Project-2_Group-4

Implementing External Sort Algorithms Using Two Layers

Kunal Kumar - 170010012
Shalaj Kumar Yadav -170010015
Sai Sandeep Bareedu -170010018

Work Plan:

1.Merge Sort -

- First of all we will create input pages/disk using function pf_createFile().
- Then we will open file/disk using function PF_OpenFile().
- Now we will allocate buffers by PF_AllocPage(a,&b,&buf).
- Now we will open the file and write code to do useful tasks using functions PF_UnfixPage() and PF_AllocPage().
- We will store ‘files_names’, ‘No of runs’ and ‘pages in last run’.
- We will run sorting Algorithm till we not get error in PF_GetNextPage() function. We will store all pages in an array of size ‘PF_MAX_BUFS-1’.
- Finally we will have No of seeks required in mergesort for read and writes, No. of transfers required in mergesort for reads and writes.

2.B+ Tree -

- First we will create an array to store chars.
- We will have another array to store file names in buffer.
- We will have variables to store record number.
- We will create index on both fields of the record using function xAM_CreateIndex().
- Open the index using function xPF_OpenFile().

- Now insert into index on character using function xAM_InsertEntry().
- Now we will run a loop to find no of records using functions xAM_FindNextKey(), xAM_OpenIndexScan().
- Finally we will close and destroy everything using functions xAM_CloseIndexScan(), xPF_CloseFile() and xAM_DestroyIndex().
- Now we have No. of seek required in bTree for reads and writes, No. of transfers required in bTree for reads and writes.

Pages are identified by their page numbers, which are essentially their physical locations within the file on disk. Access to data on a page of a file involves reading the page into the buffer pool in main memory and then manipulating its data.

Algorithms:

1.Merge Sort- Merge Sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results in a sorted list.

Idea:

- Divide the unsorted list into N sublists, each containing 1 element.
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2.
- Repeat the process till a single sorted list is obtained.

While comparing the two sublists for merging, the first element of both lists is taken into consideration. At each step a list of size M is being divided into 2 sublists of size M/2, until no further division can be done.

2. B+ Tree - A **B+ tree** is an N-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes

and leaves. The root may be either a leaf or a node with two or more children.A B+ tree can be viewed as a B-tree in which each node contains only keys (not key–value pairs), and to which an additional level is added at the bottom with linked leaves.

Algorithm:

- ➢ Searching a node in B+ tree
  - Perform a binary search on the records in the current node.
  - If a record with the search key is found, then return that record.
  - If the current node is a leaf node and the key is not found, then report an unsuccessful search.
  - Otherwise, follow the proper branch and repeat the process.

- ➢ Insertion of node in a B+ Tree
  - Insert the new node as a leaf node
  - If the leaf doesn't have required space, split the node and copy the middle node to the next index node.
  - If the index node doesn't have required space, split the node and copy the middle element to the next index page.

- ➢ Deletion of node in B+ Tree
  - Delete the key and data from the leaves.
  - if the leaf node contains less than the minimum number of elements, merge down the node with its sibling and delete the key in between them.
  - if the index node contains less than the minimum number of elements, merge the node with the sibling and move down the key in between them.

Input design:

1. Block size
2. Pagesize
3. Attr_Length
4. No of entries
5. No. of passes required

Making use of toyDB code:

1. AM Layer :

a. We are using code in AM layer to perform tasks involving opening of file, creating index, accessing and deleting indices.
b. Creating index will be done using AM_CreateIndex() function which takes attributes like filename, indexNo ... and create indices for data.
c. Deleting, inserting an entry is done using AM_DeleteEntry() and AM_InsertEntry().
d. After possible deleting and inserting we destroy the index using AM_DestroyIndex().
e. For opening and scanning through the indices for retrieving the record id is done using functions AM_OpenIndexScan() and AM_FindNextEntry().
f. And using AM_CloseIndexScan() to end the index scan.

2. PF Layer :

a. We are using code in PF layer to deal with paged files. It involves creating, accessing, updating and deleting of paged file.

b. Creating paged file is done by first initiating PF using PF_Init(), and then using function PF_CreateFile, which takes a fileName as input.
c. Opening and going through paged files is done using PF_OpenFile() and accessing each page using PF_GetFirstPage() for first page and using PF_GetNextPage() and PF_GetThisPage() for accessing a custom page using page number.
d. We can create a new page using PF_AllocPage().
e. Later we close the file using PF_CloseFile().
f. Finally we delete the paged file using PF_DestroyFile() which takes a fileName as input.

Output Obtained:

For each input set of Mergesort and BTree we will have :

1. Number of seeks required in mergesort for reads
2. Number of seeks required in mergesort for writes
3. Number of transfers required in mergesort for writes
4. Number of transfers required in mergesort for reads
5. Number of seeks required in BTree for reads
6. Number of seeks required in BTree for writes
7. Number of transfers required in BTree for reads
8. Number of transfers required in BTree for writes

We will compare their performance based on the above parameters.

****