B+ Tree Indexed Virtual File System

Vishwas Rajashekar | PES1201700704 | Section 4H
Sarang Ravindra | PES1201700972 | Section 4H

PES University
Department of Computer Science and Engineering
Bangalore

0.1 Abstract

Filesystems are the spine of any computer system that stores and retrieves files - from your smartphone to giant distributed database systems. A simple filesystem can store files contiguously and retrieve them using linear searching methods or more complex hashing techniques. However, with the rapid increase in the number of files that computers require to run and the number of files the users generate on a daily basis, the efficiency of any linear methods becomes a bottleneck, severely hindering performance, much to the chagrin of users. In order to keep up with the extreme performance requirements, a more efficient indexing method is needed. In this project, we have chosen to employ the logarithmic time complexity of the B+ Tree.

0.2 Requirements

1. A custom shell to interact with the virtual filesystem with support for typical UNIX shell commands such as:
   - ls
   - pwd
   - mkdir
   - rm
   - touch
   - cat
2. B+ Tree functions to implement the following facilities:
   - file creation
   - file deletion
   - directory creation
   - directory deletion

0.3 Literature Survey

B+ Trees are currently being used as the foundation of the popular Linux filesystem, BTRFS [1]. Important principles such as a separate tree for each subvolume and nested subvolume appearing as directories as well as a doubly-linked B+ Tree node design have been adapted to our design of the B+ Tree Indexed filesystem.

However, for the sake of simplicity, we have left out features such as checksums and hashes, log trees and extended attributes such as permissions.

0.4 Method

Our implementation involves the use of a simple text shell that only accepts valid input and performs the required function and produces any outputs required. A few important ideas to be noted are:

- Every directory has its own B+ Tree to index the files stored in it.
- The files are indexed based on file name and hence, for simplicity, all file names are to be unique.
- Every file and directory has a unique long integer id. This is intended to represent the inode values that UNIX generally used in the INODE Table, however, in this implementation, this unique number is used as the index of a memory location in the "endless memory pool". This is analogous to a memory location on secondary storage.
- Directory handling uses a data structure known as a stackqueue (Stack + Queue). Printing the current working directory is done by reading the queue from head to tail without element dequeue and the current directory is maintained by using the stack where the top of the stack holds a pointer to the current directory.

0.5 System Architecture

Structure Types Used:

```
############### Representation of a File ######################
struct file{
    int file_id;  //unique integer value id
    char file_name[100]; //file name - string
    unsigned char* file_contents; //simulation of bytes stored in a file
    long long int file_size;  //file size (length of the file_contents array)
};
typedef struct file XFILE;

############### Representation of a Directory ##################
struct directory{
    int dir_id; //unique integer value id
    char dir_name[100]; //directory name - string
    struct bnode* dir_tree; // pointer to the root of a B+ Tree
    long long int dir_file_count; //count of the number of files in the directory
};typedef struct directory DIR;

############### Representation of a Memory Pool Unit ###########
struct inner_node{
    int object_type; // 0 is file 1 is dir
    char name[100]; //name of object
    struct file* myfile; //pointer to a file struct
    struct directory* mydir;  //pointer to a directory struct
}; typedef struct inner_node INNERNODE;


############### Representation of a B+ tree's Node #############
struct bnode{
        struct bnode * parent; //parent of the node
```

```
        int controller; // 0 is leaf 1 is internal
        struct bnode** children; // all the pointers to the children of the node
        char ** names; // keys (names)
        long int* ids; // key (ids) - not used for indexing
        int length;  //keeps track of the number of keys in the node
}; typedef struct bnode BNODE;

struct sq{
        struct directory* stackqueue[100]; //locations for the stackqueue.
        //Stores pointers to directory structure
        int top; // top of stack and tail
        int head; //head of the queue
};typedef struct sq STACKQUEUE;

################################################################
```

The architecture of the system is shown below:

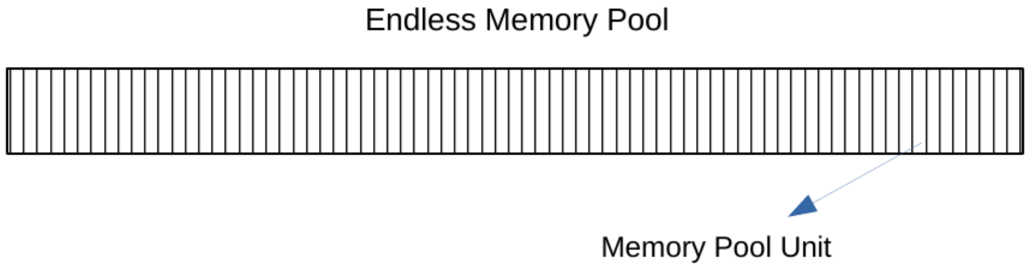

Figure 1: Memory Pool

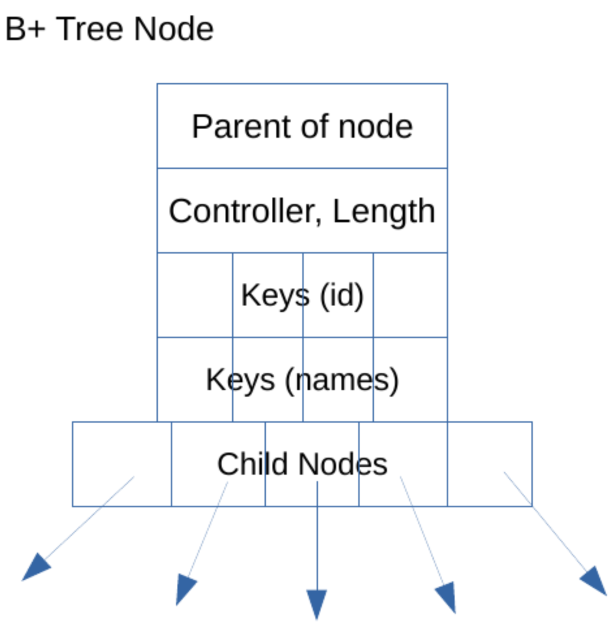

Figure 2: B+ Tree Node

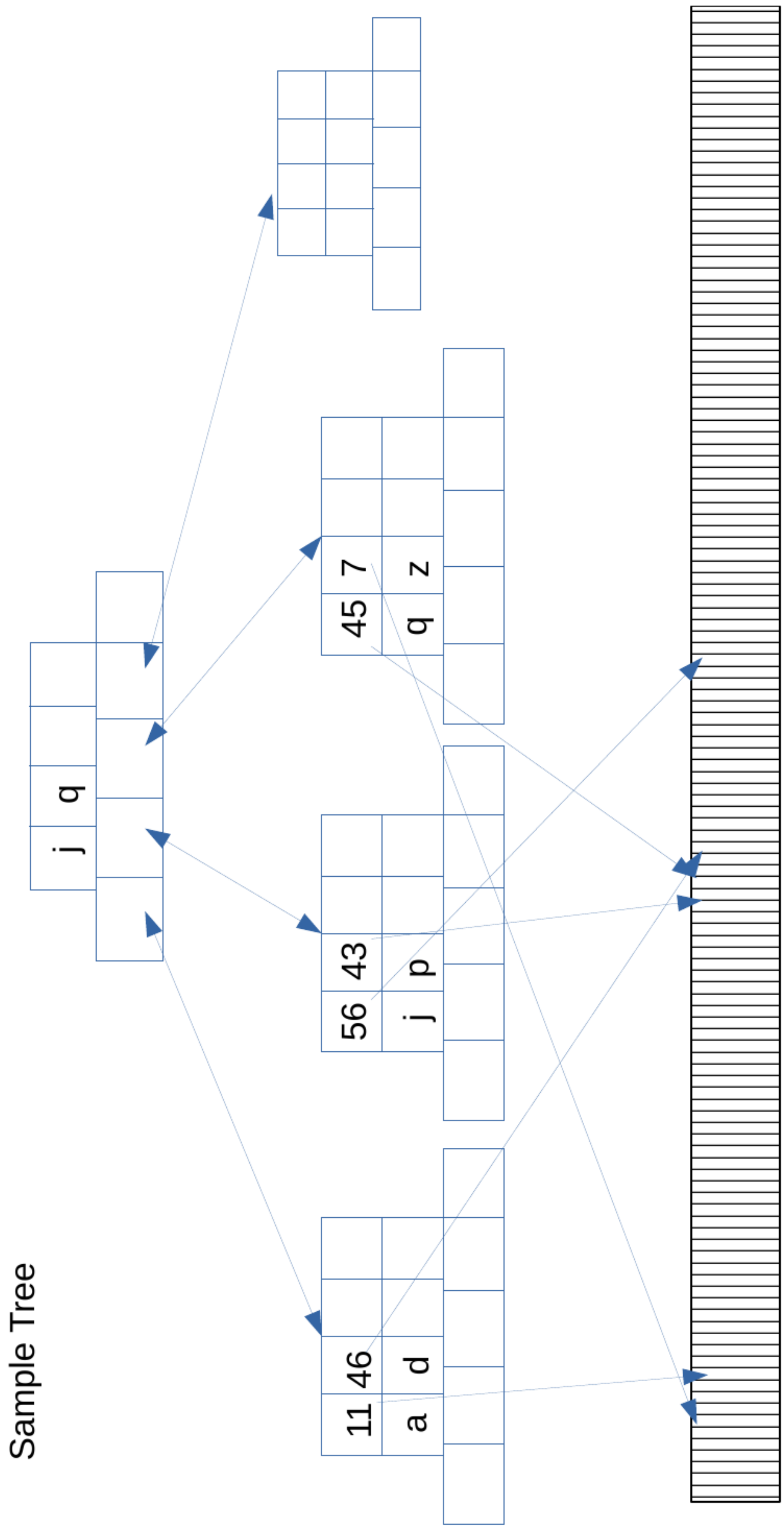

Figure 3: Sample Tree

0.6 Algorithms and Techniques

The algorithms implemented are based on the B+ Tree Visualisation Tool [3] and the helpful resources at: [2] [4]

0.6.1 Insertion Algorithm

```
ALGORITHM insert_leaf(item)
    Get the current directory from the stack
    Navigate the tree's internal nodes by comparing item with the
    names of the keys in the current node
    Find the correct leaf node to insert the item into
    Inside the leaf node:
        if the leaf is not full
            for i <- 0 to length
                if (nodes->names[i] > item) //insertion place found
                    make space for the new item and insert its name
                    make space at the same index in the ids array and
                    insert the item's id
                    increment size of the directory and length of the node
            if new item is to be entered at end of node then
                allocate space and copy id and name
        else
            for i <- 0 to length
                if (nodes->names[i] > item) //insertion place found
                    make space for the new item and insert its name
                    make space at the same index in the ids array and
                    insert the item's id

            if new item is to be entered at end of node then
                allocate space and copy id and name

            create newnode
            split the node into 2 and evenly distribute the keys between
            them
            //promote the name and id of the first element in the second
            //node (middle_item) to the parent recursively
            insert_internal(stackqueue,current_node->parent, middle_item,
            current_node,current_root,newnode)


ALGORITHM
insert_internal(stackqueue,current_root,item,currentchild,newchild)
    Get the current directory from the stack
    if the current_root is not null
        if the current_root is not full
            for i <- 0 to length
                if (nodes->names[i] > item) //insertion place found
```

```
                make space for the new item and insert its name
                make space at the same index in the ids array and
                insert the item's id
                increment size of the directory and length of the node
                adjust the children pointers to align correctly with
                the insertion of the new key
                newnode->parent = parent
                parent->children[x+1] = newnode

        if new item is to be entered at end of node then
            allocate space and copy id and name
            newnode->parent = parent
            parent->child[i+1] = newnode
    else
        for i <- 0 to length
            if (nodes->names[i] > item) //insertion place found
                make space for the new item and insert its name
                make space at the same index in the ids array and
                insert the item's id

        if new item is to be entered at end of node then
            allocate space and copy id and name

        create newnode
        split the current node into 2 and evenly distribute the
        keys and children between them
        //promote the name and id of the last element in the first
        //node
        //(middle_item)to the parent recursively
        insert_internal(stackqueue,current_root->parent,
        middle_item,current_root,bnode)
else
   //child was a root node
    create a new node newroot
    current_node -> controller = 1
    newnode-> controller = 1
    add current_root and new_node as children of newroot
    add the item as the key
    change the pointer at top of stack to point to newroot
```

0.6.2 Deletion Algorithm

```
Start at the root and go up to leaf node containing the key K
Find the node n on the path from the root to the leaf node containing K
    If n is root, remove K
        if root has mode than one keys, done
         if root has only K
            if any of its child node can lend a node
```

```
      Borrow key from the child and adjust child links
   Otherwise merge the children nodes it will be new root
If n is a internal node, remove K
   If n has at least 1 key, return
    If n has 0 keys,
       If a sibling can lend a key,
           Borrow key from the sibling and adjust
           keys in n and the parent node
           Adjust child links
       Else
           Merge n with its sibling
           Adjust child links
If n is a leaf node, remove K
   In case the smallest key is deleted, push up the next key
   If n has at least 1 element, return
   If n has 0 elements
       If the sibling can lend a key
           Borrow key from a sibling and adjust keys
           in n and its parent node
       Else
           Merge n and its sibling
           Adjust keys in the parent node
```

0.6.3 Search Algorithm

```
ALGORITHM search_for_string(current_root,target)
        while (current_root->controller != 0) do // not a leaf
                i <- 0;
                for i<- 0 to current_root->length
                        if (target < current_root->names[i]
                                break
                current_root = current_root->children[i];

        if current_root->controller =0 //leaf

                for i <- 0 to current_root->length
                       if target < current_root->names[i]
                               //found
                               found <- current_root->ids[i];
                               return MASSIVE_HASH[found];


                return NULL;
```

0.7 Sample Execution

```
Script started on 2019-05-10 18:50:29+05:30 [<not executed on terminal>]
mkdir dir1
touch file2
touch file3
touch file4
touch file5
touch file6
touch file7
touch file8
ls
pwd
cat file1
cat file2
cd dir2
cd dir1
mkdir dira
touch file11
touch file12
cat file11
ls
cd ..
cd ..
cd file2
ls
exit
[START] BTRFS initialising....
[WARNING] BTRFS is a Non-Persistent System....
[INFO] BTRFS mounting....
[INFO] BTRFS mounted and ready for operation....
[INFO] Supported commands are: ls, cd, mkdir, rmdir, cat, pwd, touch, rm, exit

<X>>>
<X>>>
<X>>>
<X>>>  [NOTE] root change complete, target : file3

<X>>>
<X>>>
<X>>>
<X>>>
<X>>> This is a new node
DIR 9383 dir1
FILE 886 file2 3
This is a new node
FILE 2777 file3 3
FILE 6915 file4 3
```

```
This is a new node
FILE 7793 file5 3
FILE 8335 file6 3
This is a new node
FILE 5386 file7 3
FILE 492 file8 3

<X>>> [PATH] //

<X>>> [ERROR] File not found

<X>>> [OUTPUT] File Contents: abc | 3 bytes

<X>>> [ERROR] Directory not found

<X>>> [INFO] Directory has been changed
[PATH] //dir1/

<X>>>
<X>>>
<X>>>
<X>>> [OUTPUT] File Contents: abc | 3 bytes

<X>>> This is a new node
DIR 6649 dira
FILE 1421 file11 3
FILE 2362 file12 3

<X>>> [INFO] Directory has been changed
[PATH] //

<X>>> [WARN] Can't go higher than root

<X>>> [ERROR] You can't change directory to a FILE

<X>>> This is a new node
DIR 9383 dir1
FILE 886 file2 3
This is a new node
FILE 2777 file3 3
FILE 6915 file4 3
This is a new node
FILE 7793 file5 3
FILE 8335 file6 3
This is a new node
FILE 5386 file7 3
FILE 492 file8 3
```

```
<X>>> TERMINATED

Script done on 2019-05-10 18:50:29+05:30 [COMMAND_EXIT_CODE="0"]

[START] BTRFS initialising....
[WARNING] BTRFS is a Non-Persistent System....
[INFO] BTRFS mounting....
[INFO] BTRFS mounted and ready for operation....
[INFO] Supported commands are: ls, cd, mkdir, rmdir, cat, pwd, touch, rm, exit

<X>>> touch a

<X>>> touch b

<X>>> touch c

<X>>> touch d
 [NOTE] root change complete, target : c

<X>>> ls
This is a new node
FILE 9383 a 3
FILE 886 b 3
This is a new node
FILE 2777 c 3
FILE 6915 d 3

<X>>> rm a
[INFO] Delete a successful.

<X>>> ls
This is a new node
FILE 886 b 3
This is a new node
FILE 2777 c 3
FILE 6915 d 3

<X>>> rm b
[INFO] Delete b successful.

<X>>> ls
This is a new node
FILE 2777 c 3
This is a new node
FILE 6915 d 3

<X>>> rm a
[ERROR] Delete unsuccessful. File Not Found
```

```
<X>>> touch a

<X>>> ls
This is a new node
FILE 7793 a 3
FILE 2777 c 3
This is a new node
FILE 6915 d 3
```

0.8 Algorithm Time Complexities

- Insertion of an item - O(log(n))
- Deletion of an item - O(log(n))
- Searching for an item - O(log(n))

0.9 Conclusion

With the logarithmic time complexity needed to perform tasks with a B+ Tree based file system, filesystems containing a very large number of files can be indexed rapidly and are inherently more efficient that having a linearly indexed filesystem. This brings more reliability and greatly improved performance for large scale systems.

Bibliography

[1] Wikipedia, Btrfs Design

https://en.wikipedia.org/wiki/Btrfs#Design

[2] B+ Tree : Search, Insert and Delete operations

https://iq.opengenus.org/b-tree-search-insert-delete-operations/

[3] B+ Tree Visualisation

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html/

[4] B+ Trees

http://www.cburch.com/cs/340/reading/btree/index.html