PROJECT REPORT

ON

"Data Storage and Retrieval using B+ Tree Indexing"

Prepared by

MANDEEP SINGH (3323)

PAWAN KUMAR (3329)

Supervised by

Ms S Lokhande

DEPARTMENT OF COMPUTER ENGINEERING

(2016-2017)

ARMY INSTITUTE OF TECHNOLOGY

Pune, Maharashtra, India

TABLE OF CONTENTS

ACKNOWLEDGEMENT

To acknowledge and thank every individual who directly or indirectly contributed to this venture personally, it would require an inordinate amount of time. We are deeply indebted to many individual, whose cooperation made this job easier.

We avail this opportunity to express our gratitude to our friends and our parents for their support and encouragement throughout project. We feel it is as a great pleasure to express our deep sense of profound thank to Ms S Lokhande, who guided us at every step and also encouraged us to carry out the project.

MANDEEP SINGH (3323)

PAWAN KUMAR (3329)

ABSTRACT

We often use various database engines but fail to know how they actually store the data inside tables or any other form.

This project acknowledges us how the data is actually stored and how it is retrieved by taking the example of relational database which stores data using tables.

The search in Relational Database is done through B+ Tree Indexing or Hash based Indexing. Our system is based on B+ Tree Indexing for data storage and retrieval. B+ tree Indexing makes the search efficient.

Another feature of our system is that it read data from and writes data on disk in blocks, which enhances the security of system and also make the access of data easier.

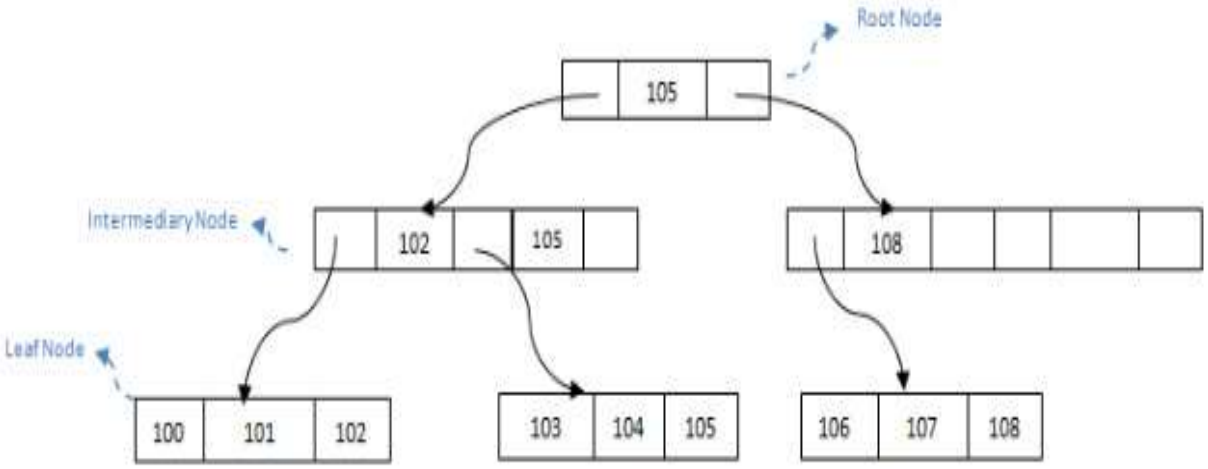
Meta Data of particular table is stored separately in file for quick information of table attributes.

Introduction

Consider the STUDENT table below. This can be stored in B+ tree structure as shown below. We can observe here that it divides the records into two and splits into left node and right node. Left node will have all the values less than or equal to root node and the right node will have values greater than root node. The intermediary nodes at level 2 will have only the pointers to the leaf nodes. The values shown in the intermediary nodes are only the pointers to next level. All the leaf nodes will have the actual records in a sorted order.

STUDENT

STUDENT_ID	STUDENT_NAME	ADDRESS
100	Joseph	Alaiedon Township
101	Allen	Fraser Township
102	Chris	Clinton Township
103	Patty	Troy
104	Jack	Fraser Township
105	Jessica	Clinton Township
106	James	Troy
107	Antony	Alaiedon Township
108	Jacob	Troy



If we have to search for any record, they are all found at leaf node. Hence searching any record will take same time because of equidistance of the leaf nodes. Also they are all sorted. Hence searching a record is like a sequential search and does not take much time.

Suppose a B+ tree has an order of n (it is the number of branches – above tree structure has 5 branches altogether, hence order is 5), and then it can have n/2 to n intermediary nodes and n/2 to n-1 leaf nodes. In our example above, n= 5 i.e.; it has 5 branches from root. Then it can have intermediary nodes ranging from 3 to 5. And it can have leaf nodes from 3 to 4.

Main Goal of B+ Tree:

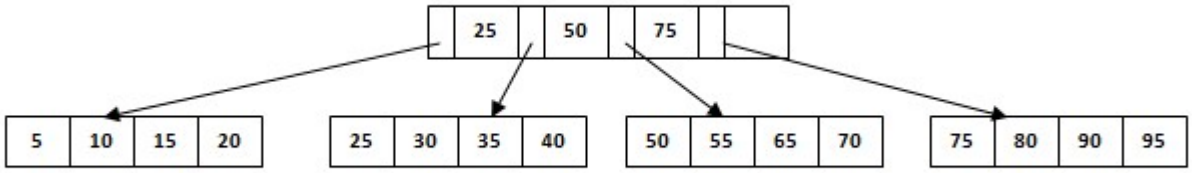
- **Sorted Intermediary and leaf nodes:** Since it is a balanced tree, all nodes should be sorted.
- **Fast traversal and Quick Search:** One should be able to traverse through the nodes very fast. That means, if we have to search for any particular record, we should be able pass through the intermediary node very easily. This is achieved by sorting the pointers at intermediary nodes and the records in the leaf nodes.

  Any record should be fetched very quickly. This is made by maintaining the balance in the tree and keeping all the nodes at same distance.

- **No overflow pages:** B+ tree allows all the intermediary and leaf nodes to be partially filled – it will have some percentage defined while designing a B+ tree. This percentage up to which nodes are filled is called fill factor. If a node reaches the fill factor limit, then it is called overflow page. If a node is too empty then it is called underflow. In our example above, intermediary node with 108 is underflow. And leaf nodes are not partially filled, hence it is an overflow. In ideal B+ tree, it should not have overflow or underflow except root node.

Searching of Record

Suppose we want to search 65 in the below B+ tree structure. First we will fetch for the intermediary node which will direct to the leaf node that can contain record for 65. So we find branch between 50 and 75 nodes in the intermediary node. Then we will be redirected to the third leaf node at the end. Here DBMS will perform sequential search to find 65. Suppose, instead of 65, we have to search for 60. What will happen in this case? We will not be able to find in the leaf node. No insertions/update/delete is allowed during the search in B+ tree.



Pseudo Code for Search

The algorithm for search _nds the leaf node in which a given data entry belongs. A pseudocode sketch of the algorithm is given in Figure 9.9. We use the notation **ptr* to denote the value pointed to by a pointer variable *ptr* and & *(value)* to denote the address of *value*.
Note that _nding *i* in *tree search* requires us to search within the node, which can be done with either linear search or a binary search (e.g., depending on the number of entries in the node).

In discussing the search, insertion, and deletion algorithms for B+ trees, we will assume that there are no *duplicates*. That is, no two data entries are allowed to have the same key value. Of course, duplicates arise whenever the search key does not contain a candidate key and must be dealt with in practice.

```
func find (search key value K) returns nodepointer
// Given a search key value, finds its leaf node
return tree_search(root, K);                          // searches from root
endfunc

func tree_search (nodepointer, search key value K) returns nodepointer
// Searches tree for entry
if *nodepointer is a leaf, return nodepointer;
else,
    if K < K_1 then return tree_search(P_0, K);
    else,
        if K ≥ K_m then return tree_search(P_m, K);        // m = # entries
        else,
            find i such that K_i ≤ K < K_{i+1};
            return tree_search(P_i, K)
endfunc
```

Fig. Algorithm for Search

Insertion of Record

The algorithm for insertion takes an entry, _nds the leaf node where it belongs, and inserts it there. The basic idea behind the algorithm is that we recursively insert the entry by calling the insert algorithm on the appropriate child node. Usually, this procedure results in going down to the leaf node where the entry belongs, placing the entry there, and returning all the way back to the root node. Occasionally a node is full and it must be split. When the node is split, an entry pointing to the node created by the split must be inserted into its parent; this entry is pointed to by the pointer variable *newchildentry*. If the (old) root is split, a new root node is created and the height of the tree increases by one.

If we insert entry 8*, it belongs in the left-most leaf, which is already full. This insertioncauses a split of the leaf page; the split pages are shown in Figure 9.12. The tree must now be adjusted to take the new leaf page into account, so we insert an entry consisting of the pair *h5, pointer to new pagei* into the parent node.

Notice how the key 5, which discriminates between the split leaf page and its newly created sibling, is `copied up.'

We cannot just `push up' 5, because every data entry must appear in a leaf page. Since the parent node is also full, another split occurs. In general we have to split a non-leaf node when it is full, containing 2*d* keys and 2*d* + 1 pointers, and we have to add another index entry to account for a child split. We now have 2*d* + 1 keys and 2*d*+2 pointers, yielding two minimally full non-leaf nodes, each containing *d* keys and *d*+1 pointers, and an extra key, which we choose to be the `middle' key. This key and a pointer to the second non-leaf node constitute an index entry that must be inserted into the parent of the split non-leaf node. The middle key is thus `pushed up' the tree, in contrast to the case for a split of a leaf page. The split pages in our example are shown in Figure 9.13. The index entry pointing to the new non-leaf node is the pair *h17, pointer to new index-level pagei*; notice that the key value 17 is `pushed up' the tree, in contrast to the splitting key value 5 in the leaf split, which was `copied up.' The di_erence in handling leaf-level and index-level splits arises from the B+ tree requirement that all data entries *k_* must reside in the leaves. This requirement prevents us from `pushing up' 5 and leads to the slight redundancy of having some key values appearing in the leaf level as well as in some index level. However, range queries can be e_ciently answered by just retrieving the sequence of leaf pages; the redundancy is a small price to pay for e_ciency. In dealing with the index levels, we have more flexibility, and we `push up' 17 to avoid having two copies of 17 in the index levels.

```
proc insert (nodepointer, entry, newchildentry)
// Inserts entry into subtree with root '*nodepointer'; degree is d;
// 'newchildentry' is null initially, and null upon return unless child is split

if *nodepointer is a non-leaf node, say N,
    find i such that K_i ≤ entry's key value < K_{i+1};          // choose subtree
    insert(P_i, entry, newchildentry);                  // recursively, insert entry
    if newchildentry is null, return;                  // usual case; didn't split child
    else,                          // we split child, must insert *newchildentry in N
        if N has space,                                                  // usual case
            put *newchildentry on it, set newchildentry to null, return;
        else,                                  // note difference wrt splitting of leaf page!
            split N:                          // 2d + 1 key values and 2d + 2 nodepointers
            first d key values and d + 1 nodepointers stay,
            last d keys and d + 1 pointers move to new node, N2;
            // *newchildentry set to guide searches between N and N2
            newchildentry = & (⟨smallest key value on N2, pointer to N2⟩);
            if N is the root,                                  // root node was just split
                create new node with ⟨pointer to N, *newchildentry⟩;
                make the tree's root-node pointer point to the new node;
            return;

if *nodepointer is a leaf node, say L,
    if L has space,                                                      // usual case
    put entry on it, set newchildentry to null, and return;
    else,                                          // once in a while, the leaf is full
        split L: first d entries stay, rest move to brand new node L2;
        newchildentry = & (⟨smallest key value on L2, pointer to L2⟩);
        set sibling pointers in L and L2;
        return;
endproc
```

Fig. Algorithm for Insertion into B+ Tree of order d

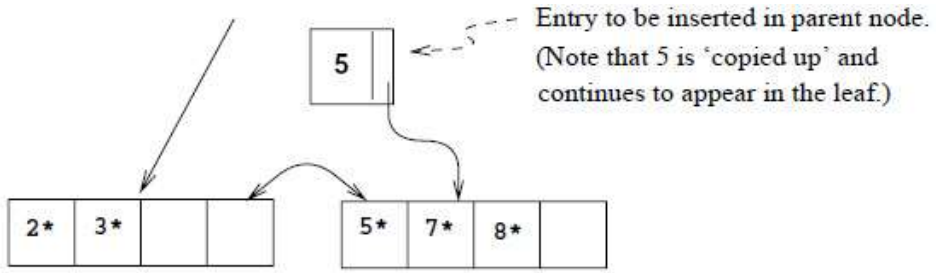


Figure 9.12 Split Leaf Pages during Insert of Entry 8*

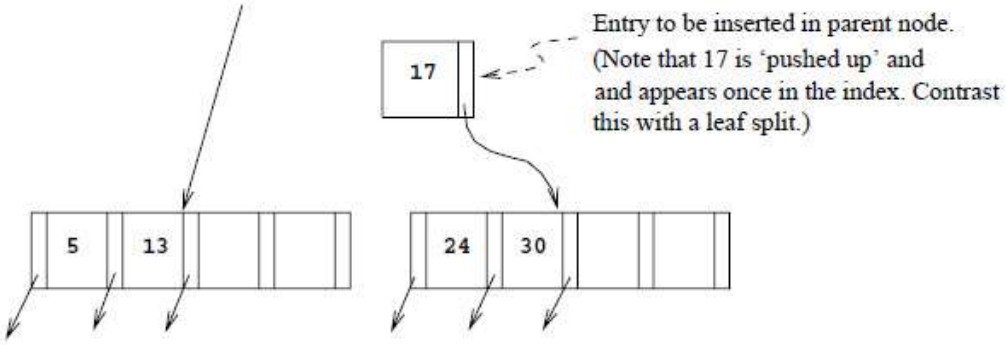


Fig. Split Index Pages during Insert of Enter 8*

Now, since the split node was the old root, we need to create a new root node to hold the entry that distinguishes the two split index pages. The tree after completing the insertion of the entry 8* is shown in below fig.

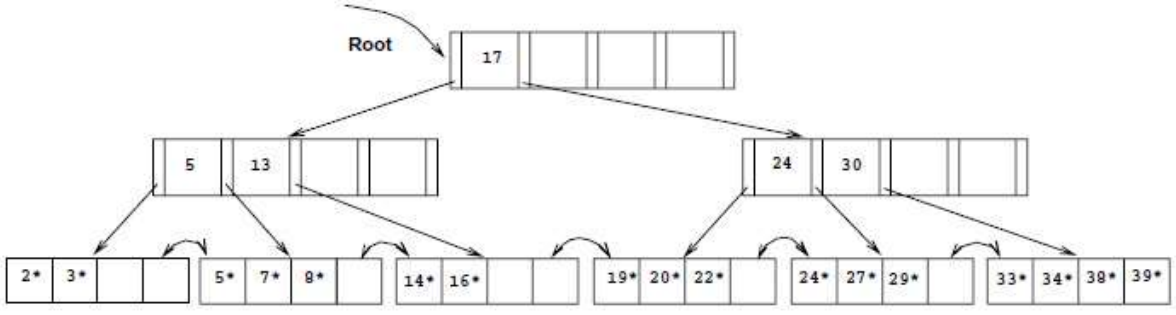


Fig. B+ Tree after inserting Entry 8*

To illustrate redistribution, reconsider insertion of entry 8* into the tree shown in Figure 9.10. The entry belongs in the left-most leaf, which is full. However, the (only) sibling of this leaf node contains only two entries and can thus accommodate more entries. We can therefore handle the insertion of 8* with a redistribution. Note how the entry in the parent node that points to the

second leaf has a new key value; we `copy up' the new low key value on the second leaf. This process is illustrated in below Fig.

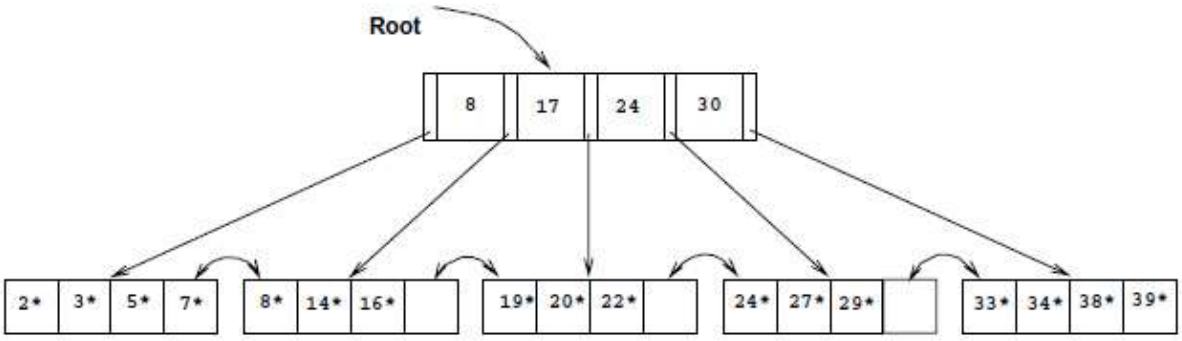
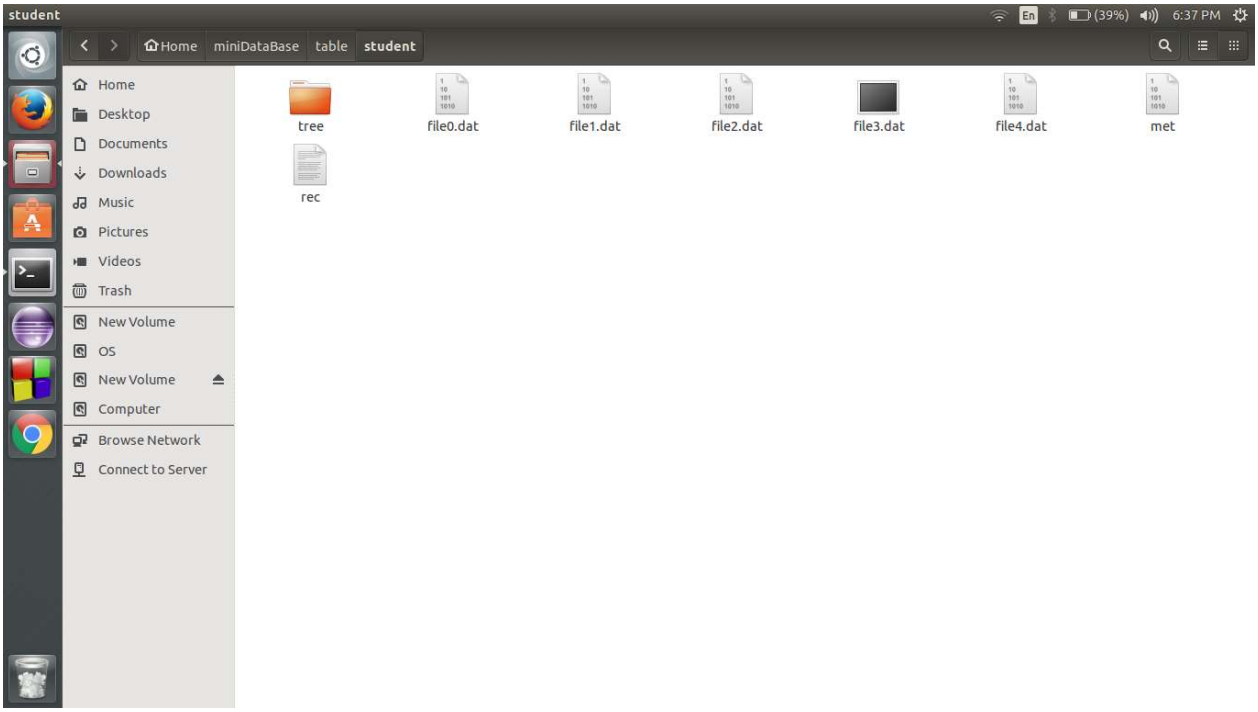


Fig. B+ Tree after Inserting Entry 8* Using Redistribution

To determine whether redistribution is possible, we have to retrieve the sibling. If the sibling happens to be full, we have to split the node anyway. On average, checking whether redistribution is possible increases I/O for index node splits, especially if we check both siblings. (Checking whether redistribution is possible may reduce I/O if the redistribution succeeds whereas a split propagates up the tree, but this case is very infrequent.) If the _le is growing, average occupancy will probably not be a_ected much even if we do not redistribute. Taking these considerations into account, *not* redistributing entries at non-leaf levels usually pays o_. If a split occurs at the leaf level, however, we have to retrieve a neighbor in order to adjust the previous and next-neighbor pointers with respect to the newly created leaf node. Therefore, a limited form of redistribution makes sense: If a leaf node is full, fetch a neighbor node; if it has space, and has the same parent, redistribute entries. Otherwise (neighbor has di_erent parent, i.e., is not a sibling, or is also full) split the leaf node and adjust the previous and next-neighbor pointers in the split node, the newly created neighbor, and the old neighbor.

Data Storage Inside Files



For each row entry, a separate file is created for storage of data. As in above fig. there are 5 rows, so 5 files are created for each row.

The tree folder contains the B+ Tree node data and the corresponding pointers.

Structure of tree file

(is_leaf, key_size, (0 to key_size) key_values, pointer_size, (0 to pointer_size) pointers, next_node);

Our System

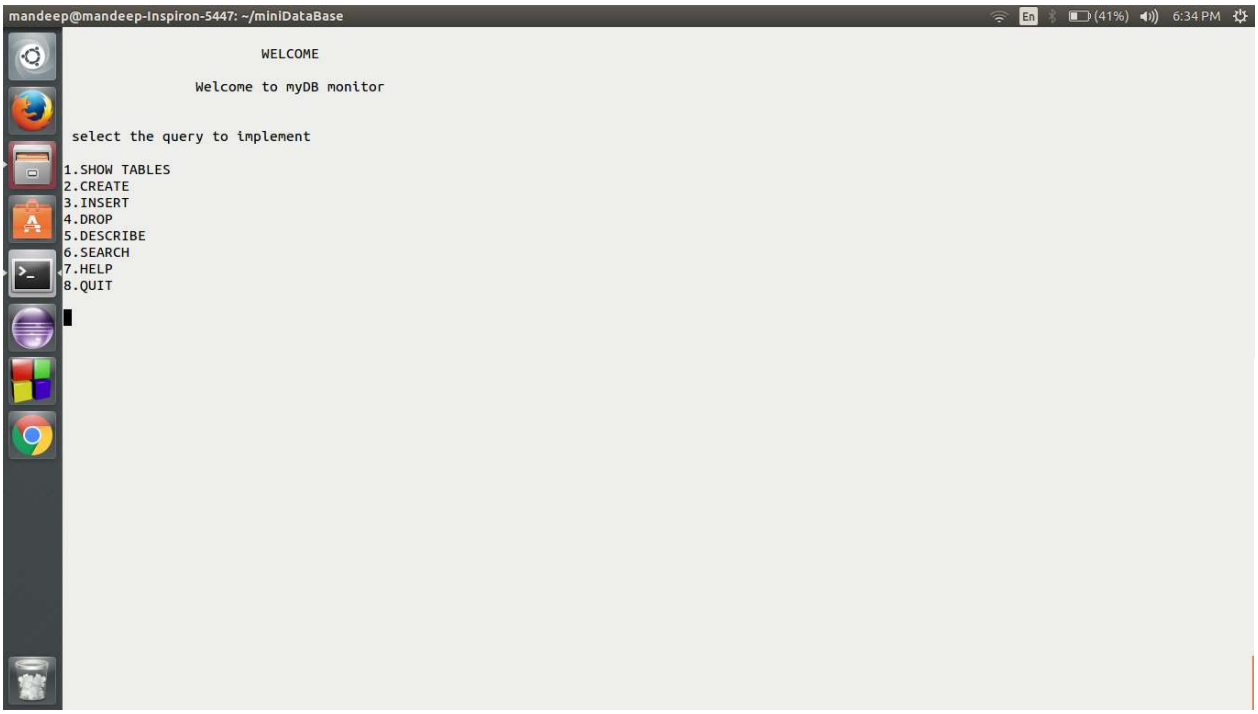


Fig. Program at Start

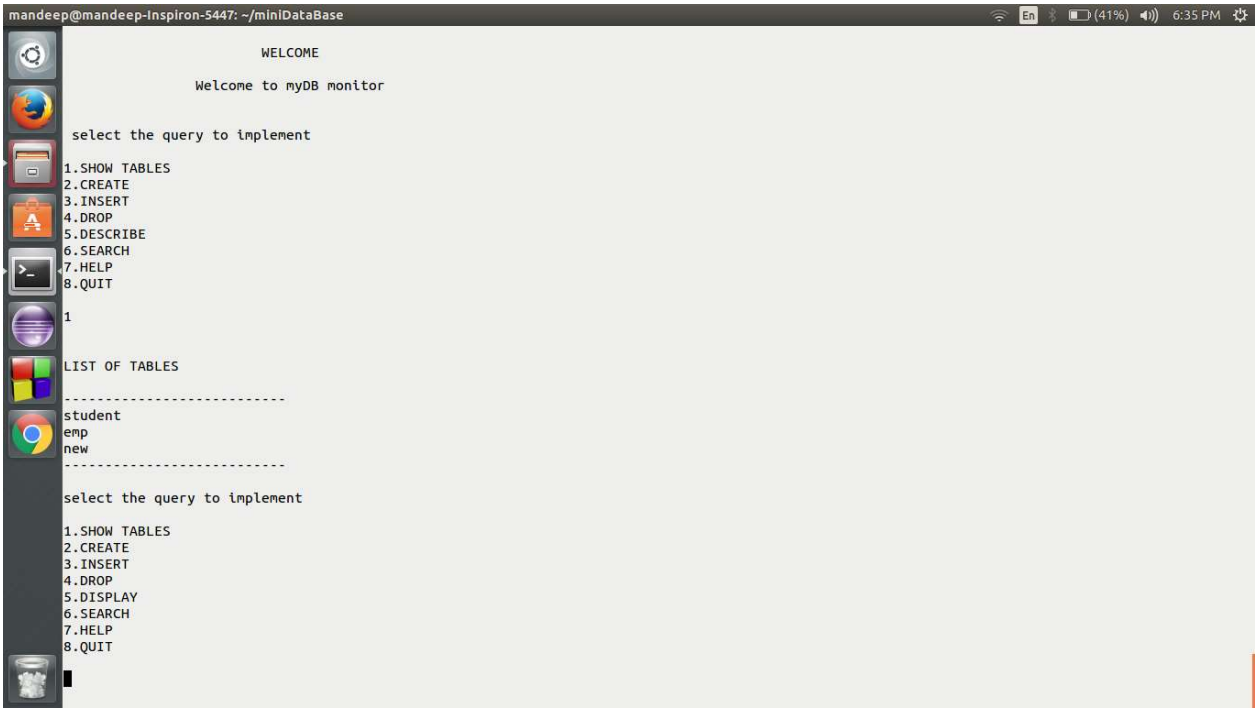


Fig. List of table entries inside database

Fig. display table contents

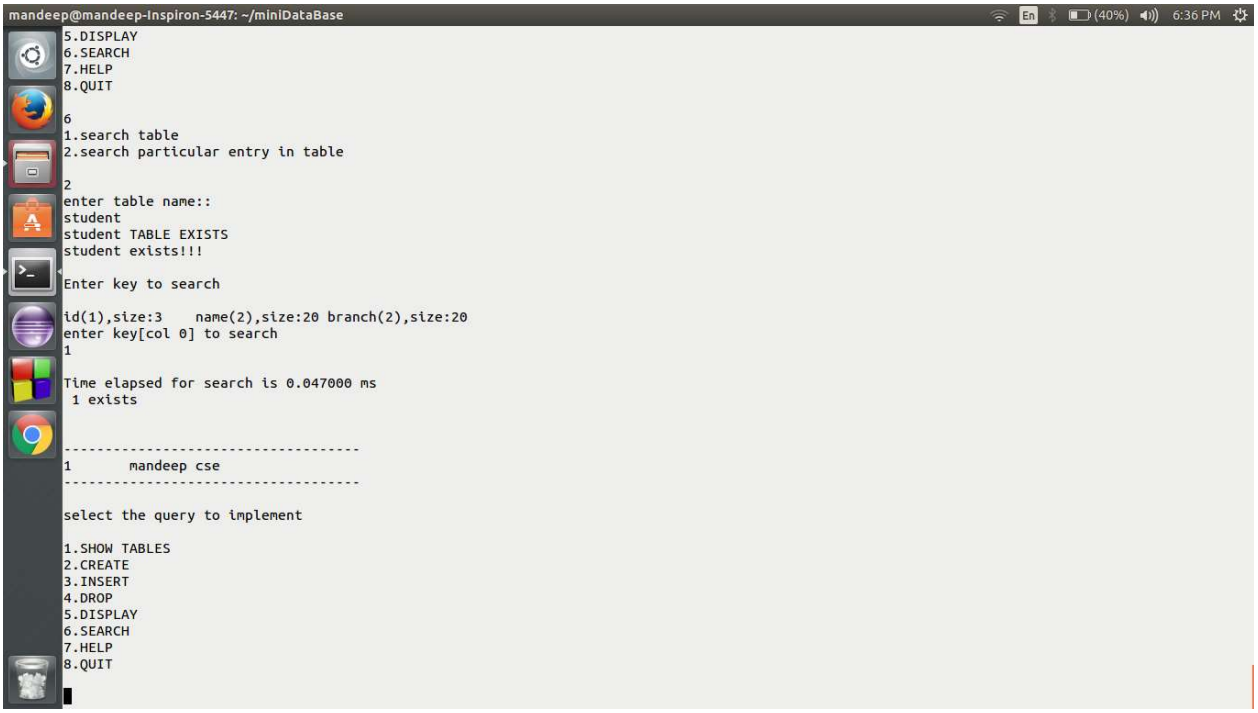


Fig. Search Inside the table.

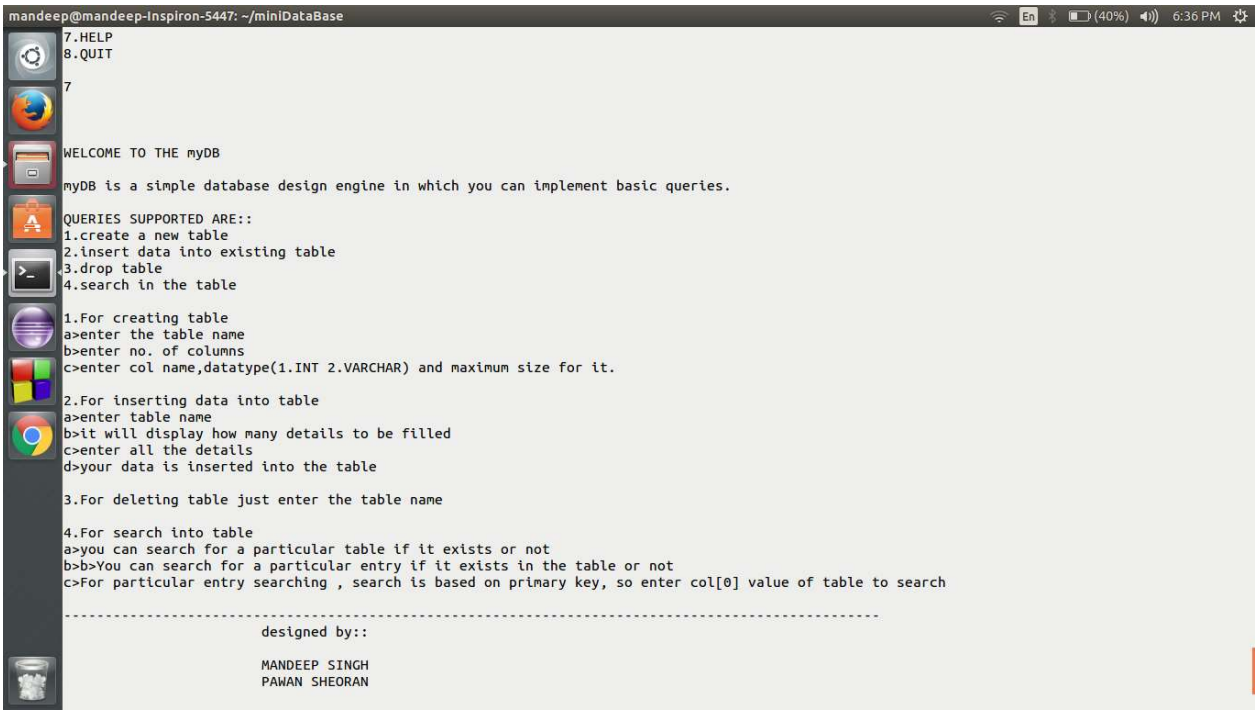


Fig. Help Window

Analysis and Conclusion

After completion of project, we get acknowledged about how data is actually stored inside the table, how Indexing is done, how file pointers are created.
Also, B+ Tree Indexing makes search efficient for huge amount of data which cannot be loaded into primary memory at the time of execution of program so simultaneously we require writing data back to secondary storage.

There is another approach for Indexing called Hash based Indexing in which search is faster but much time delay is generated for handling collisions.

Bibliography

References

[1] Database Management System by Raghu Ramakrishnan and Johannes Gehrke

[2] Using Compressed B+-trees for Line-based Database Indexes, by Hung-yi Lin and Chin-ling Chen published in Signal Processing and Information Technology, 2006 IEEE International Symposium