CS-387: DATABASE AND INFORMATION SYSTEMS LAB

B+ Trees: Bulk Loading and Analysis

Database Internals Report

Aashish Rathi(130050016) Anand Bhoraskar(130050025) Dibyendu Mondal(130050046)

Objective

- To use External Merge Sort for sorting the data
- To implement Bulk Loading for building the B+ Tree in two ways:
  - Top-down Approach
  - Bottom-Up Approach
- To compare the performance of both approaches and compare them with insertion without sorting
- Analyze the quality of indices created from each of the algorithms (height, num of nodes)

Work Summary

The External Merge Sort algorithm is implemented in the PF layer. The Bulk Loading algorithm is implemented in the AM layer. Test cases are provided as a part of the main program.

External Merge Sort:

External Merge Sort is necessary when we cannot store all the data into memory. The best we can do is break the data into sorted runs and merge the runs in subsequent passes. The length of a run is defined by the buffer size.

Bulk Loading:

Indexes are primarily optimized for inserting rows one at a time. But this method turns out to be inefficient when a bunch of data is being inserted. For instance, with a B+ Tree, the optimal way to insert a single key is very poor way of adding a bunch of data to an empty index. In such cases, bulk loading is an efficient way to create indices.

Program Design

External Merge Sort:

Since the files with missing pages are a bit problematic, we modify the file initially to condense it into contiguous sets of pages. Then the mergesort algorithm was used on that file.

There is a temp file, which is used to temporarily store the merged runs. Let's say that the buffer size is B pages. In pass 0, we sort all the pages in place by loading in the memory and writing back to temp. We call them runs now.

In the following passes, we can no longer do the operations in place since we're merging runs. So we load B-1 pages into the buffers and use B^{th} as the write buffer. The merging is identical to the merge sort algorithm. Find the buffer with the lowest/highest value(depending on sorting desc or asc), get the value and write it, then go to next value and repeat. Once we're left with 1 run, you write the rest of it to the output. The destination keeps toggling between temp and the initial file.

In these passes, the buffers do not load all the pages from each run into memory at one time. They simply load the first page in each run into the buffer. Once all the items from that page have been written into the newer larger sorted run, the next page of that run is loaded into memory. The write buffer works the same way, once it writes a whole page, it opens a new page for writing.

In the end, if the temp file contains the sorted file, then it is written to the original file. The temp file is removed at the end.

Bulk Loading:

Bulk loading is implemented in 2 ways:

1. **Top down insertion:** The data is sorted using External MergeSort, and inserted in a top down manner into the index. In this approach, the nodes are always half full, as they get filled in the sorted order and split, and none of the new entries go to the old nodes. But this approach is fast in terms of creation time.

2. **Bottom up building:** First the leaf node layer is created by populating nodes completely and connecting till all the entries are finished. Then a recursive function builds the internal node. It starts from a node which is designated as the root, and then recursively builds the subtrees corresponding to all it's children. The height at a point in the recursion is maintained. The base case is reached when the recursion reaches the node just above the leaf node level. There, the node is populated, and the pointed to the next leaf node is passed above. Then the recursion builds the rest of the internal nodes in the similar manner, while handling the keys and the pointers to the leaf nodes as the recursion passes up. This approach is very efficient in terms of space efficiency of the B+ tree, as almost all the nodes are completely filled. Hence, the height is smaller, and the accesses are faster.

Functions Used

External Merge Sort:

PF_CreateFile, PF_OpenFile, PF_GetNext Page, PF_GetThisPage, PF_UnfixPage, PF_CloseFile, PF_DestroyFile

Bulk Loading:

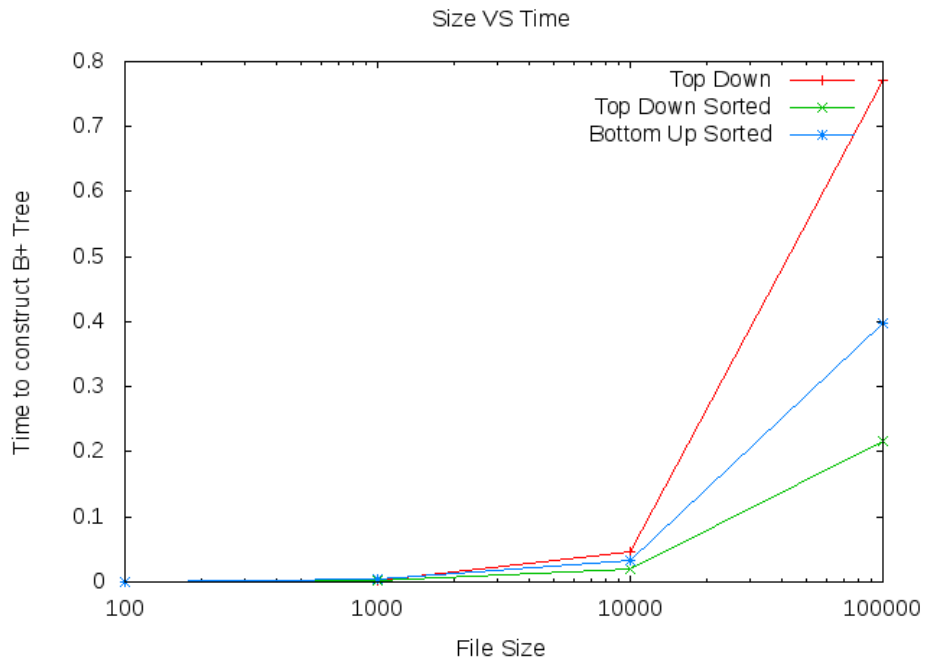
BottomUpBulkLoad, CreateLeaf, changeNextPtrLeaf, createIntLayer, bcopy, AM_InsertintoLeaf, PF_AllocPage, PF_GetNext Page, PF_GetThisPage, PF_UnfixPage, AM_AddtoIntPage In this, we have defined the 1st four functions.

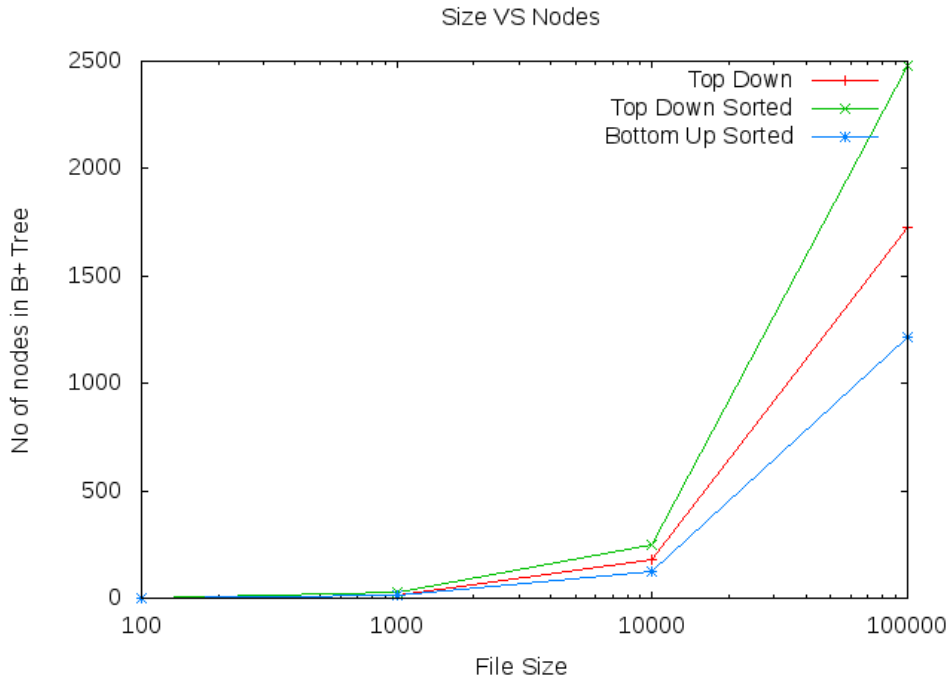
Others:

AM_GetNumOfNodes, AM_OpenIndexScan, AM_FindNextEntry, AM_CreateIndex, AM_InsertEntry, AM_CloseIndexScan, AM_DestroyIndex, clock, srand. In this, we have defined the 1st function.

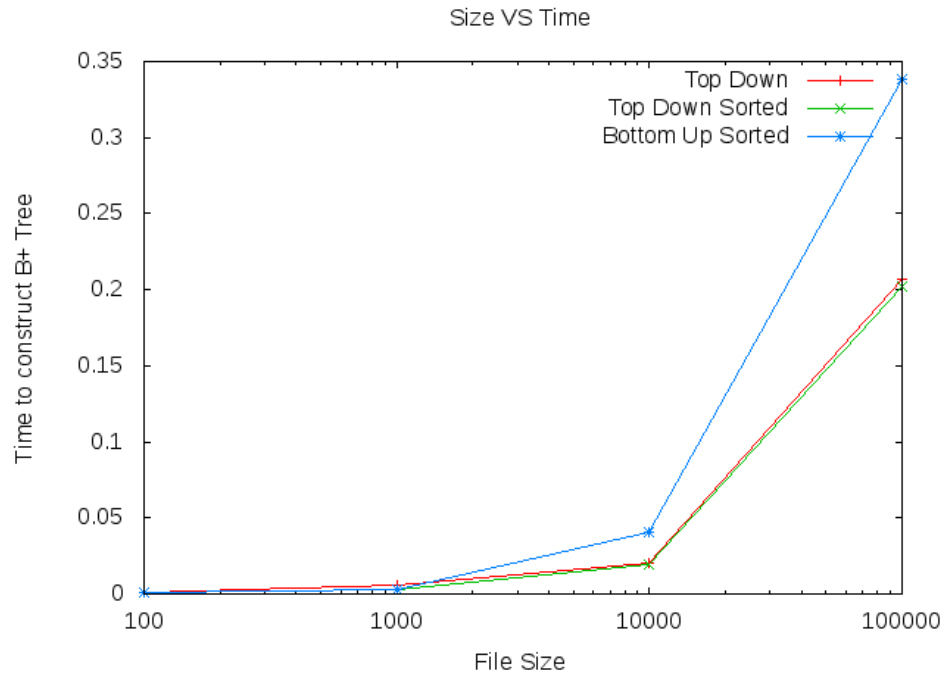
Analysis

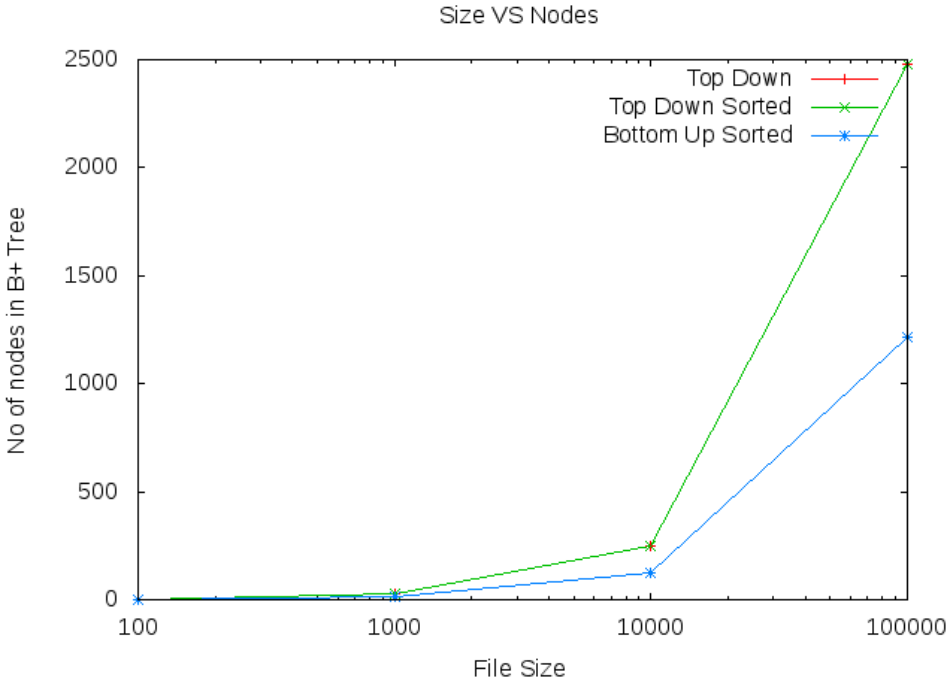
Random Data:





Sorted Data:


Size VS Time
Time to construct B+ Tree
File Size
Top Down
Top Down Sorted
Bottom Up Sorted
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
100
1000
10000
100000


Size VS Nodes
No of nodes in B+ Tree
File Size
Top Down
Top Down Sorted
Bottom Up Sorted
2500
2000
1500
1000
500
0
100
1000
10000
100000

Conclusion

After analysing the three approaches, we observed that bulk loading after sorting gives a definitive improvement in performance, in terms of time taken.
Among the two approaches, the bottom up approach gives us the optimal way to store the tree. So it is extremely efficient in terms of memory. However, the top down approach gives the least memory efficient way of storing the data.

In the case when the data is already sorted, we see that the overhead of sorting can be a bit high, but still the index obtained from bulk loading is still better in quality, so that compensates for it. The high overhead of the merge sort is also increased by the fact that it also checks for the file condition. If we assume that the file has no missing pages and just sort it, then the overhead is reduced considerably.