Internals Final Report

Performance analysis of B+ tree and insights into B tree implementation

Ankit Rathore	Amit Malav	Abhijith Dimitrov
130050029	130050032	130020012

Introduction

Overview of the B+ trees

Both B+ and B trees are important data structures used by a wide range of DBM Systems.
They are different with respect to the following two properties:

1) B+ trees don't store record pointer in interior nodes, they are only stored in leaf nodes. In B-trees both inner and leaf nodes store record pointers.
This typically leads to lower height tree and better cache-hit rate on inner nodes in case of B+ trees as they can now can fit more keys on the same block of memory.
However you need to traverse down to the leaf node to get the pointer which may not be the case in B-tree, for eg., if frequently accessed nodes lie closer to the root it can be accessed more quickly.

2) The leaf nodes of B+ trees are linked, so doing a linear scan of all keys will requires just one pass through all the leaf nodes. A B tree, on the other hand, would require a traversal of every level in the tree.
This property can be utilized for efficient search in case of range queries since data is stored only in leafs.

Design of Project and related Algorithms

We are comparing the various properties of the index tree generated and performance parameters described as follows:

Insertion, Search, Deletion Time

We use the function gettimeofday() to get the timestamps before and after performing the required operation.,Subtract to obtain the time required for operation and compare it for inserting records in sorted versus random order. In both cases we are inserting , searching for or deleting the same number of records so that we can fairly compare both.

Nodes and levels in trees

We have written functions AM_numnodes and AM_numlevels which take the root node as input and give the number of nodes and the number of levels respectively in the tree generated after inserting all records.
The pseudo code is as follows::

```
Function (root node){
        num_of_levels=0;
        num_of_nodes=1;
        for each child 'c' of this root node{
                //(c is the root node of the child tree)
                num_of_nodes += Function(c).num_of_nodes;
                num_of_levels = max (Function(c).num_of_levels, num_of_levels);
        }
        num_of_levels++;
        return (num_of_nodes, num_of_levels);
}
```

Space Utilization

We can add an extension to the above mentioned function to output the total used space in all the nodes of the tree. we can compare the space utilization by comparing the fraction (total_used_space/total_space_allocated).
We can find the total_space_allocated by multiplying num_of_nodes with PF_PAGE_SIZE (size of a node) and the total_used_space will be sum of used space (space occupied by header,keys and pointers in node so far) for all nodes.
For an **internal node**, used space is :
size of header + (attriblength + node-pointersize) * header->numKeys + pointersize.
For a leaf node, used space is :
size of header + (attriblength + record-pointersize) * header->numKeys.
The extra pointersize term in internal node is because there is (1 + number of keys) pointers in an internal node.

Buffer Hit Ratio

Whenever we access a page i.e a node in the tree for reading/writing, we need to check whether the page is in the buffer or not. If not, it has to be loaded from the disk which takes time several order magnitude greater than fetching from the cache buffer. Hence this a very important metric.

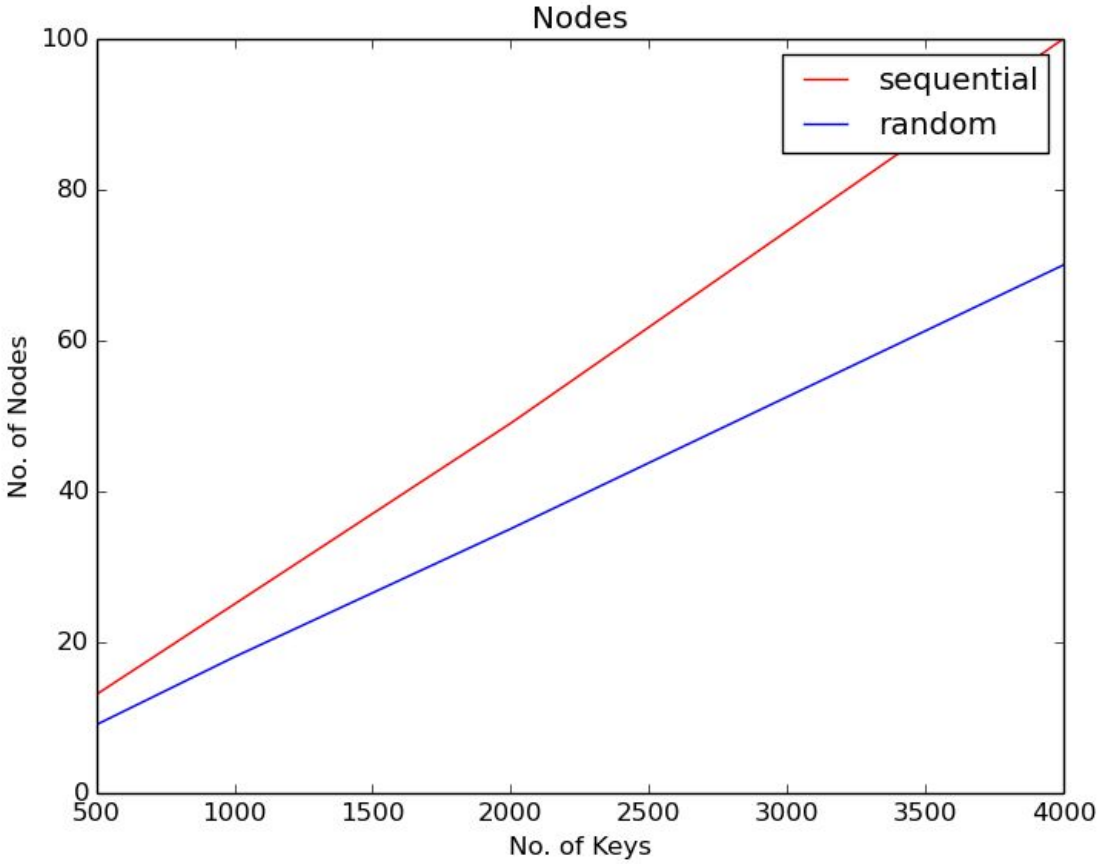
We can find whether a page access results in a buffer hit or miss depending on whether it enters the if or else branch in the function PFbufGet(). To count the total number of hits

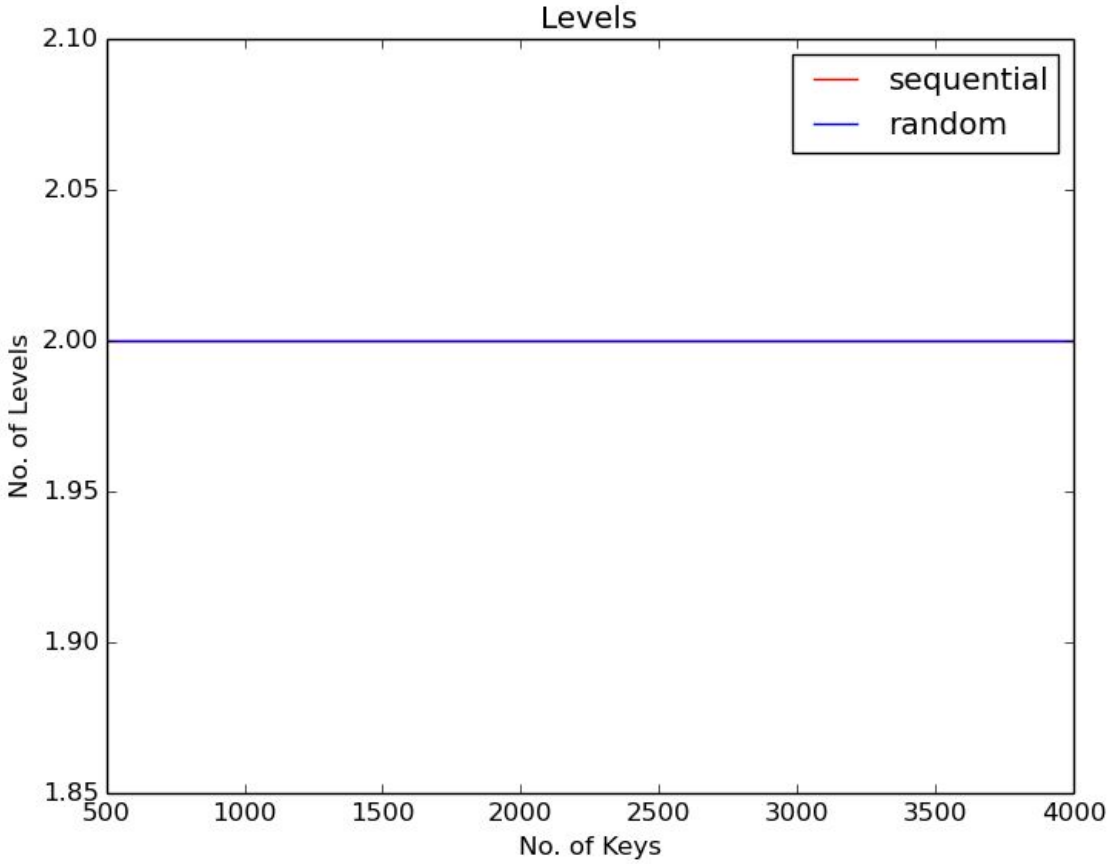
and misses which occur as the B+ tree nodes are accessed during insertion, search and deletion operations, we keep global variables buffmisses and buffhits which are incremented whenever the corresponding branch is taken in PFbufGet(). These global variables are reset everytime we want to caclulate hit-ratio for different operations,
Hit-ratio=num_of_buffer_hits/total_num_of_accessses = buffhits / (buffhits+buffmisses)
In addition to comparing the metrics for insertion in sorted versus random order, we also decided to verify whether the number of nodes and levels in tree increase as expected with increase in number of records inserted (for eg, number of levels will vary logarithmically with respect to the number of keys/records) . Another observation we made was that buffer-hit ratio decreases with increase in size of tree because only a smaller fraction of the tree can be stored in the buffer hence the probability of a node being in the buffer decreases.
We have plotted graphs of the different metrics for different tree sizes in both the cases.

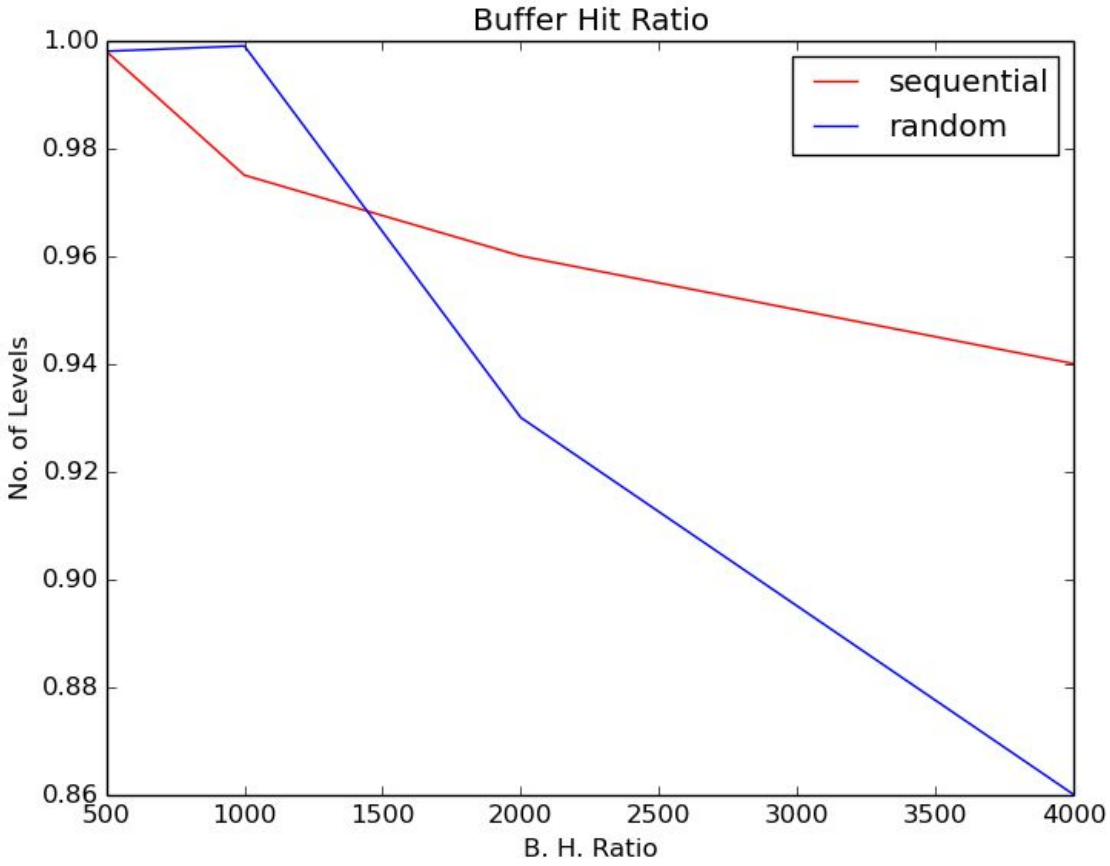
Code changes and new implementation

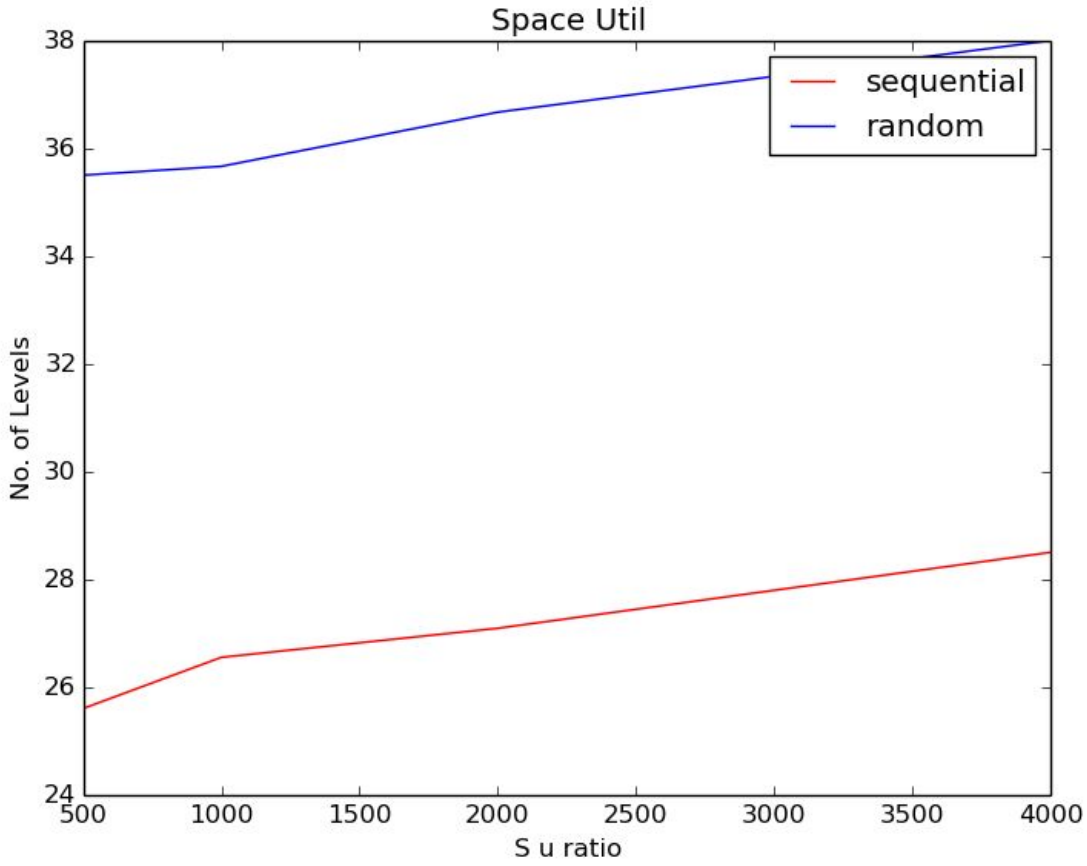
As mentioned previously, we have two new functions AM_numnodes and AM_numlevels whose roles are self-explanatory.We have calculated the total used space in all nodes in the AM_numnodes function itself by assigning a global variable occ_space and incrementing it appropriately in each node as the function is called recursively.We have modified PF_bufGet() to get number of buffer hits and misses during page accesses.
We have two testcases main.c and main1.c which insert records in sequential and random order respectively and are not using the testcases given in toydb.

Performance Measures


Nodes
sequential
random
No. of Nodes
No. of Keys
0
20
40
60
80
100
500
1000
1500
2000
2500
3000
3500
4000


Levels
sequential
random
No. of Levels
No. of Keys
2.10
2.05
2.00
1.95
1.90
1.85
500
1000
1500
2000
2500
3000
3500
4000


Buffer Hit Ratio
sequential
random
No. of Levels
B. H. Ratio
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
500
1000
1500
2000
2500
3000
3500
4000



B tree implementation

In B-trees both inner and leaf nodes store record pointers.
This typically leads to lower height tree and better cache-hit rate on inner nodes in case of B+ trees as they can now can fit more keys on the same block of memory.
However you need to traverse down to the leaf node to get the pointer which may not be the case in B-tree, for eg., if frequently accessed nodes lie closer to the root it can be accessed more quickly.
The leaf nodes of B+ trees are linked, so doing a linear scan of all keys will requires just one pass through all the leaf nodes. A B tree, on the other hand, would require a traversal of every level in the tree.
This property can be utilized for efficient search in case of range queries since data is stored only in leafs.

To implement the B trees we aimed to do the following code changes in toyDB.

Header Structures

For the leaf nodes the header structure remains same and the structure of the internal nodes is implemented in somewhat similar fashion to the leaf header with one minor change that we stored the pointer to the child nodes adjacent to key values.

AM_Search

In case of B+ tree while searching through the tree we had to search all the way through the leaf node to either get the value or it's place where it can be inserted to. While in case of B trees even if we get the desired key value in some internal page, we stop iterating down and return the probable index of the key in that particular page.

For searching in a page/node we used binary search to find the key value or the next page where it might possibly be or can be inserted and this is iteratively implemented.

AM_InsertIntoNode

Inserting an entry in the B trees is different from B+ trees. in case of B trees the values can also be inserted in internal pages as well. Along with the key values in the Internal nodes we also stored the actual pointers to Records in our relation. Hence this reduces the overall no. of keys that can be placed in one single node.
However unlike B+ tree we store one key value only once in a page of B tree.