Iowa State University
Digital Repository

Graduate Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

2009

A workbench for advanced database implementation and benchmarking

Valliappan Narayanan
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Narayanan, Valliappan, "A workbench for advanced database implementation and benchmarking" (2009). *Graduate Theses and Dissertations*. 10667.
https://lib.dr.iastate.edu/etd/10667

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

A workbench for advanced database implementation and benchmarking

by

Valliappan Narayanan

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Shashi K. Gadia, Major Professor
Johnny S. Wong
Arun K. Somani

Iowa State University

Ames, Iowa

2009

Copyright © Valliappan Narayanan, 2009. All rights reserved.

TABLE OF CONTENTS

LIST OF FIGURES vi
ABSTRACT vii
CHAPTER 1. INTRODUCTION 1
1.1. XML in our implementation 1
1.2. Introduction to CanstoreX 3
1.3. Query Engine 4
1.4. Parametric Data Model. 5
1.4.1. TempDB: a prototype for a temporal database 5
1.4.2. NC-94: A prototype for spatiotemporal database 6
1.5. Files, command-based paradigm, and a common GUI 6
1.6. Contributions under the current project 7
CHAPTER 2. SYSTEM INTEGRATION 9
2.1. Architecture of Integrated System 9
2.2. Storage Platform Component 9
2.2.1. Storage Layer. 9
2.2.1.1. Creation of Storage 11
2.2.1.2. Management of Storage 11
2.2.2. Buffer Manager Layer 12
2.3. SubSystems Component. 12
2.4. Common GUI Component 13
2.4.1. GUI Layout and Options 13
2.4.2. GUI Parser 15
2.4.3. GUI Commands 15
2.4.3.1. Declare Command. 15
2.4.3.2. Set Command 16
2.4.3.3. CreateLog Command 16
2.5. Other Components in the Integrated System 17
2.5.1. Storage Manager 17
2.5.2. File Manager 17
2.5.2.1. Pagination Utilities 17
2.5.2.2. Sorter Utility 18
2.5.3. CxDOM API Layer 18
2.6. SubSystem Organization 18
2.6.1. Physical Directory Structure of SubSystem 18

2.6.2. Registering SubSystems with the Common Storage Platform 19
2.7. Background for Integrating NC94 Subsystem 19
2.8. Steps for integrating remaining SubSystems 19
CHAPTER 3. CURRENT SUBSYSTEMS 21
3.1. Storage Manager Commands 21
3.1.1. CreateRawStorage Command 21
3.1.2. FormatStorage Command 22
3.1.3. StartBufferManager Command 23
3.1.4. UseStorage Command 23
3.1.5. ShowDirectory Command 23
3.1.6. GetPageAllocatedCount Command 24
3.1.7. GetPageDeAllocatedCount Command 24
3.1.8. GetPageAccessCount Command 24
3.1.9. GetRelativePageAccessCount Command 24
3.1.10. ResetRelativePageAccessCounter Command 24
3.2. File Manager Commands 24
3.2.1. CreateFile Command 25
3.2.1.1. CreateFile XML Command 25
3.2.1.2. CreateFile BXML Command 25
3.2.1.3. CreateFile CXML Command 26
3.2.2. CopyFile Command 26
3.2.2.1. CopyFile XML to XML Command 26
3.2.2.2. CopyFile XML to BXML Command 27
3.2.2.3. CopyFile XML to CXML Command 27
3.2.2.4. CopyFile BXML to XML Command 28
3.2.2.5. CopyFile BXML to BXML Command 28
3.2.2.6. CopyFile BXML to CXML Command 29
3.2.2.7. CopyFile CXML to XML Command 29
3.2.2.8. CopyFile CXML to BXML Command 30
3.2.2.9. CopyFile CXML to CXML Command 30
3.2.3. DeleteFile Command 31
3.2.3.1. DeleteFile XML Command 31
3.2.3.2. DeleteFile BXML Command 32
3.2.3.3. DeleteFile CXML Command 32
3.3. CanStoreX Pagination Utility 32
3.3.1. Background 33
3.3.2. Improved Attribute Representation in Depagination Algorithm 34

3.4. XQuery SubSystem 37
3.4.1. Kweelt Architecture 37
3.4.2. XQuery Parser 38
3.4.3. XQuery Evaluation Engine 39
3.4.4. Index Structure for BXML documents in the Storage 40
3.5. ElementalDB SubSystem 41
3.6. TempDB SubSystem 41
3.7. Quilt Engine 42
3.8. SQL Runner SubSystem. 42
3.9. NC94 SubSystem 42
CHAPTER 4. INTEGRATION OF NC94 SUBSYSTEM 43
4.1. NC94 System 43
4.2. NC94 XML Directory Structure 43
4.2.1. NC-94 XML Catalog 44
4.2.2. Relation XML Catalog 44
4.3. NC-94 Loader 45
4.4. NC94 Commands 46
4.4.1. LoadNC94Data Command 46
4.4.2. LoadNC94GMLData Command 47
4.4.3. OpenNC94Database Command 47
4.4.4. Variable Declaration and Query Assignment 47
4.4.5. ParseQuery Command 48
4.4.6. DisplayParseTree Command 48
4.4.7. BuildExpressionTree Command 48
4.4.8. DisplayExpressionTree Command 49
4.4.9. ExecuteQuery Command 49
4.4.10. ShowIterators Command 50
4.4.11. OpenIterator Command 50
4.4.12. GetNextTuple Command 50
4.4.13. GetRemainingTuples Command 51
4.4.14. HasNextTuple Command 51
4.4.15. CloseIterator Command 51
4.4.16. Single command Option for NC94 Query Processing 51
4.4.17. CloseNC94Database Command 52
4.4.18. DeleteFile nc94 Command 52
4.4.19. DisplayNC94Catalog Command 53
4.5. Prior Work 53

4.6. Integration of NC94 SubSystem 53
CHAPTER 5. CONCLUSION AND FUTURE WORK 55
5.1. Conclusion 55
5.2. Future Work 55
BIBLIOGRAPHY 56
APPENDIX A. 58
APPENDIX B. 59
ACKNOWLEDGEMENTS 62

LIST OF FIGURES

Figure 1. Architecture of Integrated System 10
Figure 2. Common GUI 14
Figure 3. Snapshot of RawStorageConfig.xml document 22
Figure 4. Textual representation of Sample XML document 33
Figure 5. Logical representation of Sample XML document 34
Figure 6. Binary Storage structure for the XML Document 35
Figure 7. Snapshot of Original XML document before pagination 36
Figure 8. Snapshot of Original depaginated XML document with "_A" tag 36
Figure 9. Snapshot of improved depaginated XML document 36
Figure 10. Snapshot of the Index Structure 41
Figure 11. NC-94 hybrid storage structure 44
Figure 12. NC-94 XML catalog 45
Figure 13. Snapshot of the Climate Relation XML catalog 46

ABSTRACT

This work focuses on a methodology to help bring many of our database artifacts and prototypes to reside on the top of a common workbench platform that leads to uniformity and removes overlap across different subsystems. A versatile command format has been developed to allow commands belonging to different subsystems to be interleaved in the same batch unambiguously. Through a collaborative effort carried on in parallel, existing GUIs (Graphical User Interfaces) have also been merged into a common, but simple GUI. The GUI executes a batch of commands.

Subsystem currently included are: a runner for SQL on a variety of database platforms, a runner for Quilt queries (Quilt is an early version of XQuery and runs on a platform called KWEELT), ElementalDB, an experimental database system used for instruction in a graduate database implementation course, our own XQuery engine which aims at handling data in terabyte range stored in our storage in paginated form using our pagination algorithm, a research prototype for NC94, an important spatiotemporal data set in agriculture, and a research prototype for a temporal database. The organization of the subsystems follows strict convention for ease of further development and maintenance. XML is used extensively by various subsystems. An XML based framework has been developed for benchmarking subsystems to make experiments completely repeatable at click of a button starting from creation of storage, loading of data sets, execution of commands, collecting performance data in XML-based logs to reporting using XQuery queries on the XML logs.

With a very small learning curve, the resulting workbench can be used by students, instructors, developers and researchers alike and managed easily.

CHAPTER 1. INTRODUCTION

This thesis describes the project undertaken to bring many of our database subsystems and prototypes to reside on the top of a common workbench platform in order to bring uniformity across different subsystems in the way they are used and developed in instruction as well as research projects.

This chapter provides a brief introduction to various technologies, concepts and applications which are part of this project. We will discuss about XML technology, XQuery language, parametric data model and the CanStoreX (Canonical Storage for XML) pagination utility for binary pagination of larger XML documents. We will also mention GUIs to run commands in SQL and Quilt (an early version of XQuery), and ElementalDB a database system for instruction in database implementation. It also briefs about the prior work, major contributions of this thesis and the organization of the remaining chapters.

1.1. XML in our implementation

Extensible Markup Language (XML) is a simple tree-based model for representing and standardizing information content. Its tree-based structure goes hand-in-hand with DOM API and XPath technologies making it inherently versatile, powerful, and efficient. XML is a technology that enables other technologies. We have a keen interest in XML itself as a database and also its multifaceted extensive deployment in facilitating database implementation. Our uses of XML include representation of system configuration, metadata, syntax and expression trees, physical representation of complex data, and storage of large XML documents in terabyte range and their access through our own customized DOM API, XPath, and XQuery technologies. We are also interested in XML as logical data model.

We partition our usage of XML documents into two groups. The first group consists of small XML documents where direct access from operating system is important. Configuration files, metadata, and artifacts such as syntax and expression trees are in this group. Files in this group are stored in the common text-based*.xml* format in the files in the operating system. In such cases because of small document size the commonly available DOM API is used and the efficiency issues are highly marginal.

The commonly available DOM API is reasonable only for small documents. It uses SAX (Simple API for XML) to parse a text-based physical representation of an XML document and stores the document in main memory in form of a tree materializing parent, child, and sibling pointers. This requires a great deal of memory, 5 to 10 times the size of its text-based representation. In spite of that we use it for small documents making them directly accessible without going through any specialized technologies.

In the second group we have XML documents when they are used for representation of physical or logical layers of data. XML documents in this group are potentially very large and we use our own storage and access technologies that come under the banner of CanStoreX described below. Our aim is to be able to use XML documents that can be in terabyte range. Utilities to copy XML files between the operating system and our storage are also available in our workbench.

The term logical and physical in the context of our databases implementation can be a bit puzzling. To motivate this we give following examples.

- A user, e.g. an XQuery user, may be interested in an XML document. In that case the XML document is logical. However the document can be stored in a variety of ways that are all physical.
- A relation in databases from a user's point of view is a set of tuples. The industry standard format for its physical representation consists of binary sequences of tuples that are stored in pages. In order to implement a relational database system, XML can be used to modularize the code. To see this consider three layers: query processor layer, iterator layer, and storage layer. The query processor layer can pretend that the relation is actually an XML document. The query processor depends on the iterator layer that supplies tuple at a time. The the query processor can be implemented more easily using the DOM / API, leading to code that is more human readable and easier to understand. The code can then be tested in isolation of the storage layer by creating test cases where the relations are actually represented as an XML document and an iterator, also written using DOM / API, to supply one tuple at a time that is an XML element. Once this is done, or in parallel, for the same iterator interface its implementation can be changed so that tuples are fetched

directly from the industry standard binary storage mentioned above. In this case the relation as a set of tuples is the logical representation, XML representation of the relation is the physical representation, which in turn is physically represented in industry standard binary format.

- In our implementation we often encounter data that has complex logical structure, such as a spatiotemporal relations, where tuples are functions with space and time dimensions. Such a logical tuple can be physically represented in XML that is invisible to the user but understood well by internals in the system.

We note that SAX (Simple API for XML) is suited only when one needs to parse an XML document that is stored linearly in depth first order, for example the usual text-based representation of XML document as *.xml* files. SAX is not suitable for navigation in XML documents because of their tree-based structure. For example SAX is used by DOM to create a tree-based representation of a document. Once it is represented in memory, DOM API can be used for access and maintenance. Therefore, SAX is sometimes called a parser. DOM API is also sometimes referred to as a parser, but use of the term "parser" does not express the true nature of DOM. We view the representation of XML as a text file as a transient concept in evolution of XML technology. There is no theoretical requirement that should stop us from representing an XML documents in form of a trees that always remains ready to be processed using DOM and XPath technologies.

The text-based representation is suitable only for small XML documents. The storage technologies for large XML documents can be classified into two broad categories: native and non-native. The term "native" in the context of storage technologies is used in many different ways and it is difficult to arrive at a common definition. We are only interested in native storage and define reserve it for those representations where the gap between physical and logical representations is minimal. Two well-known native storages for XML are Natix [4] and Timber [5]. We have a serious interest in native storage of XML. We are developing our own storage technology called a Canonical Storage for XML (CanStoreX) described below.

1.2. Introduction to CanstoreX

CanStoreX (a Canonical Storage for XML), breaks an XML document into pages and stores it

in the paginated form in the storage. Each page itself is organized as a self contained legal XML document. Pages within the storage are interconnected using special nodes called *storage facilitating nodes*. A multitude of such nodes have been envisioned and left to future exploration. Currently, two types of storage facilitating nodes are used: f-node and c-node. The f-node is used to group one or more siblings having the same parent. A c-node contains a pointer to a child page where a subtree rooted at an f-node resides. Thus the whole XML document becomes a tree of pages where each page, and XML document on its own right, is stored as a tree in a binary format. Obviously, an XML document in CanStoreX remains in a ready state for access. Unlike Natix, updates have not yet been implemented in CanStoreX, but they are planned for the near future.

The initial implementation of CanStoreX for storing XML document and DOM API called DiskDOM for their access was undertaken by Shihe Ma [9]. He used textual representation for pages, and a common Java based DOM API for navigation within pages. Thus every page was seen as a java based Node object by his DiskDOM. Due to heap-based management of memory for the objects in Java, the system would choke when paginating or accessing XML documents beyond 1 gigabyte range. In order to scale it up, binary page implementation was undertaken by Daniel Patanroi [10]. Currently, binary pagination algorithm has been tested for XML documents up to 1 terabytes in size. As the pace for pagination was linear, it seemed that we had a pagination technology that would handle XML documents in terabyte range and beyond. Patanroi also made improvements in the pagination algorithm making it more efficient.

The development of CxDOM, the DOM API for binary XML documents in the storage was continued by Bob Stark, Srikanth Krithivasan, and Matt Swanson. Srikanth also improved the pagination algorithm.

1.3. Query Engine

XQuery is a functional programming language recommended by XML Query working group of World Wide Web (W3C) consortium for query of XML documents [15]. XQuery provides the means to extract and manipulate data from XML documents. The syntax of XQuery is inspired by SQL; whereas SQL allows relations to be queried as predefined collections of tuples, XQuery uses

XPath to form versatile collections to query them.

Quilt is an older version of XQuery language. Sahuguet implemented Kweelt [19] a query engine for execution of Quilt queries. We have also implemented an XQuery engine. It is styled after Kweelt platform with two main differences. First, the Quilt query language has been replaced by the official version of XQuery; a parser for the latter was developed by Satyadev Nandakumar [13]. Second, instead of storing the XML documents as plain text operating system (.xml) files and using the common DOM API, Krithivasan undertook the implementation of the engine for the XML documents in the storage in binary paginated form on the top of CxDOM API. As seen before, the implementation of CxDOM, initially called DiskDOM, was undertaken by Ma, Patanroi, Stark, and Krithivasan. Krithivasan implemented the NodeList class in the CxDOM API. The XQuery engine implemented by Krithivasan also includes external (disk-based) sorting of XML forests as needed by the sort by clause of XQuery. The implementation does not yet cover all types of XQuery queries and further development of XQuery engine is expected to continue for some time to come.

1.4. Parametric Data Model

Parametric data model is a data model for data with one or more dimensions. Example of dimensions are space and time. A parametric space consists of a set of points such instants of time or points in space. In the parametric approach for implementation of parametric databases, attribute values are defined as a functions from a subset of the parametric space [8]. Every object is modeled as a single tuple creating a one-to-one correspondence between an object in the real world and a tuple in the database. This makes user queries simpler avoiding un necessary self joins. It is difficult to implement the parametric data model on top of conventional database storage because tuples have complex structure and their length is not fixed. The challenge is easily met by XML.

1.4.1. TempDB: a prototype for a temporal database

A prototype for a one dimensional temporal database was implemented by Seo-Young Noh. The temporal tuples are physically modeled as XML elements. Thus XML is used as the storage layer.

1.4.2. NC-94: A prototype for spatiotemporal database

North Central Regional Association of State Agricultural Experiment Station Directors (NCRA) decided to develop a strategy for data collection that helps in study, efficient crop management and to reduce the risk factors associated with agriculture practices, which depends on a various factors [16]. Analysis of such data collection helps predicting patterns and guide farmers to make good decisions regarding their produce. Association's primary focus was to collect and organize data on the states in the north-central region of the United States. Climate, crop and soil data were collected for these states. The result of these data collection and organization efforts was the NC-94 dataset. The NC-94 data covers a 30-year span from 1971 to 2000 and climate, crop, and soil data have their different time granularities. For climate data, measurements were on daily basis, for crop data, values were recorded yearly and soil data is relatively time invariant. The spatial granularity of all these datasets is a county with some minor but intricate variations. A prototype for NC-94 dataset for validation of parametric model for spatiotemporal data was implemented by Seo-Young Noh. Over a period of time, our storage technology in general and for XML documents in particular continued to evolve and diverged with the one deployed by Noh [7]. Niranjan Kumar [17] ported NC-94 to the existing storage platform. Many other structural changes were made. Initially only Climate dataset was available to us. After Soil and Crop datasets became available, Kumar added them to the NC-94 prototype. In addition, Kumar also implemented loading of the datasets from Microsoft Access format to our own XML-based format in our storage. More and more of the prototype was made command-centric. The command-centric approach is discussed below.

1.5. Files, command-based paradigm, and a common GUI

Krithivasan and Swanson began the implementation of XML files in multiple formats and commands to create and copy files from one format to another. As mentioned above, we already have had GUIs for running SQL and Quilt (an early version of XQuery language) commands that have been used in introductory undergraduate course in databases at Iowa State for several years. The SQL runner allows a user to connect to and execute SQL commands on any relational database system such as Microsoft Access, Oracle, or SQL

Server. But for Quilt only one platform, called Kweelt, was available. Therefore Kweelt was incorporated as the back end for the Quilt runner. The Kweelt platform was built on the top of common DOM API and therefore worked only for small XML documents. However, it did enable XML and XQuery to be tightly integrated in the undergraduate introductory database course.

At this point a clear direction emerged. It became clear to us that our entire database implementation should become command-centric. The concept of prefix, to indicate the subsystem the command applied to, was also added making it possible to interleave commands from multiple subsystems in the same batch for their sequential execution. In addition to the GUIs for SQL and Quilt commands, we also had GUIs for ElementalDB, NC-94 and temporal database prototypes. It was also decided to merge all the GUIs into a single GUI with the objective that the resulting GUI for users will be as simple as the simplest of all our GUIs and have a functionality that exceeds the sum total of functionalities of all other GUIs. This was to be accomplished by the concept of command prefix mentioned above and in addition creation of commands that are instructions to the GUI itself. A comprehensive plan for unification of all subsystems thus emerged.

1.6. Contributions under the current project

The comprehensive plan for unification of subsystems with common storage and GUI mentioned above has been initiated and reported in this thesis. We mainly focusses on the system integration to bring different database prototypes to reside on the top of common terabyte storage platform and share the single common Graphical User Interface (GUI). The project also led to modifications of some subsystems. The enhancements to the common storage platform includes adding new file management commands like copying and deleting popular file types from the storage. New commands were added for the creation of the storage following configuration specified by XML files. Not all the subsystems have been integrated. The new integrated system have placeholders to bring other subsystems like temporal database, elemental database on the common storage platform in the near future.

In System Integration, though the work resembles more of an integration of the various subsystems, the core of the development involved understanding of the subsystem and

modifying them to bring it to the common storage platform and merging the different Graphical User Interface (GUI) for subsystems into single common GUI. It also involved creating commands for all the functionalities which was available in the old GUI only through the use of specialized radio buttons. These buttons have now been absorbed by the commands. The common storage platform will be shared by many subsystems and the common GUI will be used by all the subsystems.

The rest of the thesis is organized as follows. Chapter 2 describes the architecture of integrated system, the components of the integrated system, target subsystems and how these subsystems can be efficiently integrated on the common storage platform and the common GUI. Chapter 3 provides the current status of the common storage platform, CanStoreX pagination utility, XQuery engine and briefs about other target subsystems. Chapter 4 describes the integration of NC94 subsystem on the common storage platform and the enhancements made to NC94 subsystem as a part of integration process. Chapter 5 contains conclusions and scope for future work.

CHAPTER 2. SYSTEM INTEGRATION

Over the years, our research group have developed several database subsystems that have a large overlap in the storage modules and the Graphical User Interfaces (GUIs) which lack uniformity. This chapter describes the common platform and how different subsystems can be integrated on to the common storage platform and the common GUI for achieving uniformity and consistency. In addition it is also described how the subsystems are organized in a way that the upper layers are mutually exclusive and self contained.

2.1. Architecture of Integrated System

In the new integrated system, the common storage platform is shared by most of our subsystems and the common GUI is shared by all our sub systems. The architecture of integrated system is shown in Figure [1]. Our GUI is command based. Every command will have a prefix as the part of the command.

The integrated system has 3 major components. The underlying component in the architecture is the storage platform which takes cares of data storage. Above the storage platform are the subsystems components which includes the different database subsystems. Above the subsystem is the common GUI components which interacts with the user and the subsystems. The below subsections describes about each components.

2.2. Storage Platform Component

The *Storage Engine* module in the storage platform component have two layers which are explained in the below subsections. Currently, the storage engine accesses the disk via Java random access files.

2.2.1. Storage Layer

The bottom and the most important layer of storage engine is the storage layer. Storage manager component in the integrated system manages the storage layer which is responsible for two important functionalities, creation of storage and management of storage. The storage manager uses commands for creating the storage and the suffix of the commands are *sm*.

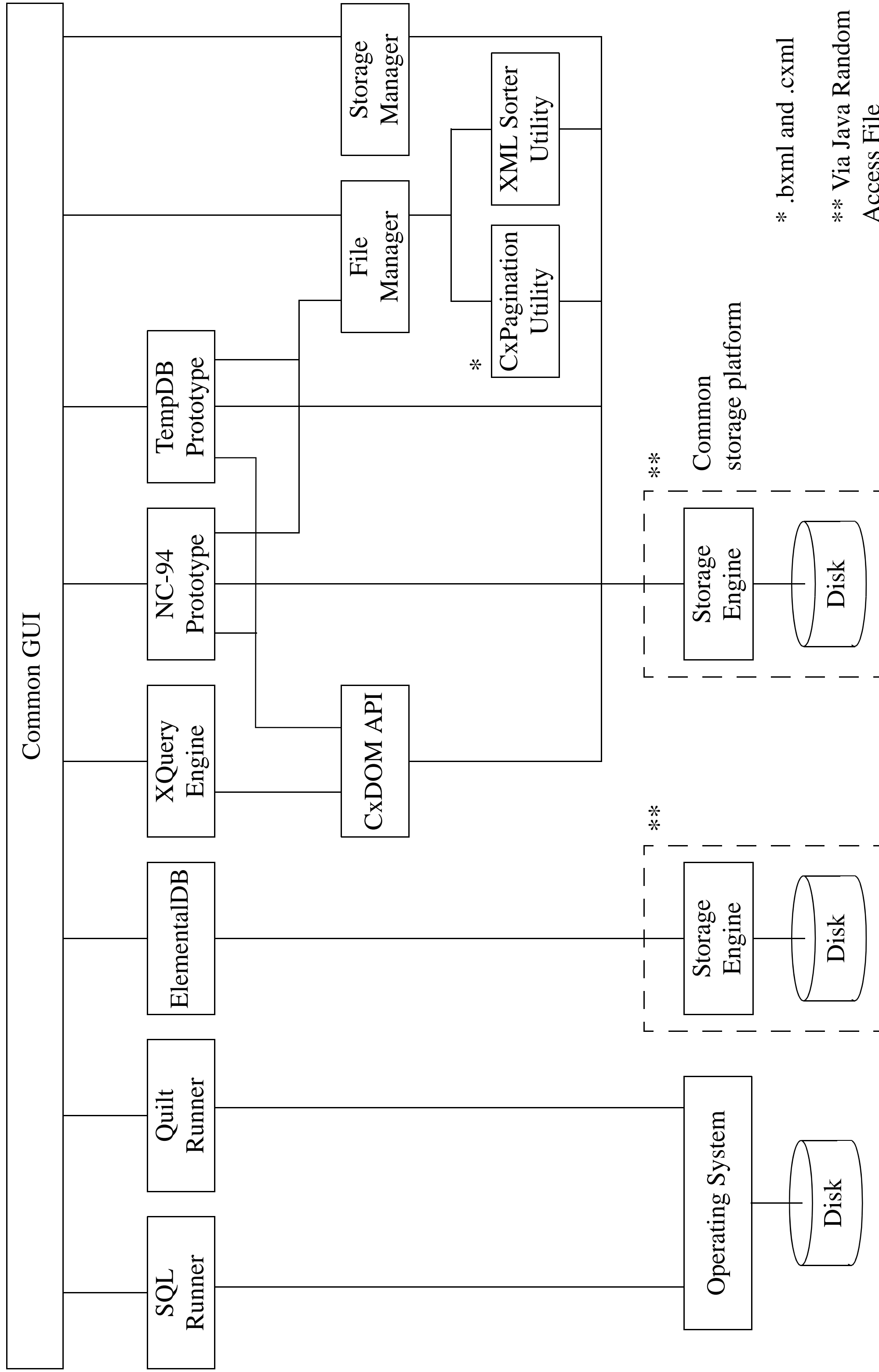

Figure 1. Architecture of Integrated System

2.2.1.1. Creation of Storage

In our common platform, byte level storage is the lowest level of storage, akin to a disk. The storage is created in two steps. In the first step we acquire raw storage, called *raw storage*, from the disk by creating an array of java random access files. The random access files can straddle any number of disks. In the next step the raw storage is paginated according to a specified page size to obtain *paginated storage* or simply *storage*.

There exists a centralized XML configuration file called *RawStorageConfig.xml* where user can configure details of the storage to be created and then execute a storage command in the GUI for creating a storage. The configuration details includes path of the storage, total size of the storage, page size and also the size of a single random access file. The storage manager determines the number of random access files from the total storage size and the size of a single random access file and then create them. Once the raw storage is created, the storage manager formats the raw storage with the given pagesize into many pages. Initial pages containing bit map is then created. For each page, a *flag bit* is included to indicate whether it has been allocated or free. The paths of the storage can be different physical directories on different disks. This entire byte-level storage is viewed as a single sequence of bytes.

2.2.1.2. Management of Storage

The storage manager supports the concept of a page as a unit of data and manages the storage using commands that *allocate*, *deallocate*, *read*, or *write* one page at a time. The client requests a free page by *AllocatePage* command, the storage manager finds a free page n by sequentially checking each page's *flag bit* and a returns the pageID of the first free page and sets its flag bit to indicate the page has been allocated. Pages are read and written using ReadPage(n,b) and WritePage(n,b) where n is the page number and b is a buffer that has same number of bytes as a page. When client executes *DeallocatePage(n)* when page n is no longer needed. The command sets the allocation bit of the page to free. The storage manager keeps track of the counters for allocated pages, deallocated pages, page accesses and the disk block accesses. The disk accesses involved in using *allocate* and *deallocate* commands are considered as system's overhead and won't be counting towards client's disk accesses count.

2.2.2. Buffer Manager Layer

The top layer of the storage engine is the buffer manager layer. It is responsible for bringing pages from the storage to main memory as required. The size of each buffer is equal to the size of the pages and the number of buffers to be used is given by the client using a storage command with *sm* prefix. Buffers are used both for reading pages from the disk and for writing pages to the disk. The collection of buffers is called the *buffer pool*. The buffer manager manages the buffer pool. When all the buffers in buffer pool are full, the buffer manager uses buffer replacement policy to decide the victim buffer. The buffer manager also keeps track of counters for the page accesses and number of pages allocated and deallocated. By monitoring the counts a client can gaze at the performance and ensure no garbage is left in the storage.

2.3. SubSystems Component

This section describes different subsystems that are targeted for integration into a single system. Different subsystems are listed below.

- **XQuery Engine** - XQuery engine uses both the common storage platform and the common GUI for executing its command and the suffix is *XQuery*. XQuery can be used to query XML documents stored as binary paginated *.bxml* documents in the storage.
- **NC94 System** - NC94 uses both the common storage platform and the common GUI. Commands with the prefix *nc94* are available for loading the NC-94 datasets in to the storage in varieties of formats, open the NC-94 database, and query them. For queries, commands are available that allows a user to step through their compilation to execution. Alternatively a query can be executed from the beginning to the end in a single step using the *nc94query* prefix.
- **Temporal Database** - Temporal database (TDB), not yet integrated, will use both the common storage platform and the common GUI. Commands along the lines of NC94 prototype will be developed. The command suffixes *tdb* and *tdbquery* will be used.
- **ElementalDB** - Elemental database has its own storage and buffer manager for managing its storage and hence it uses only the common GUI for executing its commands. The suffix for ElementalDB command will be *.edb*.

- **Quilt Engine** - The third party kweelt execution engine is used for executing Quilt query on the standard XML documents stored on the disk and hence it doesn't use the common storage platform but uses common GUI for executing its commands. The suffix for kweelt command is *quilt*.

- **SQL Runner** - SQL runner is a simple GUI used to establish a connection with a relational database and pass the query to database and then return the resultant tuples to the user. This subsystem will use only the common GUI for executing its command. The suffix for SQL runner command will be *sql*.

2.4. Common GUI Component

Originally, every subsystem had its own graphical user interface (GUI) which lacked in uniformity and look and feel. The above issue is addressed by having a single common GUI for all the subsystems which results in uniformity and easier maintenance. The common integrated GUI is shown in Figure [2]. The common GUI is command based. Every subsystem is assigned a suffix and the suffix is used along with the command to indicate the targeted subsystem for the command. A sample command is shown below:

nc94:>loadnc94data climate;

The GUI parser parses the above string and breaks into two parts with ":>" as separator, the suffix and the command. The suffix indicates the target subsystem where the command has to be delegated for execution.

2.4.1. GUI Layout and Options

The common GUI consists of a pane to input commands, a pane to display the results of execution, a console, an area where statistics such as disk accesses are displayed, and a flags area where status of several flags can be indicated. In addition there are radio buttons to load, and execute same command files. Buttons to clear input and output panes, find & replace, and to reset certain counters are included. The functionalities of common GUI is as follows.

- *Load Command* - It allows the user to load a text file containing a batch of commands into the command editor pane.

- *Clear Command* - It clears the contents of the command editor pane.

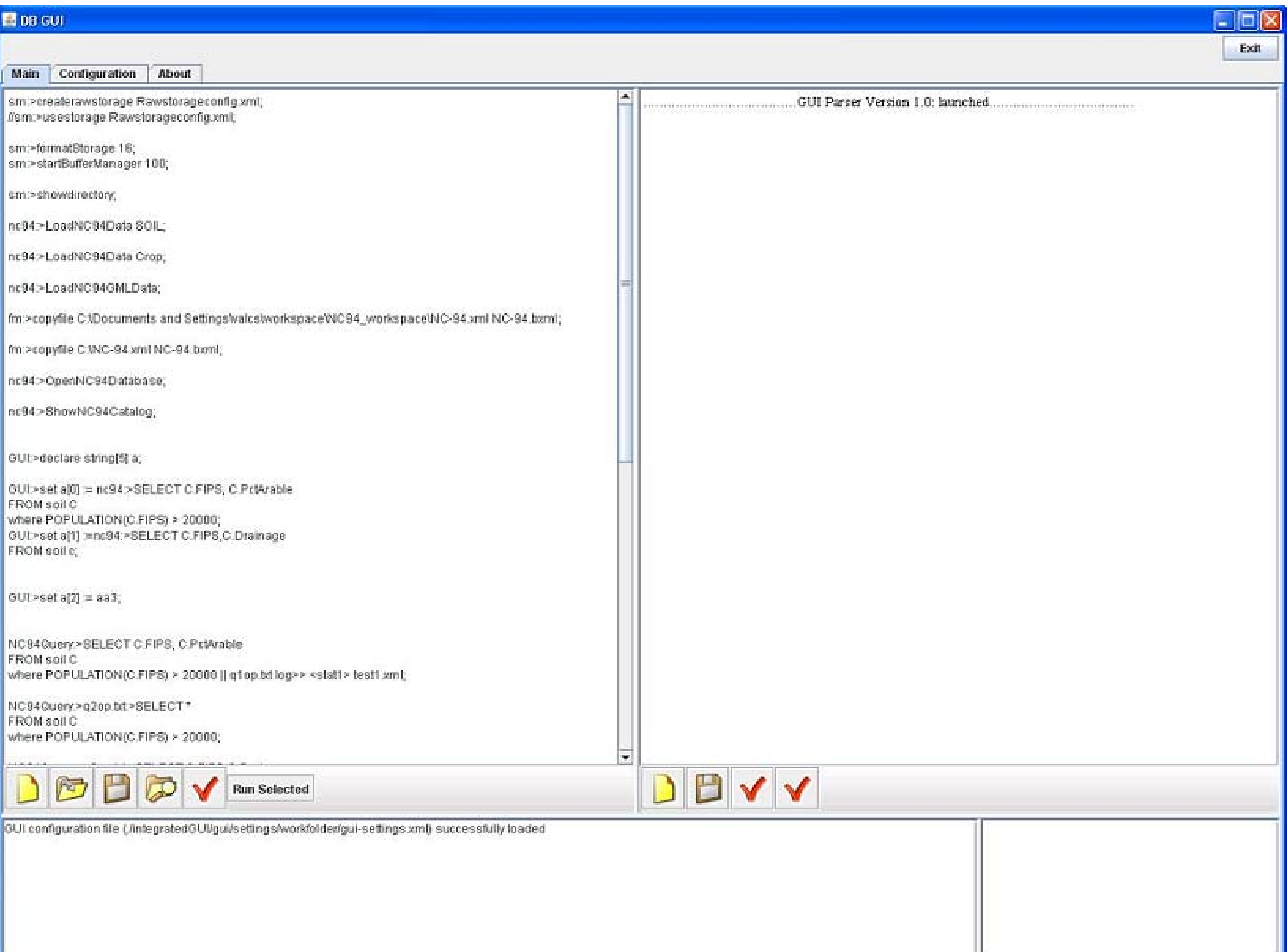

Figure 2. Common GUI

- *Save Command* -It allows the user to save the batch of commands from the command editor pane into a text file.
- *Run Command* - It executes all the command in the command editor pane. Two different commands are separated by a character *";"*.
- *Run Selected Command* - It gives the flexibility to the user to execute only the highlighted command in the command editor pane.

2.4.2. GUI Parser

GUI parser is built by Xinyuan. The parser parses the entire string given by the user and delegates the command part of the string to the target subsystem specified by a suffix part of the string. Then the specific subsystem executes the actual command given by the GUI parser and returns the result to the user.

Ex: *nc94:> loadnc94data climate;*

The above command delegates the command to NC94 subsystem for further execution.

Apart from implementing standard commands, a scripting language for executing these commands has also been implemented which will help the user to execute the batch of commands in a loop. The current version of scripting language includes basic functionalities like *variable* declaration, *array* declaration and *for loop* execution with those variables. This is convenient for executing commands, e.g, queries, on different configuration of storage platform, gather performance statistics in XML-based logs, and prepare reports from the logs. The above functionality will make it possible to repeat entire experiments starting from creation of storage, to loading data, to execution of commands, collection of benchmarks, to reporting of benchmarks with a single click.

2.4.3. GUI Commands

As of now, we have few GUI commands which will be extended in the near future to support the advanced GUI scripting language.

2.4.3.1. Declare Command

Declare Command is used by the client to declare certain variables and use those variables

in the scripting language. Currently, we support declaring and using an integer variable, string variable, integer array and string array types. The syntax of the declare commands are shown below:

In case of simple variables,

GUI:>declare data_type var_name := var_initvalue;

In case of array variables,

GUI:>declare data_type[arraySize] var_name;

Examples of declare command is shown below:

Ex: GUI:>declare int i := 2;

GUI:>declare int[10] j;

GUI:>declare string a := "abc";

GUI:>declare string[7] b;

2.4.3.2. Set Command

Set Command is used by the client to set the values of the declared variables. The syntax of the declare commands are shown below:

In case of simple variables,

GUI:>set var_name := var_value;

In case of array variables,

GUI:>set var_name[subscript] :=var_value ;

Examples of set command is shown below:

Ex: GUI:>set i := 5;

GUI:>set j[0] := 10;

GUI:>set a := "abc";

GUI:>set b[5] := "abc";

2.4.3.3. CreateLog Command

CreateLog command allows the user's to create a XML log file for reporting the query

performance. The syntax of the command is shown below:

GUI:>createlog <outerElement> log_filename.xml;

The outerElement xml tag wraps up the query performance parameters and logs it as single element into the XML file.

2.5. Other Components in the Integrated System

The other components of the integrated system includes storage manager, file manager and CxDOM API which are described in the following subsections.

2.5.1. Storage Manager

The common storage platform is managed by the *storage manager*. It gets commands from the GUI. The commands used by storage manager are described in detail in storage manager section in chapter 3 (Section 3.1).

2.5.2. File Manager

The common file types in the storage are managed by the *file manager*. Currently, the common file types includes *bxml* and *cxml*. Creation, copying and deletion of these common file types are taken care by file manager. The subsystems which wants to store their files in this format request the file manager to convert their file format to the commonly used format and file manager stores the converted file into the common storage. In the near future more file formats will be supported by file manager. It gets commands from the GUI. The commands used by file manager are described in detail in file manager section in chapter 3 (Section 3.2).

Currently file manager has two utilities, pagination utilities and sorter utility which can be used by subsystems.

2.5.2.1. Pagination Utilities

Currently we support two types of pagination algorithms for storing XML documents. The most important pagination algorithm is the binary pagination algorithm provided by CanStoreX. The binary pagination algorithm paginates the source XML documents into binary pages and stores them in the storage. The other pagination algorithm is the plain text

representation of XML documents (CXML) in the storage.

2.5.2.2. Sorter Utility

Sorter utility sorts a set of XML nodes (a forest of subtrees) and stores it in the storage. Currently the sorting is invoked internally by the XQuery engine. In the near future this will be used by different subsystems.

2.5.3. CxDOM API Layer

The other component in the integrated system is the CxDOM API. This is used by the subsystems that stores data in the storage in a binary paginated format (BXML). This is our customized version of standard DOM API for efficient access of the stored XML data from the common storage. The standard DOM API creates an in memory representation for the entire XML document and use the constructed DOM tree for node navigation which results in more memory consumption. A *.bxml* document can be consumed readily by CxDOM API. Updates are not yet supported.

2.6. SubSystem Organization

The following subsections describe the physical directory structure of the subsystems and how these subsystems are registered with the common storage platform.

2.6.1. Physical Directory Structure of SubSystem

The common integrated system have a root physical directory and root directory have many subdirectories. Each subdirectory correspond to a different subsystems. A subdirectory is also created for the common GUI and the common storage platform and the modules are placed inside the subdirectory respectively. The common library files used by different subsystems are kept in the common *lib* directory within the root directory. Every directory corresponding to a subsystem have its own *lib* subdirectory which contains the library files necessary for their system. Apart from the directories related to subsystems, the root directory also contains the centralized XML configuration document called *RawStorageConfig.xml* which contains storage configuration information. The above organization of subsystems ensures there is no overlap in directory structures of different subsystems.

2.6.2. Registering SubSystems with the Common Storage Platform

This subsection describes how the subsystems are logically organized for sharing the common GUI and common storage platform. Every subsystem has its own parser module which parses the command received from GUI parser and invokes the appropriate function for execution of their commands. Instances of parsers of different subsystems are registered with the GUI parser. The GUI parser passes the handler of the common storage platform and the handlers of the common GUI to the parsers of different subsystems. When needed, the subsystems access the storage through this common storage platform handler which ensures different subsystems use the common storage. All subsystems access the common GUI through this common GUI handler.

2.7. Background for Integrating NC94 Subsystem

Major focus of this work is towards integration of NC94 system on the common storage platform and the common GUI and making a use case that may be imitated for integrating other subsystems in the near future. NC94 subsystem uses the common storage platform for storing NC94 dataset and also for storing the catalog of the nc94 relations in a binary XML (.bxml) format in the storage. The parametric query on the nc94 dataset in the storage is executed and the results are displayed in the output pane or redirected into an output file using an option provided in the command for execution of the query. The integration of NC94 subsystem is detailed in chapter 4.

2.8. Steps for integrating remaining SubSystems

There are two types of subsystems in the list of subsystems to be integrated. Subsystems which uses only common GUI and the subsystems which uses both the common GUI and the common storage platform.

- Create a subdirectory within the integrated system's root directory and place all the modules of the targeted subsystem in the subdirectory.

- Then create a *lib* subdirectory within subsystem's directory and place all the library files that are specific to the subsystem.

- Add a parser module to the subsystem for converting all the old GUI functionalities as command based in the new GUI and get rid of the old GUI. The instance of parser is registered in the GUI parser with the suffix of the target subsystem. This will ensure that commands are redirected by the GUI parser to the appropriate subsystem on looking at the suffix of the user's commands.
- If the targeted subsystem requires only the common GUI, only the handler of the common GUI is passed by the GUI parser to the parser of the target subsystem. If the targeted subsystem requires both the common GUI and the common storage platform, the GUI parser passes the handler of both the common storage platform and the common GUI to the parser of the targeted subsystem.
- Ensure, the targeted subsystem always makes call to the common storage through the common storage platform handler and to the common GUI using common GUI handler.
- The subsystem which creates a particular data format in the common storage should ensure they implement delete command which gives the user, the flexibility for deleting data from the common storage.

The above mentioned steps ensures the subsystems are successfully integrated onto the common storage platform and the common GUI.

CHAPTER 3. CURRENT SUBSYSTEMS

This chapter provides a brief overview of the different subsystems that are targeted for integration into a single integration system. This chapter also describes the existing state of storage manager and file managers components that manage the common storage platform, CanStoreX pagination utility and XQuery evaluation engine. This includes the current features available in these systems and the implementation of these systems.

3.1. Storage Manager Commands

As we discussed in chapter 2, the storage creation in the integrated system is taken care by the storage manager component. Storage manager performs these tasks using storage commands. We already know that, storage creation happens in two steps. Creation of raw storage and paginating the storage with the specified page size. Creation of raw storage indicates acquiring storage space from the disk for system's use by creating an array of java random access files of specific size. At this point one can think of the raw storage as an unformatted disk. Now, paginating or formatting the storage indicates formatting the acquired raw storage with specific page size. This will partition the storage space into many pages of specified page size.

There exists a centralized XML configuration file called *Rawstorageconfig.xml* where user can configure details of the storage creation and then execute the storage command from the GUI for creating a storage. The configuration details includes path of the storage, size of the total storage and the size of the individual random access file. It also includes the default pagesize parameter to be used for formatting rawstorage and also number of buffers to be used for initializing buffer manager. The unit for size of the storage is in MB and the unit for pagesize is in KB. Snapshot of the *RawStorageConfig.xml* file is shown in Figure [3]. The below subsection describes syntax and functionalities of these storage commands.

3.1.1. CreateRawStorage Command

CreateRawStorage command is used by the client for creating rawstorage to be used by

```xml
<RawStorageParameters>
    <SmallestRawGranule Size = "16" Unit = "MBytes" >
    <JavaRandomAccessFileSize NofRawGranules = "1" / >
    <StoragePaths Unit = "MBytes">
        <Path Location = "X:\CanXStorage" Size = "1000" Volume = "A" />
    </StoragePaths>
    <PageSize Unit = "KBytes" Value= "16" />
    <BufferNumber Value= "100"/>
</RawStorageParameters>
```

Figure 3. Snapshot of RawStorageConfig.xml document

storage platform. This command acquires the storage space from the operating system through java random access files. The syntax of the command is shown below:

sm:>createrawstorage Rawstorageconfig.xml;

This command takes the centralized rawstorageconfig.xml file as input which contains the necessary configuration information for creating a raw storage. Once the command is executed we have unformatted raw storage.

3.1.2. FormatStorage Command

FormatStorage command is used by the client for formatting rawstorage with the specified pagesize. This command paginates the rawstorage into many pages according to the specified pagesize. Each page is associated with a *flag* bit which indicates whether the page has been allocated or not. Initially, flag bits corresponding to all pages except for the pages which hold this bitmap of pages are set to 0 indicating they have not been allocated and free to use. The sample command is shown below:

sm:>formatstorage 16;

This command takes pagesize as input parameter and the unit of pagesize is kilobytes(KB). The above example will format the storage with 16KB page size. Once this command is executed, storage is ready for usage. This command gives the flexibility to the user for formatting the storage with a new page size without deleting the raw storage. Formatting just erases the data and partition the same storage into different pages according to the new page size.

3.1.3. StartBufferManager Command

StartBufferManager command is used by client for initializing number of buffers to be used by buffermanager for bringing data from the storage to main memory and from the main memory to the storage.

The sample command is shown below:

sm:>StartBufferManager 100;

This command takes the number of buffers to be used as input parameter. The size of the buffer is equal to the pagesize. The above example will instruct the buffer manager to use 100 buffers for reading and writing data from or to the storage.

3.1.4. UseStorage Command

UseStorage command allows the user to reuse the existing storage with two options. The first option allows the user to use the existing storage along with the data present in it and the second option is to use only the raw storage and format the storage with a different page size. The syntax of the command is shown below:

sm:>usestorage Rawstorageconfig.xml;

This command takes the centralized rawstorageconfig.xml file as input parameter. It uses the existing configuration information from the rawstorageconfig.xml document and initializes the existing storage. Once the command is executed, storage is ready to use and user can view the data present in the storage using *showdirectory* command, which shows the files in the storage. If user wants to use option 2 where only raw storage to be used, then user can execute *usestorage* to instruct the storage platform to use the existing storage and *FormatStorage* command for formatting storage with different page size in succession.

3.1.5. ShowDirectory Command

The common storage platform maintains a XML based catalog that acts a directory for the documents stored in the storage. The filename, number of pages used and the root page for every stored documents are included in the catalog. *ShowDirectory* command reads XML catalog and lists the documents stored in the common storage. The syntax of the command is shown below:

sm:>showdirectory;

3.1.6. GetPageAllocatedCount Command

GetPageAllocatedCount command gives the number of pages allocated in the storage in the current session. The syntax of the command is shown below:

sm:>GetPageAllocatedCount;

3.1.7. GetPageDeAllocatedCount Command

GetPageDeAllocatedCount command gives the number of pages deallocated in the storage in the current session. The syntax of the command is shown below:

sm:>GetPageDeAllocatedCount;

3.1.8. GetPageAccessCount Command

GetPageAccessCount command gives the number of pages accessed by the system in the current session. The syntax of the command is shown below:

sm:>GetPageAccessedCount;

3.1.9. GetRelativePageAccessCount Command

GetRelativePageAccessCount command is similar to the *getpageaccesscount* command but this counter can be reset in the middle of the session. The syntax of the command is shown below:

sm:>GetRelativePageAccessCount;

3.1.10. ResetRelativePageAccessCounter Command

ResetRelativePageAccessCounter command allows the user to reset the relativepageaccess counter. The syntax of the command is shown below:

sm:>ResetRelativePageAccessCounter;

3.2. File Manager Commands

As we discussed in chapter 2, the common file formats like *xml*, binary XML (*bxml*) and textual representation of XML data (*cxml*) in the common storage are managed by file manager component. All the subsystems that wants to use common file types for storing their

data have to go via file manager which performs the processing using the file management commands described in the following subsections.

3.2.1. CreateFile Command

CreateFile command allows user to create different types of data formats like XML, BXML and CXML documents. XML documents are created in the operating system. BXML and CXML documents are created in the common storage.

3.2.1.1. CreateFile XML Command

CreateFile XML command allows user to create a XML document in the physical location given by the user. The sample command is shown below:

fm:>createfile C:\Workspace\auctions.xml 2;

The above command takes the full physical path of the destination XML document in the operating system and the size of the XML document to be created as input parameter. The unit of file size is in MegaBytes (MB). When user gives the command for creating a specific sized XML document, the system invokes XMark application [18], a third party benchmark technology for generating specific sized XML document in the path given by the user. The above sample command creates a 2 MB auctions.xml document in the given path.

3.2.1.2. CreateFile BXML Command

CreateFile BXML command allows user to create a binary paginated format of a XML document in the storage. The sample command is shown below:

fm:>createfile auctions.bxml 5;

The above command takes the destination BXML document in the storage and the size of the BXML document to be created as input parameters. The unit of file size is in MegaBytes (MB). When user gives the command for creating a specific sized BXML document (binary version of XML document), the system calls XMark for generating specific sized XML document and as characters are generated they are streamed using Java Native Interface from XMark application to the CanStoreX pagination utility which performs binary pagination and stores the pages inside the storage. The final outcome is the paginated binary XML (BXML)

document in our common storage. The above sample command creates a 5 MB auctions.bxml document in the storage.

3.2.1.3. CreateFile CXML Command

CreateFile CXML command allows user to create a plain text version of a XML document in the storage. The sample command is shown below:

fm:>createfile auctions.cxml 1;

The above command takes the destination CXML document in the storage and the size of the CXML document to be created as input parameters. The unit of file size is in MegaBytes (MB). When user gives the command for creating specific size CXML (stored as plain text version of XML document), the system calls XMark for generating specific sized XML document and as characters are generated they are streamed using Java Native Interface from XMark application to the textual version pagination utility and the streamed characters are written to the page as sequence of bytes in the common storage. The final outcome is the textual version of XML document (CXML) in our common storage. The above sample command creates a 1 MB auctions.cxml document in the storage.

3.2.2. CopyFile Command

The *copyfile* commands allows copying of files either within the storage or from the storage to outside the storage or from outside the storage to the storage. The below subsection describes syntax and the functionality of different copyfile commands.

3.2.2.1. CopyFile XML to XML Command

CopyFile XML to XML command allows user to create a copy of a operating system's XML document into another XML document. The sample command is shown below:

fm:>copyfile C:\Workspace\auctions.xml C:\Workspace\auctions_copy.xml;

The above command takes 2 input parameters, the physical path of the source XML document in the operating system and the full path of the destination XML document in the operating system. This uses the standard FileCopy method from Java File class for copying XML document into another XML document. The above sample command copies

auctions.xml into auctions_copy.xml.

3.2.2.2. CopyFile XML to BXML Command

Copy File XML to BXML command allows user to copy a operating system's XML document to a BXML document, binary version of a XML document into the storage. The sample command is shown below:

fm:>copyfile C:\Workspace\auctions.xml auctions.bxml;

The above command takes 2 input parameters, the physical path of the source XML document and the name of the destination BXML document in the storage. On executing this command, internally it invokes binary pagination algorithm of CanStoreX utility and passes source XML document as input. The pagination algorithm reads the source XML document and starts the pagination process. The end result is the binary paginated XML document, a BXML document in the underlying storage. The catalog of the common storage platform is updated with the new document information. The above sample command copies auctions.xml document into auctions.bxml document (BXML format) and stores it in the underlying storage.

3.2.2.3. CopyFile XML to CXML Command

CopyFile XML to CXML command allows user to copy a operating system's XML document to a CXML document, plain text version of a XML document into the storage. The sample command is shown below:

fm:>copyfile C:\Workspace\auctions.xml auctions.cxml;

The above command takes 2 input parameters, the physical path of the source XML document and name of the destination CXML document in the underlying storage. On executing this command, internally it invokes the textual representation algorithm of pagination utility and passes source XML document as input. The algorithm reads every character in the source XML document and writes into the storage as plain text. The end result is the plain text format of XML document, a CXML document in the common storage. The catalog of the common storage is updated with the new document information. The above sample command copies auctions.xml document into auctions.cxml document (plain text

version) and stores it in the underlying storage.

3.2.2.4. CopyFile BXML to XML Command

CopyFile BXML to XML command allows user to copy a BXML document, a binary paginated XML document from the storage into a operating system's XML document. The sample command is shown below:

fm:>copyfile auctions.bxml C:\Workspace\auctions.xml;

The above command takes 2 input parameters, the source BXML document in the storage and the physical path of the destination XML document. On executing this command, internally it invokes binary depagination algorithm of CanStoreX utility and passes source BXML document as input parameter. The depagination algorithm gets the root page of the source BXML document from the catalog of the common storage and reads the document from the storage and performs depagination process. The end result is the creation of the XML document in the specified physical location. The above sample command copies auctions.bxml document from the storage into auctions.xml document in the specified location.

3.2.2.5. CopyFile BXML to BXML Command

CopyFile BXML to BXML command allows user to create a copy of a specific BXML document in the storage. The sample command is shown below:

fm:>copyfile auctions.bxml auctions_copy.bxml;

The above command takes 2 input parameters, the source BXML document in the storage and the name of the destination BXML document in the storage. On executing this command, internally it invokes the *makeBXMLtoBXML* function of pagination utility which reads every page from the source BXML document and copies into a new page in destination document. If the scanned page is a page with no child pointers in the source document, they are directly copied into a new page for destination document without any modifications in the page and written into the disk. If scanned page is a parent page with child pointers, we copy the parent page into a new page in the destination document and update the child pointers in the destination parent page to point to the new child pageid's of destination document. The

process continues till all pages are copied. The above algorithm gets the root page of the source BXML document from the catalog of common storage and reads the document from the storage and performs conversion process. The end result is the creation of the BXML document in the common storage. The catalog of the common storage is updated with the new document information. The above sample command copies auctions.bxml document from the storage into auctions_copy.bxml document into the storage.

3.2.2.6. CopyFile BXML to CXML Command

CopyFile BXML to CXML command allows user to copy a BXML document, a binary paginated XML document from the underlying storage into a CXML document, a plain text format of XML document into the storage. The sample command is shown below:

fm:>copyfile auctions.bxml auctions.cxml;

The above command takes 2 input parameters, the source BXML document in the storage and the name of the destination CXML document in the storage. On executing this command, internally it invokes the *makeBXMLtoCXML* function of pagination utility which reads the source BXML document and uses *CharacterIteratortoStorage,* an iterator which allows writing sequence of characters into the storage. The above algorithm gets the root page of the source BXML document from the catalog of the common storage and reads the document from the storage and performs conversion process. The end result is the creation of the CXML document in the storage. The catalog of the common storage is updated with the new document information. The above sample command copies auctions.bxml document from the storage into auctions.cxml document into the common storage.

3.2.2.7. CopyFile CXML to XML Command

CopyFile CXML to XML command allows user to copy a CXML document, a plain text version of XML document from the underlying storage into a operating system's XML document. The sample command is shown below:

fm:>copyfile auctions.cxml C:\Workspace\auctions.xml;

The above command takes 2 input parameters, the source CXML document in the storage and the physical path of the destination XML file. On executing this command, internally it

invokes plain text depagination algorithm of pagination utility and passes source CXML document as input parameter. The depagination algorithm gets the root page of the source CXML document from the catalog of the common storage and reads the document from the storage and performs depagination process. The end result is the creation of the XML document in the specified physical location. The above sample command copies auctions.cxml document from the storage into auctions.xml document in the specified location.

3.2.2.8. CopyFile CXML to BXML Command

CopyFile CXML to BXML command allows user to copy a CXML document, a plain text format of XML document from the underlying common storage into a BXML, binary paginated XML document into the storage. The sample command is shown below:

fm:>copyfile auctions.cxml auctions.bxml;

The above command takes 2 input parameters, the source CXML document in the storage and the name of the destination BXML document in the storage. On executing this command, internally it invokes the *makeCXMLtoBXML* function of pagination utility which reads the source CXML document using *CharacterIteratorfromStorage,* an iterator which allows reading sequence of characters from the storage. The sequence of characters are given as input to the binary pagination algorithm which performs the pagination process. The above algorithm gets the root page of the source CXML document from the catalog of the common storage platform and reads the document from the storage and performs conversion process. The end result is the creation of the BXML document in the storage. The catalog of the common storage is updated with the new document information. The above sample command copies auctions.cxml document from the storage into auctions.bxml document into the common storage.

3.2.2.9. CopyFile CXML to CXML Command

CopyFile CXML to CXML command allows user to create a copy of a specific CXML document in the storage. The sample command is shown below:

fm:>copyfile auctions.cxml auctions_copy.cxml;

The above command takes 2 input parameters, the source CXML document in the storage and the name of the destination CXML document in the storage. On executing this command, internally it invokes the *makeCXMLtoCXML* function of pagination utility which reads every page from the source CXML document and copies into a new page in the destination document. When the algorithm scans the current page of the source document, it retrieves the nextpageid from the header of the source document's current page and saves it, then it copies the contents of current page of the source document into a newly allocated page of the destination document. Then the nextpageid of the destination document's current page is updated with the next available pageid to be allocated in the destination document. The process continues till all pages are copied. The above algorithm gets the root page of the source CXML document from the catalog of the common storage platform and reads the document from the storage and performs conversion process. The end result is the creation of the CXML document in the storage. The catalog of the common storage is updated with the new document information. The above sample command copies auctions.cxml document into auctions_copy.cxml document into the common storage.

3.2.3. DeleteFile Command

The *deletefile* command can be used by the users for deleting a BXML or CXML document from the storage. It also allows user to delete XML documents from the operating system. Each type of deletefile command is described in the below subsections.

3.2.3.1. DeleteFile XML Command

DeleteFile XML command allows user to delete a XML document from the operating system. The sample command is shown below:

fm:>deletefile C:\Workspace\auctions.xml;

The above command takes the full physical path of the XML document to be deleted as input parameter and uses the standard FileDelete method from Java File class for deleting XML document from the operating system. The above sample command deletes auctions.xml document from the operating system.

3.2.3.2. DeleteFile BXML Command

DeleteFile BXML command allows user to delete a specific BXML document from the storage. The sample command is shown below:

fm:>deletefile auctions.bxml;

The above command takes the name of the BXML document to be deleted as input parameter and gets the root page of the target BXML document from the catalog of common storage and initiates the deletion process. On executing this command, *deleteBXML* function of file manager is invoked which deallocates each page in the BXML document and adds the page to the list of available pages. While deleting pages, we ensure child pages are deleted before deleting their parent page. The end result is the deletion of the BXML document from the common storage. The entry of the deleted BXML document is removed from the catalog of the common storage. The above sample command deletes auctions.bxml from the storage.

3.2.3.3. DeleteFile CXML Command

DeleteFile CXML command allows user to delete a specific CXML document from the storage. The sample command is shown below:

fm:>deletefile auctions.cxml;

The above command takes the name of the CXML document to be deleted as input parameter and gets the root page of the target CXML document from the catalog of the common storage and initiates the deletion process. On executing this command, *deleteCXML* function of file manager is invoked which deallocates each page in the CXML document and adds the page to the list of available pages. In the deletion process, we retrieve nextpageid from the header of the current page, then we delete the current page and the process continues till all pages are deleted from the storage. The entry of the deleted CXML document is removed from the catalog of the common storage. The above sample command deletes auctions.cxml from the common storage.

3.3. CanStoreX Pagination Utility

XML documents of size greater than few Megabytes are difficult to be queried using standard DOM parser because of the in memory representation of DOM tree of the entire

document. The solution to the above problem is usage of native storage technology which exposes a logical model of storing in the common storage platform and retrieving the XML documents. CanStoreX is a binary pagination utility which successfully stores large XML documents into the common storage platform by breaking into XML pages and storing those XML pages in binary format in the storage. CanStoreX utility supports binary data format within the storage called BXML document, a binary paginated XML document.

3.3.1. Background

Larger XML documents are broken into several pages and each page should be a legal XML document on its own. Different pages of an XML document are linked together using special nodes called f-node and c-node. Special nodes helps in maintaining the hierarchical structure of the original document and ensures entire document can be retrieved without loss of contents or logical structure from the storage. The f-node is used to group multiple siblings having the same parent and the c-node contains a pointer to a child page where a subtree rooted at an f-node resides. It performs the pagination process in the depth first order and the end result of the pagination process is the *rootpageId* which points to the page where the root node of the original BXML document is present in the storage.

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

Figure 4. Textual representation of Sample XML document

CanStoreX seems to offer the flexibility to parse documents of size in the range of terabytes. Currently, CanStoreX had been tested for documents up to 1 TB in size and it had successfully parsed and paginated it. The pages can be easily navigated because they were organized in the tree hierarchical structure.

Textual representation and logical representation of sample XML documents are shown in Figure [4] and Figure [5] The corresponding binary page storage structure for the sample xml document is shown in Figure [6].

3.3.2. Improved Attribute Representation in Depagination Algorithm

As we discussed, CanStoreX utility is a native storage technology, which uses logical representation while the data is stored in the storage. In the binary pagination algorithm used by CanStoreX utility, all the elements in the xml document are represented as nodes (binary format) in the storage. Attributes in the XML documents are also represented as nodes in the storage and label of the node is "_A". It is a special type of physical node that represents the presence of attributes for a specific element in the XML document. All the attributes of the specific element in XML document will be represented as child node of "_A" node in the storage. Then this special node "_A" in turn is represented as a child node of that specific element node in the common storage. This mechanism is used while performing binary pagination of the XML document into a BXML document.

Snapshot of the XML document is shown in Figure [7]. In the original depagination algorithm of BXML document to the XML document, data is displayed in the stored format

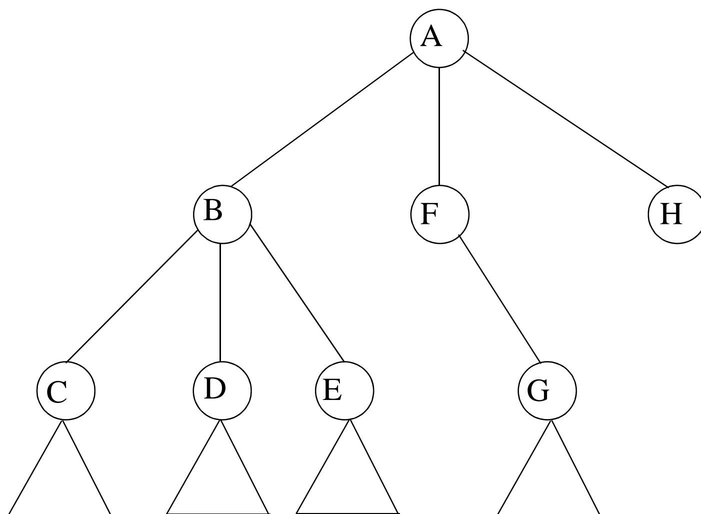

Figure 5. Logical representation of Sample XML document

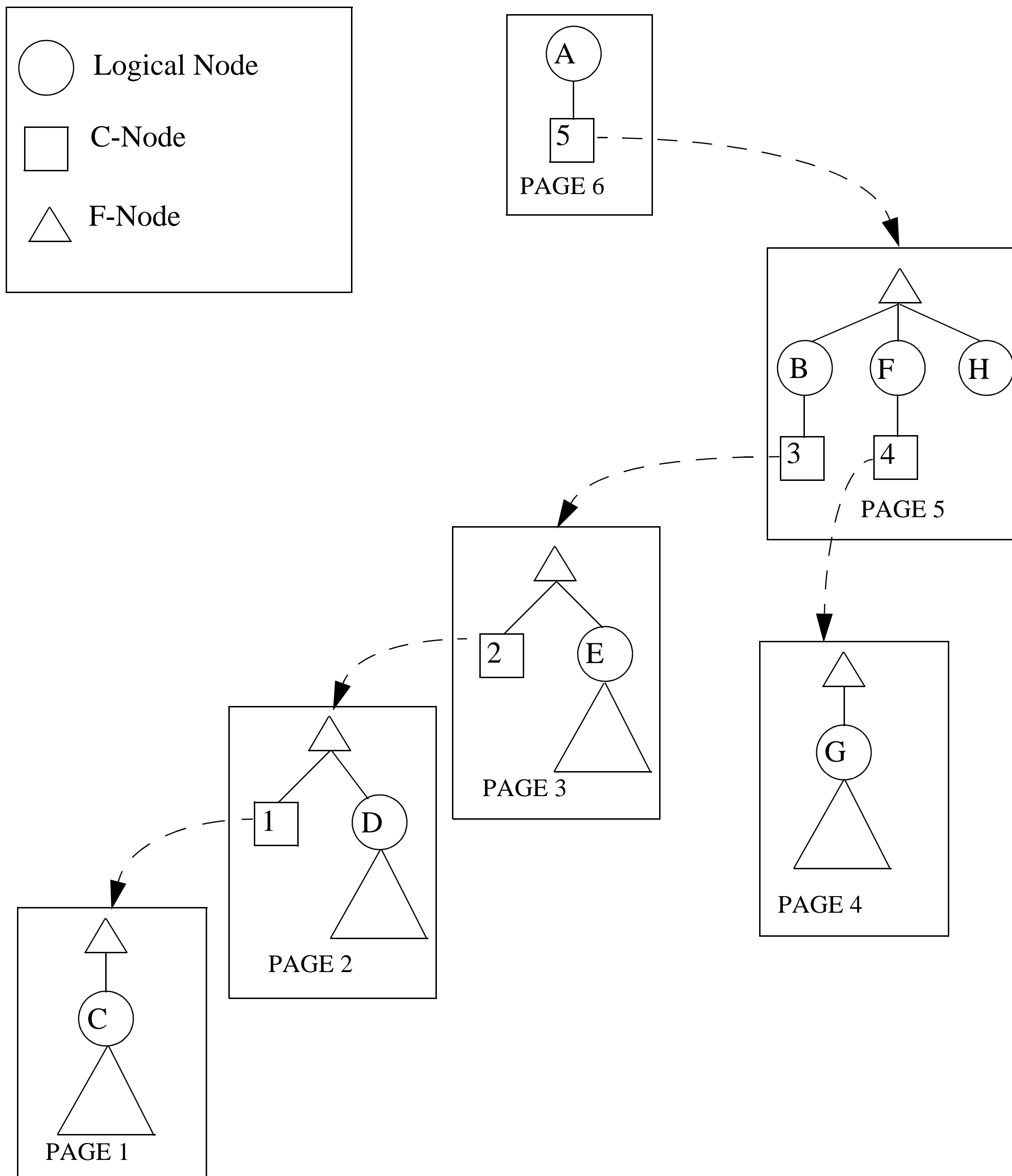

Figure 6. Binary Storage structure for the XML Document

which includes "_A" node representation in the depaginated XML document. Hence the depaginated XML document will not be in the same format as the original XML document as shown in Figure [8]. In the improved depagination algorithm, the above issue has been fixed and now the depaginated XML document is in the exactly same format as the source XML document as shown in Figure [9].

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<site>
  <regions>
        <africa>
           <item id="item0" name="gold">
                 <location>United States</location>
                 <quantity>1</quantity>
...
</site>
```

Figure 7. Snapshot of Original XML document before pagination

```
<site>
  <regions>
        <africa>
           <item>
                 <_A>
                       <id>item0</id>
                       <name>gold</name>
                 </_A>
                 <location>United States</location>
                 <quantity>1</quantity>
...
</site>
```

Figure 8. Snapshot of Original depaginated XML document with “_A” tag

```
<site>
  <regions>
        <africa>
           <item id="item0" name="gold">
                 <location>United States</location>
                 <quantity>1</quantity>
...
</site>
```

Figure 9. Snapshot of improved depaginated XML document

3.4. XQuery SubSystem

XQuery is a query language recommended by XML Query working group of World Wide Web (W3C) consortium for querying XML documents. XQuery is a functional language that consists of operators and expressions. Quilt, a preliminary version of XQuery is a query language developed by Jonathan Robie, Don Chamberlin and Daniela Florescu to process XML documents. Currently Quilt language has been replaced by w3c standard XQuery language, but the prior implementation of Quilt engine helped in developing the XQuery engine. XQuery engine involves two main modules, XQuery parser module and XQuery evaluation module. Kweelt [19] framework can be extended to create a XQuery parser for parsing the specified query onto a QuiltQuery object and evaluate the object. The document for evaluation is present in the common storage in binary format and hence for integrating our common storage with the Kweelt platform, specific primitives have been developed that instructs the Kweelt execution engine to use the common storage rather than the default main-memory storage.

3.4.1. Kweelt Architecture

Kweelt [19] is a framework for querying XML data that provides the evaluation engine for XML query language Quilt. Kweelt implements a query language for XML that satisfies all the requirements from the w3c query language requirement [19]. Kweelt is open source (GPL), completely written in Java and is easily extensible. Kweelt offers multiple XML back-ends. Hence it requires only customized implementation of Node and NodeList interface for working with different XML storage technologies. Node and NodeList are instantiated via a NodeFactory class.

The core of the Kweelt platform, specifically the query parser and the query evaluator resides in the package xacute.quilt [19]. The classes in the xacute.quilt package relies on interfaces and constants defined in xacute.common package. The entities are defined in terms of Node and NodeList classes. Node refers to a single element in the XML document whereas NodeList refers to a collection of such nodes rooted at specific node. Default implementation for interfaces xacute.common.Node and xacute.common.NodeList are located in the xacute.impl package. DOM interface based implementation is available in xacute.impl.dom

package and xacute.impl. xdom contains an implementation that is based on the Xerces DOM implementation [19]. Kweelt also allows user to add their own customized implementation to the collection.

3.4.2. XQuery Parser

XQuery uses XPath expression syntax for accessing different parts of an XML document using query patterns. XPath contains two kinds of query patterns, expressions and predicates. Sample XPath expression with different query patterns is shown below:

Display all items within region:

document("auctions.bxml")/regions//item

Find the items within region whose itemid =”item0”:

document("auctions.bxml")/regions//item[/@id="item0"]

Expressions specify the pattern the query needs to look for in the document. A ”/” pattern instructs the query engine to look only for the children of the current node and a ”//” instructs the query engine to look for all descendants of the current node. Predicates are applied to the specific node and validates whether the requested node satisfies a specified criterion or criteria. In the above sample query, ’itemid’ is a predicate that validates whether the node satisfies this criteria. The core of XQuery functionality includes FLWR expressions which is similar to Select-From-Where Clause of the SQL language. The current version of XQuery’s grammar is publicly available and includes updates to XML documents. However our current application uses only a subset of the grammar which includes the basic FLWR expressions, path expressions and a few operators and functions. The XQuery grammar has been extended successfully by Nandakumar [13] to work with Kweelt platform.

The FOR clause is used for binding a variable to a collection of nodes so that variable could be iterated over the collection for processing a element at a time [11]. The LET clause is used for variable assignment which avoids repeating same expression multiple times. The WHERE clause is used along with FOR/LET clause for evaluating predicates. The RETURN clause is used for returning elements to the outermost query or to the output stream. XQuery parser parses the XQuery and converts into QuiltQuery object and gives it to the XQuery evaluation engine.

3.4.3. XQuery Evaluation Engine

The Kweelt evaluation engine is organized in terms of QuiltExpression [19]. Every QuiltExpression must implement the method eval (EvalContext con) which performs evaluation and returns a result of type xacute.quilt.Value. It is an interface for the Kweelt base types like ValueBool, ValueString, ValueNum, ValueNode and ValueNodeList [19]. xacute.common.Node and xacute.common.NodeList objects are wrapped with ValueNode and ValueNodeList classes. EvalContext class carries the evaluation context along the various steps of the evaluation of a query. It contains information related to XPath: the current node, the position of the current node in the current nodelist and the size of the nodelist; variable bindings for LET and FOR clause; information about the NodeFactory to be used for node creation [19].

XQuery evaluation engine is customized version of Kweelt evaluation engine. Kweelt engine had a memory limitation because of the use of standard DOM parser for parsing the XML document which creates DOM tree for the entire document in the memory. This was addressed in XQuery engine by adding new primitives for accessing information directly from the common storage. Another significant modification in XQuery engine was the conversion of NodeLists to iterators which limits main memory usage [11]. An iterator behaves like a pipe where data constantly flows with respect to the to user's request. An iterator have a *open* method which opens the iterator for reading, *hasNext* method which informs the user whether there are more data available for reading from the iterator, *getNext* method which returns the next available data and a *close* method that closes the iterator and releases the used resources [11]. XML document have different types of hierarchy such as list of children, ancestors, descendants, siblings etc. which requires different type of nodelist. In order to address different type of node list, different type of iterators have been developed.

XQuery evaluation module gets the input Quilt expression from XQuery parser and starts the evaluation process. It gets the document name from the XQuery query given by the user and locates the root page of the document that is physically located in the common storage. The root page for this document is located from the XML catalog, which is a directory for the common storage that contains basic information about different files stored in the storage. XQuery engine uses the ChildNodeIterator for iterating through all the children of root node

to find the result of the Quilt expression which is formed from the XQuery given by the user. It loads only the necessary pages into main memory for processing which results in limited memory usage. Finally, it returns results in an XML file to the user. Sample query is shown below which retrieves the specific item from the document.

```
xquery:>query_a4. xml:>
        for $b in document("auctions.bxml")/regions//item[/@id="item0"]
        return <item>{$b}</item>;
```

XQuery engine uses ChildNodeIterator to iterate through all the children of *regions* node which requires more disk access, more page access and more time for retrieving the specific child that matches user's predicate from the storage. The next subsection describes how performance of this kind of XQuery can be improved by creating a index structure for BXML documents.

3.4.4. Index Structure for BXML documents in the Storage

The index structures are built for documents which have a unique key. Index structures are constructed when the original XML documents are binary paginated into the storage. Index files can be built for any element requested by the user that is a unique key in the target document.

Ex: “item” element, “Person” element from XMark generated document have a unique “id” attribute.

The format of command for building index is shown below:

fm:>copyfile E:\auctions.xml auctions.bxml item;

The above command copies a XML document into BXML format and stores it in the storage. While performing binary pagination, it creates a index file for “item” element in the document. An entry will be created in the index file for each “item“ elements from the original XML document and entry contains *id* attribute, *pageID* and *nodeOffset* of the “item” node in the target document within the storage. The structure of the index file is shown in Figure [10].

When the specific type of XQuery that looks for an element with the target *id* is evaluated, XQuery engine gets the location of specific search *id* from the index file and directly access

```
<root>
    <item0>1;2</item0>
    <item1>1;63</item1>
    <item2>1;143</item2>
    <item3>1;233</item3>
    <item4>2;2</item4>
.
</root>
```

Figure 10. Snapshot of the Index Structure

the location in the disk where node is present and returns results to the user. Experimental results showed for a 1 GB BXML document, the execution time without indexing is *385* seconds and execution time with indexing is *2.5* seconds.

XQuery engine is faster with indexing , because XQuery engine quickly locates the location (Pageid + node offset) of the specific search item from the index file and directly access it. There is also hardly any overhead involved in generating an index file because the index file is built while the original XML document is loaded into the common storage in a binary format.

3.5. ElementalDB SubSystem

Elemental DB is an artifact to learn relational database implementation. It is used for instruction purpose. ElementalDB facilitates the understanding of various layers in a relational database system to the students. ElementalDB comes packaged with a very basic form of select statement. The user enters a query in the GUI and the system generates a parse tree, expression tree for the query and execute the query. The two trees are standardised in XML. The expression tree has a built-in provision to accommodate algebraic optimization and plan development. ElementalDB also provides performance parameters like disk accesses, page accesses and query execution time. Currently, ElemetalDB have its own storage and buffer manager and its own GUI. In the near future, ElementalDB system will be integrated on the common storage platform and common GUI and the suffix will be *edb*.

3.6. TempDB SubSystem

Temporal data model is a parametric data model for modeling a real world objects where attribute values are function over a parametric space that have a time dimension. Temporal

database deals with temporal data that varies over time. TempDB is used for storing temporal data and effectively querying those data. TempDB was originally developed by Noh [7] and used hybrid storage structure for storing temporal data. Currently, TempDB uses its own storage and page format and own GUI. In the near future, TempDB system will be integrated on the common storage platform and common GUI. The suffix for TempDB subsystem is *tdb*.

3.7. Quilt Engine

Kweelt [19] is a open source framework for querying XML data that provides the evaluation engine for XML query language Quilt, a preliminary version of XQuery. Kweelt platform has been originally developed to support querying of XML documents using quilt, a query language for XML. Quilt engine works only with standard XML documents stored on the operating system. The standard Kweelt execution engine uses the standard DOM parser for parsing the XML document and to evaluate the quilt query and hence it doesn't require our common storage but requires our common GUI. Quilt engine has been integrated with our new GUI and the suffix for the subsystem is *quilt*.

3.8. SQL Runner SubSystem

SQL runner is a light weight GUI application which allows user to establish the connection with any relational database with the given connection string, passes the query to the database and returns the resultant dataset to the user. It is used for academic purpose. SQL runner requires only the common GUI and will be integrated with the common GUI in the near future. The suffix for SQL Runner subsystem is *sql.*

3.9. NC94 SubSystem

NC94 prototype works with NC94 dataset which is spatiotemporal data that varies over both time and space. NC94 subsystem was originally developed by Noh [7] and later modified by kumar [17] to use the binary pagination algorithm of CanStoreX utility. The architecture, enhancements made to the NC94 prototype for the integration of NC94 system on the common storage platform and common GUI are detailed in chapter 4.

Chapter 4. INTEGRATION OF NC94 SUBSYSTEM

This chapter briefs about the architecture and implementation details of NC94 prototype, enhancements made to the system for integrating NC94 subsystem on the common storage platform and common GUI.

4.1. NC94 System

North Central Regional Association of State Agricultural Experiment Station Directors (NCRA) decided to develop a strategy for data collection that helps in study, efficient crop management and to reduce the risk factors associated with agriculture practices, which depends on a various factors [16]. Analysis of such data collection helps predicting patterns and guide farmers to make good decisions regarding their produce. Association's primary focus was to collect and organize data on the states in the north-central region of the United States. Climate, crop and soil data were collected for these states. The result of these data collection and organization efforts was the NC-94 dataset. The NC-94 data covers a 30-year span from 1971 to 2000 and climate, crop, and soil data have their different time granularities. For climate data, measurements were on daily basis, for crop data, values were recorded yearly and soil data is relatively time invariant. The spatial granularity of all these datasets is a county with some minor but intricate variations.

A prototype for NC-94 dataset for validation of parametric model for spatiotemporal data was originally implemented by Seo-Young Noh [7] and later modified by Kumar [17]. The NC94 system uses the hybrid storage structure for storing nc-94 dataset in the storage. The hybrid storage structure is shown in Figure [11]. The hybrid storage structure includes conventional sequences of binary pages for relational data for each county and also XML directory structure that serves as an index to the relational data.

4.2. NC94 XML Directory Structure

NC-94 system have a system wide catalog (NC-94.xml) that gives the schema and characteristics for all the three relations present in the NC-94 datasets and the GML spatial data file. The characteristics of a given relation are described by different user defined

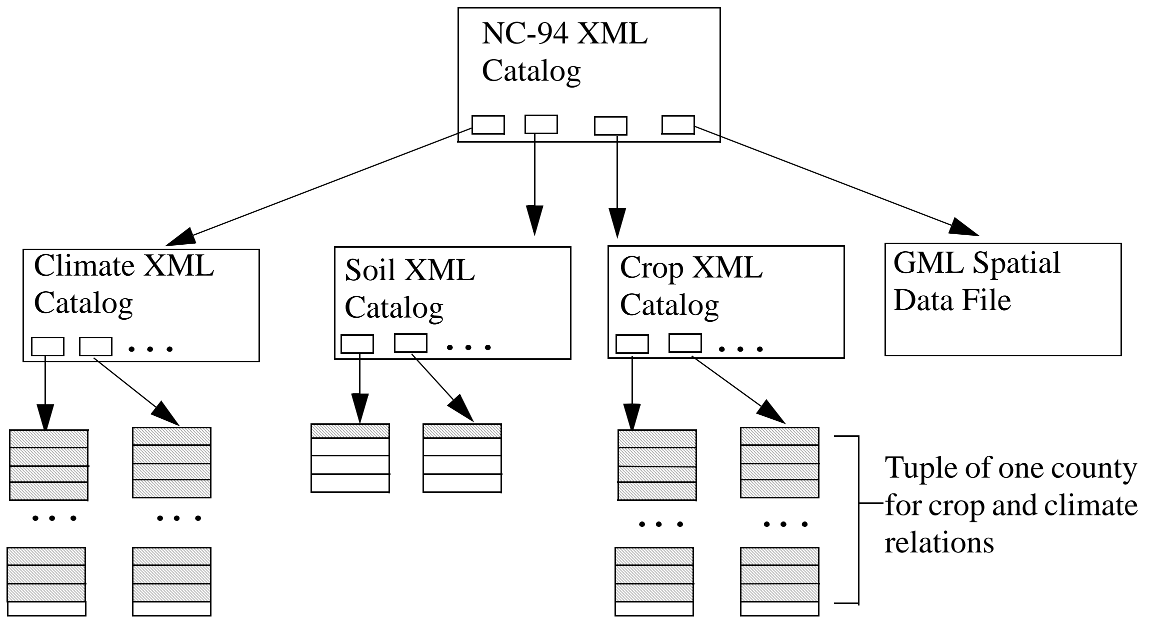

Figure 11. NC-94 hybrid storage structure

elements and *attributes* which in turn have *values*.

4.2.1. NC-94 XML Catalog

NC-94 XML catalog have a entry for each relation. In case of spatiotemporal relations, it contains a pointer to the subsequent second level XML index catalog and in case of GML file, it points to the first page of the actual relation.

Values in the catalog defines the characteristics of the relation. The characteristics includes size and type of each field in the relation and other indexing information for that relation. This can be used by any application that deals with querying nc94 dataset. A snapshot of NC-94 XML catalog is shown in Figure [12].

4.2.2. Relation XML Catalog

Second level relation catalog exist for climate, soil and crop relations. This catalog contains tuple pointer that indicates *pageId* where the actual tuple reside in the storage. It also contains key attributes and its actual values for each tuple in the specific relation. Since the key attribute values can be read from the catalog itself, it eliminates the need to retrieve key attribute values from the storage and thus saving physical disk accesses.

This catalog is binary paginated using CanStoreX utility and stored in the storage. XML

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Database Name="NC-94" homogeniety="false">
  <RelationList number="3">
    <Relation name="climate" StorageFragmentKey="FIPS"
      ctype="hybrid" SpatialGranularity="county"
      SpatialRepresentation="FIPS" TemporalGranularity="day"
      TemporalRepresentation="integer" rtype="spatiotemporal">
      <Key> <Attribute>FIPS</Attribute> </Key>
       <AttributeList number="9">
        <Attribute length="32" name="FIPS" pos="0" type="string" />
        <Attribute length="4" name="StFIPS" pos="1" type="integer" />
        <Attribute length="4" name="CoFIPS" partition="true" pos="2"
          type="integer" />
        <Attribute length="4" name="Year" pos="3" type="integer" />
        <Attribute length="4" name="Day" pos="4" type="integer" />
        <Attribute length="4" name="Radiation" pos="5" type="float" />
        <Attribute length="4" name="MaxTemp" pos="6" type="float" />
        <Attribute length="4" name="MinTemp" pos="7" type="float" />
        <Attribute length="4" name="Precipitation" pos="8" type="float" />
      </AttributeList>
      <SpatialRelation name="county_gml" />
    </Relation>
...
  </RelationList>
</Database>
```

relation catalogs are generated at run time when the actual nc94 relation data are loaded into the common storage. During the loading process, the page numbers of each tuple are determined at loading time and updated in the catalog before the actual page is written to the disk. The Snapshot of NC_Climate_Relation XML catalog is shown in Figure [13].

4.3. NC-94 Loader

The NC-94 Loader module was modified by Kumar [17] for loading NC-94 dataset which includes climate, soil and crop relational data. The original relational data is present in Access database and hence we used Java's JDBC-ODBC bridge for reading records from the access tables. NC-94 loader use the common storage platform as a storage for NC94 dataset.

Once the storage is ready to use, the loader starts paginating every nc94 relation's data and

```xml
<Relation name="climate" homogeneity="true" TupleSize="64 bytes">
  <Tuple pageID="833">
    <KeyAttributes>
      <Attribute name="FIPS" value="19001" />
    </KeyAttributes>
    <ParametricElement domain="temporal">
      <Interval start="0" end="10958" />
    </ParametricElement>
    <Info totalPages="43" totalItems="10958" />
  </Tuple>
  <Tuple pageID="876">
    <KeyAttributes>
      <Attribute name="FIPS" value="19003" />
    </KeyAttributes>
    <ParametricElement domain="temporal">
      <Interval start="0" end="10958" />
    </ParametricElement>
    <Info totalPages="43" totalItems="10958" />
  </Tuple>
...
</Relation>
```

Figure 13. Snapshot of the Climate Relation XML catalog

stores in the storage. XML relation catalogs for each relation are generated on the fly, as the relational data are paginated. NC-94 loader uses the binary pagination algorithm of CanStoreX utility for paginating XML relation catalogs and stores into the storage. The system wide NC-94 XML catalog is also updated with the information of the relation catalogs.

4.4. NC94 Commands

All the functionalities in the old ParaSQL GUI has been converted to NC94 commands. The below sets of commands facilitates the client in loading the nc94 dataset to the storage, parsing the client query and creating a parsetree, building a expression tree, executing a query, redirecting the resultant tuples to the output text file and benchmarking the performance of the query in the xml based log file.

4.4.1. LoadNC94Data Command

LoadNC94Data command is used by the client for invoking the loader module for loading

the NC94 data set into the common storage. The suffix of the command is *nc94* which denotes the targeted subsystem. The command takes the nc94 relation name as parameter which instructs the loader module to load only specific relation into the common storage. The syntax of the command is shown below:

nc94:>LoadNC94Data Relation_name;

The above command takes nc94 relation name as input parameter and loads the specific relation into the common storage. *Relation_name* parameter takes 'climate' or 'soil' or 'crop' as values. On execution of this command, the specific nc94 relation data are paginated and loaded into the common storage. XML relation catalogs for the paginated relational data are also generated on the fly and loaded into the common storage in binary XML (BXML) format using binary pagination algorithm of CanStoreX utility.

4.4.2. LoadNC94GMLData Command

LoadNC94GMLData command is used by the client for loading the NC94 GML spatial data, a XML based Geography Markup Language (GML) into the common storage. The suffix of the command is *nc94*. The syntax of the command is shown below:

nc94:>LoadNC94GMLData;

The above command loads GML spatial data which is an XML document into the common storage. The binary pagination algorithm of CanStoreX utility is used to paginate the GML data and load into the storage as BXML document.

4.4.3. OpenNC94Database Command

OpenNC94Database command opens the NC94 XML catalog and parses using DOM parser for accessing schema of the nc94 relations. It is an initialization step. The suffix of the command is *nc94*. The syntax of the command is shown below:

nc94:>OpenNC94Database NC-94.xml;

4.4.4. Variable Declaration and Query Assignment

Declaring a variable and assigning a query to the variable is a GUI command which is explained in detail in GUI commands section in chapter 2 (Section 2.4.3). Variables can be a simple or an array variable. Declaration of variables and assignment of queries allows the

clients to use the variable name with the commands instead of queries which makes the command more versatile. The syntax of variable declaration and query assignment is shown below:

GUI:>declare string[5] a;

GUI:>set a[0] :=nc94:>SELECT C.FIPS,C.Drainage FROM soil c;

4.4.5. ParseQuery Command

ParseQuery command parses the given query and creates a parse tree with select, from and where clauses corresponding to the query. The suffix of the command is *nc94.* The syntax of the command is shown below:

nc94:>ParseQuery variable_name;

The above command takes variable name which holds the target query as input parameter. The variable name can be a simple variable or an array variable. Execution of this command parses the query and creates a parsetree. Parse tree is represented as a XML document.

4.4.6. DisplayParseTree Command

DisplayParseTree command is used by the client to display the parse tree that has been created for the query. The suffix of the command is *nc94.* The syntax of the command is shown below:

nc94:>DisplayParseTree variable_name view_type;

The above commands takes variable name which holds the target query and type of view as parameters. It provide two types of tree view, XML view and the graphical view. This command has to be executed only when *parsequery* command has been executed for the target query. The variable name can be a simple variable or an array variable. The view_type parameter can take ‘xmlview’ or ‘graphicalview’ as values. ‘XML view’ opens the xml representation of the parsed tree in the web browser. ‘Graphical view’ converts the xml representation of parsed tree to the graphical view and displays in the new pop window.

4.4.7. BuildExpressionTree Command

BuildExpressionTree command is used by the client to build the expression tree for the

parsed query. It constructs expression tree for the parsed query with projection, restriction and where clauses. The suffix of the command is *nc94.* The syntax of the command is shown below:

nc94:>BuildExpressionTree variable_name;

The above command takes variable name which holds the target query as input parameter. Execution of this command creates a expression tree for the parsed query. This command has to be executed only when *parsequery* command is executed since, Expression tree is constructed based on the parse tree. Expression tree is also represented as a XML document. The variable name can be a simple variable or an array variable.

4.4.8. DisplayExpressionTree Command

DisplayExpressionTree command is used by the client to display the expression tree that has been created for the parsed query. The suffix of the command is *nc94.* The syntax of the command is shown below:

nc94:>DisplayExpressionTree variable_name view_type;

The above commands takes variable name which holds the target query and type of view as parameters. It provide two types of tree view, XML view and the graphical view. This command has to be executed only when *BuildExpressionTree* command has been executed for the target query. The variable name can be a simple variable or an array variable. The view_type parameter can take ‘xmlview’ or ‘graphicalview’ as values. ‘XML view’ opens the xml representation of the parsed tree in the web browser. ‘Graphical view’ converts the xml representation of parsed tree to the graphical view and displays in the new pop window.

4.4.9. ExecuteQuery Command

ExecuteQuery command is used by the client to execute the parsed query on the nc94 dataset. It evaluates the query on the nc94 relational data stored in the storage and redirects the entire resultant tuples to a output text file. The suffix of the command is *nc94.* The syntax of the command is shown below:

nc94:>ExecuteQuery variable_name outputfilename.txt log>> <Outerelement> log.xml;

The above commands takes variable name which holds the target query, the outputfilename

where the output tuples are redirected, the log XML filename for logging the query execution statistics and the outerelement xml tag that wraps up the query execution statistics in the XML log file as input parameters. This command has to be executed only when *parsequery* and *buildexpressiontree* commands are executed for the target query. The variable name can be a simple variable or an array variable.

4.4.10. ShowIterators Command

ShowIterators command lists the current iterators that are available for getting the resultant tuples one by one. Iterators gives the flexibility to the user to view the resultant tuples one by one using *GetNextTuples* and *GetRemainingTuples* command. The use of variables along with the iterators allows the user to retrieve the resultant tuples of different queries simultaneously without any ambiguity. The suffix of the command is *nc94.* The syntax of the command is shown below:

nc94:>ShowIterators;

4.4.11. OpenIterator Command

OpenIterator command is used by the client to open the iterator for the target query associated with the given variable name. The syntax of the command is shown below:

nc94:>OpenIterator variable_name;

The above commands takes the variable name which holds the target query as input parameter. The variable name can be a simple variable or an array variable. Without opening the iterator, client can't iterate through resultant tuples one by one using *GetNextTuple* command.

4.4.12. GetNextTuple Command

GetNextTuple command is used by the client to retrieve the next tuple of the resultant tuple set associated with the iterator of the target query indicated by its variable name. The syntax of the command is shown below:

nc94:>GetNextTuple variable_name;

The above commands takes the variable name which holds the target query as input parameter. The variable name can be a simple variable or an array variable. Once the tuple is

retrieved, the iterator points to the next tuple in the resultant tuples associated with this query.

4.4.13. GetRemainingTuples Command

GetRemainingTuples command is used by the client to retrieve all the remaining tuples starting from the iterators' current pointer till the last tuple in the resultant tuple set associated with the iterator of the target query indicated by its variable name. The syntax of the command is shown below:

nc94:>GetRemainingTuples variable_name;

The above commands takes the variable name which holds the target query as input parameter. The variable name can be a simple variable or an array variable.

4.4.14. HasNextTuple Command

HasNextTuple command indicates whether any more tuples are available in the resultant tuple set associated with the iterator of the target query indicated by its variable name. The syntax of the command is shown below:

nc94:>HasNextTuple variable_name;

The above commands takes the variable name which holds the target query as input parameter. The variable name can be a simple variable or an array variable.

4.4.15. CloseIterator Command

CloseIterator command is used by the client to close the iterator for the target query associated with the given variable name. The syntax of the command is shown below:

nc94:>CloseIterator variable_name;

The above commands takes the variable name which holds the target query as input parameter. The variable name can be a simple variable or an array variable. Once the iterator is closed, client can't iterate through resultant tuples one by one using *GetNextTuple* command.

4.4.16. Single command Option for NC94 Query Processing

NC94 subsystem provides the flexibility to the user by allowing two different options for processing nc94 queries. First option allows user to do step by step processing of nc94 queries

which includes parsing query, viewing parsetree, building and viewing the expression tree and finally execution of the query. The above mentioned sequence of steps are performed by using *ParseQuery*, *DisplayParseTree*, *BuildExpressionTree*, *DisplayExpressionTree* and *ExecuteQuery* commands discussed in the previous subsections. Second option allows user to process the entire nc94 query using single command. The suffix of this command is *nc94query*. The sample command is shown below:

```
NC94Query:>
        select C.fips, C.PctArable
        from soil C where Population(C.fips) > 20000
        || queryop.txt log>> <stat1> test1.xml;
```

It takes the nc94 query, the outputfilename where the output tuples should be redirected and the log XML filename for logging the query execution statistics as input parameters. The string "||" acts a separator that separates the actual nc94 query from the rest of the command. The rest of the command have output file name and the log xml filename.

4.4.17. CloseNC94Database Command

CloseNC94Database command closes the handler to the nc94 relation in the storage and releases all the resources. The suffix of the command is *nc94*. The syntax of the command is shown below:

nc94:>CloseNC94Database;

4.4.18. DeleteFile nc94 Command

DeleteFile nc94 command allows user to delete a specific nc94 relation data from the common storage. The syntax of the command is shown below:

nc94:>deletefile relation_name.nc94;

The above command takes the name of the specific relation to be deleted as input parameter and gets the root page of the target nc94 relation from the XML catalog of the common storage and initiates the deletion process. *Relation_name* parameter takes 'climate' or 'crop' or 'soil' as values. Execution of *deletefile* command deallocates each page in the target nc94 relation from the storage and adds the page to the list of available pages. In the deletion

process, we retrieve the nextpageid from the header of the current page and delete the current page and the process continues till all pages are deleted from the common storage. The entry of the deleted nc94 relation is removed from the catalog of the common storage.

4.4.19. DisplayNC94Catalog Command

DisplayNC94Catalog command opens the NC94 XML catalog in the web browser. The syntax of the command is shown below:

nc94:>DisplayNC94Catalog;

4.5. Prior Work

Originally Noh had designed the hybrid storage architecture [7] for storing the NC-94 dataset which contains only climate relational data. The hybrid storage structure includes conventional sequences of binary pages for relational data for each county and also XML directory structure that serves as an index to the relational data. Noh used textual pagination utility for storing XML data. Noh also created a Graphical User Interface (GUI) for executing ParaSQL queries on the NC-94 database [7].

Later Kumar [17] modified NC94 subsystem to work with climate, soil and crop relations of NC94 dataset. He also created the loader module for loading NC-94 dataset which now includes climate, soil and crop relational data into the common storage. He used CanStoreX utility for storing nc94 relation XML catalogs in the binary format (BXML) in the storage but the relational homogenous data was still maintained in the previous format in the storage structure.

4.6. Integration of NC94 SubSystem

NC94 subsystem has been successfully integrated with the common storage platform and the common GUI. The sequence of steps followed for its integration are described below:

- A subdirectory called *nc94* has been created within the integrated system’s root directory and all the modules of the nc94 subsystem has been placed in the nc94 subdirectory.
- A *lib* subdirectory is created within *nc94 directory* that contains all the library files that are required for nc94 subsystem.

- All the GUI functionalities in the old ParaSQL GUI are converted into NC94 commands in the integrated system and the old GUI is removed. A parser module called *nc94parser* is added to the subsystem which converts the old paraSQL GUI functionalities into nc94 commands and manages them. The *nc94parser* is registered with the GUI parser. This will inform the GUI parser to redirect the user's commands with *nc94* suffix to the *nc94parser* for processing and execution.
- NC94 subsystem requires both the common GUI and the common storage platform and hence, the GUI parser passes the handler of both the common storage and the common GUI to *nc94parser* which allows nc94 modules to access and use the common storage and the common GUI.
- NC94 subsystem stores the nc94 relational data in its own format in the common storage and the extension of data is *.nc94*. The *deletefile* command which is described in the previous section is implemented. This gives flexibility to the user for deleting *nc94* data from the common storage. The system also uses binary paginated XML (BXML) format for storing specific nc94 relation catalog in the common storage. It uses binary pagination algorithm of CanStoreX utility via the file manager for paginating and storing these catalogs. Deletion of BXML document is taken care by the File manager which currently manages BXML and CXML documents in the common storage.

CHAPTER 5. CONCLUSION AND FUTURE WORK

This chapter closes the thesis with the conclusion and the scope for future work.

5.1. Conclusion

The common workbench platform have been developed to bring many of our database prototypes to reside on the top of a common terabyte storage platform with the common GUI. The new architecture ensures uniformity is maintained across different subsystems that use the common storage and results in easier maintenance. NC94 subsystem have been smoothly integrated into the framework on the top of common storage platform without losing any of its functionalities. Further, as a part of integration process, enhancements were made which includes command based functionalities, simultaneous parsing of multiple queries, execution of multiple queries and benchmarking of performance parameters in an XML based log file. Placeholders have been created in the framework for smooth integration of other subsystems like TempDB and ElementalDB on the common storage platform.

5.2. Future Work

As mentioned before, ElementalDB and TempDB which currently have its own storage needs to be smoothly integrated on the common storage platform and GUI of these subsystem have to be removed and the subsystem have to be modified to work with the new common GUI. Further developments in NC-94 subsystems that are paused for the integration to the common storage platform will be resumed. Currently we use old Quilt language for evaluating smaller XML documents and the W3C recommended standard XQuery language for evaluating larger XML documents. A new application have to be created to support XQuery language on the Kweelt platform for the XML documents. This will ensure we use the standard W3C recommended XQuery language for both smaller and larger XML documents.

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] 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.

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

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

[5] 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. , and Yu, C. (2002). TIMBER: A native XML database. VLDB Journal, 11(4): 274-291, 2002.

[6] Staken, K. (2001). Introduction to Native XML Databases. http://www.xml.com/pub/a/2001/10/31/nativexmldb.html

[7] Noh, S.-Y. (2006). An XML-based implementation of the parametric data model for adhoc query of temporal and spatiotemporal data. Ph.D. thesis. Department of Computer Science. Iowa State University, 2006.

[8] Noh, S.-Y. , Gadia, S. K. , Ma ,S. An XML-based methodology for parametric temporal database model implementation Volume 81 ,Issue 6 (June 2008).

[9] Ma, S. (2004). Implementation of a canonical native storage for XML. Master’s Thesis. Department of Computer Science. Iowa State University, 2004.

[10] Patanroi, D.(2005). Binary page implementation of a canonical native storage for XML. Master’s Thesis. Department of Computer Science. Iowa State University, 2005.

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

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

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

[14] 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.

[15] 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.

[16] North Central Regional Association of State Agricultural Experiment Station Directors. Expected Outcomes. NC094: Impact of Climate and Soils on Crop Selection and Management. http://www.lgu.umd.edu/lgu_v2/pages/attachs/474_NC94ExpectedOutcomes.html. September 2004.

[17] Kumar,N. (2008) Implementation of the NC-94 hybrid storage prototype on a binary version of CanStoreX. Master's Thesis. Department of Computer Science. Iowa State University,2008.

[18] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001. http://monetdb.cwi.nl/xml/

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

APPENDIX A.

The following is a batch of storage commands which can be executed from the GUI. It uses 'sm' as prefix.

```
/* Creation or Use of Storage - Storage Commands */
sm:>createrawstorage Rawstorageconfig.xml;
sm:>formatStorage 16;
sm:>startBufferManager 100;
//sm:>usestorage Rawstorageconfig.xml;
sm:>showdirectory;
```

Sample file manager commands which can be executed from the GUI. It uses 'fm' as prefix.

```
/* Create bxml or cxml files in storage */
fm:>createfile auctions.bxml 100;
fm:>createfile sample.cxml 10;

/* Copying from xml to bxml or cxml formats - Pagination */
fm:>copyfile d:\workspace\auction.xml auction.bxml ;
fm:>copyfile d:\workspace\sample.xml sample.bxml ;
/* Copying from bxml or cxml formats to xml - DePagination */
fm:>copyfile auctions.bxml d:\workspace\auctions.xml;
fm:>copyfile sample.cxml d:\workspace\sample.xml;
fm:>copyfile sample.cxml d:\workspace\sample.xml;

/* Delete bxml or cxml files from storage */
fm:>deletefile auction.bxml;
```

APPENDIX B.

The following is a batch of commands which can be executed using [Run All] button from the GUI. The batch contains one or more instances of every type of command available in NC-94 Prototype.

```
/* Loading of NC-94 Data - NC9-94 Loader Module */
nc94:>LoadNC94Data Climate;
nc94:>LoadNC94Data Crop;
nc94:>LoadNC94Data Soil;
nc94:>LoadNC94GMLData;

/* NC-94 Commands for Execution of NC-94 Query */
nc94:>OpenNC94Database NC-94.xml;
nc94:>DisplayNC94Catalog;

// Execution of sample NC94 Queries without variables in single step - Option 1
NC94Query:>SELECT C.FIPS, C.PctArable
FROM soil C
where POPULATION(C.FIPS) > 20000
|| MyQueryOutput.txt log>> <stat1> MyLog.xml;
NC94Query:>SELECT C.FIPS,C.Drainage
FROM soil c
|| MyQueryOutput.txt log>> <stat1> MyLog.xml;

// Execution of NC94 Query with variables and step by step - Option 2
GUI:>declare int i;
GUI:>declare string[5] a;
```

```
GUI:>set a[0] :=
        nc94:>SELECT C.FIPS, C.PctArable
        FROM soil C
        where POPULATION(C.FIPS) > 20000;
GUI:>set a[1] :=
        nc94:>SELECT C.FIPS,C.Drainage
        FROM soil c;

GUI:>createlog <MyLogRoot> MyLog.xml;

nc94:>ParseQuery a[0];
nc94:>ParseQuery a[1];
nc94:>BuildExpressionTree a[0];
nc94:>BuildExpressionTree a[1];

nc94:>DisplayParseTree a[0] xmlview;
nc94:>DisplayParseTree a[0] graphicalview;
nc94:>DisplayParseTree a[1] xmlview;
nc94:>DisplayParseTree a[1] graphicalview;
nc94:>DisplayExpressionTree a[0] xmlview;
nc94:>DisplayExpressionTree a[0] graphicalview;
nc94:>DisplayExpressionTree a[1] xmlview;
nc94:>DisplayExpressionTree a[1] graphicalview;

// Redirects Resultant Tuples to the output file in NC_94 Workspace folder
nc94:>ExecuteQuery a[0] MyQueryOutput.txt log>> <Stat1> MyLog.xml;
```

```
nc94:>ExecuteQuery a[1] MyQueryOutput.txt log>> <Stat1> MyLog.xml;
/* NC94 Commands for handling Iterators- Displays Resultant tuples in the GUI */
nc94:>ShowIterators;
nc94:>OpenIterator a[0];
nc94:>OpenIterator a[1];
nc94:>GetNextTuple a[0];
nc94:>GetNextTuple a[1];
nc94:>GetRemainingTuples a[0];
nc94:>GetRemainingTuples a[1];
nc94:>HasNextTuple a[0];
nc94:>HasNextTuple a[1];
nc94:>CloseIterator a[0];
nc94:>CloseIterator a[1];
nc94:>CloseNC94Database;

/* Deletion of Nc-94 data from the storage */
nc94:>DeleteFile soil.nc94;

/* Quilt Query on the Output Log XML document */
kweelt:>let $e := document ("P:\MyLog.xml")//stat1/QueryPageAccess
return concat("Graph of Queries vs. Page Access goes here : " , avg($e)) ;
```

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, I would like to thank my major professor, Dr. Shashi K. Gadia, without whose guidance and patience this work would not have been possible. His guidence, insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. In addition, he was always accessible and willing to help his students with their research. I would also like to thank Dr. Johnny S. Wong and Dr. Arun K. Somani for their willingness to serve on my Program of Study committee.

I am also grateful to all the people whom I met at Iowa State University who have made my stay pleasant and memorable. My thanks to roommates Fred, Bharath and Kartic for their camaraderie and for encouraging me during my difficult times. My lab-mate Xinyuan deserves a special mention for assisting me with my project and the fruitful discussions we have had. I would also like to thank my other lab-mates Niranjan Kumar for giving me knowledge transfer on NC-94 system and Jia for guiding me in my thesis. Finally, I would like to thank my father Narayanan, my mother Jothi Narayanan and my brothers, who have encouraged me to pursue my graduate education and always been a constant sources of inspiration and motivation at every stage in my life.