2017

A User Configurable B-tree Implementation as a Utility

Sheng Bi
Iowa State University

A User Configurable B-tree Implementation as a Utility

by

Sheng Bi

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 Gadia, Major Professor
Ying Cai
Phillip Jones

Iowa State University

Ames, Iowa

2017

DEDICATION

Throughout my life two people have always been there during those difficult and trying times. I would like to dedicate this thesis and everything I do to my dear parents. I would not be who I am today without the love and support from them. Although recent years our time together was brief when I am studying abroad, their contributions to my life will be felt forever.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

CYDIW	Cyclone Database Implementation Workbench
PTC	Produce Transform and Consume
HDD	Hard Disk Drive
FDD	Floppy Disk Drive
LBA	Logic Block Address
NFS	Network File System

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Shashi Gadia, and my committee members, Dr. Ying Cai, and Dr. Phillip Jones, for their guidance and support throughout the course of this research.

In addition, I would also like to thank my friends, colleagues, the department faculty and staff for making my time at Iowa State University a wonderful experience. I want to also offer my appreciation to those who were willing to participate in my surveys and observations, without whom, this thesis would not have been possible.

ABSTRACT

B-trees are widely used in management of data to support good performance for storage, retrievals and updates. Many excellent implementations of B-tree exist in industry and academia. However, it is hard to find one that is easily configurable for clients who need to create and use B-trees. In this work, we undertake implementation of B-trees as a utility. The utility absorbs most of the logistical support needed in creation and maintenance of a B-tree leaving only tasks that can only be performed by the clients to them. For exchange of data a two ended iterator-based framework called PTC, short for produce, transform, consume, is offered that produces one record at a time, the record is optionally processed, and then consumed. The PTC-based exchange is quite versatile. For example, it can be used in generating data and storing it in a file, converting data from text (resp. binary) to binary (resp. text) formats, creating a B-tree by inserting one tuple at a time or bulk loading, and out streaming the data from a B-tree to a destination such as query processor. XML is used to describe essential configuration settings for B-trees, records, and keys. A Java class file is automatically generated from the record configuration to provide support to upper level modules such as page-based storage and comparisons of keys needed by B-tree algorithms.

CHAPTER 1. INTRODUCTION

In this chapter, we will cover disk system, file system, database management system and I/O Paging. Then we will introduce what are B-trees, and why B-trees are import.

Disk System

Computer disk storage is a general category of storage mechanisms where data are persistently recorded onto a surface layer of one or more rotating disks, by different optical, electronic, magnetic, or mechanical changes. Disk drives are devices that designed and implemented following this mechanism. Nowadays, popular disk types are the hard disk drive (HDD) containing a non-removable disk, the floppy disk drive (FDD) [2] with its a removable floppy disk, and the optical disk drive (ODD) with an associated optical disk.

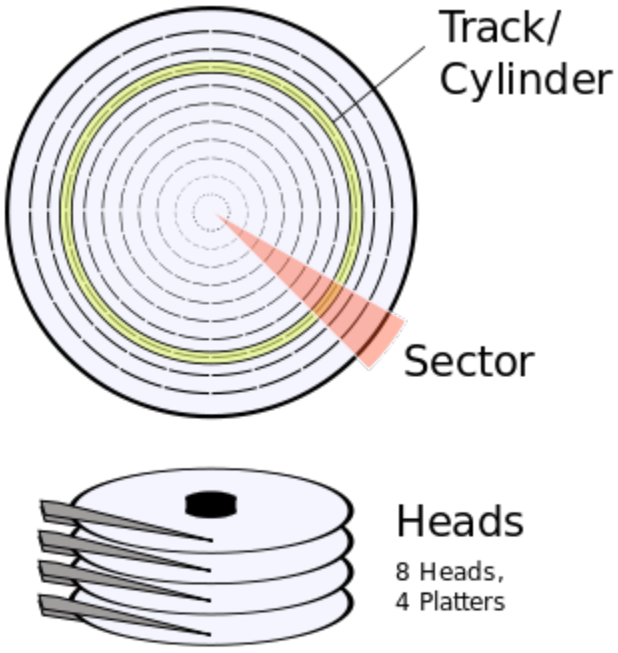

Figure 1 Cylinder-Sector-Header Structure[1]

Modern digital disk drives are designed as block storage devices. In an entire disk drive, multiple logical blocks are divided, and they are also called collection of sectors. Blocks are located by their logical block address (LBA). Read and write operations on a disk happen at the granularity of blocks. In the early stage of disk storage, the capacity was quite low and has been improved in several ways. For example, developments of mechanical design and manufacture permit smaller and more precise disk headers, which means that more tracks can be recorded on each of the disks. Also, improvements in data compression technologies have enlarged the data volume to be stored in each of the individual sectors.

Data is stored on cylinders, heads and sectors by the disk drive. For each file, the sectors unit is the smallest size of data to be stored in a hard disk drive. Usually, one file will have multiple sectors units assigned to it. The smallest unit entity in a CD is called frame, it is combined of 33 bytes, which is divided 24 bytes to store 16-bit stereo samples (2 bytes * 2 channels * 6 samples = 24 bytes), and 9 bytes to store 8 bytes CIRC error-correction bytes and another byte sub-code for control and display.

File System

As we already introduced the physical firmware of data storage, another important concept is file system. In computing, data is stored, retrieved and controlled through a file system. By separating data into various pieces and assigning each piece an unique identification, the operating system could easily manage a large volume of data. Also, the information is uniquely indicated and isolated, which brings some safety naturally. As many physical storage pieces are assigned together to be a group, we could name each group of data to be a “file”. And a “file system” is a collection of the structure and logic rules used to manage those groups of information in some physical storage drives.

Many different kinds of file systems have been proposed. Each one has different design principle and structure, speed, extensibility consideration, security and size and more. Some file systems are designed for specific academic or industry scenarios. For example, the ISO 9660 file system is proposed and implemented specifically for optical discs.

Apparently, we have numerous types of storage devices as physical infrastructures for multiple file systems. Those storage devices are using different categories of media. The most widely used physical storage device is HDD. Other kinds of media that are served as physical devices include optical discs, magnetic tapes, and solid state drive (SSD). In some cases, the computer's main memory (random-access memory, RAM) is served for a temporary file system for in-session usage.

Regarding computer networks and distributed systems, some file systems are not only used on local data storage devices, but also provide data access through a network protocol, such as NFS [3], and SMB [4]. Some file systems are supporting virtual files [5] which are computed on request or are merely a mapping into a different file system served as a storage bank. The core of file system manages access to both the content of files and the metadata. It can handle storage space arrangement, reliability, efficiency.

Modern file systems usually consist of two or three layers. For interaction with the user application, the logical file system provides the application program interface (API) for file operations, including OPEN, READ, WRITE, CLOSE, and so on. It passes users' requests to the lower layer for processing. The logical file system manages a table of entries for opened files and all preprocessed file descriptors. Generally, it provides directory access, file operations and security protection.

The one below logical layer is called virtual file system. It interacts with the upper layer, and provides support for multiple thread physical file operations and data concurrency. The third layer is called physical file system, which directly interacts with storage devices via physical operations. It operates physical blocks with reading or writing tasks. It contains a buffer and provides memory management. In most cases, physical operations are conducted with the help of the device driver or the channel to drive the storage device.

Database Management System

The need of maintaining a group data reliable and reusable keeps growing, and we are having a huge demand for data creation, update, retrieving and deletion. Sometimes, we also want the system to support data range query, join, grouping and so on. Also, we are expecting data management can be related with the concept of object-oriented design. Nowadays, a high demand for massively distributed databases [6] is raising up… As we look backward, the concept of database management system is keep developing, and also new techniques are brought onto the table as different demands come out.

The introduction of the term database started with the need of direct-access storage from mid-1960s [9]. Edgar Codd described a new system for storing and working with large database [7]. He brought up the idea of “table” of fixed-length records, with different types of attributes being included in each record. IBM started working on Codd’s concepts of relational database. And the first version was born in 1975, it started on multi-table system where data could be split into multiple tables, so that the relation did not have to be in a big chunk of single table. And in 1978, they added a standardized query language (SQL) into the system. Based on IBM’s research, Larry Ellison started Oracle Database, and finally defeated IBM to the market in 1979 with Oracle Database Version 2.

I/O Paging

As the file systems have been deeply merged into modern operating systems [8], we will discuss its paging property under the framework of the operating system. Paging is the memory management scheme, in order to store, retrieve, and communicate between the main memory (RAM) and the external storage (usually SSD or HDD). In a paging framework, both memory and disk are logically divided into multiple same-size blocks called pages. For example, under a 4 kilobytes page size setting, an 8GB memory will be divided into 2 million pages, and a 512GB external storage device will be divided into 128 million pages. This idea is fundamental in order to support virtual memory, which is using SSD or HDD as external memory, to let programs exceed the size of real physical memory.

When we create a new file in the disk, we are actually assigning a series of pages for this file. The critical paging information, like privilege, is included in the metadata. some other information, like next page, is usually contained within each page header itself. And all the OPEN, READ, WRITE, CLOSE operations, are conducted through the process of page retrieving, main memory loading, operations and writing back to disk, according to the page address.

B-tree

Trees are widely used in file system and database, dynamic search trees include binary search tree, balanced binary search tree, red-black tree, B-tree/B+tree [10, 11]. While the first three trees are typical binary structure, their search time complexity is O(logN), linear to the tree's depth. So one optimization idea could be very straightforward: increasing efficiency by decreasing depth of the tree.

As we know that disk I/O is much slower than operations inside memory. In practical, binary search trees or balanced binary search trees suffer from a larger depth and more disk

I/Os. In order to overcome the disadvantage of binary search trees, researchers tried to increase the fan-out of internal node in each level. In 1970, Rudolf Bayer and Edward M. McCreight proposed a new balanced tree data structure [12].

B-tree is a N-ary structure which is designed for disks and other external storage devices. Different from binary search trees, its fan-out is larger, and it is self-balanced. As the following figure shows a B-tree of depth equals to 3:

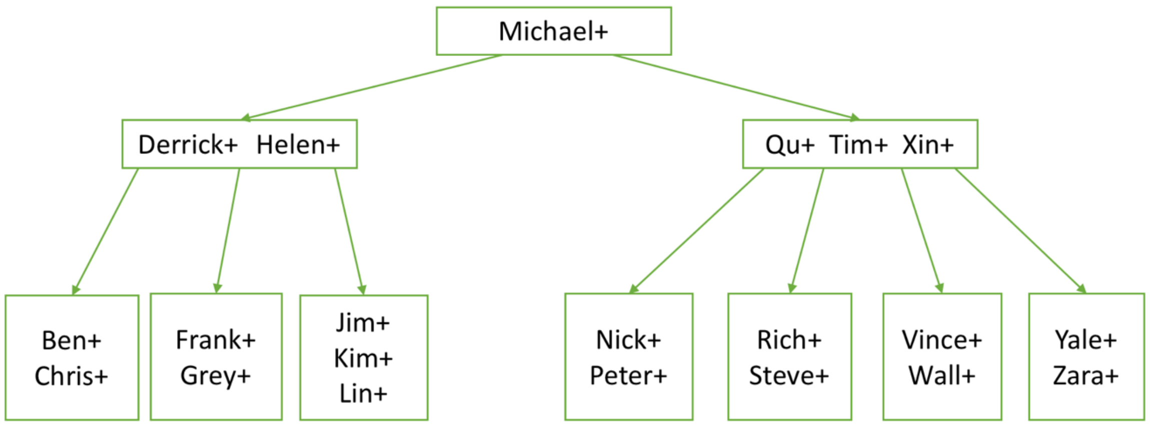

Figure 2 An example of B-tree

According to Knuth's definition, a B-tree of order m show satisfies the following properties:

Every node has at most m children (m >= 2).

Every index node (except root) is required to be half-full at least [ceil(m/2)].

The root has more than one children if it does not serve as a leaf node.

An index node keeps k records in order, and it has k+1 children.

All leaves are in the same level.

B+tree

A B+tree is also an N-ary tree but keep all integrated tuples in leaf nodes. It usually has a variable but often larger fan-out of internal nodes, comparing with B-tree. Also, different from B-tree, the root may be either as a leaf node, or a node with at least two fan-outs. Briefly speaking, the B+tree can be taken as a B-tree where each internal only contains keys, rather than entire tuples, and leaf node contains all tuples at the most bottom level.

Given the following simple example, we can see how a relation Employee is organized as a B+tree.

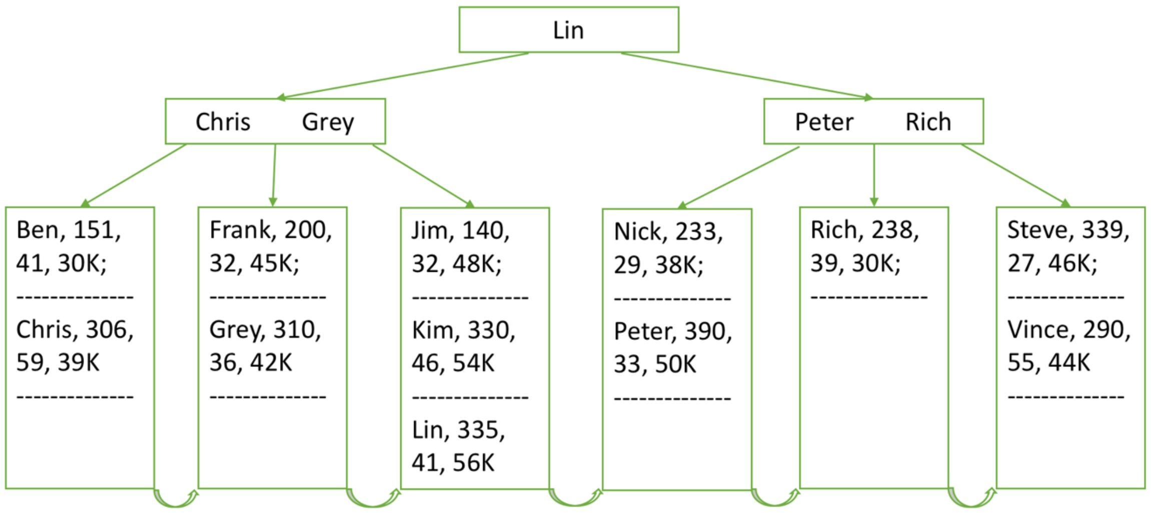

Figure 3 An Example of B+tree

In the relation Emp, all records are organized according to the key Emp_Name attribute. The height of this B+tree is 3. Each time, when user want to retrieve any record in this relation, it requires 3 page accesses, which is the height of B+tree. Sequential traversal can be done with the help of this structure, as long as the program keep the first sequence page at the right place.

The excellent property of B+tree, such as larger fan-out, smaller height, and better time complexity in keep data sorted, make it widely used in block-oriented storage context, in particular, file systems, and database management systems. With the fan-out advantage, it can reduce the number of I/O operations required to find an element significantly. Since B+tree is more widely used than B-tree, sometimes, we are calling B+tree to be B-tree if there is no need to differentiate. In the following paragraphs and chapters, the words B-tree are actually referring to B+tree if no specification.

In order to take an overview of B-tree's efficiency analysis, we can name the order, or branching factor of this B-tree as b. It indicates how many children nodes can be linked in an internal node (including root node). Assume the actual number of children nodes to be m. There is a constraint for all internal nodes except root node, [b/2] <= m <= b, as we allow root have at least two children. If the order of a B-tree is 9, then all internal node, except the root node, should have 5-9 children. There is no constraint of children number of leaf node, but the leaf node itself, should have at least [b/2]-1 but at most b-1 records. In the case where B-tree is almost empty, it only occupies one root page, and serving as leaf node. Then, there is no bottom constraint for the root node, but it should not break the page size rule.

Table 1 Math property of B-tree nodes

Node Type	**Children Type**	**Min Number of Children**	**Max Number of Children**	**Example b = 9**	**Example b = 100**
Root (Only one node)	Records	1	b-1	1-8	1-99
Root	Internal or Leaf Nodes	2	b	2-9	2-100
Internal Node	Internal or Leaf Nodes	[b/2]	b	5-9	50-100
Leaf Node	Records	[b/2]-1	b-1	4-8	49-99

Here is some other characteristics of B-tree with an order of b, and height of h from the root.

Table 2 B-tree extremes and time complexities

Maximum number of records	$b^h - b^{h-1}$
Minimum number of records	$2[\mathrm{b}/2]^{h-1} - 2[b/2]^{h-2}$
Minimum number of keys	$2[\mathrm{b}/2]^{h-1} - 1$
Maximum number of keys	$b^h - 1$
Space complexity	$\mathrm{O(n)}$
Time complexity of inserting one record	$\mathrm{O}(log_b \mathrm{n})$
Time complexity of searching one record	$\mathrm{O}(log_b \mathrm{n})$
Time complexity of removing one record	$\mathrm{O}(log_b \mathrm{n})$
Time complexity of range query for k elements	$\mathrm{O}(log_b \mathrm{n} + \mathrm{k})$

CHAPTER 2. BACKGROUND

CyDIW is abbreviation of Cyclone Database Implementation Workbench [13, 14], which can be used to implement new database prototypes, using existing command-based system, and conduct experiments. As CyDIW serves as a database workbench, the demand of B-tree utility is very natural, just like any other common database system that requires a better performance on data indexing and storing. Previous researchers and programmers have proposed and published many B-tree implementations. However, due to their incompatibilities to our workbench, we can not directly apply those elegant implementations onto our system. Therefore, we started to think about designing and implementing a CyDIW based B-tree utility for our users.

CyDIW Storage System

As we planned to build a series of B-tree APIs that serves CyDIW as an utility. We need to apply CyDIW storage system to be our implementation infrastructure, especially for page allocation, deallocation, read and write.

CyDIW storage system is fully user configurable, and it is self maintained via a catalog file. Here is an example of CyDIW storage configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<RawStorageParameters>
        <SmallestRawGranule Size="16" Unit="MBytes"/>
        <JavaRandomAccessFileSize NofRawGranules="1"/>
        <StoragePaths Unit="MBytes">
                <Path Location="H:/CanStoreXStorage/300M" Size="300" Volume="A"/>
        </StoragePaths>
        <PageSize Unit="KBytes" Value="16"/>
        <BufferNumber Value="6"/>
</RawStorageParameters>
```

This is an xml file for storage configuration. It indicates that under path "H:/CanStoreXStorage", we create a folder named 300M to store raw storage granules as our storage devices. Each granule has a size of 16 megabytes (the last one might be smaller than 16 MB). In this storage, the page size is set to be 16 kilobytes, and buffer number is 6.

In our B-tree implementation, we are using CyDIW storage. Here are some storage commands that we are using and their related APIs.

Create Storage

The create storage is done by a public boolean method in storagemanager. StorageUtils class. This method receives two parameters. The first one is the storage config which is original from the user storage configure file, and then a boolean tag which indicates to format the storage or not.

Load storage

Load storage is executed by a public method in storagemanager.StorageUtils, whose parameter only contain a storage config. The storage config is instantiated from the user's storage config file, just like what create storage did.

Allocate a page

The public method allocatePage in storagemanager.StorageManagerClient is the API for clients to allocate a page from the physical storage. And it will increase client's page allocated count by 1 for each execution.

Deallocate a page

The public method deallocatePage in storagemanager.StorageManagerClient is the API for clients to deallocate a page from the physical storage. And it will increase client's page deallocated count by 1 for each execution.

Read a page

In CyDIW environment, we apply a storage buffer for page I/Os. Page reading is done by the public method readPageWithoutPin in storagemanager. StorageManagerClient class. It will return a reference to byte array, which is pointing to a block in storage buffer. It is not sure whether page allocated count will be increased or not, but client page request count and client page request will be increased by 1 after each execution.

Write a page

Similar to page reading in CyDIW, page writing is done by the public method writePageWithoutPin in storagemanager. StorageManagerClient class. The first parameter is page id to be written, and the second parameter is the buffer pointer.

Storage catalog

CyDIW storage is maintaining a storage catalog file under the storage path folder. Each record inside the catalog file indicates one file in the storage. The record contains id, name, root page id, and size. In our B-tree utility implementation, we also take care of this catalog file, even though we already have our own configuration file to store critical informaiton, like root page id, sequence page id, which means there will be a duplicate information of B-tree's name and root page id.

CyDIW Client System

In this paper, we are not only show how we created and implemented B-tree utility based on CyDIW storage, but also represent how it was registered in CyDIW client system. The CyDIW platform consists of two major parts: CyDIW central services and various client systems. CyDIW central services include CyDIW user interface, CyDIW command parser, clients manager, clients interface, and storage services. In order to add new functionalities, each client needs to provide an adapter which can be used as a bridge between CyDIW platform and the client APIs.

All clients and their corresponding adapters information are located in SystemConfig.xml file under CyDIW root. Here is an example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<RegistrationRoot>
        <DefaultSystemRegistration>
                <Element Name="CyDIW" Prefix="CyDB"
                ClassPath=""
                LibraryPath="lib"
                WorkspacePath="CyWorkspace" />
        </DefaultSystemRegistration>
        <ExternalSystemRegistration>
                <Client Name="OS" Prefix="OS" Enabled="yes"
                        ClassPath="" LibraryPath=""
                        ClientAdapter="cysystem.clientsmanager.clients.OSAdapter" />
                <Client Name="MySQL" Prefix="$MySQL" Enabled="yes"
                        ClassPath=""
                        LibraryPath="cyclients\sql\drivers"
                        ClientAdapter="cyclients.sql.adapter.SQLClientAdapter"
                        DriverClass="com.mysql.jdbc.Driver"
                        ConnectionString="jdbc:mysql://csdb.cs.iastate.edu:3306/xfwangDB"
                        Username="xfwang" Password="xfwang-12" />
        </ExternalSystemRegistration>
</RegistrationRoot>
```

DefaultSystemRegistration

The <DefaultSystemRegistration> is where CyDIW main program get registered. The path can be modified by the user.

The type of each attribute in the "Element" element is defined as follows:

Name

"CyDIW" is a reserved keyword. The value of this attribute cannot be modified by the user.

Prefix

"CyDB" and "OS" are reserved prefixes in CyDIW. Any other clients could not use those two prefix keywords.

ClassPath

the relative path which contains CyDIW system class files. For instance, if the current path for CyDIW (set in the startDIW.bat file) is "D:\CyDIW_Root\", and the class files of

CyDIW reside at "D:\CyDIW_Root\SystemCalssPath\", then the value of the "ClassPath" attribute should be set as "SystemClassPath".

LibraryPath

the relative path to external library files of CyDIW system.

ExternalSystemRegistration

The <ExternalSystemRegistration> is where client systems being registered. A <Client> element is used to register each client. Only clients with the Enabled = "yes" is loded when CyDIW is started. No validity check is done for settings of clients with Enabled = "yes". The type of each attribute in the <Client> element is defined as follows:

Name

A string consisting of alphanumeric characters denotes the name of the client.

Prefix

A string consisting of alphanumeric characters denotes the prefix in commands for CyDIW parser. The prefixes must be unique accorss client systems for both lower can upper cases. For example, there will be a confliction if “mesql” and “MeSQL” both registered by the user.

Type "local": The client system is located in a local directory.

Type "remote": The client system is located in a remote location, such as a SQL database server. It also includes a username and password setting.

ClassPath

The absolute path which contains the class files of the client system.

LibraryPath

The absolute path which contains the library files of the client system.

ClientAdapter

The full class name including package information of the client adapter. The client adapter is responsible for certain commands which start with the prefix.

Enabled

Value of enabled "yes" means The current "Client" entry is enabled, and the client will be loaded. Otherwise, "no" means The current "Client" entry is disabled, and its information is ignored.

An example: CyDIW Client Registration for NC94

We have an example about an external client registration entry for NC94. After registering the following entry into the client system. The CyDIW system will dispatch all commands started with “$NC94:>” to the NC94Adapter.

```
<Client Name="NC94" Prefix="NC94" Enabled="yes"
        ClassPath="E:\workspace\CyDIW_v0.0"
        LibraryPath="cyclients\nc94\lib"
        WorkspacePath="cyclients\nc94\workspace"
        ClientAdapter="cyclients.nc94.commandparser.NC94Adapter" />
```

The NC94Adapter will be responsible for parsing the rest commands to its relating API to be executed. The first argument denotes the NC94 command name, and the rest of strings are parameters in the prompt commands. The NC94Adapter will call corresponding API in NC94 package.

B-tree Utility Registration

Our B-tree utility is part of Cy utilities, so we put all the code, configurations and data under cyutils folder, and name an CyUtilsAdapter to handle all commands starting with “$CyUtils:>”. The CyUtils client registration are given as follows:

```
<Client Name="CyUtils" Prefix="CyUtils" Enabled="yes"
        ClassPath=""
        LibraryPath=""
```

```
WorkspacePath=""
ClientAdapter="cyutils.parser.CyUtilsAdapter" />
```

As we can see from this registration entry, the ClientAdapter is CyDIW_Root/cyutils/parser/CyUtilsAdapter. It will be able to enable all commands like "$CyUtils:> some commands" to be executed. Also, CyUtilsAdapter class will call B-tree API during the execution of B-tree commands.

User Configuration

As we can see from the previous introduction to CyDIW platform, its storage and client system are fully user configurable. And in our implementation of B-tree utility, we also want to keep everything to be user configurable.

A user configurable tuple

The first thing came to our mind is the tuple definition in a relational database. A user configurable tuple definition is critical for database utilities implementation. Otherwise, user will have to change some setting that hardcoded in the implementation program, which does not sound attractive to users. Previous B-tree utility on CyDIW ignored this, and some of them used some static variables in class files to indicate tuple definition. However, it still required users to change the code if they want to configure tuples by their own.

In the following paragraphs, we are going to show our configurable tuple design and implementation. We believe that every tuple contains several elements, which could be a string or an integer. For an integer, we know that its corresponding bytes length is 4, but for a string, we need to specify the maximal length of the string, thus we know its corresponding bytes length is the string maximal length. As we know each element's length, we can easily calculate the entire tuple length by adding up them. The next thing matters is the order of those elements. The solution for this issue, is adding an attribute name id to each element.

Also, we need a name attribute, so that the user will have a better identification and understanding to all elements. The last thing that we want to take care about, is the primary key of this tuple. For example, suppose we have the tuple being composed of A, B, C, D, four elements, and the key is C, D, A. Our solution is to add an another attribute named key order to each element. Key orders start from 1, and will increase by 1 for the next element, the key order for non-key elements are -1. So, in the given example, A.keyorder = 3, B.keyorder = -1, C.keyorder = 1, D.keyorder = 2. In this example, let us complete other attributes for A, B, C, D, and write a configuration xml file as a conclusion. We suppose:

A is employee id, an integer, length is 4

B is department name, a string, length is 5

C is employee name, a string, length is 10

D is employee salary, an integer, length is 4

So the tuple configuration file looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<TupleConfiguration Name="Emp">
        <Attribute id="1" name="Id" type="Integer" length="4" keyorder="3"></Attribute>
        <Attribute id="2" name="DName" type="String" length="5" keyorder="-
1"></Attribute>
        <Attribute id="3" name="Name" type="String" length="10"
keyorder="1"></Attribute>
        <Attribute id="4" name="Salary" type="Integer" length="4"
keyorder="2"></Attribute>
</TupleConfiguration>
```

Our user configurable tuple definition is general, and can be applied to some other utilities, like linear hashing, internal sorting and so on. Details about how tuple class is being created and tuple comparison methods, will be introduced in the next chapter.

A user configurable B-tree

Similar to the tuple definition, we also want B-tree to be a user configurable concepts. Since B-tree is required to be persistent in storage, we need to keep track of B-tree's name, root page id, sequence page id. Also, for demoing a depth over 5 with less data, we should add a concept of used page size of B-tree nodes. At last, in order to track of how many pages being allocated for B-tree, we can have an element called number of pages. Here is the B-tree configuration file looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<BTreeConfiguration Name="BTreeEmpConfig">
        <RootPageId>-1</RootPageId>
        <SequencePageId>-1</SequencePageId>
        <BTreeName>BTreeEmp</BTreeName>
        <PageSizeUsed>256</PageSizeUsed>
        <NumberOfPages>0</NumberOfPages>
</BTreeConfiguration>
```

There are two modes in B-tree's initialization, the first one is initialization of an empty B-tree, in this case, we are only using BTreeName and PageSizeUsed information in the B-tree configuration file. The second one is initialization of an existed B-tree, therefore, we are all elements information from this configuration file.

CHAPTER 3. IMPLEMENTATION DETAILS

CyUtils B-tree Commands

As we already introduced how to register CyUtils commands in CyDIW system configuration. Now, we want to illustrate how those commands being parsed and processed. In order to recognize and handle all commands starting with the prefix "CyUtils", we need a specific adapter class and make an incremental compilation into the CyDIW system. In CyUtilsAdapter class, the constructor is receiving CyDIW gui instance as a parameter. Since we need to use CyDIW storage, so we make a method call of CyDIW gui in order to get the storageutils instance. By doing so, we are connected with CyDIW storage, and could interact with CyDIW GUI panel.

In the adapter class, we are separating commands into a string list by spaces. The first parameter in the string list denotes what utility function will be called. For example, in CyDIW panel, if we typed:

```
$CyUtils:> CreateBTreeEmpty <BTreeConfigFile> <TupleConfigFile>;
```

In this case, the first parameter is CreateBTreeEmpty, and the B-tree constructor will be called, then the newly created B-tree will be initialized as an empty B-tree. After all creating and initializing are done, the store B-tree information methods will be called, and B-tree's information will be backed up into BTreeConfigFile and storage catalog file.

In CyUtils, we have designed the following commands, majorly for B-tree utilities:

List commands

```
$CyUtils:> list commands;
```

List all commands for all CyUtils commands.

BTreeCreateEmpty

$CyUtils:> BTreeCreateEmpty <BTreeConfigXmlFile> <TupleConfigXmlFile>;

Create and initialize an empty B-tree. The rest two parameters are B-tree configuration file and tuple configuration file.

BTreeShowRootAndSequencePageId

$CyUtils:> BTreeShowRootAndSequencePageId <BTreeConfigXmlFile>;

Print an existed B-tree's root page id and sequence page id.

BTreePrepareSortedData

$CyUtils:> BTreePrepareSortedData <TupleConfigXmlFile> <TupleTxtFile>;

Prepare sorted tuples according to tuple configuration file. All data will be stored in the tuple txt file. The prepared data will be used in B-treeBulkLoad

BTreeBulkLoad

$CyUtils:> BTreeBulkLoad <BTreeConfigXmlFile> <TupleConfigXmlFile> <TupleTxtFile>;

Perform bulk loading for an empty B-tree. In order to bulk load sorted data into a B-tree. Frist, we need to create and initialize an empty B-tree according to the tuple configuration and B-tree configuration files. Then load data from tuple txt file by our data reading iterator. This iterator will produce a byte array for each tuple. Our B-tree bulk loading method will take the iterator, and load all tuples into the B-tree eventually.

BTreeInsert

$CyUtils:> BTreeInsert <BTreeConfigXmlFile> <TupleConfigXmlFile> [OneTuple];

B-tree insertion is conducted by this command in our CyDIW environment. In order to successfully insert one tuple into our B-tree. We need to offer B-tree configuration file and tuple configuration file, and load the existing B-tree according to those configuration files.

CyUtilsAdapter class will parse [OneTuple] to be a string array, then convert the string array to be a byte array which represents the tuple in a binary format. Eventually, the tuple (binary array) will be inserted through B-tree class's insertion method.

BTreeSequenceScan

$CyUtils:> BTreeSequenceScan <BTreeConfigXmlFile> <TupleConfigXmlFile>;

In some cases, we want to take a quick look at B-tree data in text format. For example, we insert several unsorted tuples into the B-tree, and want to examine whether B-tree is organized in a sorted order. To use B-tree sequential scan, we need to add B-tree configuration and tuple configuration files, so that we can load the existed B-tree by its configuration, then sequentially visit each page, and print tuples in a text format.

BTreeSecondaryBulkLoad

$CyUtils:> BTreeSecondaryBulkLoad <BTreeConfigXmlFile> <TupleConfigXmlFile> <SecondaryBTreeConfigXmlFile> <SecondaryTupleConfigXmlFile>;

We assume that index and sequence nodes in a secondary B-tree use the same size pages, and the secondary B-tree is built from the primary B-tree, with a given secondary tuple configuration and B-tree configuration. The BTreeSecondaryBulkLoad command invokes the primary B-tree to produce records iteratively, then transform them according to the secondary tuple configuration, with an origin page pointer, which will be eventually loaded into the secondary B-tree.

BTreeSecondarySequenceScan

$CyUtils:> BTreeSecondarySequenceScan <BTreeConfigXmlFile> <TupleConfigXmlFile> <SecondaryBTreeConfigXmlFile> <SecondaryTupleConfigXmlFile>;

In order to examine the correctness of the secondary B-tree, we can iteratively visit every tuple (with the page pointer), then search the primary tuple with the given secondary key inside the specific page (by the page pointer). Those primary tuples will be printed after being transformed from binary format to text format.

BTreeDelete

$CyUtils:> BTreeDelete <BTreeConfigXmlFile> <TupleConfigXmlFile>;

Sometimes we want to delete the entire B-tree to free some space for the future usage. The BTreeDelete command calls the delete method which will go through all the index and sequence nodes from top to bottom, and de-allocate all those nodes. The BTreeDelete should de-allocate exactly the same number of pages as the B-tree has.

Tuple Compare

$CyUtils:> TupleCompare [TupleOne] [TupleTwo] <TupleConfigXmlFile>;

As we mentioned previously, our tuple definition is fully configurable. In our implementation, we allow Tuple class to do the comparison on the flyer, which means all comparison with tuples inside the storage, will be conducted in the buffer memory. It significantly saves time and space wasted in generating keys from tuples.

Tuple Comparison

The logic of tuple comparison on the flyer is to create a key bytes array, and use it as an index when we are trying to compare two tuples. Let us take the tuple configuration in chapter 2, and explain our tuple comparison algorithm based on this example.

```
<?xml version="1.0" encoding="UTF-8"?>
<TupleConfiguration Name="Emp">
        <Attribute id="1" name="Id" type="Integer" length="4" keyorder="3"></Attribute>
        <Attribute id="2" name="DName" type="String" length="5" keyorder="-
1"></Attribute>
        <Attribute id="3" name="Name" type="String" length="10"
keyorder="1"></Attribute>
        <Attribute id="4" name="Salary" type="Integer" length="4"
keyorder="2"></Attribute>
</TupleConfiguration>
```

According to this configuration file, the tuple is composed of [A, B, C, D], where A takes 4 bytes, B takes 5 bytes, C takes 10 bytes, and D takes 4 bytes. So we could easily know that: bytes[0-3] represents A, bytes[4-8] represents B, bytes[9-18] represents C, and bytes[19-22] represents D. Consider the key [C, D, A], it actually is bytes[9-18], followed by bytes[19-22], followed by bytes[0-3].

If we using an array (key importance array) to indicate the importance of each byte in the key definition, then we will have:

[14,15,16,17, -1,-1,-1,-1,-1, 0,1,2,3,4,5,6,7,8,9, 10,11,12,13]

Here, -1 means non-key bytes, 0 means the most import byte, 1 means the second import byte, and so on. We can see, The first ten import bytes are actually attribute C, then attribute D, then attribute A. If we convert the key importance array to the keyr order array, we will have:

[9,10,11,12,13,14,15,16,17,18,19,20,21,22,0,1,2,3,-1,-1,-1,-1,-1]

the first element 9 means the most import byte is the 9th one, then 10th, and so on. If we encounter -1, that means there is no key bytes any more.

In tuple comparison, the key order serves like a ruler. In this example, each time when we want to compare two tuples, we first check byte 9, then byte 10, then byte 11, and so on. If at somewhere, two bytes are not equal, we could immediately determine what to output. If we encounter a -1 in key order array (the ruler), then we simply return 0, since two tuples are equal from the key perspectives.

B-tree parsing and storing

B-tree is configurable and persistent in our storage, since we are using both storage catalog file and B-tree configuration file to store critical information, such as root page id, sequence page id, B-tree name and so on. Compare to tuple definition, our B-tree parsing and storing are pretty simple and straightforward. We created a class to be responsible for information parsing and storing, and defined two modes for initializations of B-tree.

The first mode is initialization of an empty B-tree. Under this mode, we first create an empty B-tree, with a system allocated root page id (it is also a sequence page id). Then we load B-tree name and pageSizeUsed information from parsing the B-treeConfiguration file. After everything has been done, we update both B-treeConfiguration file and storage catalog file.

The second mode is initialization of an existed B-tree. Under this mode, we first create an empty B-tree without root page or sequence page information. After that, we load B-tree name, page id, numberOfPages and pageSizeUsed information from parsing the B-treeConfiguration file. After current command session is done, we update both B-treeConfiguration file and storage catalog file.

B-tree Insertion

Our B-tree insertion algorithm starts with two parameters, the first one is B-tree root page id, the second one is the byte tuple. We are using B-tree search algorithm to find an insertion point, and it will return us a parent path. There are three scenarios might occur:

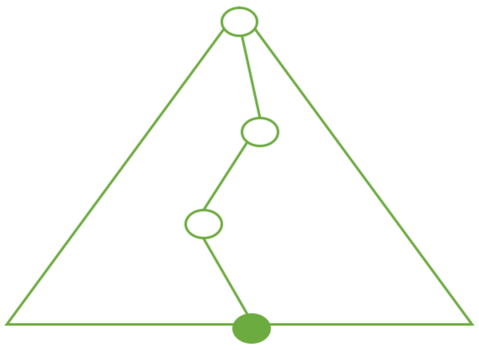

Figure 4 B-tree insertion, leaf node is not full

The best case we could encounter is the leaf node is not full at the bottom, we can directly insert the given tuple into the leaf node.

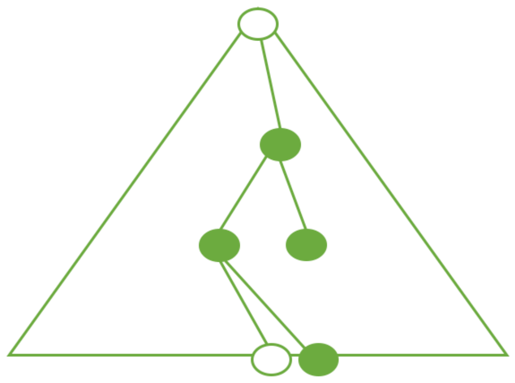

Figure 5 B-tree insertion, recursively split and insert bottom-up

Another scenario is the leaf node is not full, so that we need to split and insert bottom-up, until one internal index node is not full. During the split procedure, we divide current node into two halves, and create a new node to store the right half. A key-pointer pair will be generated after splitting. The key in the mid of the previous node, the left pointer is the previous node id, and the right pointer is the newly created node id. After current level splitting is done, insert this key-pointer pair into the upper level along the parent path.

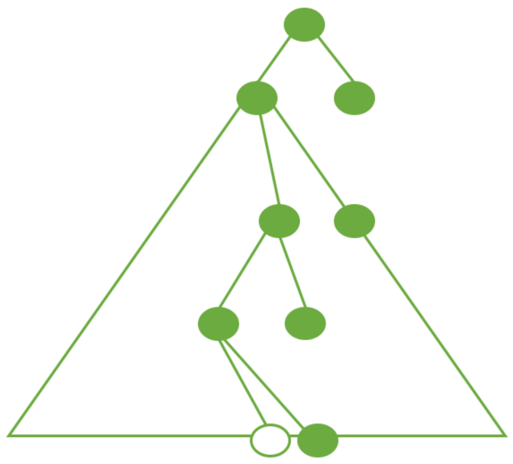

Figure 6 B-tree insertion, all nodes in the insertion way are full, create a new root node

Under the worst case of B-tree insertion, all nodes along the parent path are full, so that we have to keep splitting and inserting along the way, and create a new root node eventually. In this case, times of splitting equal to the previous height of the tree, say h, and it requires 2*h, and data transferring during splitting is linear against page size used at each level.

Full details about B-tree insertion are in the algorithm pseudo code below.

B-tree insertion algorithm pseudo code

```
B-treeInsert(b, x){
        [parent_stack] = B-treeSearch(b, x)
        pageId(leaf) = parent_stack.pop()
        offset = bisect_leaf(pageId, x)
        if(!isPageFull(pageId)){
                addOneTupleToSortedLeafPage(pageId, offset, x)
        }else{
                //consider the leaf node split
                take pageId[:offset]+x+pageId[offset:] as a whole sequence WS
                in WS find mid
                pageId <- WS[:mid+1]
                new_pageId <- WS[mid+1:]
                (leftPtr, key, rightPtr) <- (pageId, midTuple, new_pageId)
                while(parent_stack != NULL){
                        pageId(index) = parent_stack.pop()
                        offset = bisect_index(index, key/midTuple)
                        if(!isPageFull(pageId)){
                                addOneIndexToSortedIndexPage(pageId, offset, key, rightPtr)
                                return true //call a end of B-treeInsert, since everything has
been completed!
                        }else{
                                //need to split index page
                                take pageId[:offset] + key + pageId[offset:] as a whole index
WI
                                in WI find mid
                                pageId <- WI[:mid+1]
                                new_pageId <- WI[mid+1:]
                                (leftPtr, key, rightPtr) <- (pageId, midTuple, new_pageId)
                        }
                } // end while
                // split root b(), make a new root b' to replace the original b, then return true
                take pageId[:offset] + key + pageId[offset:] as a whole index WI
                in WI find mid
                pageId <- WI[:mid+1]
                new_pageId <- WI[mid+1:]
                (leftPtr, key, rightPtr) <- (pageId, midTuple, new_pageId)
                b' = createNewIndex()
                set b' to be B-tree's new root page id
                addOneIndexToEmptyIndexPage(b', leftPtr, key, rightPtr)
                return true
        }
        return false
}
```

B-tree Bulk Loading

Bulk loading is used when we have an empty B-tree and a sorted dataset, we can load all those tuples in an efficient way comparing with insert them sequentially. The major difference between insertion and bulk loading is how tuples get added into the B-tree. In B-tree insertion, a tuple is inserted in somewhere in the leaf. However, in B-tree bulk loading, since all tuples are sorted according to the primary key, we can keep track of all appending position at each level of B-tree. If current level node is not full, we directly append data at the place, otherwise, we create a new node and add the data, then keep appending on the upper level until one internal index node is not full, or a new root node is being created.

This is called B-tree bulk loading in a pyramid way. In the loading procedure of each tuple, there are also three major scenarios we need to consider.

1. The leaf node is not full. We can directly append tuple into it.

2. The leaf node is full. We should keep creating new node and appending until one internal index node is not full.

3. Under the worst case, the root node is also full. We need to create a new root node, and insert key-pointer pair into it.

Here is a figure of showing how B-tree bulk loading is done.

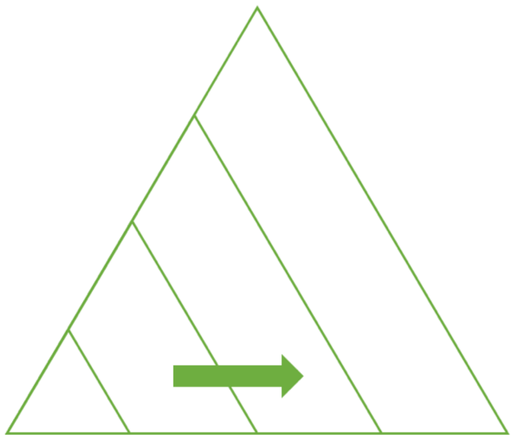

Figure 7 B-tree bulk loading in a pyramid style

The following algorithm pseudo code shows up more details about B-tree bulk loading.

B-tree bulk loading algorithm pseudo code

```
B-treebulkloading() {
    stk += [root, page_header_size],
    for (x : X) {
        pageId, offset = stk.pop()
        if (offset + len(x) < sizeof(page)) {
            // Leaf node is not full, can directly insert tuple into leaf page
            addToLeaf(pageId, offset, x)
            offset += len(x)
            stk += [pageId, offset],
        } else {
            new_leafPageId <- [x]
            (leftPtr, key, rightPtr) <- (pageId, generateKey(x), new_leafPageId)
            tmp_stk = [new_leafPageId, page_header_size+len(x)]
            while (stk != NULL) {
                indexId, offset = stk.pop()
                if (indexId not full) {
                    addToIndex(indexId, offset, key, rightPtr)
                    offset += len(key) + len(rightPtr)
                    tmp_stk += [indexId, offset],
                    while tmp_stk:
                        stk += tmp_stk.pop(),
                    continue // go to next element x
                } else {
                    // add a new index
```

```
                                        new_indexId <- (leftPtr, key, rightPtr)
                                        (leftPtr, key, rightPtr) <- (indexId, key,new_indexId)
                                        tmp_stk += [new_indexId,
default_offset+len(key+2*ptr)]
                                }
                        }
                        // add a new root
                        new_indexId <- (leftPtr, key, rightPtr)
                        set new_indexId to be B-tree's new root page id
                        tmp_stk += [new_indexId, offset+len(key+2*ptr)],
                        while tmp_stk:
                                stk += tmp_stk.pop(),
                }
        }
}
```

PTC Framework

In order to make a clear division on data operations between user and client, a two ended iterator-based framework called PTC, short for produce, transform, consume, is offered that produces one record at a time, the record is optionally processed, and then consumed.

PTC framework allows everyone works individually under a clear division, and they can collaborate with each other to perform the whole data operation process. More specifically, P concentrates on producing records iteratively on some given datasets. T plays a role on each record's operations, such as selection and projection. And C works as a consumer which can iteratively take the transformed records from T, and consume them to perform the entire process. For example, in database queries, we usually encounter some operations that require to print projected data on some attributes for a dataset. It can be translated into a producer iteratively produce data from the existing dataset, then the transformer makes a projection on each produced data, finally the consumer take each transformed data into next operation, here it is printing.

Our B-tree implementation is following PTC framework, here are two examples about PTC usage of B-tree on both sides: bulk loading and sequential printing. In our B-tree bulk loading demos, users are only concern how dataset is given, they just need to provide the P iterator that can produce text record from the given dataset. T is responsible for transforming text record to binary format. Eventually, our B-tree servers as the C side, which iteratively takes sorted binary records and appending them into the B-tree. In sequential printing, our B-tree is the P side, the P iterator produces binary records, and T transforms a binary record to text, which is then consumed by C, that simply prints it.

CHAPTER 4. EXPERIMENTS

Experiment Settings

Our B-tree experiments are organized in a demos folder. The relative path is CyDIW_root/cyutils/B-tree/demos. All user configuration and data files are organized in a workspace folder. The relative path is CyDIW_root/cyutils/B-tree/workspace. Experiments are loaded into CyDIW commands panel, and runned with CyDIW GUI.

Tuple Comparison

In this experiment, we want to examine the functionality of our tuple class and general tuple comparison idea. Here, in the first sub-experiment, we keep using the same tuple configuration as what we showed in chapter 2 and chapter 3. In the second sub-experiment we change the key order settings to test the configurability of our general tuple definition.

First sub-experiment:

According to the following tuple configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<TupleConfiguration Name="Emp">
        <Attribute id="1" name="Id" type="Integer" length="4" keyorder="3"></Attribute>
        <Attribute id="2" name="DName" type="String" length="5" keyorder="-
1"></Attribute>
        <Attribute id="3" name="Name" type="String" length="22"
keyorder="1"></Attribute>
        <Attribute id="4" name="Salary" type="Integer" length="4"
keyorder="2"></Attribute>
</TupleConfiguration>
```

We are trying to compare the relationship between two tuples:

[1, EE, Jack, 80000] and [2, COMS, Tom, 90000]

As we known, in this tuple configuration, the key is Name, followed by Salary, then followed by Id. So [1, COMS, Jack, 80000] is apparently smaller than [2, EE, Tom, 90000].

The commands output should be -1 which represents the first tuple is smaller than the second tuple.

Afterwards, we try to switch two tuples, make [2, EE, Tom, 90000] to be the first tuple, and [1, COMS, Jack, 80000] to be the second tuple. Then the commands output should be 1 which represents the first tuple is larger than the second tuple.

Then we compare two tuples that are exactly the same on key attributes. The comparison results for [1, EE, Jack, 80000] and [1, COMS, Jack, 80000] should be 0 since they are equal from the key perspective.

After that, we want to modify the key order of tuple configuration a bit. As showed in the following, we make the key to be DNAME, followed by Salary, then followed by Id.

Then the relationship between those two tuples: [1, EE, Jack, 80000] and [2, COMS, Tom, 90000]. We could easily know that the second one is smaller than the first one. So the commands output should be since [1, EE, Jack, 80000] is larger than [2, COMS, Tom, 90000]. Then we switch the first tuple and the second tuple, our expecting output is -1. Finally we compare two modified tuples where they have the same key, we should get 0 from the CyDIW output panel.

Comparison Experiment Commands:

```
$CyUtils:> TupleCompare TupleConfig.xml [1, EE, Jack, 80000] [2, COMS, Tom, 90000];
Output is 1
$CyUtils:> TupleCompare TupleConfig.xml [2, COMS, Tom, 90000] [1, EE, Jack, 80000];
Output is -1
$CyUtils:> TupleCompare TupleConfig.xml [1, EE, Jack, 80000] [1, COMS, Jack, 80000];
Output is 0
```

B-tree Insertion

In this experiment, we are going to create an empty B-tree, and insert several unsorted tuples into it. Before end this session, we call a sequential scan to print all tuples in text format onto the console. The output should be in an increasing order since B-tree can organize them to be sorted. Here are the tuple configuration and B-tree configuration we are using in this experiment.

Insertion Experiment Tuple Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<TupleConfiguration Name="Emp">
        <Attribute id="1" name="Id" type="Integer" length="4" keyorder="1"></Attribute>
        <Attribute id="2" name="Name" type="String" length="22" keyorder="-
1"></Attribute>
        <Attribute id="3" name="DName" type="String" length="5" keyorder="-
1"></Attribute>
        <Attribute id="4" name="Salary" type="Integer" length="4" keyorder="-
1"></Attribute>
</TupleConfiguration>
```

Insertion Experiment B-tree Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<BTreeConfiguration Name="BTreeConfigInsertion">
        <RootPageId>-1</RootPageId>
        <SequencePageId>-1</SequencePageId>
        <BTreeName>BTreeInsertion</BTreeName>
        <PageSizeUsed>256</PageSizeUsed>
        <NumberOfPages>0</NumberOfPages>
</BTreeConfiguration>
```

Insertion Experiment Commands:

```
$CyUtils:> BTreeCreateEmpty BTreeConfig.xml TupleConfig.xml
```

Output: Create an empty B-tree according to two user configuration files, and store the critical information of B-tree in BTreeConfig.xml. Deliver a message “Successfully create an empty B-tree” if no error prompt, otherwise deliver the error information.

```
$CyUtils:> BTreeInsert BTreeConfig.xml TupleConfig.xml [1, EE, Jack, 80000]
```

Output: Convert the tuple from text format to binary format, then insert the binary tuple into B-tree, update BTreeConfig.xml if necessary. Deliver a message "Successfully insert a tuple into B-tree" if no error prompt, otherwise deliver the error information.

```
$CyUtils:> BTreeSequenceScan BTreeConfig.xml TupleConfig.xml
```

Output: Sequentially scan all B-tree leaf nodes, and print them onto console in text format, so that we could see if our B-tree could manage unsorted insertion data in a sorted order.

Tuples to be inserted:

```
[2, Michael, COMS, 100000]
[3, Jack, EE, 90000]
[7, Helen, PHYS, 80000]
[8, Jeff, CHEM, 90000]
[1, Kate, ENGL, 60000]
[6, Kevin, ALG, 70000]
[4, Osborn, ME, 80000]
[5, Jason, SPORT, 120000]
```

Sequentially scan output is:

```
Id: 1, Name: Kate, DName: ENGL, Salary: 60000
Id: 2, Name: Michael, DName: COMS, Salary: 100000
Id: 3, Name: Jack, DName: EE, Salary: 90000
Id: 4, Name: Osborn, DName: ME, Salary: 80000
Id: 5, Name: Jason, DName: SPORT, Salary: 120000
Id: 6, Name: Kevin, DName: ALG, Salary: 70000
Id: 7, Name: Helen, DName: PHYS, Salary: 80000
Id: 8, Name: Jeff, DName: CHEM, Salary: 90000
```

B-tree Bulk Loading

In this experiment, we are going to generate 10000 sorted tuples, and save them into a txt file in the workspace. Afterwards, we create an empty B-tree according to our tuple and B-tree configurations, then bulk load all tuples in the data file to our B-tree in the storage.

The bulk loading process are divided into two parts. First, we use a data generator to generate a sequence of sorted tuples to txt files in workspace folder. After data generation has been done, we can open an iterator which can read the data file and produce a byte tuple each time. Then, the B-tree bulkload method will take the iterator as the parameter, and iteratively load all tuples in the data file. A sequential scan can be called to test the correctness after bulk loading is done.

Bulk Loading Experiment Commands:

```
$CyUtils:> BTreeCreateEmpty BTreeConfigBulk.xml TupleConfigBulk.xml;
```

Output: Create an empty B-tree, message is delivered accordingly.

```
$CyUtils:> BTreePrepareSortedData TupleConfigBulk.xml Tuples.txt;
```

Output: Generate a sorted dataset according to the given tuple configuration, and store the data in Tuples.txt file.

```
$CyUtils:> BTreeBulkLoad BTreeConfigBulk.xml TupleConfigBulk.xml Tuples.txt;
```

Output: A binary tuple generator will read Tuples.txt file line by line, and prompt binary arrays (binary tuple) iteratively. B-tree bulkload method takes the generator, and iteratively load all binary tuples into newly created B-tree. Deliver error messages if necessary.

```
$CyDB:> getPageAllocatedCount;
```

Output: get page allocated count for the B-tree utility client.

PageSizeUsed in B-tree bulk loading:

In B-tree bulk loading experiment, we used page size used property, in order to get a small fan-out, so that we could have a large height without huge dataset. In the first bulk loading experiment, we set PageSizeUsed to be 256 bytes. It can help us get a B-tree of height 4 with 10000 tuples. We also conduct another bulk loading experiment with full page size, that one can give us a B-tree of height 2 with 10000 tuples when full page size is equal to 16 KB. The experiment where PageSizeUsed equals to 256, is using the same B-tree configuration. The bulk loading experiment of full page size is using the following B-tree configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<BTreeConfiguration Name="BTreeConfigBulkLoad">
        <RootPageId>-1</RootPageId>
        <SequencePageId>-1</SequencePageId>
        <BTreeName>BTreeBulkLoad</BTreeName>
        <PageSizeUsed>16384</PageSizeUsed>
        <NumberOfPages>0</NumberOfPages>
</BTreeConfiguration>
```

16KB size of page can significantly increases the fan-out of each node, and decrease the page allocated for loading the same data

Secondary B-tree

Tuples inside the secondary B-tree consist of two parts: the secondary key, and the page pointer where the primary tuple (denoted by the secondary key) exists. Ideally, the user should give an independent secondary key which can also uniquely identify full records in the primary B-tree. But here, we are using the same key of the primary B-tree as a secondary index. Configurations of the primary B-tree and tuple, and the secondary B-tree configuration, are similar to the previous insertion and bulk loading (PageSizeUsed equals to

256) experiment, but with a different BTreeName to differentiate with the previous created B-trees. The secondary tuple configuration is given as the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<SecondaryTupleConfiguration Name="Secondary">
        <Attribute id="1" name="Id" type="Integer" length="4" keyorder="1"></Attribute>
        <Attribute id="2" name="PagePointer" type="Integer" length="4" keyorder="-
1"></Attribute>
</SecondaryTupleConfiguration>
```

We divide this experiment into two major parts, in the first one, we create and load a secondary B-tree, then sequential scan the secondary B-tree. The creation of the secondary B-tree is just like normal B-tree creation, the system will allocate a new root page, and update with the secondary B-tree configuration file. Our secondary B-tree loading invokes the primary B-tree as the producer, which can iteratively produce records in a binary format. The transformer converts a full record into the secondary key format with a page pointer (which is an integer). Finally, those converted bytes are consumed by the newly created secondary B-tree via C. In the secondary B-tree sequential scan experiment, the SecondaryBTreeSequenceScan command calls secondary B-tree as an iterator to produce secondary key and page pointer iteratively, then the transformer will search the secondary key in the given page pointer, and convert a full record into text format sequentially, then consume by C, which simply prints it.

Secondary B-tree Creation and Loading Commands:

$CyUtils:> BTreeCreateEmpty SecondaryBTreeConfig.xml SecondaryTupleConfig.xml;

Output: Create an empty B-tree, which is a secondary index of an existing dataset.

$CyUtils:> BTreeSecondaryBulkLoad BTreeConfig.xml TupleConfig.xml SecondaryBTreeConfig.xml SecondaryTupleConfig.xml;

Output: Produce full records iteratively from the primary B-tree, bulk load secondary tuples and page pointers into the secondary B-tree.

Secondary B-tree Sequence Scan Commands:

```
$CyUtils:> BTreeSecondarySequenceScan BTreeConfig.xml TupleConfig.xml
SecondaryBTreeConfig.xml SecondaryTupleConfig.xml;
```

Output: Iteratively produce secondary keys with page pointers, search full tuples in the page pointers with the corresponding secondary keys. Simply convert the full tuples to text format and print them.

REFERENCES

[1] LionKimbro https://en.wikipedia.org/wiki/Cylinder-head-sector

[2] Kent, Karen, et al. "Guide to integrating forensic techniques into incident response." *NIST Special Publication* 10 (2006): 800-86.

[3] Sahni, Priyanka, and Abhinav Batra. "Network File System." *International Journal of Research* 2.4 (2015): 894-896.

[4] Karlsson, Johan, and Magnus Luu. "Server Message Block."

[5] Ligon III, Walter B., and Robert B. Ross. "An overview of the parallel virtual file system." *Proceedings of the 1999 Extreme Linux Workshop*. 1999.

[6] Özsu, M. Tamer, and Patrick Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[7]. Codd, Edgar F. "A relational model of data for large shared data banks." *Communications of the ACM* 13.6 (1970): 377-387.

[8] Arpaci-Dusseau, Remzi H., and Andrea C. Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books, 2015.

[9] Berkeley, D. B. "Berkeley DB." (1968).

[10]. Berliner, Hans. "The B* tree search algorithm: A best-first proof procedure." *Artificial Intelligence* 12.1 (1979): 23-40.

[11] Lomet, David B. "Digital B-trees." *Proceedings of the seventh international conference on Very Large Data Bases-Volume 7*. VLDB Endowment, 1981.

[12] Bayer, Rudolf, and Edward McCreight. "Organization and maintenance of large ordered indexes." *Software pioneers*. Springer Berlin Heidelberg, 2002. 245-262.

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

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

APPENDIX A. B-TREE UTILITIES API

TupleAttribute

```
public class TupleAttribute{
	private String name;
	private String type;
	private int length;
	private int keyOrder;

	public void setName();
	public void setType();
	public void setLength();
	public void setKeyOrder();
	public String getName();
	public String getType();
	public int getLength();
	public int getKeyOrder();
}
```

Tuple

```
public class Tuple {
	private int length;
	private int keyLength;
	private int[] keyBytes;
	private int attributeNum;
	private List<TupleAttribute> tupleAttributes;

	// Constructor
	public Tuple(String tupleConfigXmlFile);

	// public methods
	public int getLength();
	public int getKeyLength();
	public int getAttributeNum();
	public List<TupleAttribute> getTupleAttributes();
	public byte[] generateKey(byte[] record);
	public byte[] geberateKey(byte[] buffer, int offset);
	public int compare(byte[] one, byte[] two);
	public int compare(byte[] buffer, byte[] record, int offset);
	public int compareKey(byte[] buffer, byte[] record, int offset);
}
```

B-tree

```
public class BTree {
        private String name;
        private StorageUtils stoUtils;
        private StorageManagerClient stoMgrClient;
        private StorageDirectoryDocument xmlParser;
        private int rootPageId = -1;
        private int sequencePageId = -1;
        private int pageSizeused = 256;
        private Tuple tupleDefinition;
        public StorageUtils getStorageUtils();
        public StorageManagerClient getStorageManagerClient();
        public int getRootPageId();
        public int getSequencePageId();
        public getName();
        public getTupleDefinition();
        public setRootPageId(int rootPageId);
        public setSequencePageId(int sequencePageId);
        // constructor
        public BTree(StorageUtils stoUtils, Tuple tupleDefinition);
        // other methods
        public void initializeBTreeFromEmpty(String BTreeConfigXmlFile);
        public void initializeBTreeFromXml(String BTreeConfigXmlFile);
        public void storeB-treeToXml(String BTreeConfigXmlFile);
        public int bisect_leaf(int pageId, byte[] tuple);
        public int bisect_index(int pageId, byte[] key);
        public int bisect_pointer(int pageId, byte[] key);
        public void addOneTupleToSortedLeafPage(int pageId, int offset, byte[]
        tuple);
        public void addOneIndexToEmptyIndexPage(int pageId, int leftPtr, byte[] key,
        int rightPtr);
        public void addOneIndexToSortedIndexPage(int pageId, int offset, byte[] key,
        int rightPtr);
        public List<Integer> BTreeSearch(int rootpageId, byte[] record);
        public boolean isLeafPage(int pageId);
        public int createNewLeaf();
        public int createNewIndex();
        public boolean isPageFull(int pageId);
        private List<Object> splitLeafNode(int pageId, int offset, byte[] record);
        private List<Object> splitIndexNode(int pageId, int offset, byte[] key, int
        rightPtr);
        public void BTreeBulkLoad(Iterator iter);
        public void BTreeInsert(int rootPageId, byte[] record);
        public int printAllTuples(int pageId);
}
```

APPENDIX B.   COMMANDS

B-tree Insertion

```
/*
Step 0. Create and Load Storage
*/
$CyDB:> createStorage QuickStorageConfig.xml;
$CyDB:> loadStorage QuickStorageConfig.xml;

/*
Step 1. Dispaly configuration files
*/
$CyDB:> displayXML  .\cyutils\btree\workspace\BTreeConfig.xml;
$CyDB:> displayXML .\cyutils\btree\workspace\TupleConfig.xml;

/*
Step 2. Declare variables for btree and tuple configuration file names
*/
$CyDB:> declare string
$$BTreeConfigFileName := .\cyutils\btree\workspace\BTreeConfig.xml;
$CyDB:> declare string
$$TupleConfigFileName := .\cyutils\btree\workspace\TupleConfig.xml;
$CyDB:> list variables;

/*
Step 3. Lookup for pageAccesses
and list CyUtils commands
*/
$CyDB:> resetPageAccessRelativeCount;
$CyDB:> getPageAllocatedCount;
$CyDB:> getPageDeallocatedCount;
$CyUtils:> list commands;

/*
Step 4. Build an empty BTree, and show its root&sequence page Id
*/
$CyUtils:> BTreeCreateEmpty $$BTreeConfigFileName $$TupleConfigFileName;
$CyUtils:> BTreeShowRootAndSequencePageId $$BTreeConfigFileName;

/*
Step 5. Insert 4 tuples into storage, show its root&sequence page Id afterwards.
Then sequencially visit all leaf page, print tupels in console.
*/
$CyUtils:> BTreeInsert $$BTreeConfigFileName $$TupleConfigFileName
[2,Michael,COMS,100000];
```

```
$CyUtils:> BTreeInsert $$BTreeConfigFileName $$TupleConfigFileName
[3,Jack,EE,90000];
$CyUtils:> BTreeInsert $$BTreeConfigFileName $$TupleConfigFileName
[7,Helen,PHYS,80000];
$CyUtils:> BTreeInsert $$BTreeConfigFileName $$TupleConfigFileName
[8,Jeff,CHEM,90000];
$CyUtils:> BTreeShowRootAndSequencePageId $$BTreeConfigFileName;
$CyUtils:> BTreeSequenceScan $$BTreeConfigFileName $$TupleConfigFileName;

/*
Step 6. Insert another 4 tuples into storage, show its root&sequence page Id,
and print tuples in console afterwards.
This experiment is on purpose of showing btree page splitting ability.
*/
$CyUtils:> BTreeInsert $$BTreeConfigFileName $$TupleConfigFileName
[1,Kate,ENGL,60000];
$CyUtils:> BTreeInsert $$BTreeConfigFileName $$TupleConfigFileName
[6,Kevin,ALG,70000];
$CyUtils:> BTreeInsert $$BTreeConfigFileName $$TupleConfigFileName
[4,Osborn,ME,80000];
$CyUtils:> BTreeInsert $$BTreeConfigFileName $$TupleConfigFileName
[5,Janathan,SPORT,120000];
$CyUtils:> BTreeShowRootAndSequencePageId $$BTreeConfigFileName;
$CyUtils:> BTreeSequenceScan $$BTreeConfigFileName $$TupleConfigFileName;

/*
Step 7. Show Directory and Lookup for pageAccesses
*/
$CyDB:> showdirectory;
$CyDB:> getPageAllocatedCount;
$CyDB:> getPageDeallocatedCount;
```

B-tree Bulk Loading (PageSizeUsed Equals to 256)

```
/*
Step 0. Create and Load Storage
*/
//$CyDB:> createStorage QuickStorageConfig.xml;
$CyDB:> loadStorage QuickStorageConfig.xml;

/*
Step 1. Dispaly configuration files
*/
$CyDB:> displayXML  .\cyutils\btree\workspace\BTreeConfigBulk1.xml;
$CyDB:> displayXML .\cyutils\btree\workspace\TupleConfigBulk1.xml;

/*
```

```
Step 2. Declare variables for btree and tuple configuration file names
*/
$CyDB:> declare string
$$BTreeConfigBulk1 := .\cyutils\btree\workspace\BTreeConfigBulk1.xml;
$CyDB:> declare string
$$TupleConfigBulk1 := .\cyutils\btree\workspace\TupleConfigBulk1.xml;
$CyDB:> declare string $$TuplesDataBulk1 := .\cyutils\btree\workspace\tuples_bulk1.txt;
$CyDB:> list variables;

/*
Step 3. Lookup for pageAccesses
and list CyUtils commands
*/
$CyDB:> resetPageAccessRelativeCount;
$CyDB:> getPageAllocatedCount;
$CyDB:> getPageDeallocatedCount;
$CyUtils:> list commands;

/*
Step 3. Build an empty BTree, and show its root&sequence page Id
*/
$CyUtils:> BTreeCreateEmpty $$BTreeConfigBulk1 $$TupleConfigBulk1;
$CyUtils:> BTreeShowRootAndSequencePageId $$BTreeConfigBulk1;

/*
Step 4. Prepare sorted data for BTree
*/
$Cyutils:> BTreePrepareSortedData $$TupleConfigBulk1 $$TuplesDataBulk1;

/*
Step 5. BulkLoad auto generated sorted data into btree
, and show page ids
*/
$CyUtils:> BTreeBulkLoad $$BTreeConfigBulk1 $$TupleConfigBulk1
$$TuplesDataBulk1;
$CyUtils:> BTreeShowRootAndSequencePageId $$BTreeConfigBulk1;
/*
$CyUtils:> BTreeSequenceScan $$BTreeConfigBulk1 $$TupleConfigBulk1;
*/

/*
Step 6. Show directory
*/
$CyDB:> showdirectory;

/*
```

```
Step 7. Lookup for pageAccesses
*/
$CyDB:> getPageAllocatedCount;
$CyDB:> getPageDeallocatedCount;
```

B-tree Bulk Loading (Full Page)

```
/*
Step 0. Create and Load Storage
*/
//$CyDB:> createStorage QuickStorageConfig.xml;
$CyDB:> loadStorage QuickStorageConfig.xml;

/*
Step 1. Dispaly configuration files
*/
$CyDB:> displayXML  .\cyutils\btree\workspace\BTreeConfigBulk2.xml;
$CyDB:> displayXML .\cyutils\btree\workspace\TupleConfigBulk2.xml;

/*
Step 2. Declare variables for btree and tuple configuration file names
*/
$CyDB:> declare string
$$BTreeConfigBulk2 := .\cyutils\btree\workspace\BTreeConfigBulk2.xml;
$CyDB:> declare string
$$TupleConfigBulk2 := .\cyutils\btree\workspace\TupleConfigBulk2.xml;
$CyDB:> declare string $$TuplesDataBulk2 := .\cyutils\btree\workspace\tuples_bulk2.txt;
$CyDB:> list variables;

/*
Step 3. Lookup for pageAccesses
and list CyUtils commands
*/
$CyDB:> resetPageAccessRelativeCount;
$CyDB:> getPageAllocatedCount;
$CyDB:> getPageDeallocatedCount;
$CyUtils:> list commands;

/*
Step 3. Build an empty BTree, and show its root&sequence page Id
*/
$CyUtils:> BTreeCreateEmpty $$BTreeConfigBulk2 $$TupleConfigBulk2;
$CyUtils:> BTreeShowRootAndSequencePageId $$BTreeConfigBulk2;

/*
Step 4. Prepare sorted data for BTree
*/
```

```
$Cyutils:> BTreePrepareSortedData $$TupleConfigBulk2 $$TuplesDataBulk2;

/*
Step 5. BulkLoad auto generated sorted data into btree
, and show page ids
*/
$CyUtils:> BTreeBulkLoad $$BTreeConfigBulk2 $$TupleConfigBulk2
$$TuplesDataBulk2;
$CyUtils:> BTreeShowRootAndSequencePageId $$BTreeConfigBulk2;
/*
$CyUtils:> BTreeSequenceScan $$BTreeConfigBulk2 $$TupleConfigBulk2;
*/

/*
Step 6. Show directory
*/
$CyDB:> showdirectory;

/*
Step 7. Lookup for pageAccesses
*/
$CyDB:> getPageAllocatedCount;
$CyDB:> getPageDeallocatedCount;
```

Secondary B-tree

```
/*
Step 0. Create and Load Storage
*/
//$CyDB:> createStorage QuickStorageConfig.xml;
$CyDB:> loadStorage QuickStorageConfig.xml;

/*
Step 1. Dispaly configuration files
*/
$CyDB:> displayXML  .\cyutils\btree\workspace\BTreeConfigS.xml;
$CyDB:> displayXML .\cyutils\btree\workspace\TupleConfigS.xml;

/*
Step 2. Declare variables for primary B-tree and tuple configuration file names
*/
$CyDB:> declare string $$BTreeConfigS := .\cyutils\btree\workspace\BTreeConfigS.xml;
$CyDB:> declare string $$TupleConfigS := .\cyutils\btree\workspace\TupleConfigS.xml;
$CyDB:> declare string $$TuplesDataS := .\cyutils\btree\workspace\tuples_dataS.txt;
$CyDB:> list variables;

/*
```

```
Step 3. list CyUtils commands
*/
$CyUtils:> list commands;

/*
Step 3. Build an empty BTree, and show its root&sequence page Id
*/
$CyUtils:> BTreeCreateEmpty $$BTreeConfigS $$TupleConfigS;
$CyUtils:> BTreeShowRootAndSequencePageId $$BTreeConfigS;

/*
Step 4. Prepare sorted data for BTree
*/
$Cyutils:> BTreePrepareSortedData $$TupleConfigS $$TuplesDataS;

/*
Step 5. BulkLoad auto generated sorted data into btree
, and show page ids
*/
$CyUtils:> BTreeBulkLoad $$BTreeConfigS $$TupleConfigS $$TuplesDataS;
$CyUtils:> BTreeShowRootAndSequencePageId $$BTreeConfigS;

/*
$CyUtils:> BTreeSequenceScan $$BTreeConfigS $$TupleConfigS;
*/

/*
Step 6. Show directory
*/
$CyDB:> showdirectory;

/*
Step 7. Lookup for pageAccesses
*/
$CyDB:> getPageAllocatedCount;
$CyDB:> getPageDeallocatedCount;

/*
Step 8. Declare variables for secondary B-tree and tuple configuration file names
*/
$CyDB:> declare string
$$SecondaryBTreeConfigS := .\cyutils\btree\workspace\SecondaryBTreeConfigS.xml;
$CyDB:> declare string
$$SecondaryTupleConfigS := .\cyutils\btree\workspace\SecondaryTupleConfigS.xml;
$CyDB:> list variables;
```

```
/*
Step 9. Build an empty secondary B-tree, and show root&sequence page Id of both primary
and secondary B-trees
*/
$CyUtils:> BTreeCreateEmpty $$SecondaryBTreeConfigS $$SecondaryTupleConfigS;
$CyUtils:> BTreeShowRootAndSequencePageId $$BTreeConfigS;
$CyUtils:> BTreeShowRootAndSequencePageId $$SecondaryBTreeConfigS;

/*
Step 10. Load for secondary B-tree via a PTC framework
*/
$CyUtils:> BTreeSecondaryBulkLoad $$BTreeConfigS $$TupleConfigS
$$SecondaryBTreeConfigS $$SecondaryTupleConfigS;
$CyUtils:> BTreeShowRootAndSequencePageId $$SecondaryBTreeConfigS;

/*
Step 11. Secondary B-tree scan
*/
$CyUtils:> BTreeSecondarySequenceScan $$BTreeConfigS $$TupleConfigS
$$SecondaryBTreeConfigS $$SecondaryTupleConfigS;

/*
Step 12. Show directory
*/
$CyDB:> showdirectory;

/*
Step 13. Lookup for pageAccesses
*/
$CyDB:> getPageAllocatedCount;
$CyDB:> getPageDeallocatedCount;
```