CS 386 - DB Internal Project - Final Report

B+ Tree Scan Vs External Merge Sort Analysis

Team Members

Palak Jain (130050031)
Vaibhav Bhosale (130050007)
Siddharth Bulia (130050012)

Introduction

B+ Trees

B+ tree is an organizational structure for information storage and retrieval in the form of a tree in which all terminal nodes are the same distance from the base, and all non-terminal nodes have between *n* and 2 *n* subtrees or pointers (where *n* is an integer). It keeps data sorted and allows searches, sequential access, insertions and deletions in logarithmic time. It is optimised for systems that read and write large amount of data. Hence, it is used widely in databases and file systems.

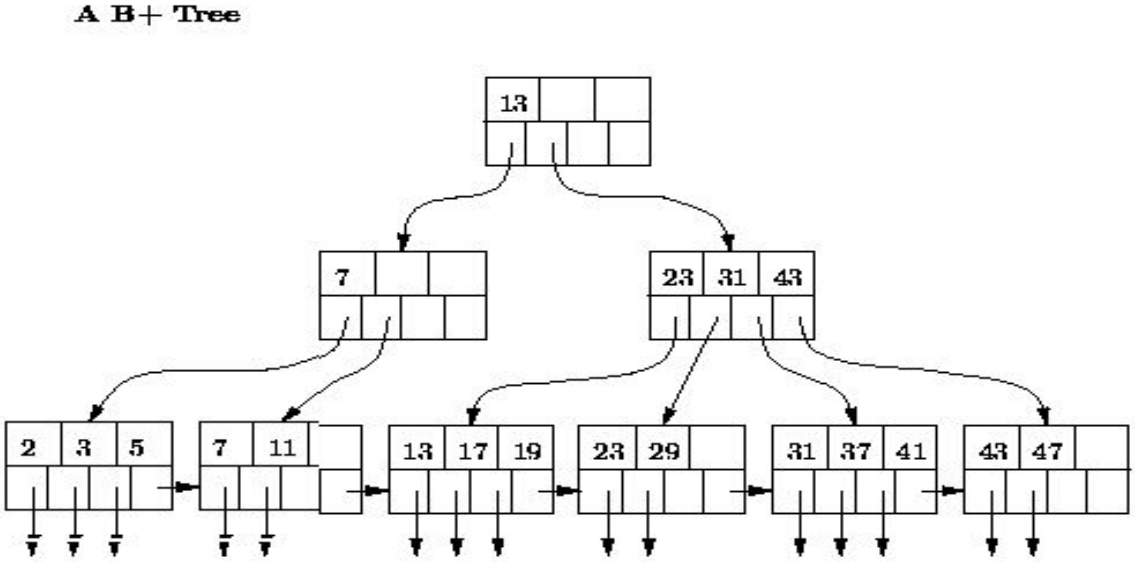


image reference - http://infolab.stanford.edu/~ullman/dbsi/win98/gifs/B+tree.gif

External Merge Sort

It is a type of sorting algorithm which is capable of handling massive amount of data. This is required when the data being sorted does not fit in main memory (RAM) of a computing device and instead resides in the slower external memory (hard disk). External merge sort loads chunk of data which is small enough to fit in main memory, sorts it and writes it to a temporary file. In the merge phase all these sorted files into a final output file.

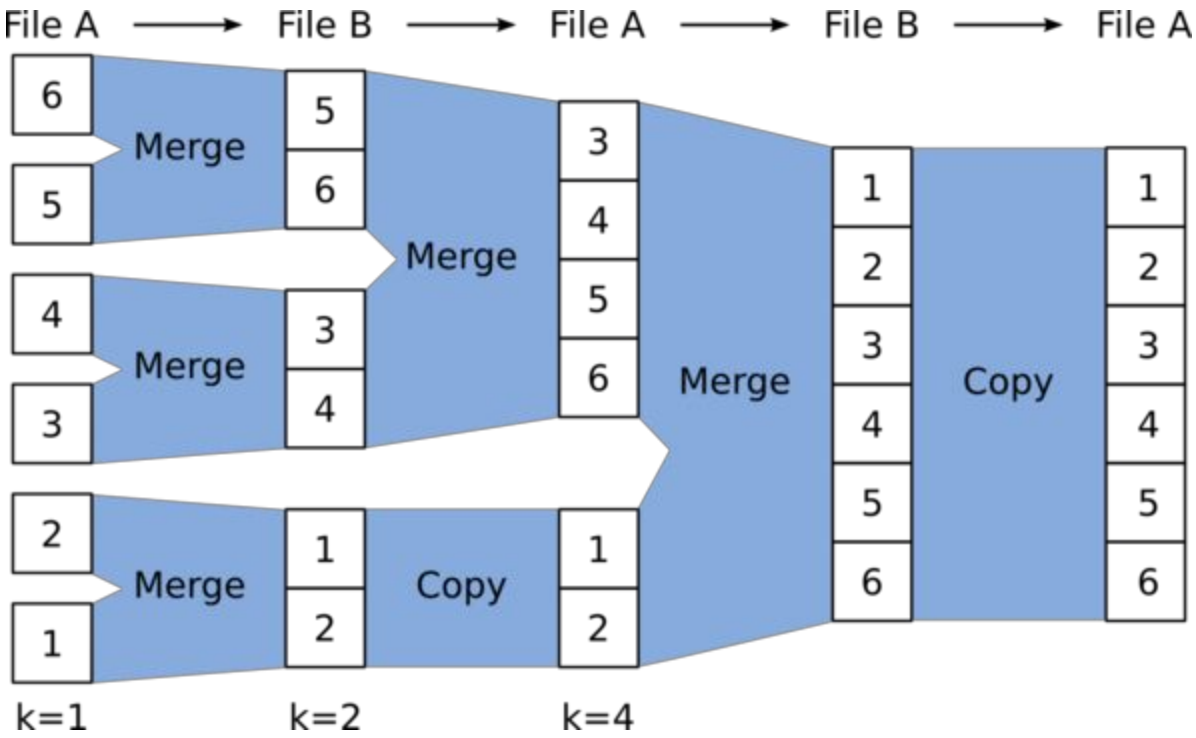


image reference - https://www.cl.cam.ac.uk/teaching/0910/lbAsExBrfg/figures/merges-small.png

---

Design

We were provided with toydb code which had B+ tree functionalities implemented. So we studied this code and used its functions for our analysis.

B+Tree

B+ tree is implemented in the AM layer. With the help of AM layer functionalities, the key idea here being creating an index over a file and then inserting entries into it. To access a particular set of values, this can be scanned with help of pointers at the leaf nodes.

External MergeSort

We implemented external merge sort in the PF layer using PF layer functionalities. Our major idea here is to sort a file whose size is bigger than the buffer size. For this, we individually wrote content of file into buffers, sorted each buffer and created a new file by merging all the buffers step by step.

Implementation Details

B+ Trees

Insertion into B+ trees

- We initialize a file using **PF_Init** and create an index over it with the help of **AM_CreateIndex**.
- Using the file descriptor obtained from the paged file, we insert entries into the created indexed file using **AM_InsertEntry**
- The function takes as argument file descriptor, attribute type, size of attribute, address of the attribute value to be inserted into the index and record id which indicates the record corresponding to the fle.
- For insertion, Search the tree to find the leaf node where the new element should be added using n-ary search (**AM_Search**). It attempts inserting the (value, recId) tuple into the tree (**AM_InsertintoLeaf**). If the node is full, the function returns false. In that case, split the node i.e. create a new node, insert the tuple and add its pointer to parent.

Scanning B+ trees

- We use AM_OpenIndexScan to initiate scanning of an indexed file.
- This function takes as arguments a file descriptor, attribute type, the length of the attribute, the desired operation and the value which is being compared with.
- This is done to get the records whose indexed attribute value compares in the desired way(greater than/less than/equal to) with the value parameter.

External Merge Sort

Creating Runs

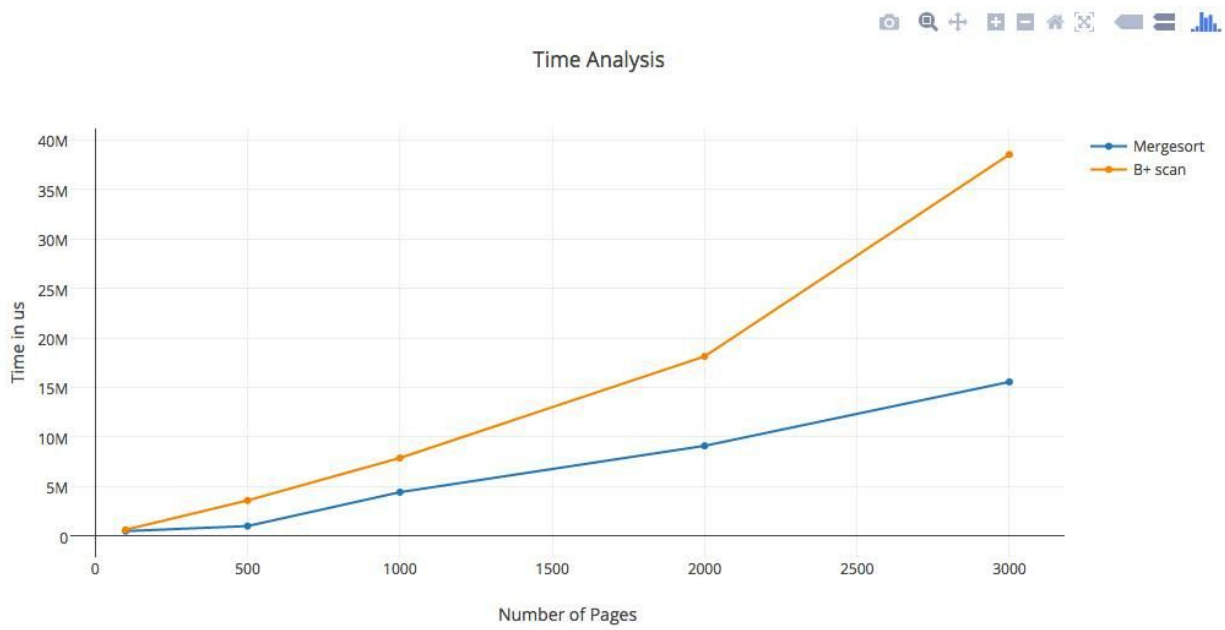
1. Open input file using **PF_OpenFile**.
2. The page numbers and buffer addresses are iteratively stored into two arrays using function **PF_GetNextPage**.
3. The data on the buffer is copied to a temporary buffer using **WritePage** and sorted with function **mysort.**
4. The buffers are allocated to files using **PF_AllocPage** and then finally written on to the file using function **PF_UnfixPage**. This function also clears the memory assigned to the buffer.

Merge Phase

1. Files are created for the current pass based on the number of files in previous pass. For the first pass, files are same as the initial runs created **Run phase**.
2. File names are created using function **createname** and files created and opened using **PF_CreateFile** and **PF_OpenFile**.
3. A buffer array is initialised with size one less than maximum number of buffers available and in all these arrays, we read in the files created in the current pass to prepare them for the merge.
4. All the buffers read now are merged using function **merge**.
5. All the open files are closed using **PF_CloseFile**.
6. Steps 1-5 are repeated to generate all the necessary files for the next pass until all the files of this pass is read.
7. Steps 1-6 are repeated till we are left with only a single file.
8. This final file contains the required output.

Performance Measures

To measure the performance, we have primarily used the sum of initialisation time + traversal time for both the cases.
We tried to compute the above times for the method for different different number of entries. Graph for the same is below.



Conclusion

In our experiment, we found out that merge sort is fast and efficient when we are scanning the entire database.

Citations:

http://people.cs.aau.dk/~simas/aalg04/esort.pdf
http://www.db-book.com/
