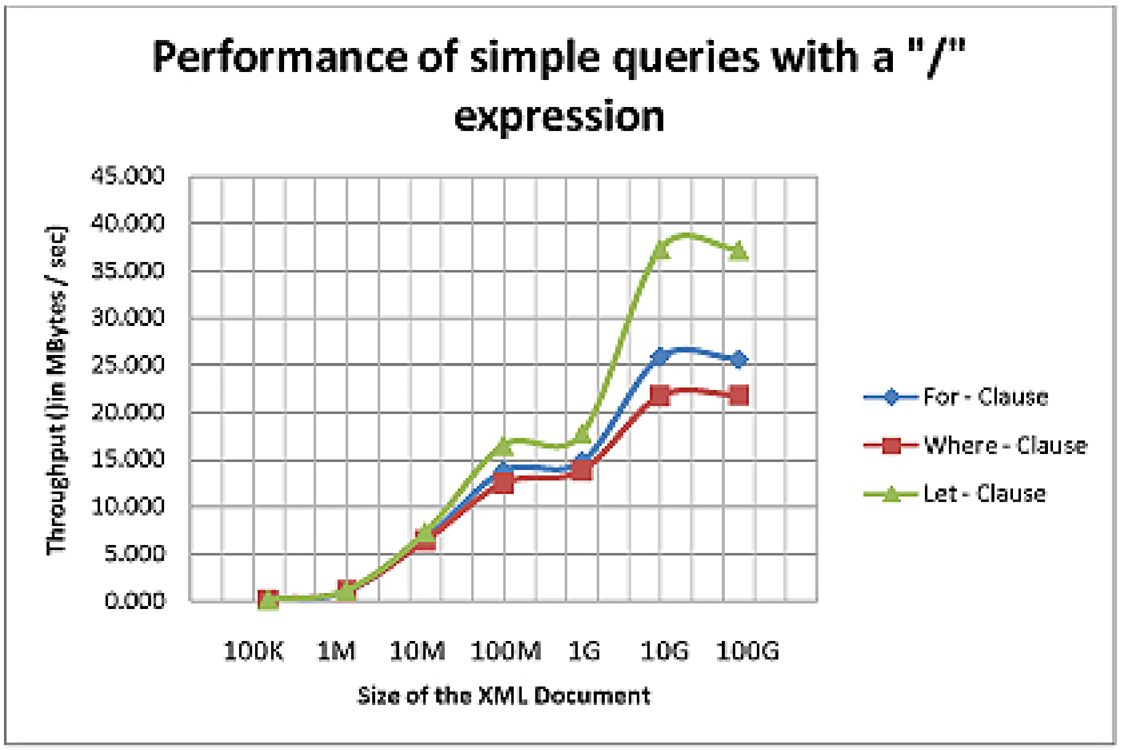


Figure 7.1 Throughput of query engine on simple queries

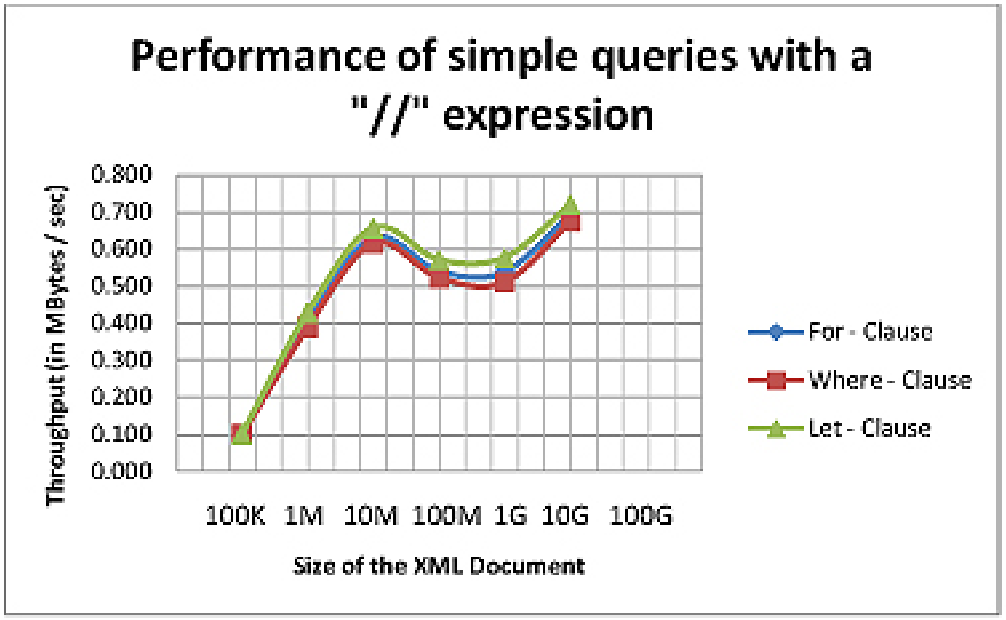


Figure 7.2 Throughput of query engine on complex queries

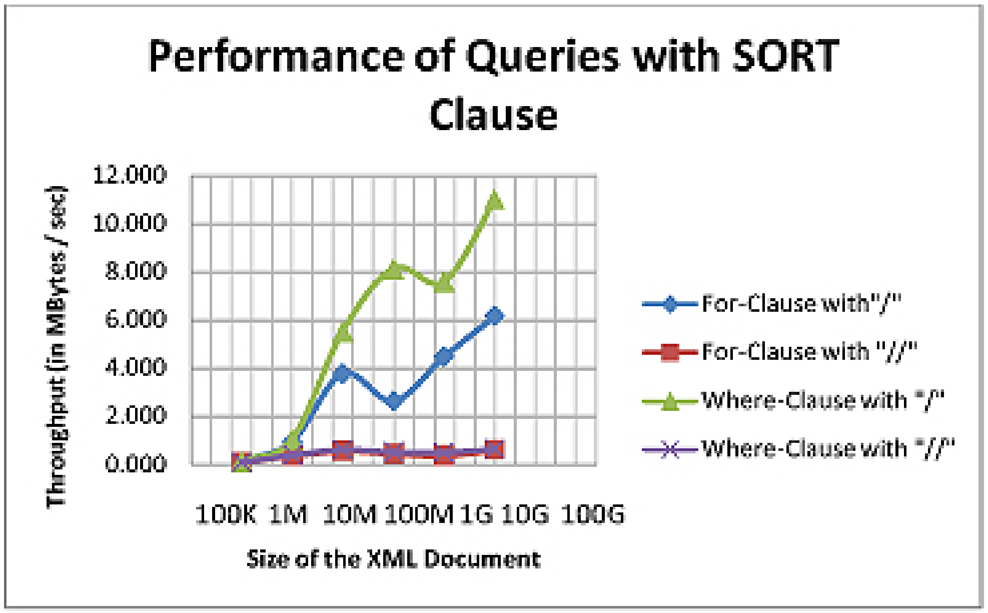


Figure 7.3 Throughput of query engine on SORT queries

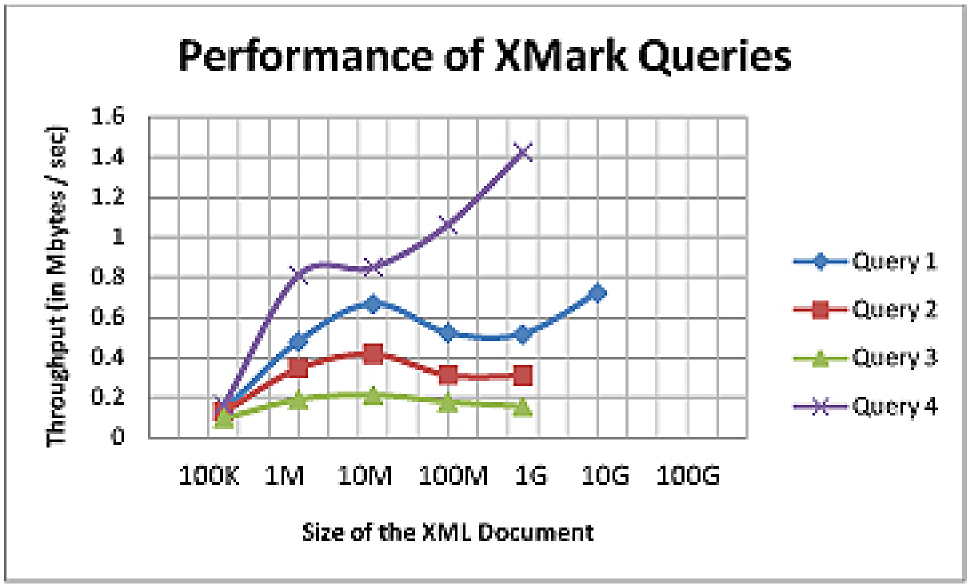


Figure 7.4 Throughput of query engine on XMark queries

A general increase in the throughput is observed as the document increases linearly in size. The increase in the throughput is due to the extra time the application requires to parse the input query and construct the query tree before traversing the document. This extra time is quite evident in the case of small documents where the processing time involved with the document is very less. Since huge documents take some time to be traversed completely, the throughputs obtained from these are appropriate and hence shows a trend of increase in throughput. The values may not be accurate for documents up to 100 MB in size because the application takes very little time to process these documents and the time may not be captured very precisely.

The results indicate the flexibility and the extensibility of the application in being able to process documents up to 100 GB in size. While the results mentioned pertain to the basic query syntax, there are still quite a lot of complex query patterns that could be used to evaluate the performance of the application which would give a more realistic estimate of the application's efficiency. One good source would be from the XMark benchmark queries which could be used to evaluate the utility comprehensively.

CHAPTER 8. CONCLUSIONS AND FUTURE WORKS

The chapter closes the thesis with the conclusion and some suggested future works.

8.1 Conclusion

The XQuery engine provides the extension to the binary version of CanStoreX to evaluate and execute queries on huge XML documents. It removes the current problems existing with the Kweelt implementation such as main-memory exhaustion and in-memory storage of intermediate results. The storage structure provided for intermediate node storage and sorting technique makes the query engine extensible and allows it to scale to documents of any size and structure. The utility executes almost all of the basic query patterns associated with XQuery on documents of any size.

The experimental results indicate the throughput of the application to increase linearly in accordance with the size of the document and the complexity of the query. The throughput varies accordingly for queries involving single or multiple documents.

This proves that CanStoreX has indeed been extended from a raw storage format for XML documents to a complete storage structure for XML documents which could be queried and the required results could be extracted. A lot of query patterns still need to be included in the application but the current implementation would serve as a platform to store and retrieve huge documents.

8.2 Future Work

As mentioned before, there are still a lot of query patterns and operators/functions that need to be integrated into the platform to make it a full-fledged XQuery runner. Besides, object-orientation and reference queries need to be incorporated into the query engine. A user interface containing the various options needs to be provided to make the application user-interactive and reduce the hassles required on part of the end-user.

The XML documents are currently being generated from XMark onto a XML file which is then being paginated. This could be replaced with an utility which directly reads the XMark generated feeds without writing them onto a XML file. This would remove the storage space required to store such XML files which could in turn be used as part of the application storage. A project involving this extension is currently in the pipeline and is expected to be integrated into the application sooner. Several applications like these could be coupled on top of this application to enable quick, easy and user-interactive way of storing and handling huge XML documents.

The sorter currently uses a simple sort-and-merge technique through a 2-way merging algorithm. This could be replaced with a more efficient sorter involving long runs, k-way merging, using 2 disks for the algorithm and using simple / advanced techniques to sort the nodes. Further, several other operators such as UNION, INTERSECT, DISTINCT could be implemented on the sorted nodes.

On the optimization part, a lot of features such as indexing, directories, plan generation etc.. needs to be introduced into the query runner. Besides, XQuery is an expression oriented language. The query engine could handle only FLWR expressions; XQuery needs to be made schema-aware to incorporate optimization and object-orientation while advanced features such as dispatching are yet to be introduced.

Currently only XMark is being used as the benchmarking technique to determine the performance of the application. This could be further improved by using other benchmarking techniques such as XMach, XQuery test suite etc. Using diverse set of benchmarking techniques would be useful in comparing the throughput of the query engine with other existing XQuery implementations.

APPENDIX A. EXPERIMENTAL RESULTS

This appendix focuses on the performance metrics of the XQuery application. The chapter contains the results of the various experiments conducted on the application with documents of different sizes and with queries of varying levels of complexity. Some of the features covered in this appendix are the query description, the size of the document(s) the query is acted upon, the running time in seconds and the throughput of the application in megabytes per second. Some entries may be labeled with n/a implying that the results are not available for those due to time and space limitations. The value of throughput may not be accurate for documents of size up to 100 MB due to the very little time consumed with these documents. The time measurements in these cases are usually small and are not captured accurately.

Query Results

The tables are classified based on the type of queries they were tested with. Every table contains documents starting with size 100 KB reaching up to 100 GB in size. The running time of the query in seconds and the throughput in MBytes/sec are captured and are presented in the tables. In addition to the basic FLWR expressions, the application is tested with a few object-oriented queries and benchmark queries from XMark the results of which are specified too.

CanStoreX Performance Results

This section discusses about the general performance of the CanStoreX architecture in terms of paginating and depaginating XML documents. The results of creating the binary storage from the raw XML documents and re-creating thee original XML documents are provided in Table A.1 and Table A.2 respectively to compare the performance of CanStoreX application with the performance of the XQuery.

Table A.1 Pagination Results

Document size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.656	0.152
1	1.031	0.970
10	4.297	2.327
100	66.247	1.510
1000	647.600	1.544
10000	3547.099	2.819
100000	40703.337	2.457

Table A.2 De-pagination Results

Document size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.703	0.142
1	1.000	1.000
10	5.344	1.871
100	77.639	1.288
1000	771.491	1.296
10000	4776.175	2.094
100000	n/a	n/a

FOR-Clause Performance Results

This section is used to describe the performance of a simple FOR-clause without any predicates or filters on documents of various sizes. The clause is evaluated for two different types of expressions; the first type involves clauses with the "/" path expression the results of which are provided in Table A.3 and the second type involves clauses with "//" path expressions for which the results are provided in Table A.4.

Table A.3 Simple FOR-Queries with a "/" path expression

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.797	0.125
1	0.922	1.085
10	1.515	6.601
100	7.264	13.767
1000	67.575	14.798
10000	387.141	25.830
100000	3913.728	25.551

Table A.4 Simple FOR-Queries with a "//" path expression

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	1.000	0.100
1	2.484	0.403
10	15.811	0.632
100	184.950	0.541
1000	1854.781	0.539
10000	14357.222	0.697
100000	n/a	n/a

A variant of FOR-clause uses the "ORDER BY" operator which is used to sort the results according to the specified criteria and in the prescribed order. The FOR-expression along with the sorting mechanism is evaluated for various documents and the results could be found in Table A.5 and Table A.6.

Table A.5 FOR-Queries with an ORDER BY clause along with a ”/” path expression

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.844	0.118
1	1.062	0.942
10	2.234	4.476
100	16.234	6.160
1000	265.303	3.769
10000	3801.642	2.630
100000	n/a	n/a

Table A.6 FOR-Queries with an ORDER BY clause along with a ”//” path expression

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	1.047	0.096
1	2.765	0.362
10	17.718	0.564
100	218.211	0.458
1000	2693.787	0.371
10000	16310.540	0.613
100000	n/a	n/a

FOR-WHERE-Clause Performance Results

The FOR-WHERE clause is used to iterate over document nodes evaluating each node for the specified criteria or filter specified and returning only the filtered set of nodes. This section describes the performance of such clause on documents of various sizes. As with FOR, this expression is evaluated for both "/" and "//" path expressions and the results are depicted in Table A.7 and Table A.8 respectively.

Table A.7 FOR-WHERE-Queries with a "/" path expression

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.812	0.123
1	0.922	1.085
10	1.563	6.398
100	7.999	12.502
1000	72.227	13.845
10000	459.029	21.785
100000	4580.399	21.832

Table A.8 FOR-WHERE-Queries with a "//" path expression

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	1.000	0.100
1	2.469	0.405
10	15.655	0.639
100	188.124	0.532
1000	1886.065	0.530
10000	14407.170	0.694
100000	n/a	n/a

The FOR-WHERE clause is used along with the ”ORDER BY” operator to sort the filtered nodes in the specified order and criteria. The performance results of a FOR-WHERE clause with the sorting technique are specified in Table A.9 and Table A.10

Table A.9 FOR-WHERE-Queries with an ORDER BY clause along with a ”/” path expression

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.828	0.121
1	0.969	1.032
10	1.813	5.516
100	12.300	8.130
1000	131.695	7.593
10000	907.370	11.021
100000	n/a	n/a

Table A.10 FOR-WHERE-Queries with an ORDER BY clause along with a ”//” path expression

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	1.016	0.098
1	2.548	0.392
10	16.145	0.619
100	192.104	0.521
1000	1974.347	0.506
10000	15384.864	0.650
100000	n/a	n/a

LET-Clause Performance Results

The LET Clause is used to evaluate a collection of nodes rather than iterating through each node at a time. The experiments conducted on the LET clause mostly focus on the aggregate operations such as sum,count,max,min,avg. Multiple operators are specified to estimate the efficiency of the application in handling the same. As before, the clause is evaluated for two different types of path expressions and the results are tabulated in Table A.11 and in Table A.12.

Table A.11 LET-Queries with a "/" path expression

Document Size	**Running Time**	**Throughput**
(in MBytes)	**(in seconds)**	**(in MBytes/sec)**
0.1	0.812	0.123
1	0.890	1.124
10	1.375	7.273
100	6.061	16.499
1000	56.376	17.738
10000	268.005	37.313
100000	2685.853	37.232

Table A.12 LET-Queries with a "//" path expression

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.984	0.102
1	2.421	0.413
10	15.465	0.647
100	178.206	0.561
1000	1795.595	0.557
10000	13866.641	0.721
100000	n/a	n/a

FOR-LET-Clause Performance Results

The LET clause could also be used in conjunction with the FOR clause to iterate through the forest of trees evaluating each node for any specified filters in addition to the set operations. This operation involves the functionality of both FOR and LET combined and is evaluated for the path expressions "/" and "//" with documents of different sizes. The experimental results are documented in Table A.13 and Table A.14.

Table A.13 FOR-LET-Queries with a "/" path expression

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.593	0.169
1	0.719	1.391
10	1.188	8.418
100	5.078	19.693
1000	44.574	22.435
10000	287.285	34.809
100000	2675.322	37.379

Table A.14 FOR-LET-Queries with a ”//” path expression

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	1.015	0.099
1	2.765	0.362
10	19.261	0.519
100	233.209	0.429
1000	2337.212	0.428
10000	13977.870	0.715
100000	n/a	n/a

FOR-FOR-Clause Performance Results

The FOR-FOR clause is used to process multiple documents by joining them and extracting information from all of the input sources. For the sake of simplicity, the queries involve joining only two documents to estimate the overall efficiency of the query. The throughput of these queries depend on the size of both of the documents and is apparently much higher when compared to queries operating on single documents. The results provided in Table A.16 and in Table A.17 depict the performance of these queries.

The tables provided below contain the size of both the documents (referred to as outer and inner documents) along with the number of nodes present in the outer document. The number of nodes would vary depending on whether the path expression is ”/” or ”//”. This number determines the number of iterations the inner document needs to be processed to compute the overall size of the join operation as specified in the Table A.15. The throughput is now computed to be the size of the join operation upon the total time spent in processing the request as depicted in the Figure

Table A.15 Node Count on Documents

Document Size (in MBytes)	**No. of item nodes in /region/namerica**	**No. of item nodes in /region/africa**	**No. of item nodes in the entire document**
0.1	10	1	22
1	100	5	217
10	1000	55	2175
100	10000	550	21750
1000	100000	5500	217500
10000	1000000	55000	2175000
100000	10000000	550000	n/a

Table A.16 FOR-FOR-Queries with a ”/” path expression

Outer/Inner Document (in MBytes)	**Size of the join operation (in MBytes)**	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1 / 0.1	0.2	0.734	0.272
0.1 / 1	1.1	1.406	0.782
0.1 / 10	10.1	4.812	2.099
0.1 / 100	100.1	50.789	1.971
0.1 / 1000	1000.1	589.328	1.697
0.1 / 10000	10000.1	4344.445	2.302
0.1 / 100000	100000.1	n/a	n/a
1 / 1	6	1.203	4.988
1 / 10	51	3.171	16.083
1 / 100	501	27.889	17.964
1 / 1000	5001	305.098	16.391
1 / 10000	50001	2268.723	22.039
1 / 100000	500001	n/a	n/a
10 / 10	560	24.156	23.183
10 / 100	5510	294.723	18.696
10 / 1000	55010	3547.692	15.506
10 / 10000	550010	n/a	n/a
10 / 100000	5500010	n/a	n/a
100 / 100	55100	3455.946	15.944
	Data n/a for documents	of higher size	

Table A.17 FOR-FOR-Queries with a ”//” path expression

Outer/Inner Document (in MBytes)	**Size of the join operation (in MBytes)**	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1 / 0.1	0.2	7.296	0.315
0.1 / 1	1.1	67.135	0.329
0.1 / 10	10.1	651.680	0.338
0.1 / 100	100.1	7897.143	0.279
0.1 / 1000	1000.1	n/a	n/a
0.1 / 10000	10000.1	n/a	n/a
0.1 / 100000	100000.1	n/a	n/a
1 / 1	6	650.288	0.335
1 / 10	51	6255.511	0.347
	Data n/a for documents	of higher size	

FOR-FOR-WHERE-Clause Performance Results

The FOR-FOR-WHERE clause introduces predicates and filters to the FOR-FOR clause and provides only the document nodes that satisfy a given criteria. The clause is evaluated against the path expressions ”/” and ”//” the results of which are summarized in Table A.18 and in Table A.19 respectively.

Table A.18 FOR-FOR-WHERE-Queries with a "/" path expression

Outer/Inner Document (in MBytes)	**Size of the join operation (in MBytes)**	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1 / 0.1	0.2	0.813	0.246
0.1 / 1	1.1	0.953	1.154
0.1 / 10	10.1	1.703	5.931
0.1 / 100	100.1	8.139	12.299
0.1 / 1000	1000.1	76.169	13.130
0.1 / 10000	10000.1	555.967	17.987
0.1 / 100000	100000.1	n/a	n/a
1 / 1	6	1.266	4.739
1 / 10	51	3.827	13.326
1 / 100	501	33.992	14.739
1 / 1000	5001	361.319	13.841
1 / 10000	50001	2796.192	17.882
1 / 100000	500001	n/a	n/a
10 / 10	560	31.429	17.818
10 / 100	5510	370.113	14.887
10 / 1000	55010	4230.877	13.002
10 / 10000	550010	n/a	n/a
10 / 100000	5500010	n/a	n/a
100 / 100	55100	3842.612	14.339
	Data n/a for documents	of higher size	

Table A.19 FOR-FOR-WHERE-Queries with a "//" path expression

Outer/Inner Document (in MBytes)	**Size of the join operation (in MBytes)**	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1 / 0.1	0.2	7.185	0.320
0.1 / 1	1.1	65.500	0.337
0.1 / 10	10.1	627.823	0.351
0.1 / 100	100.1	8556.843	0.257
0.1 / 1000	1000.1	n/a	n/a
0.1 / 10000	10000.1	n/a	n/a
0.1 / 100000	100000.1	n/a	n/a
1 / 1	6	630.276	0.346
1 / 10	51	6216.832	0.349
	Data n/a for documents	of higher size	

XMark Benchmark Queries Performance Results

This section tabulates the performance of the application in executing the benchmark queries generated by XMark. These queries are complex involving extensive traversal of the documents and could be used to evaluate the efficiency of the application. Since the queries involve XQuery compatible expressions and operators, the application does not support all queries since it is not a full-fledged XQuery implementation but contains only a subset of the query patterns with the queries modified to remove the unsupported functions. The tables provided below depict the throughput of the application for different queries on documents of various sizes. The XMark generated queries are available at [37].

Table A.20 Query 1 Performance Results

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.515	0.194
1	0.594	1.684
10	1.047	9.551
100	5.453	18.339
1000	48.076	20.800
10000	315.751	31.671
100000	3180.840	31.438

Table A.21 Query 2 Performance Results

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.532	0.188
1	0.703	1.422
10	1.453	6.882
100	11.999	8.334
1000	111.838	8.942
10000	684.766	14.604
100000	6871.183	14.554

Table A.22 Query 3 Performance Results

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.578	0.173
1	0.797	1.255
10	2.093	4.778
100	22.499	4.445
1000	219.084	4.564
10000	1378.516	7.254
100000	n/a	n/a

Table A.23 Query 5 Performance Results

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.719	0.139
1	2.078	0.481
10	14.999	0.667
100	191.351	0.523
1000	1934.835	0.517
10000	13866.641	0.721
100000	n/a	n/a

Table A.24 Query 6 Performance Results

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.766	0.131
1	2.891	0.346
10	23.952	0.418
100	318.735	0.314
1000	3213.819	0.311
10000	n/a	n/a
100000	n/a	n/a

Table A.25 Query 7 Performance Results

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	1.328	0.075
1	5.453	0.183
10	45.969	0.218
100	559.515	0.179
1000	6390.191	0.156
10000	n/a	n/a
100000	n/a	n/a

Table A.26 Query 13 Performance Results

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	2.047	0.049
1	2.062	0.485
10	2.281	4.384
100	2.984	33.512
1000	8.375	119.403
10000	39.952	250.300
100000	380.030	263.137

Table A.27 Query 15 Performance Results

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	2.109	0.047
1	2.297	0.435
10	2.578	3.879
100	8.922	11.208
1000	69.795	14.328
10000	370.613	26.982
100000	3661.805	27.309

Table A.28 Query 16 Performance Results

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.531	0.188
1	0.562	1.779
10	0.812	12.315
100	2.859	34.977
1000	23.093	43.303
10000	139.603	71.632
100000	1369.408	73.024

Table A.29 Query 17 Performance Results

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.531	0.188
1	0.641	1.560
10	1.000	10.000
100	5.531	18.080
1000	49.389	20.247
10000	322.114	31.045
100000	3171.134	31.534

Table A.30 Query 18 Performance Results

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.547	0.183
1	0.609	1.642
10	0.954	10.482
100	4.891	20.446
1000	42.999	23.256
10000	236.883	42.215
100000	2363.800	42.305

Table A.31 Query 19 Performance Results

Document Size (in MBytes)	**Running Time (in seconds)**	**Throughput (in MBytes/sec)**
0.1	0.610	0.164
1	1.234	0.810
10	7.031	1.422
100	94.325	1.060
1000	1177.181	0.849
10000	n/a	n/a
100000	n/a	n/a

BIBLIOGRAPHY

[1] Bray, T. , Paoli, J. , Sperberg-McQueen, C. M. , Maler, E. , Yergeau, F. (2006). Extensible Markup Language (XML) 1.0 (Fourth Edition) W3C recommendation, 16 August 2006.

[2] Bohannon, P. , Freire, J. , Roy, P. , Simeon, J. (2002). From XML schema to relations: a cost-based approach to XML storage. In *Proceedings of the 18th International Conference on data Engineering (ICDE'02)*, pages 64-75, San Jose, CA, USA, 2002.

[3] Tatarinov, I. , Viglas, S. , Beyer, K. , Shanmugasundaram, J. , Shekita, E. , Zhang, C. (2002). Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204-215, Madison, WI, USA, 2002.

[4] Shanmugasundaram, J. , Shekita, E. , Kiernan, J. , Krishnamurthy, R. , Viglas, E. , Naughton, J. , Tatarinov, I. (2001). A general technique for querying XML documents using a relational database system. *Special section on advanced XML data processing at ACM SIGMOD*, Volume 30, Issue 3, pages 20-26, 2001.

[5] Bohannon, P. , Freire, J. , Roy, P. , Haritsa, J. , Simeon, J. , Ramanath, M. (2002). LegoDB: Customizing Relational Storage for XML Documents. In *Proceedings of the 28th VLDB Conference*, Hong Kong, China, 2002.

[6] Florescu, D. , Kossmann, D. (1999). Storing and querying XML data using an RDBMS. IEEE Data Eng Bull 22(3):27-34, 1999.

[7] Fiebig, T. , Helmer, S. , Kanne, C. -C. , Moerkotte, G. , Neumann, J. , Schiele, R. , Westmann, T. (2002). Anatomy of a native XML base management system. Springer-Verlag, 2002.

[8] Jagadish, H. V. , Al-Khalifa, S. , Chapman, A. , Lakshmanan, L. V. S. , Nierman, A. , Paparizos, S. , Patel, J. M. , Srivastava, D. , Wiwatwattana, N. , Wu, Y. , Yu, C. (2002). TIMBER: A native XML database. VLDB Journal, 11(4): 274-291, 2002.

[9] Kanne, C. , Moerkotte, G. (2000). Efficient Storage of XML Data. In *Proceedings of the 16th International Conference on Data Engineering*, page 198, 2002.

[10] Meng, X. , Luo, D. , Lee, M. L. , An, J. (2000). OrientStore: A Scheme Based Native XML Storage System. In *Proceedings of the 29th VLDB Conference*, Berlin, Germany, 2003.

[11] Cognetic Systems, Inc. XQuantum: A XML Native Data Store. http://www.cogneticsystems.com/

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

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

[14] BlueStream, Corp. XStreamDB. http://www.bluestream.com/products/xstreamdb32

[15] Ives, Z. G. , Halevy, A. Y. , Weld, D. S. (2002). An XML query engine for network-bound data. VLDB Journal, Volume 11, Number 4, pages 380-402, 2002.

[16] Fegaras, L. , Elmasri, R. (2001). Query Engines for Web-Accessible XML Data. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 251-260, 2001.

[17] 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.

[18] Chamberlin, D. , Robie, J. , Florescu, D. (2000). Quilt: An XML query language for heterogeneous data sources. In *Proceedings of WebDB 2000 Conference*. In *Lecture Notes in Computer Science*, Springer-Verlag, 2000.

[19] Clark, J. , DeRose, S. (1999). XML path language (XPath) version 1.0. Technical report, World Wide Web Consortium (W3C) recommendation 16 November 1999.

[20] DeHaan, D. , Toman, D. , Consens, M. P. , Ozsu, M. T. (2003). A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of the ACM SIGMOD*. ACM Press, June 2003.

[21] Fankhauser, P. , Groh, T. , Overhage, S. (2002). XQuery by the book: The IPSI xquery demonstrator. In *Proceedings of the International Conference on Extending Database Technology*, 2002.

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

[23] Choi, B. , Fernandez, M. , Simeon, J. (2002). The XQuery Formal Semantics: A Foundation for Implementation and Optimization. May 2002.

[24] Simeon, J. , Fernandez, M. Galax: An implementation of XQuery. http://db.bell-labs.com/galax/optimization.

[25] Pal, S. , Cseri, I. , Seeliger, O. , Rys, M. , Schaller, G. , Yu, W. , Tomic, D. , Baras, A. , Berg, B. , Churin, D. , Kogan, E. (2005). XQuery implementation in a relational database system. In *Proceedings of the 31st international conference on very large data bases*, pages 1175-1186, Trondheim, Norway, 2005.

[26] Graves, M. (2002). Designing XML Databases. Prentice Hall.

[27] Grust, T. (2002). Accelerating XPath location steps. In *Proceedings of the 21st ACM SIGMOD international conference on management of data*, 2002.

[28] Ma, S. , Gadia, S. K. , Berleant, D. , Huang, X. (2004). Implementation of a canonical native storage for XML. Master's Thesis. Department of Computer Science. Iowa State University, 2004.

[29] Patanroi, D. , Gadia, S. K. , Leavens, G. T. , Hyde, W. G. (2005). Binary page implementation of a canonical native storage for XML. Master's Thesis. Department of Computer Science. Iowa State University, 2005.

[30] Ramakrishnan, R. , Gehrke, J. (2000). Database Management Systems, third edition, McGraw Hill.

[31] Sahuguet, A. , Dupont, L. , Nguyen, T-L. . Kweelt. http://kweelt.sourceforge.net/

[32] Nandakumar, S. , Gadia, S. K. (2005). Implementing a parser for the XQuery grammar on Kweelt platform. Department of Computer Science. Iowa State University, 2005.

[33] Stark, R. , Gadia, S. K. (2006). Implementing a primitive version of DOM Interface for CanStoreX. Department of Computer Science. Iowa State University, 2006.

[34] Krithivasan, S. , Swanson, M. , Gadia, S. K. (2006). Building a XQuery application for CanStoreX on the Kweelt platform. Department of Computer Science. Iowa State University, 2006.

[35] Le Hors, A. , Le Hegaret, P. , Wood, L. , Nicol, G. , Robie, J. , Champion, M. , Byrne, S. (2004). Document Object Model (DOM) level 3 core specification. Technical report, World Wide Web consortium (W3C) recommendation 07 April 2004.

[36] Megginson, D. (2001). SAX: A Simple API for XML. Technical Report, Megginson Technologies, http://www.saxproject.org/

[37] Schmidt, A. XMark - An XML Benchmark Project. http://monetdb.cwi.nl/xml/