2010

A user configurable implementation of B-trees

Soumya B. Shetty
Iowa State University

A user configurable implementation of B-trees

by

Soumya B. Shetty

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
Simanta Mitra
Wallapak Tavanapong

Iowa State University

Ames, Iowa

2010

TABLE OF CONTENTS

LIST OF FIGURES iv
ABSTRACT v
1. Introduction 1
2. Related Works 5
3. Background 7
3.1. XML 7
3.2. CanStoreX 8
3.3. XQuery Engine 9
3.4. Need for Indexing in CyDIW 10
3.5. Different Type of B-trees 10
3.6. Indexing CanStoreX using B+ tree 11
3.7. Implementation Decisions 11
4. Use Case 14
4.1. XMark 14
4.2. XMark Profile and Analysis 16
5. Implementation platform and style 18
5.1. CyDIW Platform 18
5.1.1. Workbench Organization 18
5.1.2. GUI 20
5.1.2.1. GUI Layout and Options 20
5.1.3. Command orientation 21
5.2. Parser 21
5.3. BTreeConfig 21
5.3.1. Page Config 21
5.3.2. Indexes 23
5.4. Iterators 24
5.5. Variables and settings 25
6. Implementation 26
6.1. Data Structures 26
6.1.1. Integer Key Leaf Node 28
6.1.2. Integer Key Index Node 28
6.1.3. String Key Leaf Node 29

6.1.4. String Key Index Node 29
6.2. Major Modules 29
6.2.1. Bulkloading 30
6.2.1.1. Level by Level 30
6.2.1.2. Pyramid Style 33
6.2.2. Retrieval 35
6.2.2.1. Single Value Retrieval 35
6.2.2.2. Range Value Retrieval 35
6.2.3. Insertion 36
7. Commands 38
7.1. B-tree Commands 38
7.2. Additional CyDIW Commands 40
8. Performance Evaluation 42
9. Configuration, Deployment and Further Development of B-trees 45
9.1. Variables 45
9.1.1. Space utilization and page usage 46
9.1.2. Page header 46
9.1.3. Index config 46
9.1.4. Sequence and subsequences 47
9.1.5. Primary file 47
9.2. Supporting unsorted data 47
9.3. Deployment of the tree 48
9.3.1. A client of CyDIW 48
9.3.2. A client of CyDIW storage 48
9.3.3. A client with own storage 48
9.4. Deletion in B-trees 48
9.5. Concurrency and recovery 48
9.6. Support for non-unique keys and compression of keys 49
9.7. Deployment of the tree in a database query language 49
10. Conclusion 50
11. Future work 51
REFERENCES 52
APPENDIX A. BTreeConfig.xml 54
APPENDIX B. Batch of Commands 56
ACKNOWLEDGEMENTS 62

LIST OF FIGURES

Figure 1. Pagination using CanStoreX 8
Figure 2. Secondary B-tree 11
Figure 3. Schema for XMark document (5.12). 14
Figure 4. Analysis of XMark document for item and person nodes. 15
Figure 5. Logical representation of Xmark document and B-trees created in storage 17
Figure 6. The Cyclone Database Implementation Workbench. 19
Figure 7. Integer and string key node structure 27
Figure 8. Bulkloading 31
Figure 9. Performance Evaluation of Creation of B-trees 42
Figure 10. Performance evaluation of queries for fixed length keys 43
Figure 11. Performance evaluation of queries for variable length keys 43

ABSTRACT

The use of B-trees for achieving good performance for updates and retrievals in databases is well-known. Many excellent implementations of B-trees are available as well. However it is difficult to find B-trees that are easily configured and deployed into experimental systems. We undertake an implementation of B-trees from scratch that specifically addresses configurability and deployablility issue. An XML file is used to store as well as document information such as page formats of the nodes of the B-trees and details about the nature of records and keys. The behavior of the tree is encapsulated by commands for creation of B-trees, insertions of records in the tree, and make retrievals via the tree. The XML based configuration together with commands make the deployment and functionality of the tree completely clear and straightforward.

1. Introduction

We have developed CyDIW, the Database Implementation Workbench, to support our needs for research, instruction, development, and implementation of multiple *database prototypes* and also support access to existing *command-based systems*. These prototypes and existing systems are all full fledged self-contained systems on their own right. However, for the sake of clarity in the context of the workbench, they may be referred to as *subsystems*. The workbench also includes several services and utilities that can be used by various subsystems.

The workbench includes a GUI that serves as an editor to develop a batch of commands as well as a launchpad for execution of commands. The GUI has a low profile but it is very powerful with almost no learning curve. In order to use the workbench for a subsystem the latter has to be registered in a centralized XML document. The registration requires a unique identifier to be associated with a subsystem in order to identify the system uniquely. The identifier is used as a prefix in all commands that constitute a batch. Different commands are separated by semicolons.

The objective of this thesis is to implement B-tree as a utility that would become available to subsystems, especially our own prototypes, to achieve good performance. The term B-tree refers to a specific design or it may refer to a general class of designs. The general class includes many variations like B+-tree and B*-tree. B+-trees are the most popular of all and practically used. So we would be referring to B+-trees as B-trees in this thesis. There is nothing new about B-trees, they are well understood and several implementations exist. The problem however is that existing implementations of B-trees seem to be difficult to configure and deploy. We have implemented a B-tree from scratch that specifically address these issues. A two-pronged approach is used. First, the desired configuration of the tree is expressed via a high level XML document. This centralizes numerous parameters that would otherwise be scattered and buried deep in the code. The parameters include information about binary format of pages, information about keys such as their name, type, and whether the keys are fixed length or variable length. Information about how the client will provide a stream of data on which the B-tree is to be created. The name of the file in CyDIW system where the B-tree will

itself be stored and page numbers where the root and the start of the sequence set of the B-tree are stored. Information about whether the B-tree is a primary indexed collection of records or an index on existing records. Second, strictly following our style for CyDIW the entire implementation is command based. Commands are used to create the B-tree, insert records, and collection of benchmark statistics. The parameters used in the commands are worked out carefully so that there is no mystery about hidden parameters. The list of parameters is clear, complete, as well as non-redundant.

The entire behavior of the B-tree is completely encapsulated by the configuration file and commands. As required in CyDIW, experiments are encapsulated by batches of commands. It is required that an entire experiment, however large, can be executed from scratch at the click of a button in the GUI. Every subsystem is packaged with one or more batches of commands that help understand the details of the system unambiguously. It is not an exaggeration to say that the batches of commands, that are plain text files, serve as a very high level description from a user stand point. A batch of commands is included with the B-tree implementation as well.

Prototypes include ElementalDB: a relational database system for instruction in a database implementation course, TDB: a temporal database system, NC94 Prototype: a use case consisting of a well known agricultural dataset for validation of our model and query language for spatiotemporal databases, and XQuery Engine: an engine for execution of XQuery queries.

As stated above, any command based system can be easily registered on the workbench. Examples of such systems are SQL-based database systems such as Oracle and Microsoft Access. It is a little known fact that Access, a light-weight SQL-based database system – a part of Microsoft Office suite of products – is built into the windows operating system via a command-based interface. Access can be used easily from CyDIW. Also included as standard on the workbench is Kweelt a platform to execute and use Quilt, an early version of XQuery. Thus, ElementalDB, TDB, NC94 Prototype, our own XQuery Engine, MS/Access, any other SQL-based database system in possession of a user, Kweelt, and any other command based system after having been registered can be used simultaneously on CyDIW.

CyDIW includes several utilities and services that can be used by subsystems. Some of

these utilities, including B-trees, are complete and self-contained systems their own right. However they are termed utilities because they generally support other subsystems. Services include a paginated storage, buffer management, infrastructure for collection and reporting of benchmarks, and redirecting outputs to other places instead of the default output pane of the GUI. Users are able to create variables for storing parameters as well as commands. The latter provision allows commands to be executed via other commands. Control structures are available too for repetitive and conditional execution of commands.

The workbench also includes a storage technology, called a Canonical Storage for XML (CanStoreX), that paginates large – terabyte size – XML documents and stores them as a tree of binary pages that are ready to be consumed efficiently. We have implemented CsxDOM, our own Java-based DOM API for XML. In fact our XQuery engine is implemented on the top of the storage and CsxDOM. As it is our view that XML, together with DOM API is an assembly language for information, we view CanStoreX pagination and CsxDOM as utilities in CyDIW.

XMark is a benchmark that creates an interesting XML document of desired size and a suite of queries for performance benchmarking. Included in CyDIW is a command that internally invokes CanStoreX and XMark to create a paginated XMark document of a desired size as a file in our storage. In order to validate our implementation of B-trees, we have chosen indexing of XMark document in CanStoreX as our use case. This will eventually be integrated into our XQuery engine.

The rest of the thesis is organized as follows. In section 2 we discuss the related work in this field. In Section 3 we discuss XML, our storage structure and querying mechanism. Also in this section we talk about the different decisions taken by us during the implementation of both fixed and variable length key indexes. In Section 4 we discuss XMark a well-known 3rd party XML benchmark technology used to generate XML documents of given size. We have used the documents created by XMark to test our application. This section also gives a brief analysis of the different type of nodes present in an XMark document. In section 5 we discuss the implementation platform which includes CyDIW, the configuration files, iterators, parsers etc. Section 6 discusses the major modules in our implementation and the data structures used

by us. Section 7 talks about the different commands supported by our system followed by Section 8 which discusses the usability and deployability of this system. Section 9 evaluates the performance of the system. Finally in Section 9 and 10 we present the conclusion and future work.

2. Related Works

This section discusses the different B-tree implementation studied by us before implementing our own version of B-tree. Also it discusses the different indexing techniques used in other xml storage and some existing bulkloading techniques.

It was very difficult to find a good implementation of a B-tree which fit our requirements of being easily configurable and deployable in our system. We looked at two major implementation of B-trees provided by Minibase [10] and Berkeley DB [11], [12]. Minibase is an educational data management system. The only way of creating a B-tree in Minibase is by inserting one node at a time. Also the B-trees created in it were not easily configurable. On closer look at its implementation we found that the code was object oriented and created a Java object for each page it accessed in the B-tree. Also most of functions in it were based on recursion. These two factors reduce the performance of the system drastically for large documents. The implementation provided by Berkeley DB is a very high level one and the best one we could find. An important and good feature of Berkeley DB was that during an insertion it split the nodes unevenly instead of splitting it into half so as to make maximum utilization of available space.

We also looked at the way indexing is implemented in other xml storage systems. The strategies used for indexing XML documents can be classified into three types; node indexing [21], [22], path indexing [23], [24] and sequence based indexing [25], [26], [27], [28]. Node indexing decides the relationship between nodes by assigning a sequence to each node in the document (using preorder, postorder or both). So every time a query comes in; it has to decompose it to an atomic unit in the XML document and join the intermediate results. Path indexing processes path expressions in a query efficiently. Dataguides [23] and APEX [24] are examples of path indexing. They process path expressions efficiently but require joins for branching queries. ViST [26] and PRIX [27] are examples built on sequences. XICS [25] (XML Indices for Content and Structural search) is another such method. It uses a path identifier and Sibling Dewey Order (numbering system for nodes in XML document) to create a key. These keys are indexed using a B-tree.The main disadvantage of all the indexing tech-

niques mentioned above are that the indexing is bound very tightly to the XML document. A small change in the document would lead to restructuring of these indices all over again.

We also looked at the different approaches used for bulkloading indexes. The bulkloading techniques are primarily divided into three classes; sorting based, sample based and buffer based [15]. In the sorting based technique one sorts the data first and builds the tree bottom up. Sample based technique relies on a recursive partitioning of the data set (the key pointer pairs) into two as it is known from Quicksort. This is a top down approach. The disadvantage of this technique is that it results in a large I/O overhead since the data set has to be read and written quite often. The buffer based bulkloading is a technique which is conceptually identical to the repeated insertion algorithm (where keys are inserted one at a time) except that the insertions are done lazily. All the internal nodes have a buffer associated with them. So instead of continuing the traversal down to the leaf, the record is inserted into the buffer. Whenever the number of records in the buffer exceeds a pre-defined threshold, a large portion of the records of the buffer is transferred via individual calls to the insert method.

The work done in this thesis tries to come up with an indexing technique which is easily configurable, easy to use, uses main memory efficiently, supports different bulkloading techniques for efficient creation of B-trees and tries to cover a range of retrieval and insertion queries.

3. Background

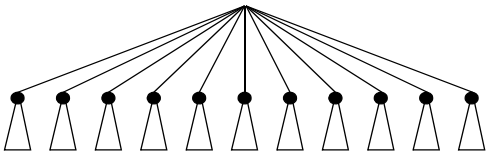
In this section we give a background of the different components that are part of our storage. Also we discuss the need for indexing and the decisions made in our implementation.

3.1. XML

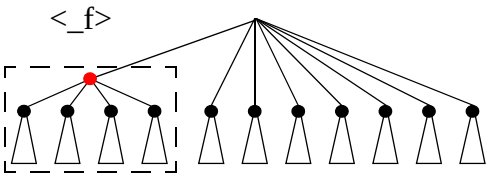
Extensible Markup Language (XML) is a simple tree-based structure for representing and standardizing information content. Its tree-based structure goes hand-in-hand with DOM API and XPath technologies making it inherently versatile, powerful, and efficient.We have a keen interest in XML itself as a database 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.

In our implementation we use XML documents in two ways. The first usage is to use them as configuration files. These XML documents are small in size and stored directly on the operating system. We use DOM API to parse these documents. It uses SAX to parse the test based physical representation of the XML document and stores the document in a tree format in the main memory by building the parent, child and sibling relationship between the nodes. DOM API works with only smaller documents because the size of the XML document once its brought to main memory is 5 - 10 times bigger than its original size. Inspite of this shortcoming we still use it for smaller documents so not to not go through specialised technologies to access them.

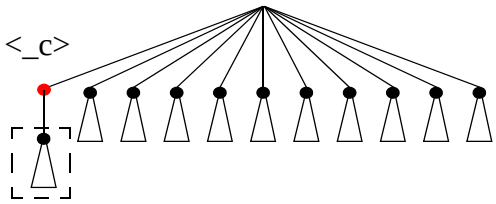
The second category of XML documents are large documents in Terabyte range. Accessing these documents using DOM API is impossible as the main memory requirements of DOM API are very high. The main challenge is storing and retrieving this large amount of data. To overcome this challenge we use our own storage and access technologies called CanStoreX (Canonical Storage for XML) [1].



(a) The source XML document



(b) Case A. The fanout is too large



(c) Case B. A child is very large

Figure 1. Pagination using CanStoreX

3.2. CanStoreX

CanStoreX breaks an XML document into pages and stores it in a paginated format in the storage. Each page itself is organized as a self contained XML document. Once the data is stored in a binary paginated form as tree, it is processed using CsxDOM, which is our version of the classical DOM API.

Figure [1] shows the basic idea behind pagination. Figure 1(a) shows an XML document where the root has a variable number of children. The child trees are XML elements on their own right and vary in complexity and physical size. Recursively the base case is when the whole document fits on a page. If this is not true then there can be one of two reasons for it. First, considered in Figure 1(b), is where the number of children is so large that even pointer to all of them will not fit in a page. In this case some children can be grouped together and a dummy parent can be created to represent all of them via a single pointer. Second, shown in

Figure 1(c), is when a child is so large that it would not fit in a page. In this case the child can be differed to a page of its own and be represented by a pointer. This argument is the basis from where CanStoreX starts. Here, the two types of nodes have been used to facilitate pagination.

CanStoreX does not place any limits or expectations on what kinds of nodes will be needed or used to accomplish pagination. However, CanStoreX requires an XML document to be broken into pages where the pages themselves are XML documents on their own right. (Except when information has no structure and requires multiple pages).

3.3. XQuery Engine

XQuery is a functional programming language recommended by XML Query working group of World Wide Web (W3C) for querying XML documents. It provides 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.

XQuilt is an older version of XQuery language. Sahuguet implemented Kweelt [4] a query engine for execution of Quilt queries. We also have our own implementation of 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 [2]. 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 [3]. 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.

3.4. Need for Indexing in CyDIW

We saw the need to incorporate indexing in our storage to improve the performance of xqueries and indirectly our system. Without indexing the retrieval time of xqueries is very high. In this thesis we have tried to integrate indexing in our storage so as to make it fast and efficient.

To understand the need for indexing we need to understand the storage better. As shown in Figure [1] the binary representation of an XML document in the storage is made of three significant nodes Logical nodes, C nodes and F nodes. The pages are linked to each other through a hierarchical structure thereby maintaining the relationship between the nodes so that the entire document could be reproduced or navigated without any loss in content or in structure. The F-node is used to group a sequence of siblings having the same parent. A subtree rooted at the F-node is stored on a single page. The C-node contains a pointer to a child page where a subtree rooted at a f-node resides. It is clear that to retrieve any node or subnode in the XML document we have to start from the root of the tree, search the whole tree till we reach the node we are looking for. This is where indexing helps. Index structures are necessary to efficiently perform queries on large, unconstrained document.

3.5. Different Type of B-trees

There are two type of B trees that can be used for indexing. The first type of B-tree is a primary B-tree where the leaf nodes contain the actual data. The second type of B-tree is called secondary B-tree where the leaf nodes contain pointers to the actual data. The difference between these two type of B-trees is that a primary B-tree does not have a fixed length record in its leaf nodes while a secondary B-tree always has fixed key pointer pairs. Accessing data in a secondary B-tree would required an additional page access as the leaf nodes contain just pointers to the actual data. The advantage of secondary B-tree over primary is that it is independent of the underlying system and can be easily integrated in any system.

Different type of keys can be supported by a B-tree. We have classified the keys into two basic types; fixed length and variable length. Key is nothing but an identifier which defines a node in the XML document. A key pointer pair is required to reach any node in a document

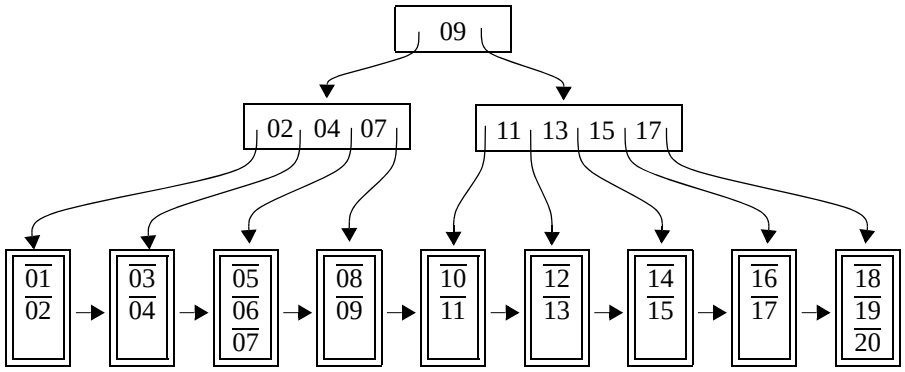


Figure 2. Secondary B-tree

and the B-tree holds a series of these pairs. The only difference between these two type of keys is the way the keys are stored in a B-tree node.

3.6. Indexing CanStoreX using B+ tree

This section talks in brief about the way we incorporated indexing in CanStoreX.

The three areas related to indexing that we concentrated on in this thesis are:

- Creation of B-tree using bulkloading
- Retrieval of data from the B-tree
- Insertion of data into the B-tree

We chose B-trees for our implementation as they are the most versatile of all data structures for indexing which provide good performance for single retrieval, sequential retrieval, insertion and deletion. We concentrated on bulkloading a B-tree as it helps in building an index structure quickly and efficiently for a storage. We implemented two different ways of bulkload B-trees called level by level and pyramid style. We will discuss these in detail in the following sections. The second most important operation supported by our application is retrieving data from the index structure. We support both single value selection and range value selection. Our application also supports insertion so as to handle any changes made to the XML document. The following section talks about the decisions made by us on our way.

3.7. Implementation Decisions

For our B-tree implementation we decided to consider both primary and secondary B-tree.

The only way to implement indexing as a primary B-tree was to make the B-tree an integral part of the storage structure. To achieve this goal we would need to integrate the creation of the B Tree in the creation of the storage structure of an xml document. If implemented this way the b tree would become an integral part of the bxml file (storage structure of an XML document in our system) and every time the bxml file would be traversed it would have to go through all the B-trees created for that document. One disadvantage of using primary index is that the best time to create it is at the time of storing the document in the storage. It can be created later in time but this would require a lot of resources and remodeling of the stored document. Similarly deleting a B-tree created for a particular document would be difficult as this too would require a lot of restructuring the storage.

To make our implementation a general purpose algorithm we decided to keep the index structure separate from the creation of the storage file i.e. we implemented it as a secondary structure. This gave us the flexibility of creating indexes as and when required in the system. Also keeping it aloof from the stored document gives us a better way of maintaining the index structure without affecting the stored file.

In our implementation we support both fixed length and variable length keys. Supporting fixed length keys is easier than variable length keys. This is because when a key pointer pair has a fixed structure and length it is easier to process it and apply binary search and splits on a node. When the keys become variable length it is difficult to assume anything and new ways have to be found to process these keys and nodes. We have tried to come up with a mechanism for variable length keys which gives comparable performance to fixed length keys. In the following section we will discuss the different ways we considered for our variable length implementation.

One way we considered for variable length key indexes was to use a hash table and write a hash function which mapped each key to a unique number and using this number in the B-tree instead of the actual key. The hash table created would be stored in an external file. The disadvantage of this technique was that every time a comparison was made between two keys we would have to read the external file. Numerous comparisons need to be made between keys in the B-tree during creation, retrieval and insertion and these comparisons prove expensive if

using a hash table. Another disadvantage of this technique is that it is very difficult to come up with a hashing function which generates a unique number for each key.

Another option considered by us was storing the key pointer pairs in the same way as fixed length keys are stored. The disadvantage of this technique is that binary search cannot be easily applied to a node in the B-tree. The key pointer pair need to be accessed sequentially which brings down the performance of the system considerably.

We finally came up with a way to store the string key within a node and represent it as an integer which acts as the key in the key pointer pairs. The integer basically defines the location where the string key is stored and its length. This implementation being close to the fixed key length implementation makes it easier to apply binary search and splits on the B-tree. Also different users might want to use different patterns in the variable length keys. In this case the only change a user needs to make to the implementation is to reimplement the compare() function so that it knows how to compare keys with a user defined pattern. We will talk in detail about the page structures for variable length keys in further sections.

4. Use Case

This section talks in brief about the document used by us to test our application. It includes a description of XMark a well-known 3rd party XML Benchmarking technology used to generate XML documents of required size. Also we talk about the distribution of the nodes in the document when stored in our storage and present an estimated size of the B-trees created for different type of keys.

4.1. XMark

XMark is a benchmark technology used to generate XML documents of given size in the path specified by the user. We use XMark to test the capabilities and performance of our system.

There are two ways of creating a storage file in CanStoreX. One way is to use the createfile

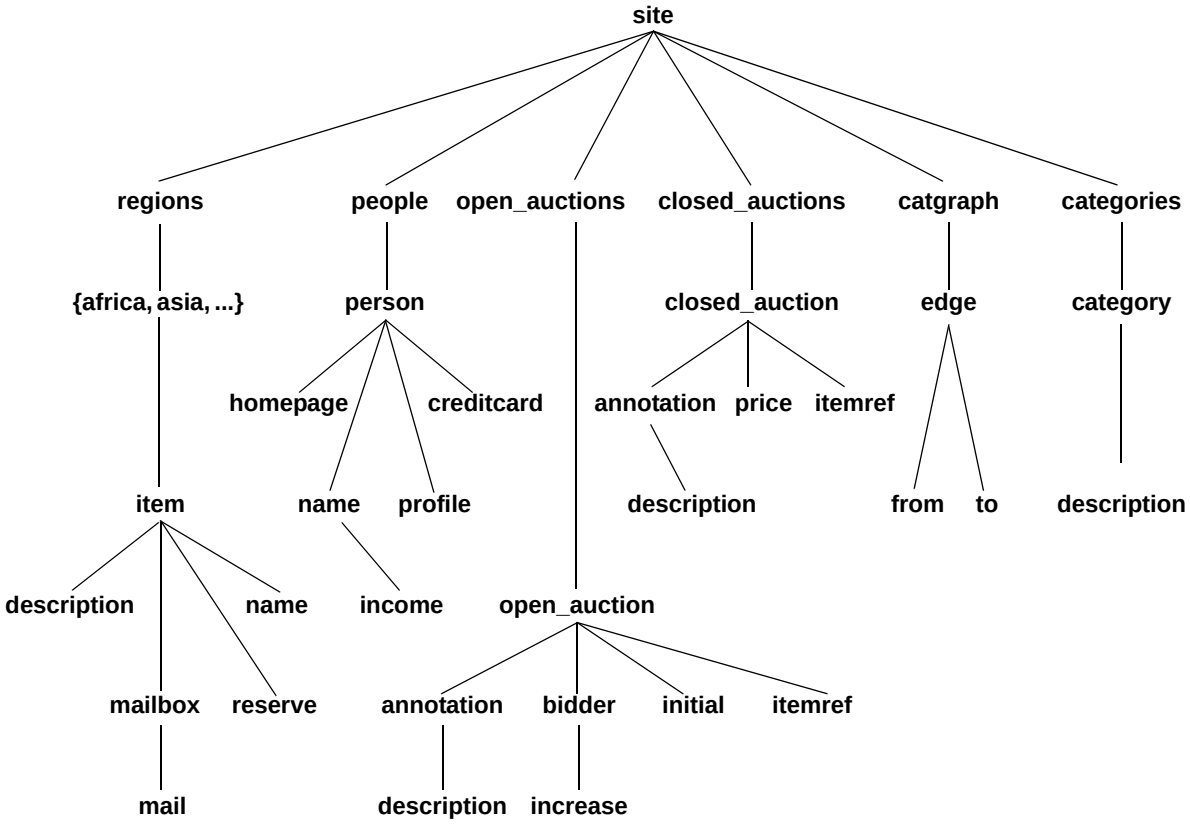


Figure 3. Schema for XMark document (5.12)

Type of nodes	Avg number of nodes per page
Item Nodes	6
Person Nodes	30

(a) Average number of nodes per 16K page in CanStoreX

		Number of pages			
File Size in MB	Number of Item nodes	Leaf Level	Upper levels	Height	Actual / Estimated
1	193	2	1	2	Actual
10	1,957	2	1	2	Actual
100	19,570	13	1	2	Actual
1000 (1 GB)	195769	130	1	2	Actual
100,000	20,000,000	13,000	9	3	Estimated

(b) Rough Analysis of B-tree for item nodes

		Number of pages			
File Size in MB	Number of Person nodes	Leaf Level	Upper levels	Height	Actual / Estimated
1	229	2	1	2	Actual
10	2,295	2	1	2	Actual
100	22,949	30	1	2	Actual
1000 (1 GB)	2,29,490	300	1	2	Actual
100,000	20,000,000	30,000	31	3	Estimated

(c) Rough Analysis of B-tree for person nodes

Figure 4. Analysis of XMark document for item and person nodes

command in CyDIW to create an XML document of a specified size by using XMark. The copyfile command can then be used to paginate and store this XML document in CanStoreX. Another way is to directly create a storage file by paginating the document on the fly as fragments of the document become available. As XMark is a C program, for direct pagination JNI (Java Native Interface) is used to pass the character stream created by XMark directly to the Java-based pagination algorithm.

Figure [3]. gives an over view of the XMark document. The document is modelled after a database deployed by an Internet auctions site. The subtrees that form bulk of the data are

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

XMark also provides a set of benchmark queries which can be used to test the performance of any XML storage system. XMark divides these queries into subcategories. Some of the categories are exact match, casting, regular path expressions etc. We concentrated on two data sets which form the bulk of the document; they are item and person nodes. We used this data to test the two key types supported by us; fixed length and variable length. The different queries that can be run on items are; creation of B-tree, get an exact item, get all the items in the document defined by an xpath, get all the items for a particular region, get the first item in a given region and inserting new items in the B-tree. The identifier of person data set is a string and the different queries that can be run for a person node are creation of B-tree, exact match, selection of all person nodes in the document and insertion of new person nodes in the B-tree.

4.2. XMark Profile and Analysis

As mentioned above the two nodes we are interested in are item and person nodes. Figure [4] gives a brief and rough analysis of the number of item and person nodes present in XMark documents of varying sizes. It also tells more about how many item and person nodes are stored on an average on a 16K page in our storage; 16K being the page size ideally used for pages in our storage.

From Figure [4] (a) it is clear that on an average 6 item and 30 person nodes are stored on a single page in our storage. This figure remains constant for XMark documents of varying sizes as the only thing that changes with the size of the document is the number of item and person nodes and not the size of each node.

Figure [4] (b) and (c) give an analysis of the number of item and person nodes in the XMark document of varying sizes and the number of nodes present at each level of the B-tree created

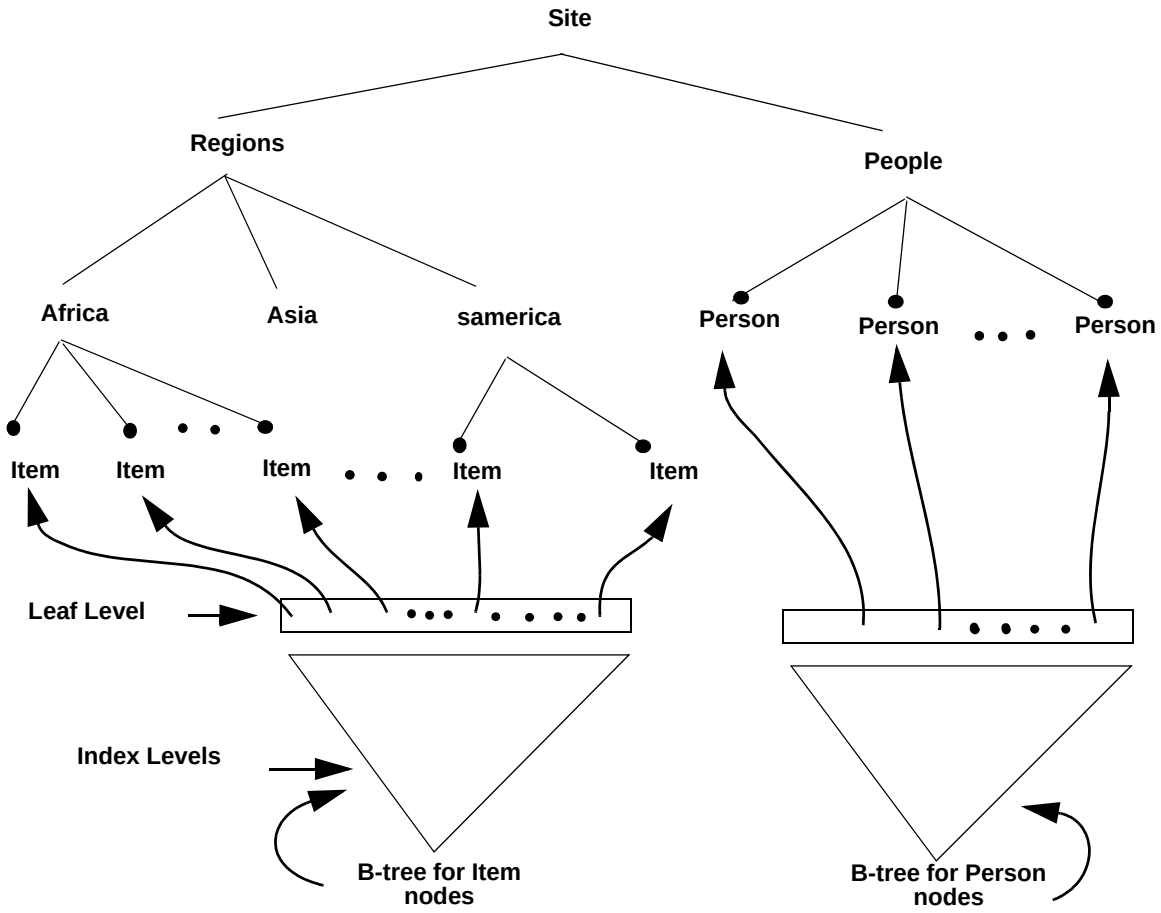


Figure 5. Logical representation of Xmark document and B-trees created in storage

for each key type. As observed in XMark documents the number of these nodes grow proportional to the size of the document. From the table it is clear that the B-trees created always grows horizontally rather than vertically for increased document size. This is an important feature of B-trees and is very helpful in improving the performance of the system. The benefit of keeping the height small is that the number of hops required to reach the leaf nodes is less.

Figure [5] gives a logical representation of the XML document created by XMark and the secondary B-trees created for indexing the item and person nodes.

5. Implementation platform and style

This section talks about the platform used for building the B-tree utility. It talks about the common workbench and GUI which forms the basis of our utility. Also this section discusses the components necessary for the creation of an index structure like the configuration file, B-tree parser and iterators etc. Let us take a closer look at all these components.

5.1. CyDIW Platform

This section talks about the common workbench used by the subsystems developed by the members of our research group.

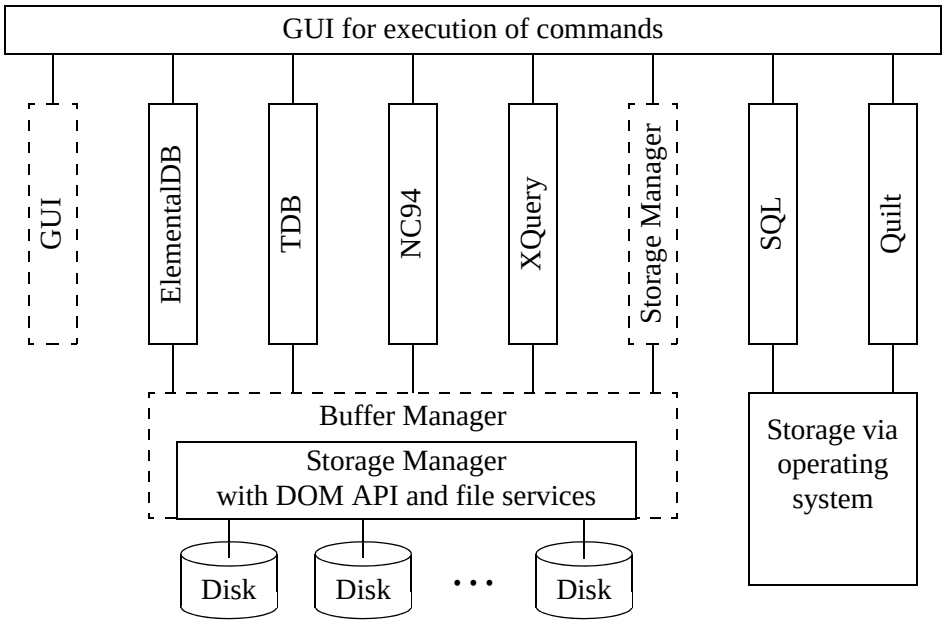
Initially for each subsystem developed by our research group we had a different GUI. Over the years several database subsystems have been developed which largely overlap in the storage modules and the Graphical User Interfaces (GUIs). The subsystems shared common needs for storage, buffer management and GUI interface to run the commands and view the results. So an integrated System was developed by Valliappan Narayanan [5] which minimized configuration and organizational overlaps among multiple systems. The Database Integrated Workbench (CyDIW) was an effort to create a common platform which brought together all these subsystem under one umbrella which resulted in uniformity and easier maintenance. Figure [6] (a) depicts the organization of the workbench. The integrated GUI provides us with a general purpose page-based storage, buffer management services, pagination and DOM API services for large XML documents.

5.1.1. Workbench Organization

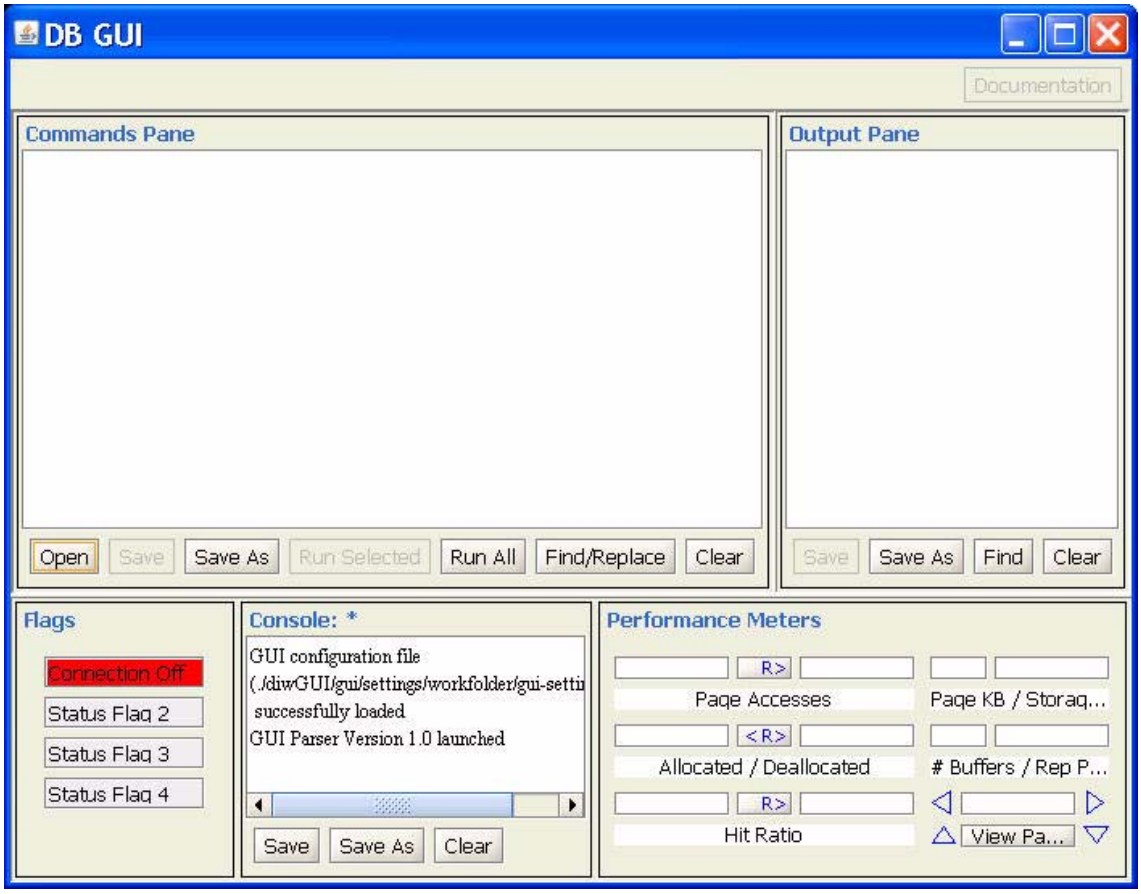
The lowest layer in the architecture is the storage system. It is responsible for the creation and management of the storage. All storage manage commands have a prefix CyDB:>. For example:

CyDB:>CreateRawStorage storageConfig.xml

This commands creates a raw storage with parameters mentioned in the storageConfig file. The storage config file contains information about where to create the storage, size of the stor-



(a) Organization of the workbench



(b) The centralized GUI

Figure 6. The Cyclone Database Implementation Workbench

age to create, what page size to use, number of buffers to use, buffer replacement policy etc.(only Least recently used policy is available)

The layer above it is the buffer manager. It ties the storage to the main memory. The buffer manager is used to read and write pages from the disk. Every time a read is called it first looks if the page exists in the Buffer pool, if no it gets it from the disk. Similarly while writing a page it is written to the Buffer pool. If the buffer pool is full a page is selected to be replaced and if the page is dirty its contents are written to the disk. It also tracks the number of times a page was accessed and the number of pages allocated and deallocated.

Above this layer lie all the other subsystems. All the subsystems use the storage manager and buffer manager to read and write to the disk but have their own set of commands. These commands can be executed with the help of the common GUI.

5.1.2. GUI

Figure [6] (b) depicts a common integrated GUI. It is command based. Every subsystem has a prefix which distinguishes it from the other subsystems. A sample command is as follows:

btree:>createbtree e:\BTreeConfig.xml AuctionsItems.btree Level_By_Level;

The GUI parser breaks the command into two parts using the separator ‘:>’ as prefix and command. The prefix depicts which subsystem it belongs to and the command is the command to be executed. The GUI parser redirects the command to the dedicated parser for that subsystem.

5.1.2 1. GUI Layout and Options

As shown in Figure [6] (b) the GUI contains a pane to enter commands, an output pane to view results of query executed, a console which displays success and failure messages, an area to view statistics like disk access made, pages allocated and deallocated and a flags area where status of several flags can be indicated. Below the commands pane there are several buttons to open and save a commands file, run all or selected commands, clear the commands pane etc. Some of the functionalities are as follows:

- Load Commands - It allows a user to open a text commands file in the commands pane.

- Save Commands - There are two commands; Save and Save As to save changes made to the commands file.
- Run Commands - There are two commands one to run all the commands in the commands pane the other to run only selected commands.
- Clear Commands - This command clears the commands in the commands pane.

5.1.3. Command orientation

We have implemented our own set of commands for creation of a B-tree, retrieval of data from the B-tree and insertion of data in the B-tree. Apart from these commands implemented for B-tree we also use a set of storage manager commands to set up the storage and buffer manager. We will talk in detail about our implementation and the different commands supported by our system in the following sections.

5.2. Parser

A new parser called BTree parser was implemented to handle B-tree commands. Every command in the GUI has a prefix which determines the subsystem it belongs to. The B-tree subsystem knows how to handle B-tree commands. The parser verifies the validity of the command by checking that it contains all the required parameters. If it is valid it redirects it to the necessary logic which knows how to execute it.

5.3. BTreeConfig

BTreeConfig is a configuration file which holds information about the b trees to be created for XML documents. The config file is quite flexible as a user can store information regarding multiple B-trees which may or may not belong to the same XML document in the same config file. If need be a single config file can hold information regarding a single B-tree. This configuration file is created manually by a user and stored on the disk. This file has to be mentioned while running commands related to the B-tree subsytem.

The config file is made up of two important parts:

- Page Config
- Indexes

5.3.1. Page Config

Page Config is the area in the config file which defines a node in the B-tree. The page config has some parts which are configurable and some are not. The parameters like page utilization and space utilization can change for one config file to another but the structure of the header cannot change as it is tied down to the implementation. Configurable page and space utilization parameters help us stress test our application. Changing these parameters can result in trees of varying heights. Also while inserting a new key in an existing B-tree; splits can occur. In our implementation we have taken special care of handling even and odd keys so as to not unbalance the two new nodes created. The management of available space helps us test all these feature easily.

The following section talks about the fields that make up the page config:

- SubPageSize - This parameter decides what fraction of the page would always be available inside a B Tree node.
- SpaceUtilization - This parameter defines what percent of the whole space available on a page (defined by SubPageSize) to use initially while creating a B-tree. We introduced this parameter so as to improve the performance of future insertions. If all the nodes in a B-tree are completely filled while creating the B-tree insertions would result in a lot of splits. As splits are costlier this parameter proves beneficial for the performance of insertions.
- PageHeader - This node defines the header of a b tree node. The different parts that make up the header are:
  - CurrentPagePtr - This field defines the location in the node where the current page id is stored.
  - NextPagePtr - This field defines the location in the node where the next page id is stored.
  - PageTypePtr - This field defines the location in the node where the page type of current page is stored. It could be either a leaf node or index node.
  - KeyTypePtr - This field defines the location in the node where the key type is stored. The two types supported now are fixed length and variable length.
  - NumOfKeysPtr - This field defines the location in the node where the number of keys present on this node is stored.
  - FreeSpacePtr - This field defines the location in the node where the free space available on this page is stored. By default it is set to the location where one can start writing data

on the page.

- FreeSpaceForContentPtr - This field defines the location in the node where the free space for content available on this page is stored. This field is required only if the key type is variable length.
- LargestKeyPtr - This field defines the location in the node where the largest key on this page is stored.
- DataStartIndex - This field defines the location in the b tree node where the header ends and data area starts.

5.3.2. Indexes

This area is where the information related to all the indexed nodes is stored. An ‘Indexes’ node can have one or more IndexConfig nodes as a single config file can have information about more than one index node.

Every IndexConfig is uniquely identified by a index file name. Along with that it holds other important attributes required during creation, retrieval and insertion in a B-tree which includes the input mode, key type and the presence of subsequences. The IndexConfig is a static structure in which some fields are configurable and some are populated by a program. The fields like input mode, key type etc are configurable but the information related to the B-tree like root page id, sequence page id are populated by a program. The information stored inside every IndexConfig node is as follows:

• PrimaryFile - This field tells more about the storage file whose node has been indexed. It contains attributes which define the document, the name of the node being indexed and the xpath of the node in the expression tree of the XML document.

• BTreeAccessPointers - This node tells more about the B-tree. It contains pointers which can be used to traverse the B-tree.

- RootPageId - This attribute holds the page id of the B-tree root. The root of the B-tree is where we start looking while search for a key or inserting a new key in the tree.
- SequencePageId - This attribute holds the page id of the first leaf node. In our implementation all the nodes at the same level are linked to each other which makes traversal at the same level easy. This attribute is used while executing commands which retrieve all the

keys stored in the B-tree.

- SubSequences - Subsequences are nothing but information about nodes which hold together groups of subnodes. In the document created by XMark all the item nodes are grouped under different regions where they are available. In our implementation we have support for retrieving these items based on their regions. This node contains a series of subsequence which store the additional information about each region. The attributes contained in each subsequence are as follows:
  - Name - The name of the region.
  - PageId - The pageid is an id of a leaf page in the B-tree which contains information about items belonging to this particular region.
  - Offset - The offset is the offset in the leaf page in the B-tree which contains information about items belonging to this region.
  - Next - This field tells more about the region following the current region. This field helps us identify where to stop in the B-tree.

5.4. Iterators

Iterators form a very important part of our implementation. Iterators provide a higher level and a more structural way of accessing the storage. Iterators was an important enhancement added by Krithivasan [3]. An iterator can be visualized as a pipe were data is inserted at one end and comes out of the other. Iterators have an open method which opens a specific iterator for reading, hasnext method to know if there is any more data available in the pipe, getnext which returns the next data available in the pipeline and close to close the iterator and release all resources. Different type of iterators were developed like childrenNodeIterator, ancestor-NodeIterator, descendantNodeIterator, siblingNodeIterator etc. These iterators act as a black-box for accessing nodes in a stored XML document.

We use iterators to get all the nodes to be indexed for an XML document. In our implementation we have used descendant node iterator to get all the descendants of a given node. While iterating through the nodes we make an entry in the B-tree if the node is a type of node to be indexed.

5.5. Variables and settings

Every time a B-tree command is run we initialise a set of variables in our application. It involves reading the page config from the B-tree config file and initializing the variables in our application so that they do not have to be read from a config file every time they are required. It also involves initializing the constants which decide what fraction of page to use for the B-tree and space utilization on a page while creating the B-tree.

6. Implementation

This section gives an overview of our implementation. It talks about the different page structures used by us for fixed length and variable length keys in our application. Also it talks about the major modules that are part of our implementation.

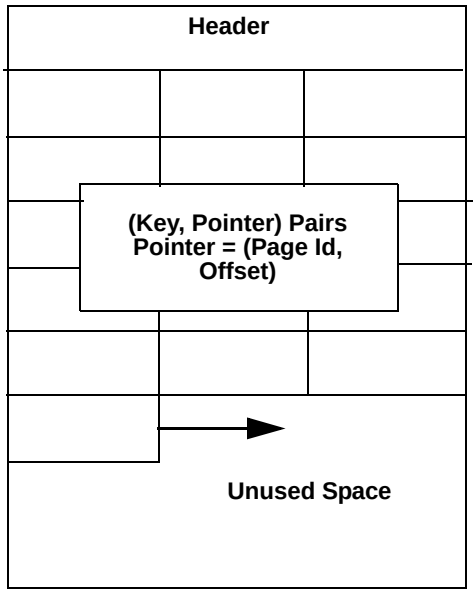
6.1. Data Structures

This section talks about the page structures used by us for leaf and index nodes. The page structure also differ for the two key types supported by our implementation; i.e. fixed length and variable length.

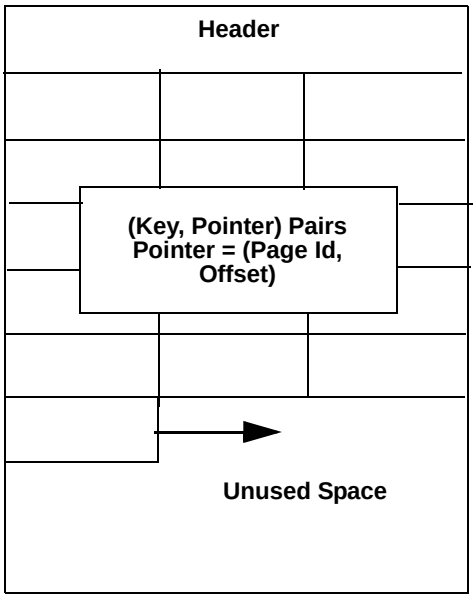
All the different page structures share a common header. The settings in the header decide if the page type is leaf or index and if the key type is fixed or variable. As mentioned above the structure of the header is decided by the BTreeConfig file. The header of a node is as important as the content inside the node as it holds important information regarding the available space, pointers, number of keys etc.

The following sections talks in detail about the different parameters that make up the header in a B-tree node:

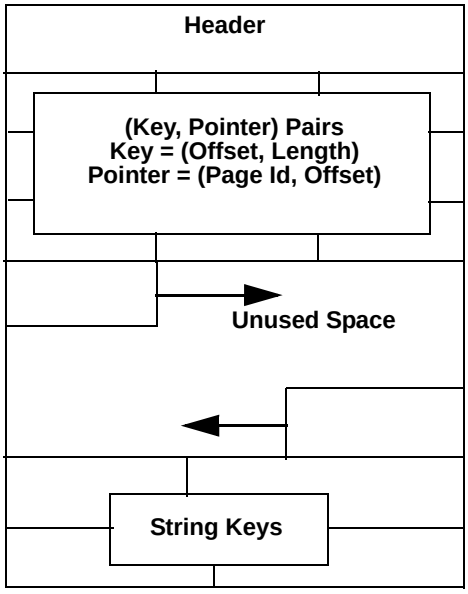
- CurrentPageId - This field contains the page id of the current page.
- NextPageId - This field contains the page id of the next page. Storing the next page id has two advantages. First being it makes range value selections easier and second is it is useful while bulkloading the B-tree.
- PageType - PageType defines the type of the current node. It could be either a leaf node or an index node.
- KeyType - This field defines the type of key stored on the current page. The two types supported now are fixed length and variable length.
- NumOfKeys - This field keeps track of the number of keys present on the current page. This field makes processing a node easier while inserting a new key or doing a binary search within a node.



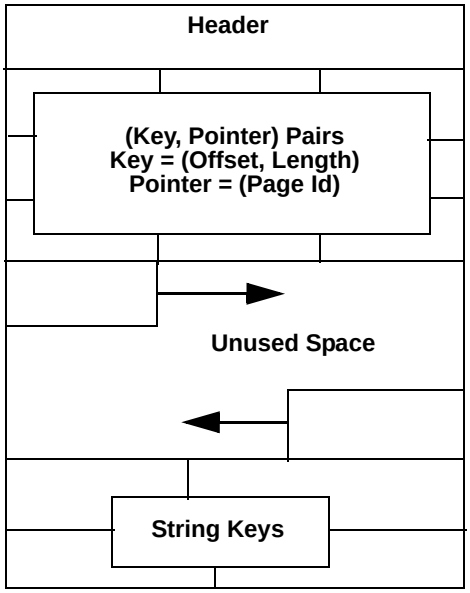
(a) Page Structure of Leaf Node for Integer Key



(b) Page Structure of Index Node for Integer Key



(c) Page Structure of Leaf Node for String Key



(d) Page Structure of Index Node for String Key

Figure 7. Integer and string key node structure

• FreeSpace - This field defines the location in the node from where one can start writing key pointer pairs in a node. The keys are written in a top down fashion in the data area.

• FreeSpaceForContent - This field is required only when the key type is variable length. This field defines the location in the node from where one can start writing the variable length keys. The variable length keys are stored bottom up in the data area.

• LargestKey - This field defines the largest key stored on the current page. This field is useful while bulkloading a B-tree as it saves the time required to calculate the last key stored in a node.

Figure [7] shows the physical representation of each type of node used in our implementation. In the following section we will talk in detail about the content area of the different page types for both int and string keys:

6.1.1. Integer Key Leaf Node

Figure [7] (a) represents the page structure of a leaf node when key type is integer. The header always occupies the top most area of the page. Following the header is the data area. The data area consists of key pointer pairs. In a leaf page the number of keys are same as the number of pointers. The key is an integer stored as 4 bytes. The pointer consists of two parts; page id and offset. The page id and offset point to a location in the storage where the indexed node actually resides. In our implementation Page Id requires 4 bytes and offset requires just 2 bytes. The 4 bytes of Page Id is enough to represent any page in the storage. Also we use 2 bytes for the page offset because we use pages of size 16K and the maximum number of bits required to represent a 16K page is 14bits or at the max 2 bytes. For integer key types the whole data area is available for storing the key pointer pairs. The free space field in the header points to the location in the data area from where one can start writing the new key pointer pairs.

6.1.2. Integer Key Index Node

Figure [7] (b) represents the page structure of an index node when key type is integer. The difference between the leaf node and index node is the data area. The data area is made up of key pointer pairs. The number of keys in an index page is one less than the number of pointers. The key is a positive integer stored as 4 bytes. The pointers in the index nodes are made up of just page id. The pointers in an index node point to the nodes in the immediate lower level of the B-tree. As it contains a page id of a page stored in the storage it required 4 bytes.

As the key type is integer the whole data area is available for the key pointer pairs. The free space field in the header points to the location in the data area from where one can start writing.

6.1.3. String Key Leaf Node

Figure [7] (c) represents the page structure of a leaf node when key type is string. The header occupies the top most area of the page followed by the data area. In the case of string keys the data area is divided into two parts. From one end we write the key pointer pairs and the other end we write the string keys. The definition of key changes slightly in this case. When the key type is string the key in the key pointer pair is just a pointer to the string key stored on the same page. This helps us treat the key pointer pairs as fixed length tuples. The key is made up of two parts; the start location of the string key and the length of the key. The first 2 bytes of the key indicate the location of the string stored in the content area. As discussed above two bytes are enough to represent any location in a 16K page. Similarly 2 bytes are enough to represent length of the string key as it never would exceed 16K.The pointers in the leaf nodes are made up of page id and offset which point to the location where the node identified by the key is stored.

6.1.4. String Key Index Node

Figure [7] (d) represents the page structure of an index node when key type is string. The string keys are stored in a similar way as the string keys in the leaf pages. The pointer is made up of 4 bytes which points to the nodes in the immediate lower level of the B-tree.

In our implementation we have made an assumption that the length of a string key is always less than half of the space available for storing the string keys. This assumption is made because in our bulkloading algorithms we always need the largest keys stored at any level to build the higher levels. In the case of integer keys it is easier to store the key as it is of fixed length but in the case of string keys the only place to store it is in the content area along with other strings. By guaranteeing that the string key size is less than half we make sure that there is always space available for storing the largest key in a node without keeping the node empty.

6.2. Major Modules

This sections talks in more detail about the major modules that are part of our implementa-

tion. Our major goal in this project was to come up with ways of bulkloading the B-tree so as to provide an efficient way of building indexes for XML documents. We also proved the usefulness of indexing by integrating B-trees with our storage. Also we support updates to an existing B-tree. This section talks in detail about the different bulkloading algorithms supported by our application, and the retrieval and insertion queries that can be run through our subsystem.

6.2.1. Bulkloading

CanStoreX is a storage structure for very large XML documents. Creating index structure for nodes in these documents wold be inefficient if the keys would be inserted one at a time. The time required and the number of page accesses made in this technique are very high. The number of comparisons and splits required also make this technique slower. We wanted to come up with an efficient way to build these index structures. An efficient way of building the B-trees is by bulkloading the tree. Bulkloading requires the minimal number of page access required while building the tree. Also while bulkloading the B-tree no comparisons are required and nodes do not have to be split which further reduces the time and page access required.

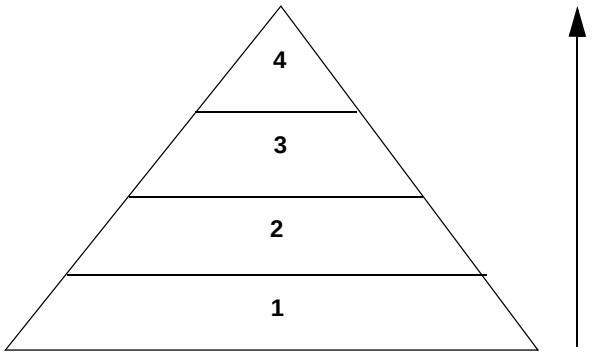
An important assumption made by us is that the keys to be indexed are coming in an increasing order of their values. If the keys to be indexed are not in order some mechanism needs to be implemented to sort the keys and then use the bulkloading algorithm or else the technique to insert one node at a time can be used.

The two ways which we came up with bulkloading a B-tree are:

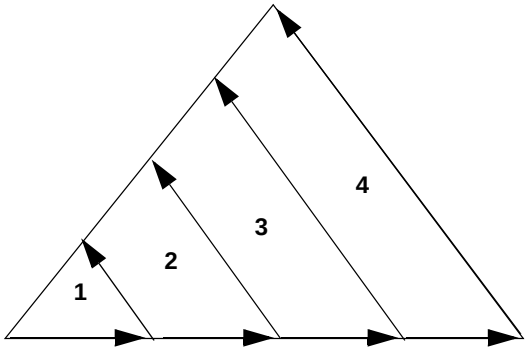
- Level By Level
- Pyramid Style

6.2.1 1. Level by Level

The basic idea behind level by level bulkloading is to build one level of the b tree at a time. We start with the last level in the tree which contains leaf nodes and build the tree upwards till we reach the root of the tree which contains just one node. Each level in the tree is built from left to right. Also at every level in the tree we take care that each node created is a proper b



(a) Level By Level Bulkloading



(b) Pyramid Style Bulkloading

Figure 8. Bulkloading

tree node. Figure [8] (a) tries to summarize this idea.

The descendant node iterator returns a series of (key, pointer) pair where the key is an identifier of a node in the XML document and pointer is a set (pageId, pagePointer) which points to a location in CanStoreX where the node is physically stored. For example if we are indexing "site/region/*//item"/ nodes the iterator returns a series of (id, pageId, pagePointer) triplets. All these key pointer pairs form the leaf of our secondary b tree. As mentioned earlier the actual data is contained only in the leaf nodes while the index nodes contain pointers to the leaf nodes.

The basic idea behind building the leaf level of the tree in this method is to allocate a page and insert key pointer pairs till the page is full. Once the current page is full; allocate the next page, create a link between the currently allocated page and previous page and continue inserting key pointer pairs on the newly created page. This process is continued till all the keys are inserted in the leaf level of the tree. As discussed above the parameters subpagesize

and space utilization in the config file decide how much space in a node to use while building the tree. Also while building the leaf level at any given point of time only one node is active at a time.

From the way we insert keys in the leaf nodes it is clear that the last node may or may not be a proper B-tree node. Key pointer pairs are inserted in a page till it is full and then we move to the next node. But to make the tree a valid B-tree we need to take care that all the nodes are proper B-tree nodes and all the nodes are at least half full. There are many ways of making the last node a valid B-tree node and ensuring that it gets at least enough nodes to make it half full without disturbing the integrity of the other nodes at the same level. We decided to use a methodology in which we would have to touch the minimum number of nodes to ensure valid B-tree nodes at this level. The minimum number of nodes to be touched apart from the current node to make it a valid B-tree node is 1. Also the minimum number of keys to be moved from the previous page to the current page is equal to half of the keys in that node. Let us see how and why.

We know all the nodes except the current node are full and proper b tree nodes. The last node at this level may have less than half of the number of keys that can fit on that page which makes it an invalid node. Also all the pages in a B-tree have the same size. If the number of keys moved from the previous page to the current page is more than half then we cannot guarantee than the current page would be overflown or no, also this would make the previous page an invalid node. Similarly if the number of keys moved from the previous page to the current page is less than half we cannot guarantee than the current page would be half full and ultimately a proper b tree node. If we move half of the keys from the previous page to the current page, the previous page would have exactly half of the keys and the current page would have more than half keys but less than what would fit on the page. Thus moving half of the keys ensures that the previous node and current node are proper B-tree nodes.

Once the leaf level is created we start building the upper levels in the tree. This process is continued till only a single node is created in the tree which is nothing but the root of the tree. The only information required to build any index level in a B-tree is a series of page id and the largest key that the page points to from the lower level in the tree. To get this information we

store the page id of the first page at the immediate level below the current level. All the pages at a given level are linked to each other so it is easy to get the page id and the largest key of all the nodes in the lower level.

There are two main differences between a leaf node and index node. Leaf nodes have p number of keys and pointers while a index nodes have p number of keys and p+1 pointers. The second difference is the largest key stored in the two type of nodes. As the number of keys and pointers are same in the leaf nodes the largest key of a leaf node is the actual last key stored in the page. In the case of index nodes there is one more pointer than the keys, so the range of keys pointed by this node is different than the last key stored in the node. The largest key field in the header helps store the actual largest key pointed by the current node. This is a necessary field as it is required to build the upper levels in the B-tree.

The same problem where the last node may or may not be a proper B-tree node can occur for the index levels. The same methodology as discussed for leaf nodes is applied to make all the nodes in the index level proper B-tree nodes. The largest key is also required while moving the keys between the last and second last nodes in the index levels of the tree.

- Advantages
  - This is the easiest way of bulkloading the b tree.
  - The time required to build the B-tree by this technique is less.
- Disadvantages
  - This method of building the tree is expensive in terms of page accesses required. This happens because in level by level implementation every time a new level is built the lower level has to be read to know the page id and largest key stored on that page so as to insert it in the current level. So once a page is full and written to the disk it is not completely done.

6.2.1 2. Pyramid Style

This is the second and a more economic way we came up to build a b tree in our implementation. The idea behind pyramid style B-tree creation is to build all the levels of the b tree at the same time. This method is more complex than the one discussed above. Figure [8] (b) tries

to give a better idea about how this method works.

In this method we do not build the B-tree level by level instead we build all the levels of the tree at the same time. The advantage of building the tree this way is that once a node is full and written to the disk we are done with it. There are a few exceptions to it as we might need to reread the second last node at each level to make the last nodes valid B-tree nodes if required. B-trees expand horizontally rather than vertically so the exception applies to a very small percentage of the whole tree created which is negligible. In the following section we discuss in detail the creation of B-tree using this method.

In this method the basic idea is to start with the minimal number of nodes and levels required in the tree and expand the tree on the go as new and new keys are inserted in it. We start with just a single level and a single leaf node in the B-tree. Once this node is full and more keys need to be inserted we need to expand this level by adding a new node. Also it becomes necessary to make an entry in the upper level of the tree so that the tree built so far is properly linked. This holds true for any level in the tree. At any level if a new node is created an entry needs to be made in the upper level which contains a pointer to the lower level and the largest key stored at that level so that the tree built so far is properly linked and complete. We do not postpone this step to the end but do it at runtime as we are building the tree. Also we do not build any level in the tree until it is required. If an entry needs to be made at a level and the level does not exist then we build it.

Even in pyramid style bulkloading we start from the left and move to the right. the difference between this method and level by level bulkloading is that we build all the levels of the tree at the same time so once a node is written to the disk it does not need to be read again.

While building the tree in pyramid style we can encounter the same problem where the last nodes at any level in the tree may or may not be proper B-tree nodes. So there are exceptions were we need to read the second last node to balance the last node and made it a valid B-tree node. As mentioned above with the page size we use i.e. 16K the height of the tree is less as the tree grows horizontally and so this is negligible.

- Advantages
  - The biggest advantage of pyramid style bulkloading is that the page accesses required to

create the tree are less. Once a node is written to the disk it is done.

- Disadvantages
  - In this technique the number of nodes active at any given point would be <= height of the tree.

6.2.2. Retrieval

Retrieval is the most frequent operation done on a storage. Indexing is a one time job, updates could be made to the storage resulting in updates in the B-tree. But the ultimate goal of indexing a storage is to improve retrievals from the storage. Most queries run against any storage are retrieval queries. The following section talks about the different type of retrievals supported by our system.

The two basic types of retrieval supported by our system are

- Single value retrieval
- Range value retrieval

6.2.2 1. Single Value Retrieval

Single Key Retrieval means retrieving a single node information from the B-tree at a time. An example of this type of retrieval is to get an item which has an xpath "site/region/*//item"/ and whose identifier is 100. If an index structure has been built for the nodes sharing the given xpath it is easy to retrieve this item. The config file has the details required to retrieve the node information from the B-tree. The only thing required to know is the root of tree. Starting from the root node we do a binary search to find the next page to go to. We jump to the next page and continue this process till we reach the leaf node which holds the page id and offset of the required node. We again apply binary search inside the leaf node to get the page id and offset of the node identified by the given key. Once we have this information it is easy to rebuild the required xml node from the storage.

6.2.2 2. Range Value Retrieval

There are two type of range value retrieval supported by our system; selecting all the nodes stored in a B-tree and selecting just a subset of the nodes stored in the B-tree.

The only information required to get all the nodes stored in the B-tree is the page id of the first leaf node. While creating the B-tree we store this information in the configuration file. Also as mentioned above while creating the B-tree we link all the leaf pages together which makes retrieval of all nodes easier.

The second type of retrieval supported by our system is range value selection. In this project range value selection has very limited capability but the idea can be expanded to support and improve the performance of more diverse queries. As shown in Figure [3] we see that the items are grouped together into various regions in which they are available. In range value selection we have proved that the performance of queries which retrieve a range of values like all the items which belong to a particular region can be improved by storing some additional information in the config file about the region nodes. This feature is limited to the regions node in the XMark document but this can be made generic and applied to simplify many queries in CanStoreX.

6.2.3. Insertion

Insertion facilitates updates to the B-tree. Updates to the B-tree are necessary when the xml document is modified. This helps us keep the tree up to date to the changes made to the XML document.

The algorithm for insertion is directly inspired by the concept of split in a search tree. The idea is to traverse the tree till we find the leaf node where the (key, pointer) belongs. If the leaf node can accommodate the pair the insertion is complete. On the other hand, if the leaf overflows, the node is split, the (key, pointer) pair is added to one of the split nodes and a (key, pointer) pair is added to the parent node. The key in this case is the middle record of the overflowing node, and the pointer is the address of the newly created node by the split. If the parent node has space in it, the insertion is completed, or else the parent is split, and a (key, pointer) pair migrates to its parent. This continues until a node that can absorb a record is encountered.

In our implementation we have taken special care about even number of keys and odd number of keys in a node while a split is applied. If we do not make this distinction then the split could lead to an imbalanced and invalid B-tree node. The implementation varies for leaf and

index nodes as the number of keys and pointers stored in leaf and index nodes is different.

Insertion can also be used to build the whole B-tree by inserting one node at a time. It is not an economic way of building the tree. In bulkloading we do not need to make comparisons required to insert a (key, pointer) pair but using insertion for creating the B-tree would contain all these comparisons. Also split in a B-tree is costly as in the worst case a split in the leaf can result in a split in all the levels of the tree. Insertion can be used as a last resort if the incoming keys are not sorted. Bulkloading cannot be used if the keys are not sorted; one option would be to sort the keys before using bulkloading to create the B-tree or use insertion.

7. Commands

The following section discusses the commands that are part of this implementation. Also it talks about additional CyDB commands required and used by us in our implementation.

7.1. B-tree Commands

In the following section we discuss all the commands related to B-tree which includes creation of a B-tree and retrieval and insertion of data to and from the B-tree.

- btree:>createbtree <config_file> <index_file> <creation_method>;

This command is used to create a b tree for a particular node in an XML document. The input parameters to this command are the config file which is manually created and stored on the disk. The index file which uniquely defines the B-tree to be created. The index file helps retrieve other parameters required to build the tree like the key type, input mode and whether this B-tree contains subsequences. The last parameter defines the method to be used for creation of the B-tree. It could be either One_By_One, Level_By_Level or Pyramid_Style.

- btree:>get <indexed_node> from[<config_file>,<index_file>] where id=<value> <output_file>;

This command is used to retrieve a single node stored in the B-tree for a given indexed node and write it to an output file on the disk. The input parameters are the indexed node (a node for which a B-tree was created), the config file which has the page configurations, the index file which contains information related to the B-tree created for the indexed node and an id which identifies the node in the xml document to be retrieved. To retrieve a single node we need to traverse the whole B-tree. The information stored in the config file is useful here. The config file stores the root of the B-tree which is the starting point of the B-tree. We need to traverse the tree till we reach the leaf node which contains the page id and offset of the node identified by the id. The output file is the file where the results are to be stored.

- btree:>get <indexed_node> from[<config_file>,<index_file>] all <output_file>;

This command is used to retrieve all the nodes stored in the B-tree for a given indexed node

and store it in an output file on the disk. The input parameters are the indexed node (a node for which a B-tree was created), the config file which has the page configurations, the index file which contains information related to the B-tree created for the indexed node. The starting leaf node or the page id of the first sequence node is required to retrieve all the keys stored in the B-tree. All the sequence pages are linked so it is easy to retrieve all the keys in one go. The key word all tells the parser to retrieve all the nodes in the B-tree. The output file is the file where the results are to be stored.

• btree:>get <indexed_node> from[<config_file>,<index_file>] where region=<value> <output_file>;

This command is used to retrieve all the nodes for a particular region in the B-tree for a given indexed node and store it in an output file on the disk. The input parameters are the indexed node (a node for which a B-tree was created), the config file which has the page configurations, the index file which contains information related to the B-tree created for the indexed node and the region for which to retrieve all the indexed nodes. We store additional information related to the indexed node for example in the benchmark document that we use if we have indexed all the nodes having an xpath "site/region/*//item"/ it means that the items are grouped under particular regions like asia, africa, europe etc. This command helps retrieve all the items in a particular region. To do this we store additional information in the config file. This is stored in the subsequences for a particular indexed node. It contains the page id and offset of a B-tree sequence page from where the items belonging to that region are stored.

• btree:>get <indexed_node>[first] from[<config_file>,<index_file>] where region=<value> <output_file>;

This command is used to retrieve the first node for a particular region in the B-tree for a given indexed node and store it in an output file on the disk. The input parameters are the indexed node (a node for which a B-tree was created), the config file which has the page configurations, the index file which contains information related to the B-tree created for the indexed node and the region for which to retrieve the first node. This command is just a part of storing additional information related to the indexed node.If the node to be indexed has an xpath "site/region/*//item"/ it means that the items are grouped under particular regions like

asia, africa, europe etc. This command helps retrieve the id of the first item in a particular region. Once we know the id we can traverse the tree to find the page id and offset of it in the B-tree leaf pages and store it in the config file.

- btree:>insertbtree <config_file> <index_file> <key> <page_id> <offset>;

This command is used to insert a page id and offset for a particular key in the b tree. This key belongs to a particular node in an XML document that has already been indexed using one of the methods mentioned above. The input file to this command is a B-tree config file which resides on the disk. The next input field is the index file which contains information related to the B-tree created for the indexed node followed by key, pageid and offset to be inserted in the B-tree.

7.2. Additional CyDIW Commands

Along with the B-tree commands we use an additional set of commands from other sub-sytem like the storage manager. Storage manager commands are used to create the storage, set up the buffer manager, get different parameters like number of pages allocated, number of pages deallocated, number of disk pages accessed, displaying the output files etc. Following is a brief listing of all the commands used by our implementation.

- CyDB:>createRawStorage storageconfig.xml;

The above command creates a raw storage by using the configuration mentioned in the storageconfig file. The configuration file includes details like where to create the storage, size of the storage, page size etc.

- CyDB:>useStorage Rawstorageconfig.xml;

The above command tells the system to use an existing storage whose location is mentioned in the storageconfig file.

- CyDB:>formatStorage <page_size>;

This command can be used to format the storage. The parameter page_size specifies the new page size to be used by the storage.

- CyDB:>startBufferManager <num_of_buffers>;

This command is used to start the buffer manager. One can specify the number of buffers to use for the current session.

- CyDB:>createfile <file_name> <size>;

  This command can be used to create a file either xml or bxml of the specified size.

- CyDB:>copyfile <source_file_name> <dest_file_name>;

  This command is used to convert a give file into another format.

- CyDB:>displayfile <file_name>;

  This command can be used to display any file on the disk in the browser. This is useful to view the results of any command in the browser.

- CyDB:>getpageallocatedcount;

  This command returns the number of pages allocated in the storage till the last query was executed.

- CyDB:>getpagedeallocatedcount;

  This command returns the number of pages deallocated in the storage till the last query was executed.

- CyDB:>getPageAccessCount;

  This command returns the number of pages accessed in the storage till the last query was executed.

8. Performance Evaluation

This sections talks about the performance of the different type of bulkloading algorithms. Also it talks about the difference in performance as a result of indexing the storage. For all the tests we used documents ranging from 1MB to 1GB.

Figure [9] compares the time required and page accesses required by the creation of B-tree using insertion and the two bulkloading algorithms implemented by us. Figure [9] (a) and (b)

Document Size	One_By_One	Level_By_Level	Pyramid_Style
1 MB	0.08	0.07	0.09
10 MB	0.42	0.94	0.73
100 MB	11.45	8.5	21.69
1000 MB (1GB)	129.79	118.68	188.60

(a) Time required (in mins) for creation of B-tree for item nodes

Document Size	One_By_One	Level_By_Level	Pyramid_Style
1 MB	387	5	3
10 MB	3935	24	22
100 MB	39203	43	31
1000 MB (1GB)	391995	259	139

(b) Page Access required for creation of B-tree for item nodes

Document Size	One_By_One	Level_By_Level	Pyramid_Style
1 MB	0.06	0.05	0.10
10 MB	0.84	0.62	0.65
100 MB	7.52	5.43	18.57
1000 MB (1GB)	133.32	73.76	209.61

(c) Time required (in mins) for creation of B-tree for person nodes

Document Size	One_By_One	Level_By_Level	Pyramid_Style
1 MB	459	4	2
10 MB	4789	7	4
100 MB	48045	61	31
1000 MB (1GB)	1173716	649	325

(d) Page Access required for creation of B-tree for person nodes

Figure 9. Performance Evaluation of Creation of B-trees

Document Size	With BTree	Without BTree
1 MB	0	0
10 MB	1	398
100 MB	1	3866
1000 MB (1 GB)	3	38654

(a) Page Access required to get a single item node

Document Size	With BTree	Without BTree
1 MB	0	0
10 MB	395	794
100 MB	3821	7722
1000 MB (1 GB)	38306	77240

(b) Page Access required to get all item nodes

Document Size	With BTree	Without BTree
1 MB	Eur = 0, NA = 0	Eur = 0, NA = 0
10 MB	Eur = 103, NA = 181	Eur = 179, NA = 367
100 MB	Eur = 1048, NA = 1759	Eur = 2121, NA = 3562
1000 MB (1 GB)	Eur = 10547, NA = 17624	Eur = 21293, NA = 35512

(c) Page Access required to get all item nodes in a single region

Figure 10. Performance evaluation of queries for fixed length keys

Document Size	With BTree	Without BTree
1 MB	0	0
10 MB	1	79
100 MB	1	754
1000 MB (1 GB)	3	7628

(a) Page Access required to get a single person node

Document Size	With BTree	Without BTree
1 MB	0	0
10 MB	36	74
100 MB	733	1507
1000 MB (1 GB)	8058	15243

(b) Page Access required to get all person nodes

Figure 11. Performance evaluation of queries for variable length keys

compare the time required and page accesses required to create a B-tree for the item nodes (fixed length keys) in XML documents of varying sizes. Figure [9] (c) and (d) compare the time required and page accesses required to create a B-tree for the person nodes (variable length keys). Among the two bulkloading algorithms Pyramid_Style takes more time than Level_By_Level. Ideally Pyramid_Style should have required less time but it takes more as pinning is not properly integrated in the buffer manager. So in pyramid style bulkloading the current pages at each level in the tree are being replaced more often and have to be read from the disk. The number of disk accesses required by pyramid style is the least as in this technique once a node is written to the disk we are done with it. The maximum number of disk accesses are required by the insertion technique.

Figure [10] gives a comparison of the disk accesses required for queries with and without B-trees for fixed length keys and Figure [11] gives a comparison between the queries with variable length keys. The performance of the queries implemented in the B-tree subsystem are compared to the queries in CanStoreX. We see an increase in the number of disk accesses as the size of the document increases. The best performance is achieved for single value retrieval. Without a B-tree these queries require more time which also increases as the id one is looking for increases. On the other hand the implementation with a B-tree gives the same performance for retrieving any key from the tree and it is so small that the performance of the system improves drastically. In all the other queries the page accesses required by a storage with indexes is half or less than half as compared to the storage without indexes.

Overall the performance of the system has improved because of indexing. Also the bulkloading algorithms implemented by us have improved the way B-trees are created.

9. Configuration, Deployment and Further Development of B-trees

In this section we discuss how the B-tree could be configured and deployed. We will also give some hints on how to improve certain aspects for further development. Some of our observations will be in a context that is wider than that of B-trees.

It helps to keep in mind that the storage in CyDIW consists of heaps of pages. A bit map to indicate the pages that are being used is stored in initial pages. The pages containing data follow immediately. Pages must be allocated before they can be used. The storage can be small, e.g. in a directory of choice on a laptop or span multiple disks. Currently CyDIW allows only one centralized storage with a buffer manager that all clients take for granted.

The page size is a global setting in the storage and clients are not free to change it. Although a page size of 16 KB was chosen for experimentation, it can vary from 1 KB to 64 KB. Therefore, with in a page 16 bit integers suffice in order to record offset of every byte in a page. In order to support a large number of pages 32-bit page addresses are chosen. With these settings our storage can be up to 2^{32} * 64 KB = 256 Terabytes in size. This itself can be tweaked easily to allow even larger storage. As we have a bias toward a small number of large file we have not cared to consider page addresses that are smaller than 32 bits. If one were to choose 16 bit inter page pointers, the maximum storage size would be 2^{16} * 64 KB = 4 GB, which may be enough for many applications.

One thing to keep in mind is that currently the storage in CyDIW can not be expanded on the fly. There are a couple of solutions to deal with this. A small amount of code can be modified to leave a larger number of pages in the bit map area for later use allowing the address space to be expanded without jeopardizing the inter-page pointers in the existing storage. Another solution is to modify the code so that the bit map has its own disjoint address space. Neither of these options should require much effort. The config file for storage is quite versatile to accommodate such changes in the code.

9.1. Variables

Our style of implementation requires that all variables used in the code for B-tree to be

listed in the XML-based config file for B-trees. These variables in various categories vary in terms of their complexity and how and when they are bound. It is best to think of a path expression in the XML document as a place holder for a variable where its definition, documentation, and value are to be stored. Some variables can only be read and such variables should not be modified or modified with considerable care – sometimes needing changes in the code. Some variables are written when some module is executed. These should never be altered manually.

9.1.1. Space utilization and page usage

Space utilization is an important parameter in file structures that, for performance reasons, is deliberately almost never chosen to be 100% for files that change dynamically. This parameter is well understood in databases.

The PageFractionUsed is a variable created by us. This is because 16 KB pages support a large fanout of 1600 in our use case and unless one is willing to spend weeks in testing with huge files, the height of the tree will not go beyond 2 or 3. The page size in CyDIW is a global setting for all clients and unless the B-tree is the only client it does not have the freedom of choosing a different page size. In order to stress test one may use only a small part of the page, say 6.25%, so the contents will be stored in the first 16K * 0.625 = 1 KB to be used to reduce the fanout and have a tree with a greater height. This amounts to having smaller pages. The global page size cannot be enlarged by the B-tree client. Even simulation of smaller page size if meant for testing.

9.1.2. Page header

A field length of 4 bytes is chosen for all variables stored in header of materialized pages. Some of these variables point to other pages and some specify offset with in a page. The values of these variables are copied into 4-byte integers in their binary format to be stored in the corresponding field of the page header. Using 16 bits instead of 32 bits can improve the performance significantly and support very large primary datasets specially when the B-tree is to be used as secondary index. Such a venture will require changes in the code.

9.1.3. Index config

A B-tree itself can be stored as a file in the CyDIW storage. However, the pointer to the root

and the sequence are stored as well. If the B-tree were to be deleted, these pointers are helpful. These pointers are written by the module that creates the tree.

9.1.4. Sequence and subsequences

In a sequence or a given subsequence nodes are siblings. But across subsequences they are first cousins. Currently they cannot be more distantly related then cousin-ship. The pointer to the first nodes in the subsequences are recorded by a module when the subsequences are created.

9.1.5. Primary file

Currently it is assumed that the primary records to be indexed reside in a paginated XML document in our storage and an XML path to these records is given. This kind of parameter passing can be improved via iterators. Note that instreaming the secondary or primary records to be indexed should be considered responsibility of a client needing the B-tree. A good way of handling this is to expect the client to create an iterator. The iterator should be opened by the B-tree and nodes should be extracted via calls to getNext. Only the name of the iterator and the prefix of the client system should be provided as the parameters.

9.2. Supporting unsorted data

We have assumed that the data is pre-sorted by keys. What if the data were not sorted? An iterator for sorting in XML is available in CyDIW. There the assumption is that the records (subtrees) can vary in length and even span multiple pages – whereas in relational databases the records are typically small, have a fixed length, and several records fit on a page. Swapping records is the staple operation to carry-out sorting and swapping large records is not prudent. Therefore the sorting available in CyDIW first sorts key-pointer pairs to records and in the end coalesces the records in order. Sorting in relational database setting, although much easier, has not yet been tackled and would need to be implemented. Once the sorting is in place the instreaming iterator should take a detour via a sorter. A composition should be developed carefully – in the last pass of sorting, instead of writing records to the disk and then outstream them, they can be outstreamed readily.

9.3. Deployment of the tree

The procedure for deployment of the tree depends upon the type of user. Three types of users are envisioned: (a) a client in CyDIW; (b) a client who only wishes to use CyDIW storage and buffer managers; and (c) a client with its own page-based storage.

9.3.1. A client of CyDIW

Not much needs to be said about the client of the storage. The implementation can be deployed directly by such a client via commands with B-tree prefix given in Section 7.

9.3.2. A client of CyDIW storage

A client can use CyDIW storage, and storage and buffer managers. Commands for creating a storage anywhere on the disk are included in CyDIW and sampled in Appendix B. The tree can be a primary or a secondary structure. In case it is a secondary structure, the primary file can reside elsewhere but their pointers can be recorded in sequence nodes in the B-tree. If the client is the only user of the storage in CyDIW, it has the freedom to choose a page size.

9.3.3. A client with own storage

A client who wishes to use their own storage will have to provide implementation for reading, writing, allocating, and de-allocating pages. If the names of the functions are to conform to the names used by the client in its storage management the names should be changed in all calls to these functions from the code that implements the B-tree.

9.4. Deletion in B-trees

Sometimes deletions are not carried out immediately and done in batches later on. This can be supported by marking records with a tombstone-bit (or some other means) to indicate its logical absence. This will not disturb most algorithms as the relative order of records in the nodes of the B-tree will remain intact. Deletions, currently not available, will need to be added. Obviously appropriate strategy for managing deletions will need to be developed as well.

9.5. Concurrency and recovery

Support for concurrent use of a B-tree by multiple users without corrupting it is an important and very interesting problem. Recovery in presence of software or hardware crashes is

also an important issue. Any industrial strength implementation has to take these aspects into account.

9.6. Support for non-unique keys and compression of keys

Currently our B-tree only works for unique keys. Support for non unique keys can be added. Also, some keys, specially alphanumeric ones can be long but only some prefix may suffice for navigation within the tree. Such compression can lead to a larger fanout, thereby reducing the height of the tree, that would in turn lead to considerable improvement in performance.

9.7. Deployment of the tree in a database query language

We have not integrated the B-tree in our XQuery Engine. In general this is a separate issue. A query engine will need to access different modules of the tree directly. In addition algorithms in query engine has to consider parsing issues in making them aware of the existence of the tree and access paths in the B-tree.

Implementation of B-trees is a challenging problem. As stated before, numerous good implementations of B-trees exists in industry and academia. We have basically added a style to make configuration, deployment, and testing much easier. This style should be maintained for all future developments.

10. Conclusion

We successfully came up with a B-tree implementation which is easy to configure and deploy. The configuration file gives the user the flexibility to create trees with different node sizes. The B-Tree implementation presented by us is a general purpose indexing mechanism which can be easily integrated in any system. The methodology when used as a secondary index structure is largely independent of the primary data. Our implementation supports both fixed length and variable length keys. Also it gives the user the freedom to decide the page size to use for each node in the B-tree. In this project we also came up with better ways of creating index structures for large documents by bulkloading. By integrating the B-tree subsystem with CanStoreX we successfully proved the usefulness of indexing in improving the performance of the system.

11. Future work

We already discussed some future development that can be done for B-trees in Section 9. The things needed to be integrated in the existing implementation are support for non unique keys and compression of keys, deletion of records in the B-trees, making XQuery aware of the B-trees and finally concurrency and recovery of the B-tree subsystem. In this section we will discuss more about improving the features required for better performance and maintenance of the system.

One important feature that would enhance the performance of the system would be caching a part of the B-tree in main memory. This boosts the performance of the system as it reduces the number of page access required to find the key in the B-tree. We have already seen that B-trees have small height as the fanout of each node is large. So just pinning the root of a B-tree would help save a lot of time and resources.

Another important addition that would help maintain a clean storage is support for deletion of B-trees. If a B-tree is no longer needed in the system the space used by the B-tree should be deallocated and made available to the storage manager. This feature is currently not supported in our implementation but would lead to a clean storage.

REFERENCES

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

[2] 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, Ames, Iowa

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

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

[5] Narayanan, V., 2009. A workbench for advanced database implementation and benchmarking

[6] Gadia, S.K.. An elemental approach to databases

[7] Gadia, S.K.. Database Implementation Workbench DIW

[8] Bayer, R., McCreight, E., 1972. Organization and maintenance of large ordered indexes. Acts Informatica, 1:173-189

[9] Ramakrishnan, R., Gehrke, J., 2002. Database Management Systems, McGraw-Hill Science/Engineering/Math

[10] Ramakrishnan, R.. Minibase. http://www.cs.wisc.edu/coral/mini_doc/minibase.html. University of Wisconsin.

[11] Olson, M.A., Bostic K., Seltzer M.. Berkeley DB. Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference

[12] Oracle Berkeley DB XML. http://www.oracle.com/us/products/database/berkeley-db/

[13] Frankes, W.B., Baeza-Yates, R., Ed., 1992. Information Retrieval: Data Structures and Algorithms. Prentice Hall

[14] Knuth, D.E., 1998. The art of computer programming, volume 3: (2nd ed.) sorting and searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA

[15] Bercken, J.V.D., Seeger, B., 2001. An Evaluation of Generic Bulk Loading Techniques. Proceedings of the 27th International Conference on Very Large Data Bases, p.461-470, September 11-14

[16] Berchtold S., Böhm, C., Kriegel, H.P., 1998. Improving the Query Performance of High-Dimensional Index Structures by Bulk-Load Operations. Proceedings of the 6th International Conference on Extending Database Technology: Advances in Database Technology, p.216-230, March 23-27

[17] Bercken, J.V.D., Seeger, B., Widmayer, P., 1997. A Generic Approach to Bulk Loading Multidimensional Index Structures. Proceedings of the 23rd International Conference on Very Large Data Bases, p.406-415, August 25-29

[18] Arge, L., Hinrichs, K., Vahrenhold, J., Vitter, J.S.. Efficient Bulk Operations on Dynamic Rtrees.ALENEX 1999: 328-348

[19] Jermaine, C., Datta, A., Omiecinski, E.. A Novel Index Supporting High Volume Data Warehouse Insertion. VLDB 1999: 235-246

[20] Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I., Busse, R., 2002. XMark: A Benchmark for XML Data Management. Proceedings of the 28th VLDB Conference (2002), pp. 974-985

[21] Li, Q., Moon, B., 2001. Indexing and Querying XML Data for Regular Path Expressions. Proceedings of the 27th International Conference on Very Large Data Bases, p.361-370, September 11-14

[22] Jiang, H., Lu H., Wang, W., Ooi, B.C., 2003. XR-Tree: Indexing XML data for efficient structural joins. Proceedings of the 19th International Conference on Data Engineering, pp.253–264, Bangalore, India

[23] Goldman, R., Widom, J., 1997. Dataguides: Enabling query formulation and optimization in semistrucutred databases. Proceedings of the 23rd International Conference on Very Large Data Bases, pp.436–445, Athens, Greece

[24] Min, J., Chung, C., Shim, K., 2005. An adaptive path index for XML data using the query workload. Information Systems, vol.30, no.6, pp.467–487

[25] Shimizu, T., Masatoshi, Yoshikawa, 2005. Full-Text and Structural XML Indexing on B+-Tree. Proceedings of the 16th International Conference on Database and Expert Systems Applications (DEXA 2005)

[26] Wang, H., Park, S., Fan, W., Yu, P.S., 2003. ViST: A dynamic index method for querying XML data by tree structures. Proceedings of 2003 ACM SIGMOD International Conference on Management of Data, pp.110–121, San Diego, USA

[27] Raw, P.R., Moon, B., 2004. PRIX: Indexing and querying XML using pr¨ufer sequences. Proceedongs of the 20th International Conference on Data Engineering, pp.288–300, Boston, USA

[28] Wang, H., Meng, X., 2005. On the sequencing of tree structures for XML indexing. Proceedings of the 21st International Conference on Data Engineering, pp.372–383, Tokyo, Japan

[29] eXist-Open Souce Native XML Database. http://exist.sourceforge.net/xquery.html

[30] Milo, T., Suciu, D., 1999. Index Structures for Path Expressions. Proceeding of the 7th International Conference on Database Theory, p.277-295

[31] Madria, S., Chen, Y., Passi, K., Bhowmick, S., 2007. Efficient processing of XPath queries using indexes. Information Systems, v.32 n.1, p.131-159

[32] Rao, J., Ross, K.A.. Making B+- trees cache conscious in main memory. Proceedings of the 2000 ACM SIGMOD international conference on Management of data

[33] Kahveci, T., Singh, A.K.. Efficient Index Structures for String Databases. Proceedings of the 27th International Conference on Very Large Data Bases 2001

[34] Jin, L., Li, C., Koudas , N., Tung, A.K.H, 2005. Indexing mixed types for approximate retrieval. Proceedings of the 31st international conference on Very large data bases

APPENDIX A. BTreeConfig.xml

This section shows a sample BTreeConfig file. The config file has information regarding the node structures to be used while creating the B-trees. As mentioned earlier a config file can hold information about more than one indexes for the same or different bxml documents. Similarly a different config file can be used for each index created. So different types of B-trees can co exist in the system. Following is a sample BTreeConfig File which contains index information for both item and person nodes.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><!-- A configuration file to create and maintain a secondary B-tree --><!-- The sequence contains key-pointer to primary records --><!-- Our B-Tree also allow a sequence to be broken into subsequences --><!-- Multiple indexes can be installed on the same primary file --><BTreeConfig>
  <PageConfig>
    <!-- A page consists of a sequence of bytes -->
    <!-- Nodes in the B-Tree are implemented as pages in a central storage -->
    <!-- Pages have a fixed size specified in StorageCongig.xml document -->
    <!-- A page consists of a sequence of bytes programmed by developer -->
    <SubPageSize PageFractionUsed="0.06"/>
    <!-- Limiting to part of the page requires smaller test data -->
    <!-- After testing the page size must be set to "1.0" -->
    <SpaceUtilization SU="1"/>
    <!-- Limiting to part of the page requires smaller test data -->
    <PageHeader Length="10" unit="byte">
      <!-- Client of this utility must ensure: offset of next field = offeset+length of current -->
      <!-- Redundancy is for ease of programming -->
      <!-- Data will start immediately after the header -->
      <HeaderField Length="4" Name="CurrentPagePtr" Offset="0"/>
      <HeaderField Length="4" Name="NextPagePtr" Offset="4"/>
      <HeaderField Length="4" Name="PageTypePtr" Offset="8"/>
      <HeaderField Length="4" Name="KeyTypePtr" Offset="12"/>
      <HeaderField Length="4" Name="NumOfKeysPtr" Offset="16"/>
      <HeaderField Length="4" Name="FreeSpacePtr" Offset="20"/>
      <HeaderField Length="4" Name="FreeSpaceForContentPtr" Offset="24"/>
      <HeaderField Length="4" Name="LargestKeyPtr" Offset="28"/>
      <HeaderField Length="4" Name="DataStartIndex" Offset="40"/>
    </PageHeader>
  </PageConfig>
  <Indexes>
    <IndexConfig IndexFile="AuctionItems.btree" InputMode="Iterator" KeyType="FixedLength" SubSequences="Yes">
      <!-- Addresses consist of PageID and Offset within the page -->
      <!-- All addresses are populated by the B-Tree utility -->
      <PrimaryFile DocName="Test.bxml" Key="item" Seq="site/region/*//item"/>
      <BTreeAccessPointers RootPageId="85" SequencePageId="83"/>
```

```
        <SubSequences SubSeq="/site/region/*">
          <!-- Next item is included for ease of programming -->
          <SubSequence Name="africa" Next="asia" Offset="40" PageId="83"/>
          <SubSequence Name="asia" Next="australia" Offset="80" PageId="83"/>
          <SubSequence Name="australia" Next="europe" Offset="260" PageId="83"/>
          <SubSequence Name="europe" Next="namerica" Offset="450" PageId="83"/>
          <SubSequence Name="namerica" Next="samerica" Offset="40" PageId="84"/>
          <SubSequence Name="samerica" Next="none" Offset="470" PageId="86"/>
        </SubSequences>
      </IndexConfig>
    <IndexConfig IndexFile="AuctionPersons.btree" InputMode="Iterator" KeyType="VariableLength"
SubSequences="No">
        <!-- Addresses consist of PageID and Offset within the page -->
        <!-- All addresses are populated by the B-Tree utility -->
        <PrimaryFile DocName="Test.bxml" Key="person" Seq="site/people/person"/>
        <BTreeAccessPointers RootPageId="81" SequencePageId="79"/>
      </IndexConfig>
    </Indexes>
</BTreeConfig>
```

APPENDIX B. Batch of Commands

This section lists a batch of commands for B-trees. The batch is completely self contained as it includes commands from creation of storage, creation of bxml file to creation of B-trees for indexes, and retrieval and insertion to and from the B-trees. Along with that it also contains commands to display the output of each retrieval in a browser and also log the performance of the system by calculating the time and page accesses required by any command. This batch of commands tries to summarize the functionality of the whole system.

Demo File:

```
// Setup the storage;
// Create the storage;
CyDB:>createRawStorage Rawstorageconfig.xml;
// To use an existing storage;
CyDB:> useStorage StorageConfig.xml;
// Format the storage;
CyDB:>formatStorage 16;
// Start Buffer Manager;
CyDB:>startBufferManager 100;

// Create an xml document of 10 MB;
CyDB:>createfile Test.bxml 10;
// Number of pages allocated
CyDB:>getpageallocatedcount;

// Create BTree for given bxml file using level by level bulkloading where key type is fixed length;
btree:>createbtree e:\BTreeConfig.xml AuctionItems.btree Level_By_Level;
// Number of pages allocated
CyDB:>getpageallocatedcount;

// Get all items from the btree;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] all e:\opLAllItem.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLAllItem.xml;

// Get single item from btree where key = 18820;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where id=18820 e:\opLItem.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLItem.xml;

// Get items that belong to a particular region;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=africa e:\opLAfrica.xml;
```

```
// Display the output file in the browser;
CyDB:>displayfile E:\opLAfrica.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=asia e:\opLAsia.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLAsia.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=australia e:\opLAus.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLAus.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=europe e:\opLEur.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLEur.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=namerica e:\opLNA.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLNA.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=samerica e:\opLSA.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLSA.xml;

// Get first item that belongs to a particular region;
// Internally used by get the first item for each region and populate the btreeconfig file
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=africa e:\opLFAf-
rica.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLFAfrica.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=asia e:\opLFAsia.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLFAsia.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=australia e:\opL-
FAus.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLFAus.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=europe e:\opL-
FEur.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLFEur.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=namerica
e:\opLFNA.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLFNA.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=samerica
e:\opLFSA.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLFSA.xml;

// Insert an item into the btree
btree:>insertbtree e:\BTreeConfig.xml AuctionItems.btree 50000 100 200;

// Create BTree for given bxml file using level by level bulkloading where key type is variable length;
btree:>createbtree e:\BTreeConfig.xml AuctionPersons.btree Level_By_Level;
```

```
// Get all person nodes from the btree;
btree:>get person from[e:\BTreeConfig.xml,AuctionPersons.btree] all e:\opLAllPerson.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opLAllPerson.xml;

// Get single person node from btree where key = person18600;
btree:>get person from[e:\BTreeConfig.xml,AuctionPersons.btree] where id=person186 e:\opLPer-
son.xml;
// Display the output file in the browser;
NC94:>displayfile E:\opLPerson.xml;

// Insert a person node into the btree
btree:>insertbtree e:\BTreeConfig.xml AuctionPersons.btree person30000 100 200;

// Create btree using Pyramid Style Bulkloading;
// To use an existing storage;
CyDB:> useStorage StorageConfig.xml;
// Format the storage;
CyDB:>formatStorage 16;
// Start Buffer Manager;
CyDB:>startBufferManager 100;

// Create an xml document of 10 MB;
CyDB:>createfile Test.bxml 10;
// Number of pages allocated
CyDB:>getpageallocatedcount;

// Create BTree for given bxml file  Pyramid Style bulkloading where key type is fixed length;
btree:>createbtree e:\BTreeConfig.xml AuctionItems.btree Pyramid_Style;
// Number of pages allocated
CyDB:>getpageallocatedcount;

// Get all items from the btree;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] all e:\opPAllItem.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPAllItem.xml;

// Get single item from btree where key = 18820;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where id=18820 e:\opPItem.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPItem.xml;

// Get items that belong to a particular region;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=africa e:\opPAfrica.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPAfrica.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=asia e:\opPAsia.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPAsia.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=australia e:\opPAus.xml;
```

```
// Display the output file in the browser;
CyDB:>displayfile E:\opPAus.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=europe e:\opPEur.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPEur.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=namerica e:\opPNA.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPNA.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=samerica e:\opPSA.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPSA.xml;

// Get first item that belongs to a particular region;
// Internally used by get the first item for each region and populate the btreeconfig file
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=africa e:\opPFAf-
rica.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPFAfrica.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=asia e:\opPFAsia.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPFAsia.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=australia e:\opP-
FAus.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPFAus.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=europe e:\opP-
FEur.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPFEur.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=namerica
e:\opPFNA.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPFNA.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=samerica
e:\opPFSA.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPFSA.xml;

// Insert an item into the btree
btree:>insertbtree e:\BTreeConfig.xml AuctionItems.btree 50000 100 200;

// Create BTree for given bxml file using Pyramid Style bulkloading where key type is variable length;
btree:>createbtree e:\BTreeConfig.xml AuctionPersons.btree Pyramid_Style;

// Get all person nodes from the btree;
btree:>get person from[e:\BTreeConfig.xml,AuctionPersons.btree] all e:\opPAllPerson.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opPAllPerson.xml;

// Get single person node from btree where key = person186;
```

```
btree:>get person from[e:\BTreeConfig.xml,AuctionPersons.btree] where id=person186 e:\opPPerson.xml;
// Display the output file in the browser;
NC94:>displayfile E:\opPPerson.xml;

// Insert a person node into the btree
btree:>insertbtree e:\BTreeConfig.xml AuctionPersons.btree person30000 100 200;

// Create btree using One By One Insertion;
// To use an existing storage;
CyDB:> useStorage StorageConfig.xml;
// Format the storage;
CyDB:>formatStorage 16;
// Start Buffer Manager;
CyDB:>startBufferManager 100;

// Create an xml document of 10 MB;
CyDB:>createfile Test.bxml 10;
// Number of pages allocated
CyDB:>getpageallocatedcount;

// Create BTree for given bxml file using Insertion where key type is  fixed length;
btree:>createbtree e:\BTreeConfig.xml AuctionItems.btree One_By_One;
// Number of pages allocated
CyDB:>getpageallocatedcount;

// Get all items from the btree;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] all e:\opIAllItem.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opIAllItem.xml;

// Get single item from btree where key = 18820;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where id=18820 e:\opIItem.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opIItem.xml;

// Get items that belong to a particular region;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=africa e:\opIAfrica.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opIAfrica.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=asia e:\opIAsia.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opIAsia.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=australia e:\opIAus.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opIAus.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=europe e:\opIEur.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opIEur.xml;
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=namerica e:\opINA.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opINA.xml;
```

```
btree:>get item from[e:\BTreeConfig.xml,AuctionItems.btree] where region=samerica e:\opISA.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opISA.xml;

// Get first item that belongs to a particular region;
// Internally used by get the first item for each region and populate the btreeconfig file
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=africa e:\opIFAfrica.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opIFAfrica.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=asia e:\opIFAsia.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opIFAsia.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=australia e:\opIFAus.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opIFAus.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=europe e:\opIFEur.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opIFEur.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=namerica e:\opIFNA.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opIFNA.xml;
btree:>get item[first] from[e:\BTreeConfig.xml,AuctionItems.btree] where region=samerica e:\opIFSA.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opIFSA.xml;

// Insert an item into the btree
btree:>insertbtree e:\BTreeConfig.xml AuctionItems.btree 50000 100 200;

// Create BTree for given bxml file using Insertion where key type is variable length;
btree:>createbtree e:\BTreeConfig.xml AuctionPersons.btree One_BY_One;

// Get all person nodes from the btree;
btree:>get person from[e:\BTreeConfig.xml,AuctionPersons.btree] all e:\opIAllPerson.xml;
// Display the output file in the browser;
CyDB:>displayfile E:\opIAllPerson.xml;

// Get single person node from btree where key = person186;
btree:>get person from[e:\BTreeConfig.xml,AuctionPersons.btree] where id=person186 e:\opIPerson.xml;
// Display the output file in the browser;
NC94:>displayfile E:\opIPerson.xml;

// Insert a person node into the btree
btree:>insertbtree e:\BTreeConfig.xml AuctionPersons.btree person30000 100 200;
```

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my heartfelt gratitude to all those who helped me with the various aspects of my research work and the writing of this thesis. First and formost I would like to thank Dr. Shashi Gadia for his expert guidance, constant support and patience throughout my research. This thesis would not have been possible without his vision and encouragement.

I would like to thank my committee members Dr. Simanta Mitra and Dr. Wallapak Tavanapong for their contribution to this work.

I would like to thank my colleagues in the Department of Computer Science for whom I have great regard. I would like to personally thank Xinyuan Zhao for his help throughout this thesis. Also I would like to thank Kartic Ramesh, Sugam Sharma and Valliappan Narayanan for their help and support.

I would like to express my loving thanks to my parents Babu and Prema Shetty, family and friends for their support and encouragement during this endeavor.