DB Internals - Final Report

Jagannath Vishal.R - 130050043

Aditya Kusupati - 130050054

Sanampudi Venkata Sailesh - 130050058

Aim:

The aim of the project is to compare the performance of External Merge Sort and B+ trees, on the basis of time and space required. In particular, we look at the trade offs, based on Initialisation Time, Access Time and Memory requirements.
This helps the programmers make a wise choice depending on their scale and avoid wastage and use the computing device to the fullest extent.

Introduction:

Data is generally stored in file systems. There are many algorithms, targeting efficient reading and writing of data into the file systems. Two popular algorithms, in this context, are B+ trees and External Sorting. We create B+ tree index on a key. To access any element, we then use the tree to find out the position of the data, and then access the data. In external sorting, we store the data sorted in the order of the key. We only need to access the pointer corresponding to the location.

Data used:

We used sequential data and have shuffled it using random function written within the code. We have used this as test data and data size is given as input. We have used this in order to ease the implementation for access and insertions.

Final Detailed Design:

We have implemented External Sorting and simulations in C/C++ languages on ToyDB. The tool has two layers namely the Paged File Layer (PF Layer) and the Access Method Layer (AM Layer).

We used the functions implemented in PF layer for various purposes like page access, page manipulations, page creation, using the inbuilt data structure of B+ tree and the pre implemented scan, insertion and delete functions on the same.

The entire implementation of the External merge sort takes place in AM layer. The design for the same is as follows including algorithm and other intrinsic details:

1. Assume a file(named disk) to be the disk in our simulation. We have various pages/blocks of data entries in it assuming its size to be large enough for any size of data
2. Assume a character array(main memory) to be the RAM in our simulation. We have a fixed size of it in terms of number of pages/blocks which can be changed as a part of the simulation. Let us call this size to be M
3. Creating Runs: Repeat the following steps until the disk is completely read
    a. Read M blocks into the main memory
    b. Sort the blocks in memory, using regular sorting techniques, quicksort in this implementation
    c. Write sorted data into a new file, call this a run

    Let the number of such runs created be N.
4. Merging Runs:
    a. Case 1: N<M
        i. Read one block from each of the N runs into the main memory. Note that since N<M, we still have at least one vacant block in the main memory. This vacant block is used as the output buffer
        ii. Repeat the following steps until all the runs are empty:

1. Select the smallest record among the N buffer blocks in the memory and write it into the output buffer. If the output buffer is full, then write it into the disk
2. Delete the selected record from the main memory. If the block from which the record was selected is now empty, move the next block of the same run as the deleted record, if available, into the main memory

b. Case 2: N>=M. Here the N first blocks picked above can not fit in the main memory. Hence, here we will have multiple merge passes
    i. In each pass, groups of **min(total remaining runs, M-1)** runs are merged into a single run with size equal to the combined size of all the runs merged
    ii. Repeated merge passes are done until the runs have been merged into one single run
    iii. This single run is written back to disk in the place of original data. Thus we have sorted data in the disk file and can be accessed later.

Accesses:

We generate given number of random indices within range and which are existent in order to perform the experiment of random accesses on B+ tree and Sorted data.

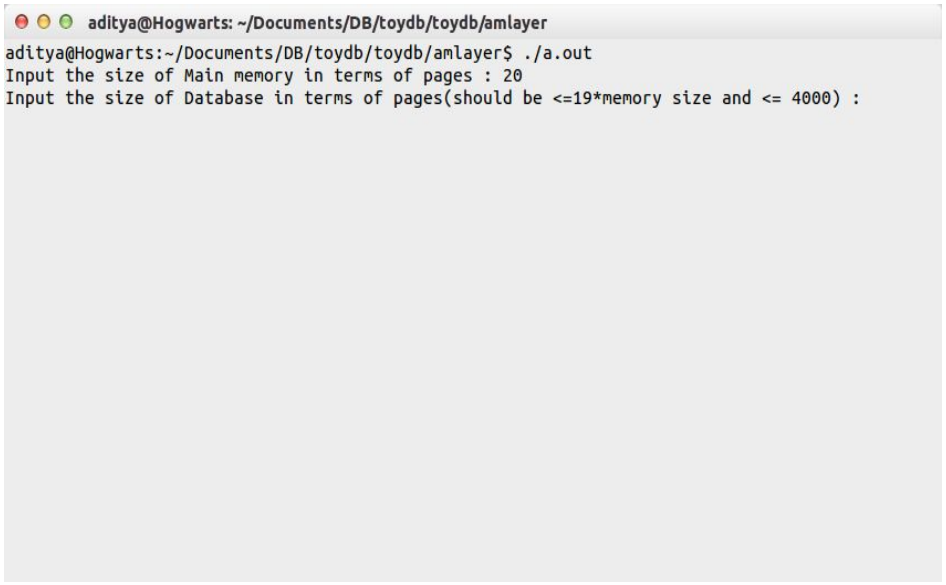
Input/Output:

Input-1 : Main memory size in terms of pages.

This gives us the RAM size we are using in the simulation.

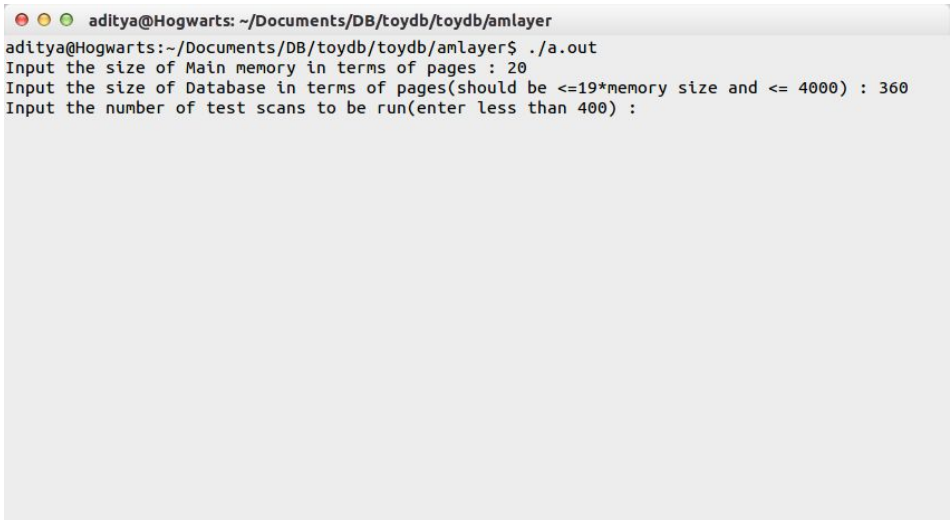
Input-2 : Total size of Database in terms of pages.

There are certain limitations in this case which will be explained in limitations.



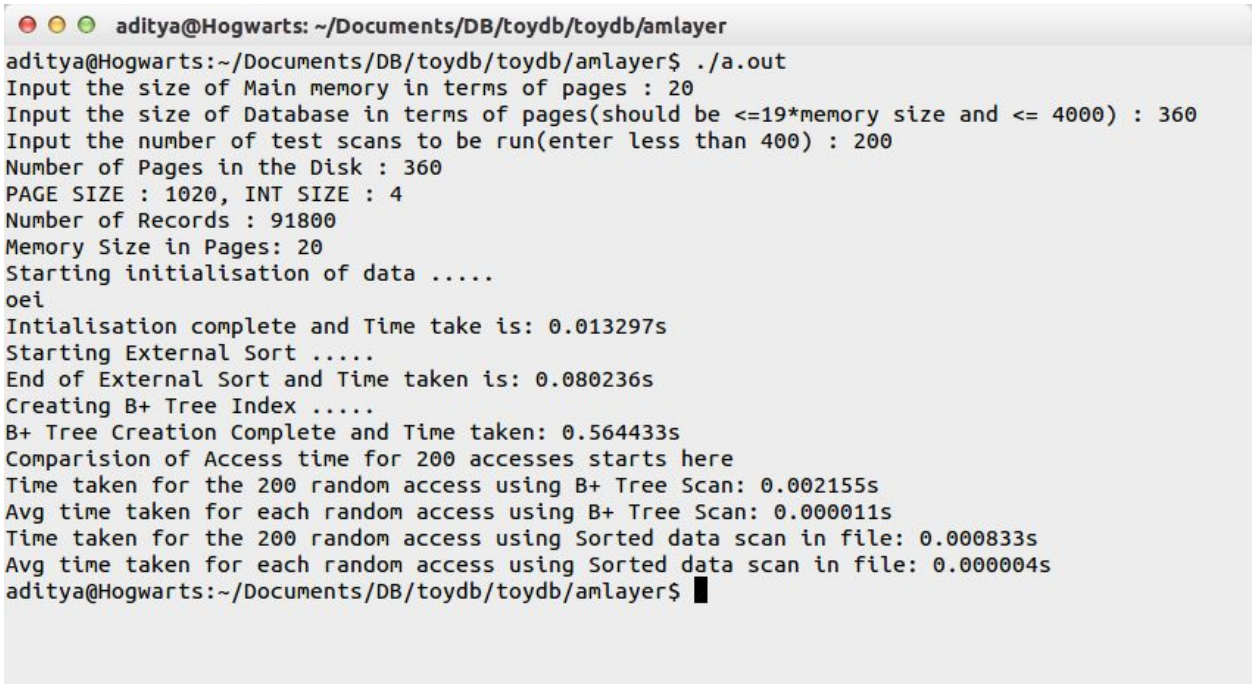
Input-3 : Total no:of scans we wish to make for comparision.

There are certain limitations in this case which will be explained in limitations.



Final Output:

1) Total initialisation time for data creation and other program based initialisations
2) Total time for external merge sort implementation
3) Total time for B+ tree population
4) Total time for given number of random accesses using B+ tree scan
5) Average time taken for each random access using B+ Tree Scan
6) Total time for given number of random accesses using Sorted data scan
7) Average time taken for each random access using Sorted data scan



Comparison between B+ Trees and External Merge Sort:

Initialisation Time:

The initialisation time of B+ trees that are implemented are around 5 times more than the external mergesort we have implemented on an average. The initialisation time of B+ tree can be at best twice the External mergesort initialisation time and worst being around 10 times.

Access Time:

On an average the access time for each access is thrice for B+ trees that are inbuilt in ToyDb when compared to our External mergesort implemented. This difference is because, while in External Sorting, we know the index of the required file, in B+ trees, we do not know the index in the file. This has to be found through a call to the B+ tree.

Limitations of ToyDB found:

1) We can't have more than 19 files related to a single program at any given point of time
2) B+ tree scan using EQ_OP can handle only less than 400 access safely for any given valid input

Limitations of the program written apart form ToyDb's intrinsic limitations:

1) We are using an inbuilt function to create random shuffle of sequential data given the size of the data, this causes segmentation fault if Database size exceeds 4000 pages as we can't create an array of int size 10^6 in C using stack

Conclusion:

Hence, when main memory size is constrained and ease of implementation is required we can implement external merge sort and quickly use in nested join queries and other places. We can also get away from the complex implementation of B+ trees without compromising the efficiency.

P.S:
We couldn’t plot graphs to show the comparisons as the comparisons don't have significant differences in a continuous fashion to be shown, so we have modelled the input and output in such a fashion so as to give the user a feel of simulation done.

References:

1) www.db-book.com
2) ToyDB documentation
3) http://stackoverflow.com/