Toy DB Project Report

Shivam Garg (12D020036), Aditya Nambiar (12D070012)

November 25, 2014

1 Project Statement

Implement construction of B+ tree when the file is sorted by the key already, and where the tree is constructed from leaf level onwards, level by level. Use adequate buffers to get better performance (study for different size of buffers). Get B+ tree performance data, and compare the performance with the AM layer index facility

2 Algorithm

2.1 Algorithm Description

The algorithm consists of following steps:

1. Create a root node (this node is the only node in the B+ tree initially)
2. Add (key,pointer) pairs of the records in table to the root node.
3. When the root node is full, create another node to keep the (key, pointer) pairs and make a new root node which has (key,pointer) pairs pointing to these two nodes. In our convention, the 2 leaf nodes are at level 0 and the root node is at level 1. Keep adding the (key,pointer) pairs to the right most leaf node.
4. When this rightmost leafnode is full, create another leaf node and add the new (key,pointer) pairs to this leaf node. Also add (key,pointer) pairs (pointing to this 3rd leaf node) in the root node (at level 1) .When the root node at level 1 is full, create another node at level 1 and add the new keys to that node. Also, create a new root node at level 2 and make it point to the nodes at level 1.
5. This process will continue and B+ tree will continue to grow in height as well as width.

Note that at any time during this construction of B+ tree, we only need the rigtmost nodes at each level in the buffer. So we can create 2 array of buffer pointers and page numbers corresponding to these right most nodes at each level.

2.2 Pseudocode

Algorithm 1 Bulk Loading in B+ Tree

```
1: function make_index(fp)                                   ▷ fp ← file pointer to the file
2:     curr_leaf ← new empty_node
3:     vector right_most_buf          ▷ contains the buffer pointer to the right most node at each level
4:     right_most_buf.push_back(curr_leaf)
5:                                  ▷ right_most_buf[0] will contain the buffer pointer to right most leaf
6:     while fp.next_record do
7:         if curr_leaf is not full then
8:             right_most_buf[0].add(fp.key,fp.RecordPointer)
9:         else                                                          ▷ curr_leaf is full
10:            right_most_buf[0].unfix()          ▷ Write back the right most leaf node to the hard disk
11:            new_leaf ← new empty_node                                  ▷ Create new leaf node
12:            right_most[0] ← new_leaf          ▷ Set the right most leaf node buffer pointer to new_leaf
13:            right_most_buf[0].add(fp.key,fp.RecordPointer) ▷ Add key,pointer in the right most leaf node
14:            add_parent(0,fp.key,right_most_buf[0].disk_address)
15:                          ▷ Make the right most node on level 1 the parent of right most leaf node
16:        end if
17:    end while
18:    Redistribute() ▷ Redistribute the keys present in last and second last node at each level (if required)
19: end function
20:
21: function add_parent(level,key,disk_address)
22:    if right_most_buf.size() == level+1 then                          ▷ Node has no parent
23:        right_most_buf.push_back(new empty_node)  ▷ Create a new parent and store its buffer pointer
24:    end if
25:    parent ← right_most_buf[level+1]
26:    if parent is not full then
27:        parent.add(key,disk_address)              ▷ Add key and pointer (to the child) in the parent
28:    else                                                              ▷ Parent is full
29:        right_most_buf[level+1].unfix()   ▷ Write back the right most node at level+1 to the hard disk
30:        new_parent ← new empty_node                  ▷ Create a new parent for the child
31:        right_most_buf[level+1] ← new_parent
32:                        ▷ Set the right most node buffer pointer (at level=level+1) to new_parent
33:        new_parent.add(key,disk_address)          ▷ Add key and pointer (to the child) in the parent
34:        add_parent(level+1,key,right_most_buf[level+1].disk_address)
35:                ▷ Make the right most node on level=level+2 the parent of right most node at level+1
36:    end if
37: end function
```

3 Implementation

We have made a file insert.c which contains all the functions needed for bulk loading. It consists of the following functions:

1.
```
errVal=InsertEntry(fileDesc,attrType,attrLength,value,recId,last)
int fileDesc; /* file Descriptor */
```

```
char attrType; /* 'i' or 'c' or 'f' */
int attrLength; /* 4 for 'i' or 'f', 1-255 for 'c' */
char *value; /* value to be inserted */
int recId; /* recId to be inserted */
int last; /*Wheteher the value to be inserted is the last value*/
```

This routine has to be called by the main function whenever we need to enter a value-recId pair. If the current value-recId pair to be entered is the last one, then paramter last should be set to 1, else it should be set to 0. This function calls the function InsertintoLeaf() to add the value-recId pair to the leaf. If InsertintoLeaf() return TRUE, that means the value has been inserted. If it return FALSE, that means there is no more space in the leaf and a new leaf has to be created. This function calls the function AddtoParent() for inserting the corresponding value-pointer pairs to the the parent node.

2.
```
errVal=InsertintoLeaf(pageBuf,attrLength,attrType,value,recId)
char *pageBuf;/* buffer where the leaf page resides */
int attrLength;/* 4 for 'i' or 'f', 1-255 for 'c' */
char attrType;/* 'i' or 'c' or 'f' */
char *value;/* attribute value to be inserted*/
int recId;/* recid of the attribute to be inserted */
```

This route inserts the given value-recId pairs to the leaf whose buffer location is given by pageBuf.If the value-recId can be inserted, it return TRUE. If the leaf is full and the value-recId can't be inserted, it return FALSE.

3.
```
errVal= AddtoParent(fileDesc,level,rightmost_page,rightmost_buf,value,attrLength, length)
int fileDesc;/* file Descriptor */
int level;/*Level at which the node whose pointer has to be added to parent is present*/
int *rightmost_page;/*Array which stores the page number of right most node at each level*/
char **rightmost_buf;/*Array which stores the buffer pointer of right most node at each
    level*/
char *value; /* pointer to attribute value to be added*/
int attrLength;/* 4 for 'i' or 'f', 1-255 for 'c' */
int *length;/*Length of rightmost_page and rightmost_buf arrays*/
```

When a node leaf is full a new leaf node needs to be created. The AddtoParent function is used to add the pointer to this new leaf to the parent node. When the node to which the pointer is to be added is full, the AddtoParent function recursively calls itself. If the node to which the pointer is being added is full and it is also the root node. A new root node needs to be created.

4.
```
void Print_check(fd,rightmost_page,rightmost_buf,length,buff_hits)
int fd;
int *rightmost_page;
char **rightmost_buf;
int length;
```

This function is used to print the entire B+ Tree level by level. The function is meant only for testing purposes. This function is only called if the global variable *print_check* is set to 1.

4 Performance Analysis

For comparing the performance of bulk-loading vs normal insertion we have used several parameters such as number of nodes created ,number of read accesses , number of write accesses, buffer hits and time. We have calculated the number of nodes by counting the number of times we have called the function $PF\_AllocPage$ The number of read accesses is calculated by counting the number of times the function $PFreadfcn$ is called. The number of write accesses is calculated by counting the number of times the function $PFwritefcn$ is called. We have also kept track of the write transfers during the closing of the file. Bufferhits have been calculated by keeping track of the number of calls to $PF\_GetThisPage$ and $PF\_GetFirstPage$. Both these functions call a function $PFbufGet$. If the page exists in $PFhashFind$ then its a buffer hit else it is a miss.

Note: Buffer Hits parameter is not very relevant for bulk loading as we only write pages into disk or allocate new pages. So there are no reads of pages from disks. So buffer hits is not required for bulk loading.

We have fixed the page size to 64 Bytes and varied the number of records and maximum buffer capacity. Page size has been deliberately kept low so that better testing can be done using relatively less number of records. At this page size, a leaf node can hold upto 3 keys and any internal node can hold upto 5 keys. Various performance analysis results obtained are as follows:

Algorithm	N_{rec}	B_{size}	N_{node}	B_{hit}	N_{read}	N_{write}	T
Bulk Loading	200	20	81	-	1	84	2
Normal Insertion	200	20	263	1075	1	266	3
Bulk Loading	1000	20	403	-	1	406	3
Normal Insertion	1000	20	1328	6624	1	1331	9
Bulk Loading	1000	6	403	-	1	406	3
Normal Insertion	1000	6	1328	4453	2172	1680	15
Bulk Loading	1000	30	403	-	1	406	3
Normal Insertion	1000	30	1328	6624	1	1331	10
Bulk Loading	10000	10	4003	-	1	4006	22
Normal Insertion	10000	10	13328	81703	674	13458	96

N_{rec}: Number of records inserted
B_{size}: Block Size (In number of pages)
N_{node}: Number of nodes in B+ tree
B_{hit}: Number of read buffer hits
N_{read}: Number of read disk accesses
N_{write}: Number of write disk accesses
T : Time for insertion (in ms)

4.1 Observations

1. Number of nodes in B+ tree is much more for normal insertion as compared to bulk loading. This is because in bulk loading most of the nodes are fully filled whereas in normal insertion, splitting of nodes lead to increase in number of nodes.

2. Number of read disk accesses for Bulk Loading is always 1 which is to get the first page. After that, no more pages are read from disk. We only need to write a node (page) whenever it is full.

3. Number of read disk accesses are generally 1 for normal insertion except when the buffer size is low. This is because, most of the time the page required is there in the buffer which leads to buffer hit. For example, with number of records=1000, when we changed the buffer size from 20 to 6, number of disk

read accesses increase from 1 to 2172 whereas number of buffer hits decrease from 6624 to 4453. Note that, in both the cases the sum of number of disk accesses and buffer hits remain constant as any read operation will be either from buffer or disk. Read buffer hits are not there for bulk loading as we read only 1 page (to get the first page of index).

4. There is no recognizable change in any parameter when we change the buffer size from 20 to 30 keeping the number of records=1000 (for both normal insertion and bulk loading). This is because after sufficient size of buffer, there are no more buffer conflicts. So increase in buffer size does not change anything.

5. When the number of records have been increased to 10000 and buffer size has been set to 10. Time required by normal insertion drastically increases (as compared to bulk loading). Some of the buffer accesses have resulted in buffer misses for normal insertion leading to increase of read disk access from 1 to 674. This is because buffer size is insufficient as compared to the number of records. So, as the number of records to be inserted increase, difference in performance of normal insertion and bulk loading will be more apparent.

5 Conclusion

Performance of bulk loading is much better than normal insertion. Although, future updates/inserts in normal insertion can be faster as most of the nodes will have empty space and no splitting will be needed. But this benefit will not be able to overcome the extra cost that normal insertion has for first time insertion. So whenever we have sorted data to be inserted in new B+ tree index, we should prefer bulk loading.