DataBase Internals Final Report

Prof. NL Sarda CS387

Anchit Gupta	Samarth Mishra	Pradyot Prakash
13D100032	130260018	130050008

November 7, 2015

1 Introduction

B+ trees are a very important data structure used by a wide range of DBM Systems. The small height of the B+ tree makes it optimal for storing indexes in database systems, as compared to a binary tree, due to the reduced number of disk access. We worked on the B+ tree implementation provided in the toyDB code, and added more functionality to it, as described in the next section.

2 Objectives

We have added Bulk Loading functionality to the existing B+ tree implementation provided to us with toyDB. Bulk loading refers to the initial insertion of a large number of keys into an index.
We have implemented two types of bulk loading.

- Sort the input and directly insert into the tree
- Build the tree bottom up after sorting the data

The performance of these have been compared with the naive bulk-loading with no sorting. We have also added support for secondary indices in the form of maintaining a bag of pointer for a given key

3 Implementation artifacts

3.1 Algorithm

The basic algorithm relies on the fact that the tree can be constructed bottom up.

- Initially we have a single node, the root. We iteratively insert elements into the node until we run out of space

- We create a new node for the leaf layer and also create a node to act as the parent of the new nodes
- We continue filling the leaf and creating a new node whenever we run out of space. We add the pointer to the newly created node to its parent. If the parent is also out of space, we create a new node which acts as the parent for the parent.
- This way the tree grows in height as well as width. This continues until the entire tree is constructed.

A little diversion from this scheme would occur when the last value is being inserted. It may happen the last node at the leaf level contains fewer elements than $\lceil \frac{n-1}{2} \rceil$ or an internal node has fewer than $\lceil \frac{n}{2} \rceil$ nodes, in which case we will have to adjust the number of elements in the last two nodes at that level.

3.2 Data structures

We would need to store data for two purposes:

- Keeping track of the last node at each layer, for insertion into the parent
- Keeping track of the second last node at each layer, for readjusting the nodes after the insertion of the last element
- Keeping track of the last bag of pointers for the secondary index

Both of the first 2 tasks can be accomplished by using arrays to store the corresponding last and second last nodes at each level. The third part can be done by maintaining the pointer to the bag page for each distinct key. Our task would be split into two parts:

1. The inserted item is not the last item: In this case we simply need to create a new leaf node. Insert this into the parent, if space is available. Else, create a new parent node and insert the pointer to the new node into that. Recursively continue up the tree
2. The inserted item is the last item: In this case, we will need to see if the last leaf node has the required number of elements. If yes, then continue as before. If not, then merge the last two leaf nodes and then split them. Though this leaf merging is required for maintaining the balance, it does not affect the complexity of operations, hence this hasn't been done in our implementation. Propagate the same condition up the tree. Unfix all rightmost_buf pages.

3.3 Secondary Index Support

To support creation of secondary index we have to use a new leaf page format. We implement the following leaf node structure, with support for bagging.

Each leaf node has a header names AM_BAGHEADER which contains the essential information regarding the number of keys and the next page number. For each key entry in the

index, there are three additional fields. The first field tells us if we have a single occurrence of the key. If yes, then the next two fields, give the page number and the address within the page respectively, of the actual record. If no, then the next two fields give the address of the bag page where all the addresses to the corresponding records are stored. So this mapping essentially serves as the pointer to the base of the node from where we can get all the record pointers directly.

The structure of the bag page is as follows. At the start of each page, we have the offset to the first free location. This is useful when we are finding a repeated key value for the first time. All the set of pointers for a particular key value, start after a header. This header keeps track of the number of records there are corresponding to the key value. Furthermore, this needs to keep track of the location where next record pointer can go. It may happen that there is not sufficient space in the current node, and so we might need to employ another page to keep track of these values as well. So the header, contains the next page number so that continuity can be maintained.

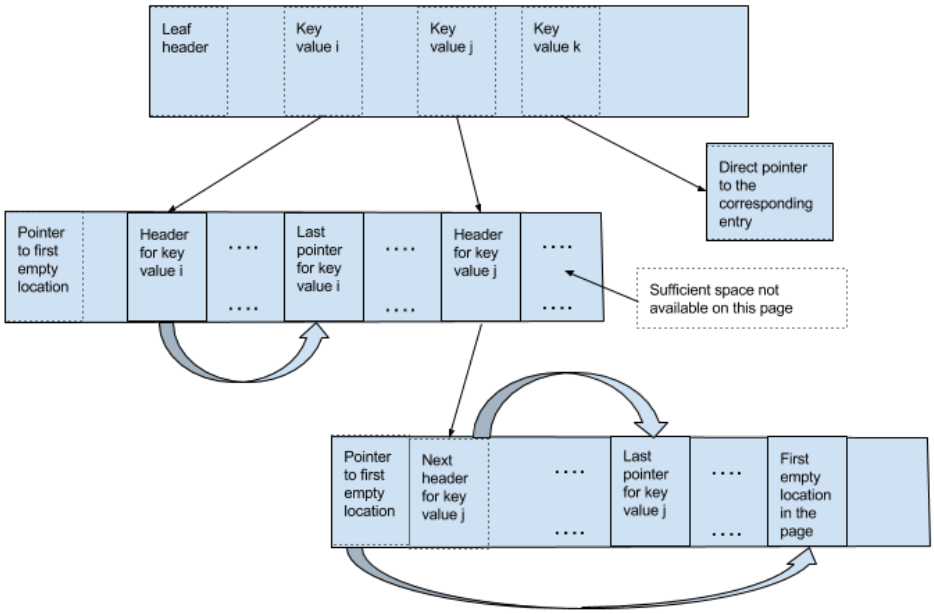

Figure 1: Storing secondary index in a bucket of pointers

4 Function descriptions

We have added all the functions required for implementation of bulk loading in the file bulk_load.c. It consists of the following three functions.

4.1 function InsertEntry()

This function handles the addition of an entry to the B+ tree, according to the algorithm, i.e. into the right most leaf, calling the functions AddtoParent and InsertintoLeaf to help it do so.

```
InsertEntry()
int *temp; // current bag page number
int fileDesc; // file Descriptor
char *value; // value to be inserted
int recId; // recId to be inserted
char attrType; // 'i' or 'c' or 'f'
int attrLength; // 4 for 'i' or 'f', 1-255 for 'c'
int last; // whether the value to be inserted is the last value
```

4.2 function AddtoParent()

This function inserts the required key to the parent level, which results from addition of a key to the lower pointer and makes the appropriate pointer changes to make the upper level entries point to correct pages on the lower level. This is done recursively.

```
AddtoParent()
int fileDesc;
char **rightmost_buffer;
int *num_nodes;
char *value; /* key value */
int level; /* level at which the node which has to be added to parent is present */
int *rightmost_pageNum;
int attrLength;
int *length; // length of rightmost_pageNum and rightmost_buffer array
```

4.3 function InsertintoLeaf()

InsertintoLeaf function inserts an entry into a leaf and return TRUE on a successful insert. If there is not enough space in the rightmost leaf, it returns FALSE, and then the function InsertEntry() allocates a new page, and then calls it again to insert the leaf into the newly created leaf. It also inserts the recId into the bag of pointers of the corresponding key, if the key already exists.

```
InsertintoLeaf()
int fd; // file descriptor
int attrLength;
char attrType;
int *bagPage;
char *pageBuf; // buffer where the leaf page resides
char *value; // key value to be inserted
int recId; // recid of the attribute to be inserted
```

---

5 Visual display

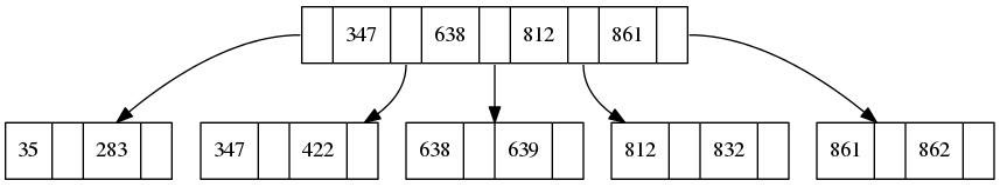

Figure 2: Dot graph generated by print function for a small tree

6 Performance Metrics

In the performance analysis, we test the above two methods of bulk loading on a range of data (varying the number of keys to be bulk loaded) and measure certain performance metrics to compare against the naive unsorted-keys insertion algorithm. Note that we will only be testing performance for the case when the initial B+ tree is empty, because otherwise, the bottom up building cant be used. The metrics used are:

- Time taken to build the index
- Number of read buffer hits
- Number of write disk accesses
- Number of read disk accesses

Insert Type	#Records	Page Size	#Nodes
Bottom Up	5000	4096	17
Bottom Up	5000	1024	67
Sort	5000	4096	21
Sort	5000	1024	86
No Sort	5000	4096	19
No Sort	5000	1024	84
Bottom Up	7000	4096	23
Bottom Up	7000	1024	91
Sort	7000	4096	34
Sort	7000	1024	115
No Sort	7000	4096	33
No Sort	7000	1024	111
Bottom Up	10000	4096	31
Bottom Up	10000	1024	123
Sort	10000	4096	38
Sort	10000	1024	176
No Sort	10000	4096	37
No Sort	10000	1024	167

Figure 3: Values of number of nodes in the index tree as a function of number of records inserted and the page size

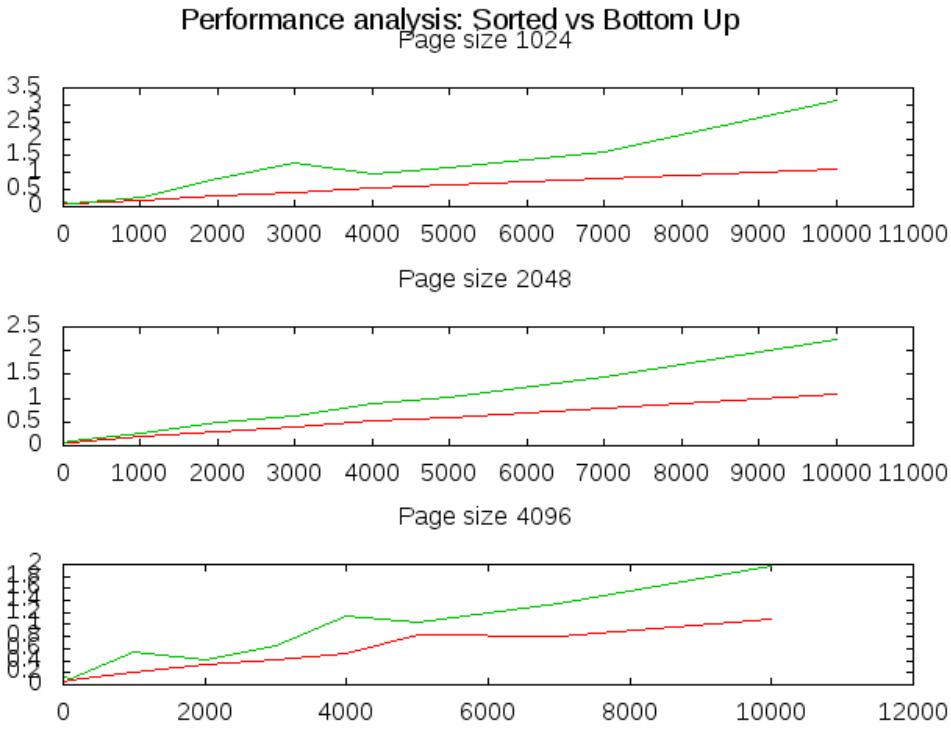

Figure 4: Red: Bottom up, Green: Sorted . x: #keys, y: time in ms

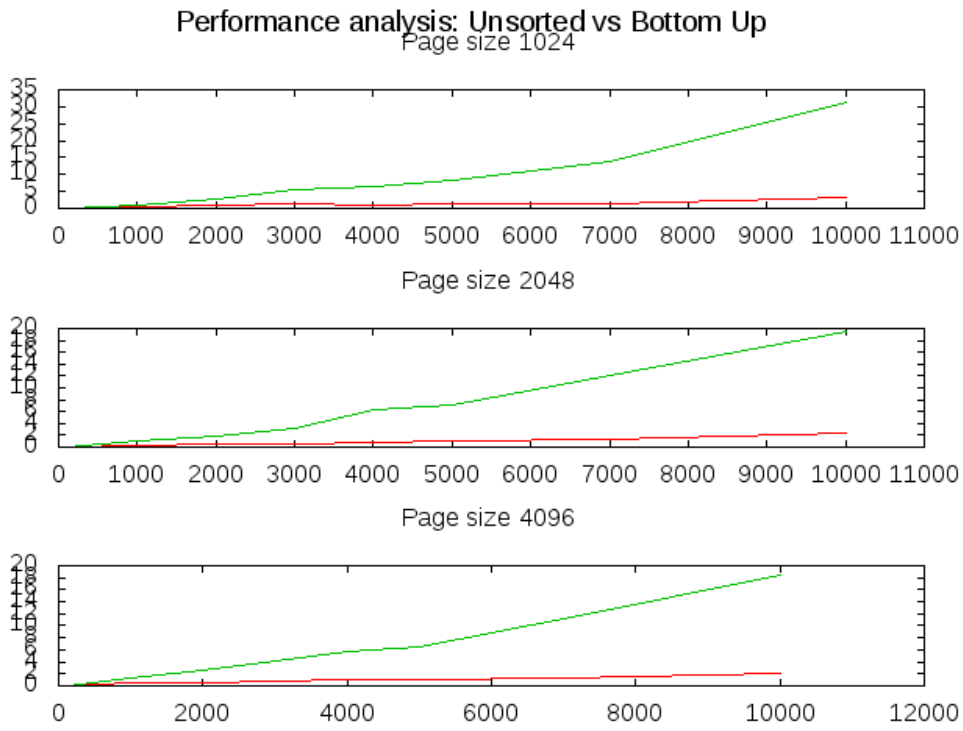

Figure 5: Red: Bottom up, Green: Unsorted. x: #keys, y: time in ms